seL4 Overview

Prof Gernot Heiser FACM
NICTA and UNSW, Australia

This workshop is supported by: The NATO Science for Peace and Security Programme

Overview

- **What is seL4, what are Microkernels?**
- What is formal verification, what does it achieve?
- What can you do with seL4?

What is seL4?

NICTA

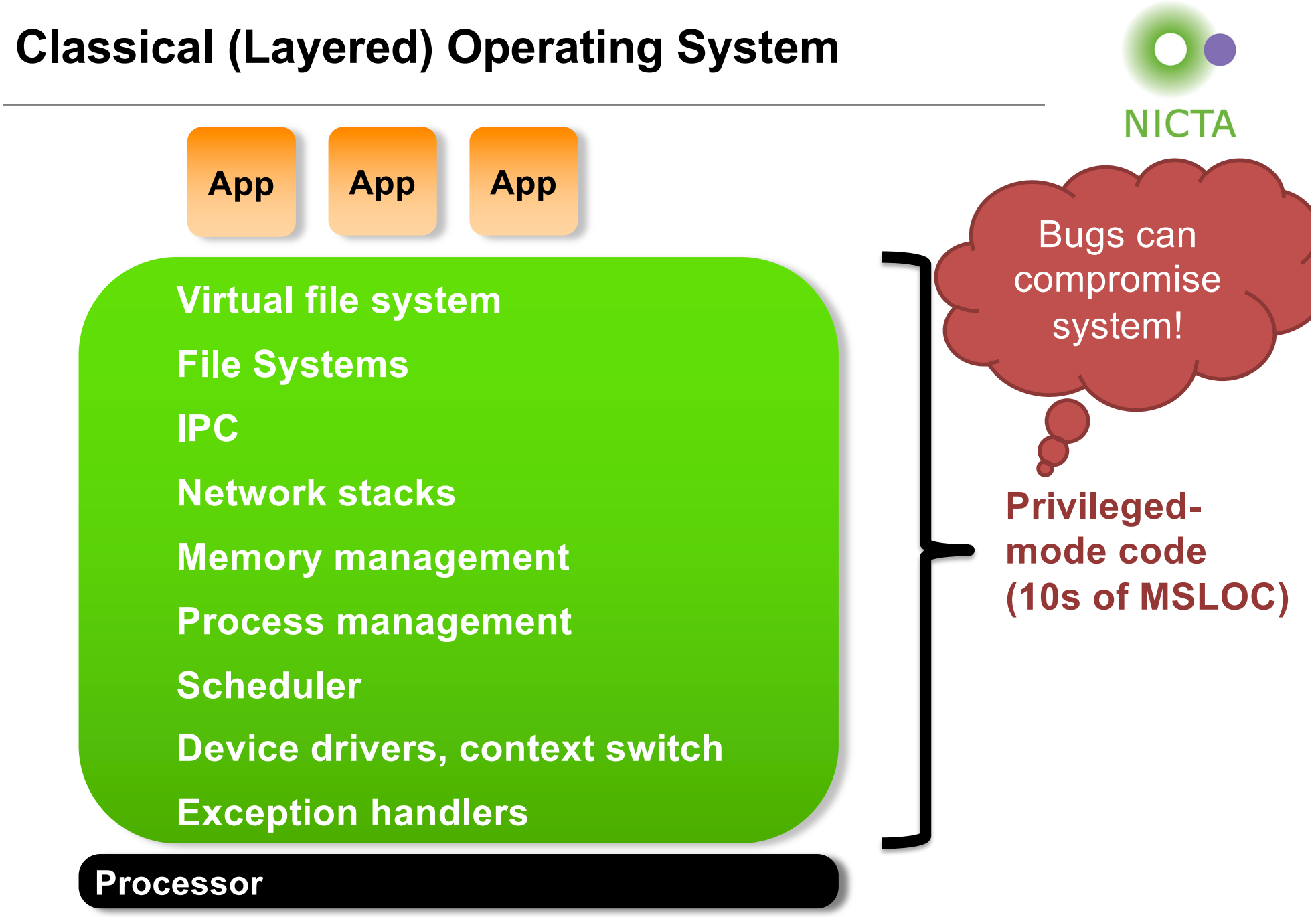
- An operating system microkernel
- … that is proved to be bug-free
- … and proved to enforce security

seL4: The world's only provably secure OS kernel

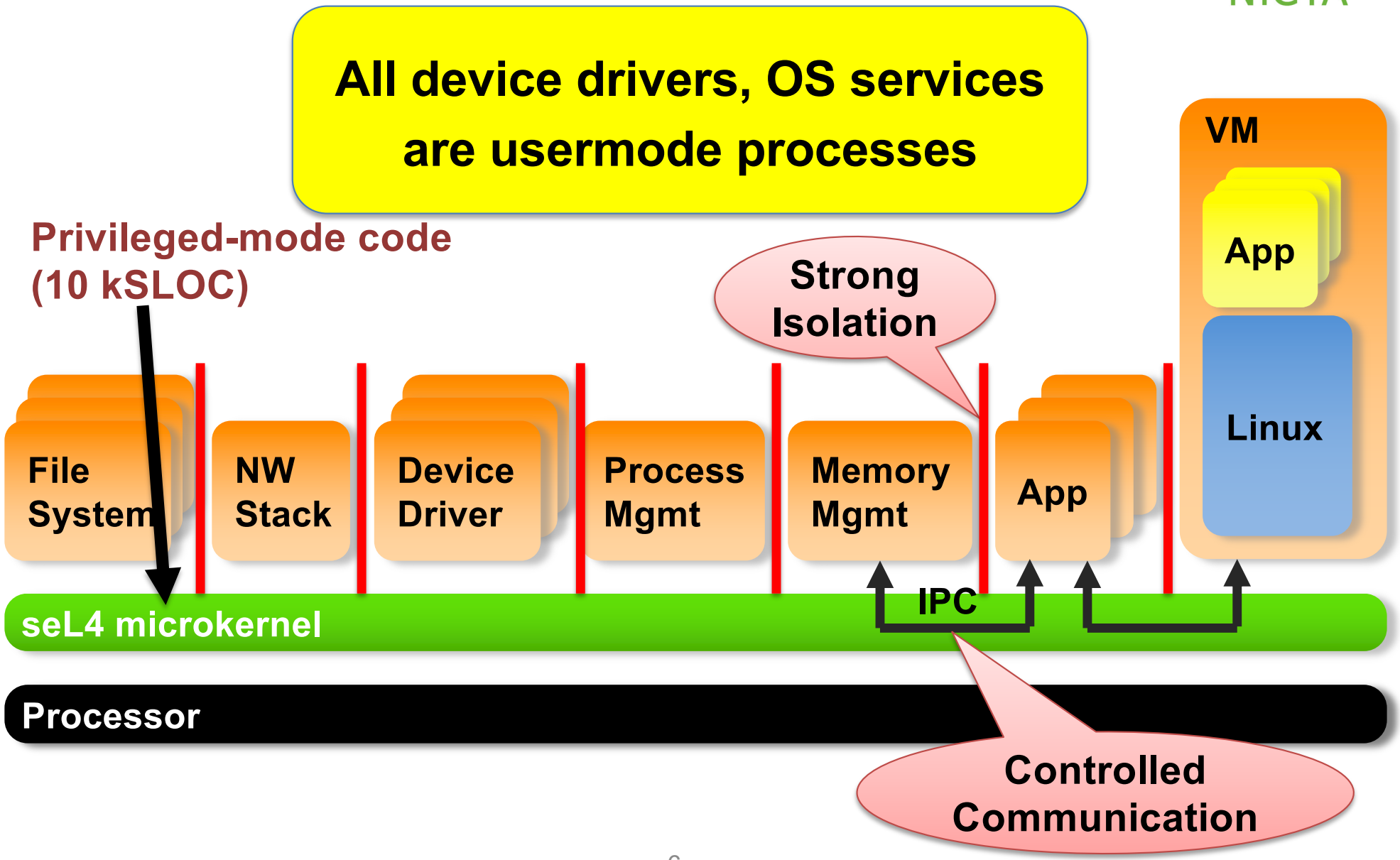
What Is a Microkernel?

- Basic idea :
  - Flexible, minimal platform
  - Mechanisms, not policies
  - Goes back to Nucleus [Brinch Hansen, CACM'70]

Classical (Layered) Operating System



Microkernel-Based Operating System



Why Does Size Matter?

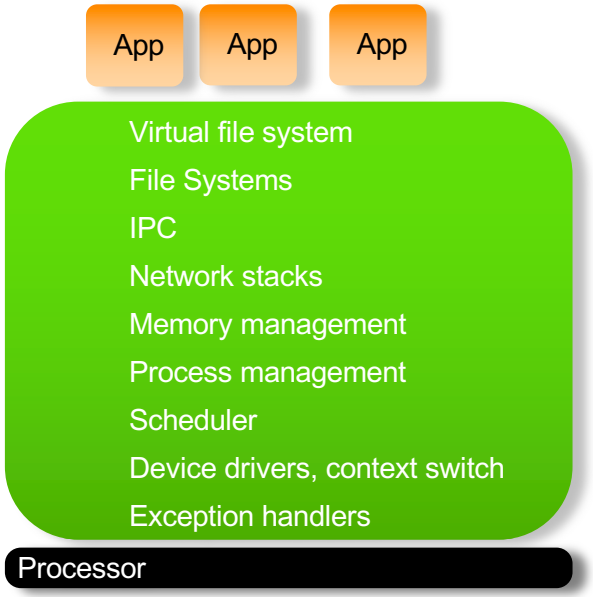
Bug density:

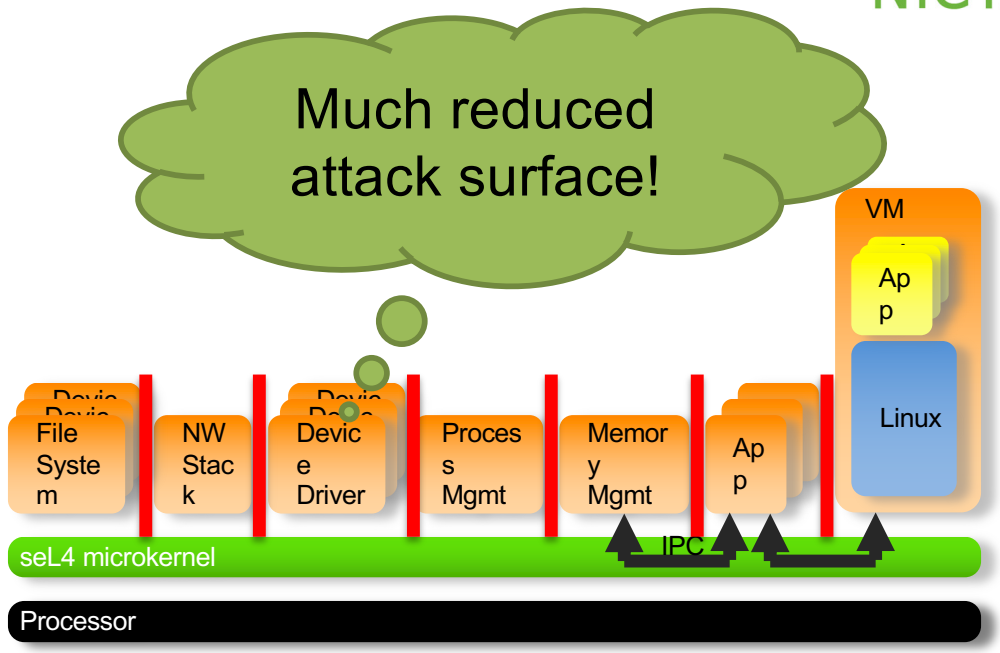
- Normal “quality-assured” code: 2–5 bugs / kSLOC
- “High-assurance” code is lower, but
  - expensive: $500–1000 / SLOC
  - unscalable: cannot maintain low bug density for large code sizes

Bug severity:

- 10–25% of OS bugs are security-relevant (vulnerabilities)

“Monolithic” Kernel vs Microkernel





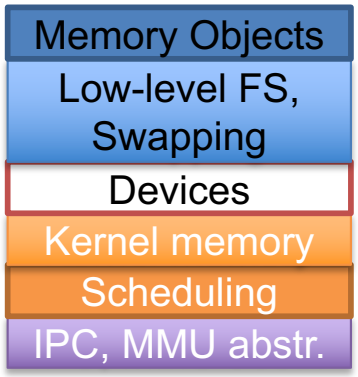
- Policy built in
- Large: 10,000,000 LOC
  - Huge attack surface
  - “trusted computing base” (TCB)

- Mechanisms only, policy at user level
- Small, 10,000 LOC
- **Inherent overhead from IPC: IPC performance critical**

Microkernel Evolution

First generation

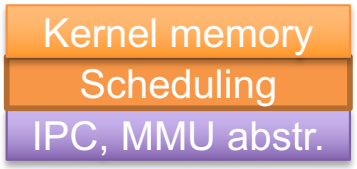
- Eg Mach [’87]



- 180 syscalls
- 100 kLOC
- 100 µs IPC

Second generation

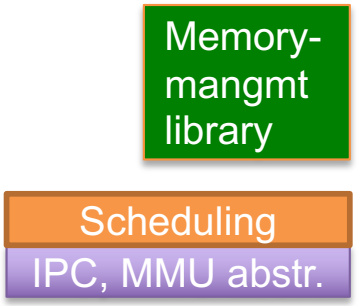
- Eg L4 [’95]



- ~7 syscalls
- ~10 kLOC
- ~ 1 µs IPC

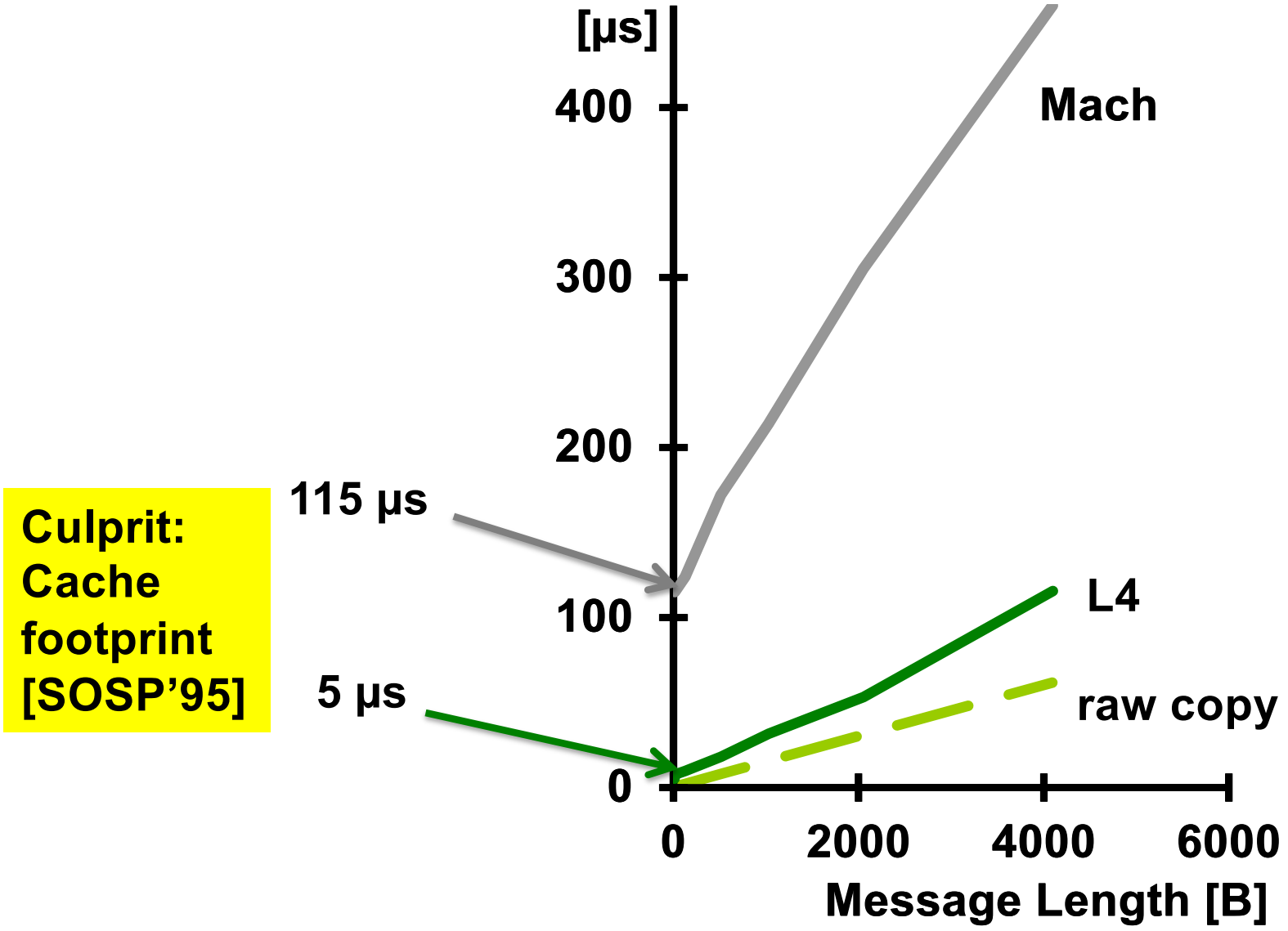
Third generation

- seL4 [’09]



- ~3 syscalls
- 9 kLOC
- 0.2–1 µs IPC

1993 “Microkernel” IPC Performance



Culprit: Cache footprint [SOSP’95]

i486 @ 50 MHz

Core Microkernel Principle: Minimality

A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

[Liedtke SOSP'95]

Examples of Microkernels:

- Mach (CMU): mid-'80s
- QNX Neutrino: 2001, predecessors from '82
- Green Hills Integrity: ca '95
  - Designed for safety-critical use
  - Deployed in avionics
- Minix: since '87

Slow

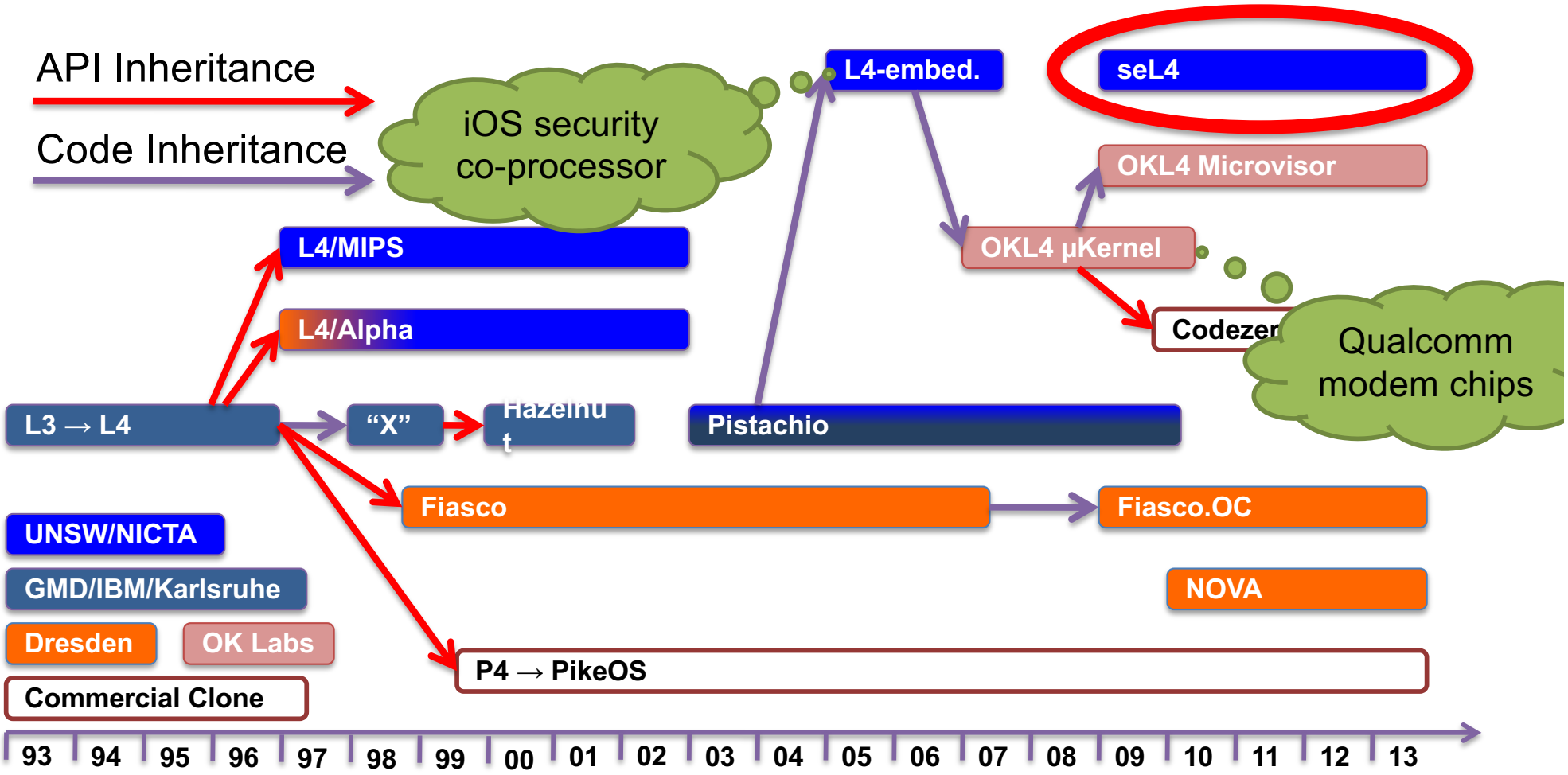
- L4 Microkernel Family (since ~93)

Fast

The L4 Microkernel Family

NICTA

seL4: The latest (and most advanced) member of the L4 microkernel family – 20 years of history and experience



Minimality: L4 Microkernel Source Code Size

Name	Architecture	C/C++	asm	total kSLOC
Original L4	i486	0	6.4	6.4
L4/Alpha	Alpha	0	14.2	14.2
L4/MIPS	MIPS64	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

L4 Microkernel IPC Performance over 20 Years

Name	Year	Processor	MHz	Cycles	µs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	i7 Bloomfield (32-bit)	2,660	288	0.11
seL4	2013	i7 Haswell (32-bit)	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

What Is a Microkernel?

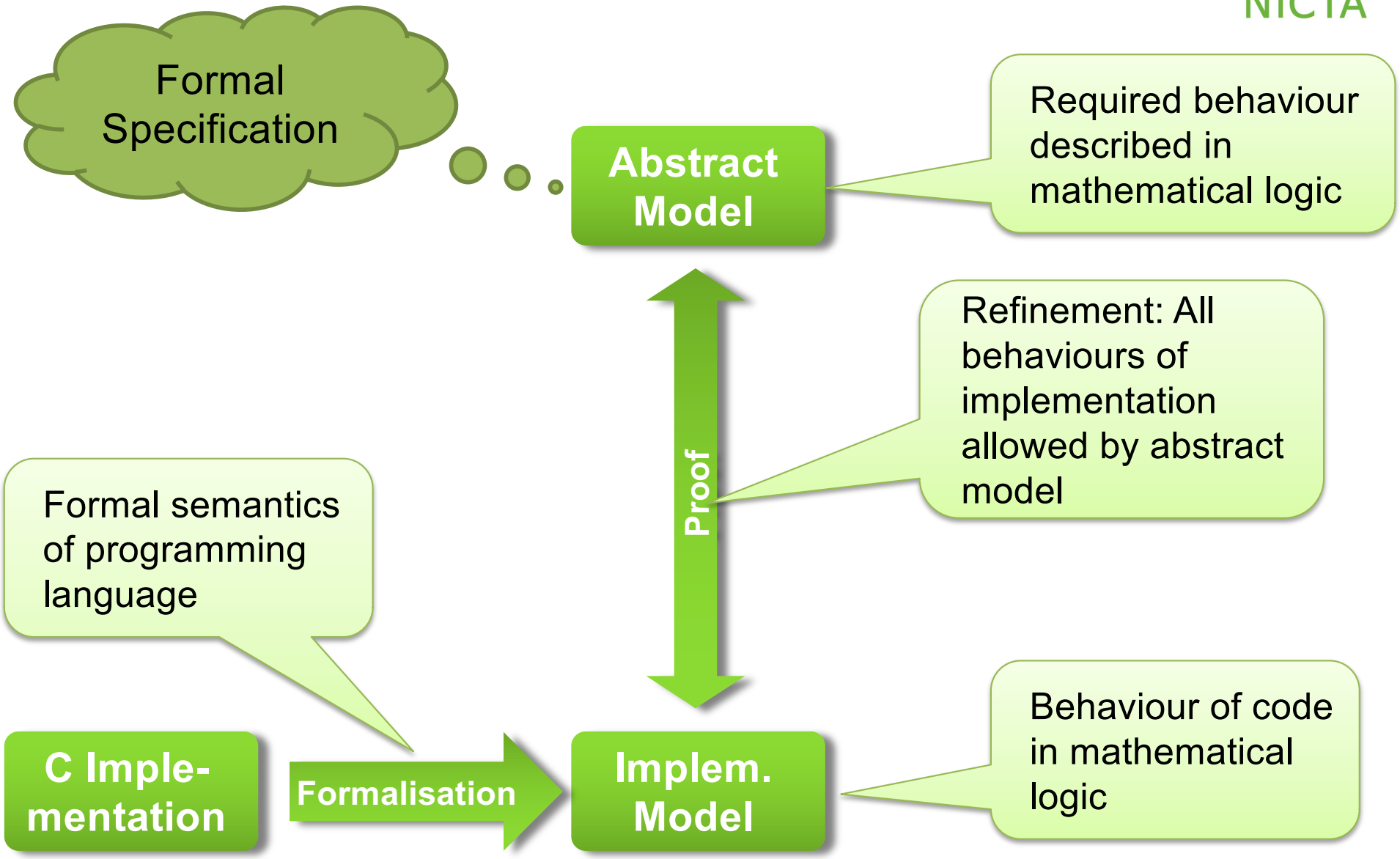
- Basic idea :
  - Flexible, minimal platform
  - Mechanisms, not policies
  - Goes back to Nucleus [Brinch Hansen, CACM'70]

- New idea:
  - If it's so small, maybe we can ***prove*** it correct?

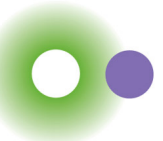
Overview

- What is seL4, what are Microkernels?
- **What is formal verification, what does it achieve?**
- What can you do with seL4?

Proving Software (Functionally) Correct



Proving seL4 (Functionally) Correct




```
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"
```

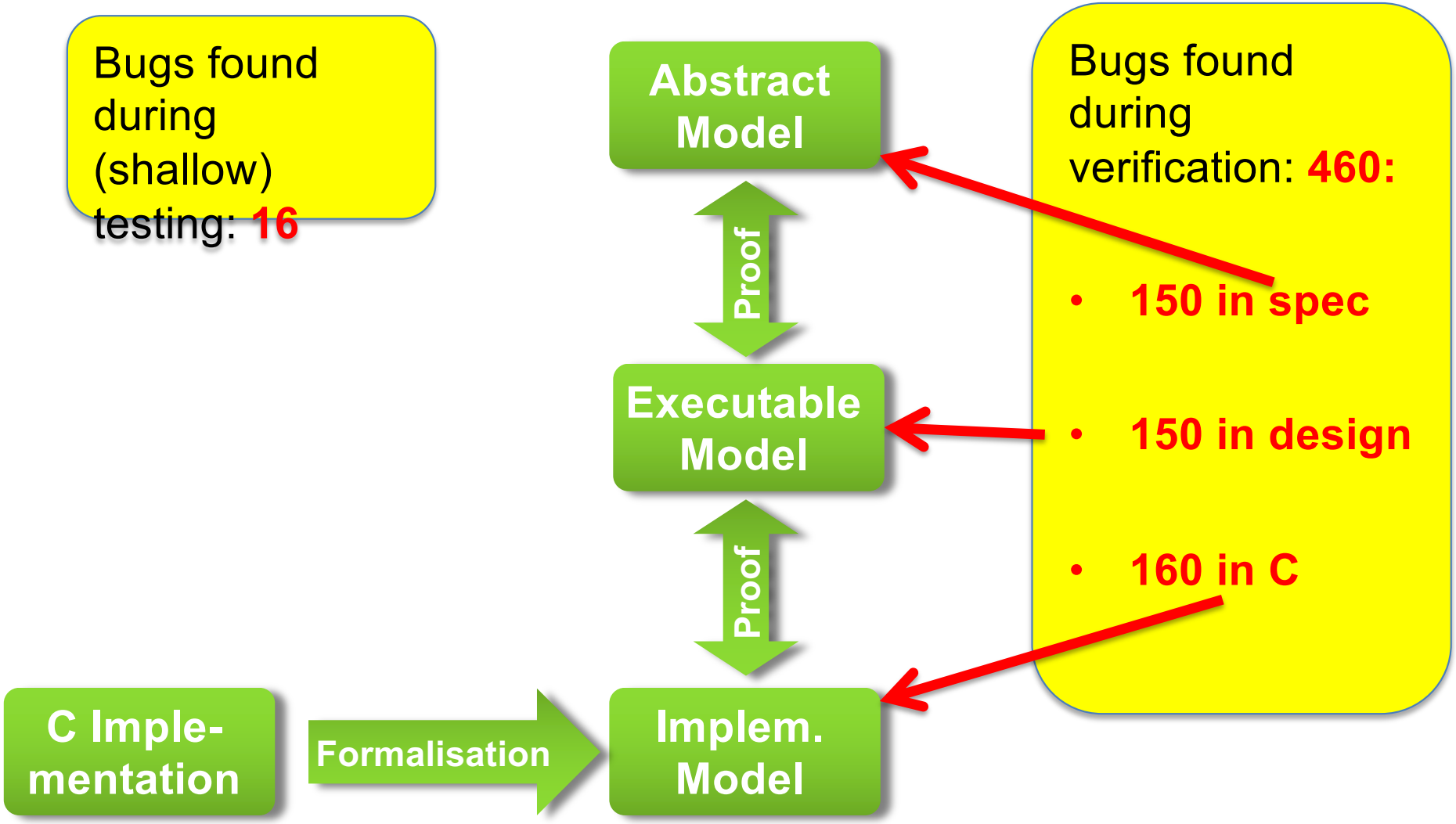
```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
```

```
ad
curThread
meSlice curThread
ime == 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;
```

Proving seL4 (Functionally) Correct

NICTA



From Functional Correctness to Security

- Functional correctness: *Kernel always behaves as specified*
  - No undefined behaviour specified ⇒ no undefined behaviour possible
  - Hence no code injection, control-flow attacks etc, etc

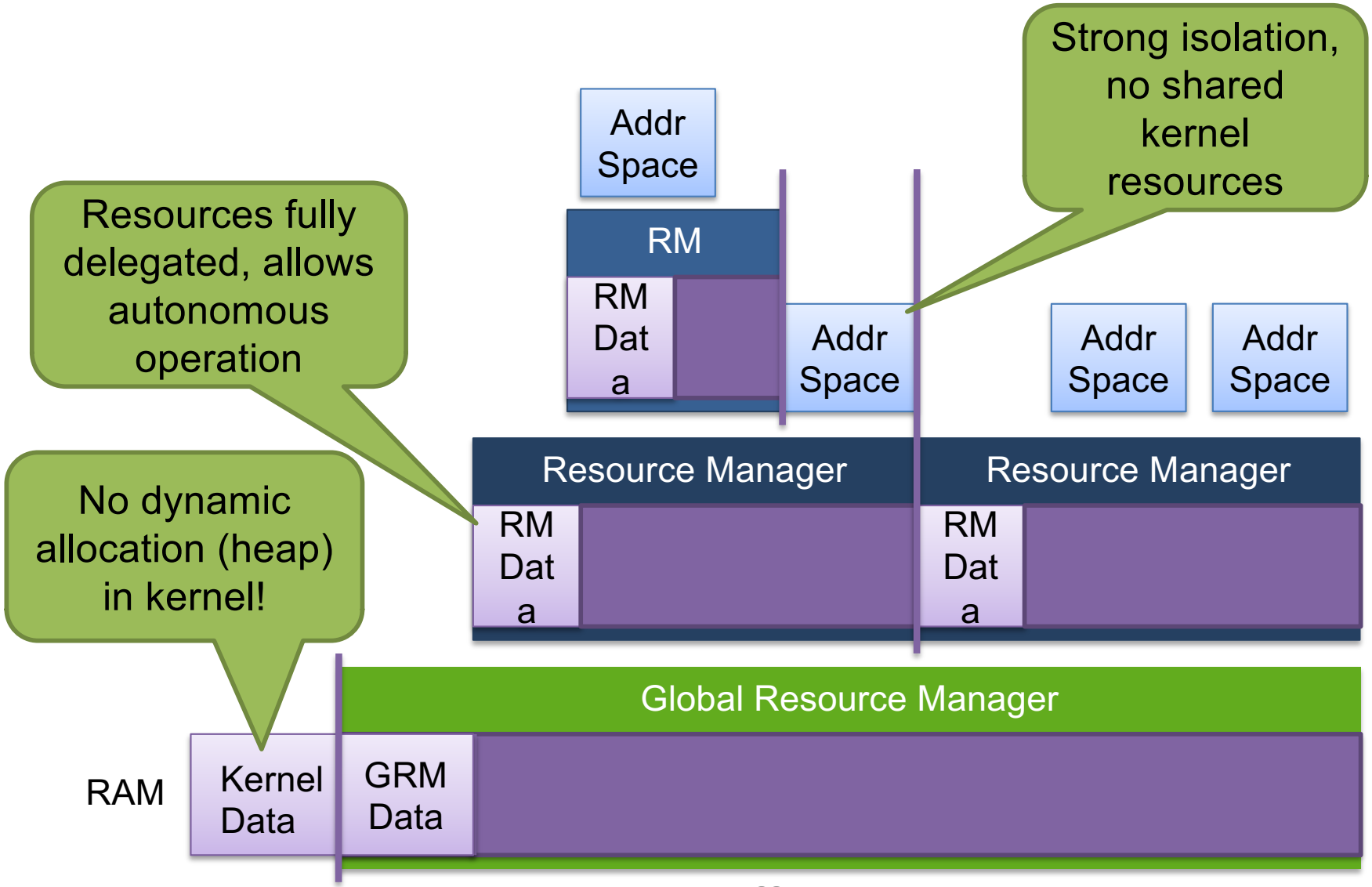
- "Security" requires more: *System must never enter insecure state*
  - … assuming initial state is secure

- Kernel must enforce *isolation* – "CIA properties":
  - **Confidentiality**
  - **Integrity**
  - **Availability**

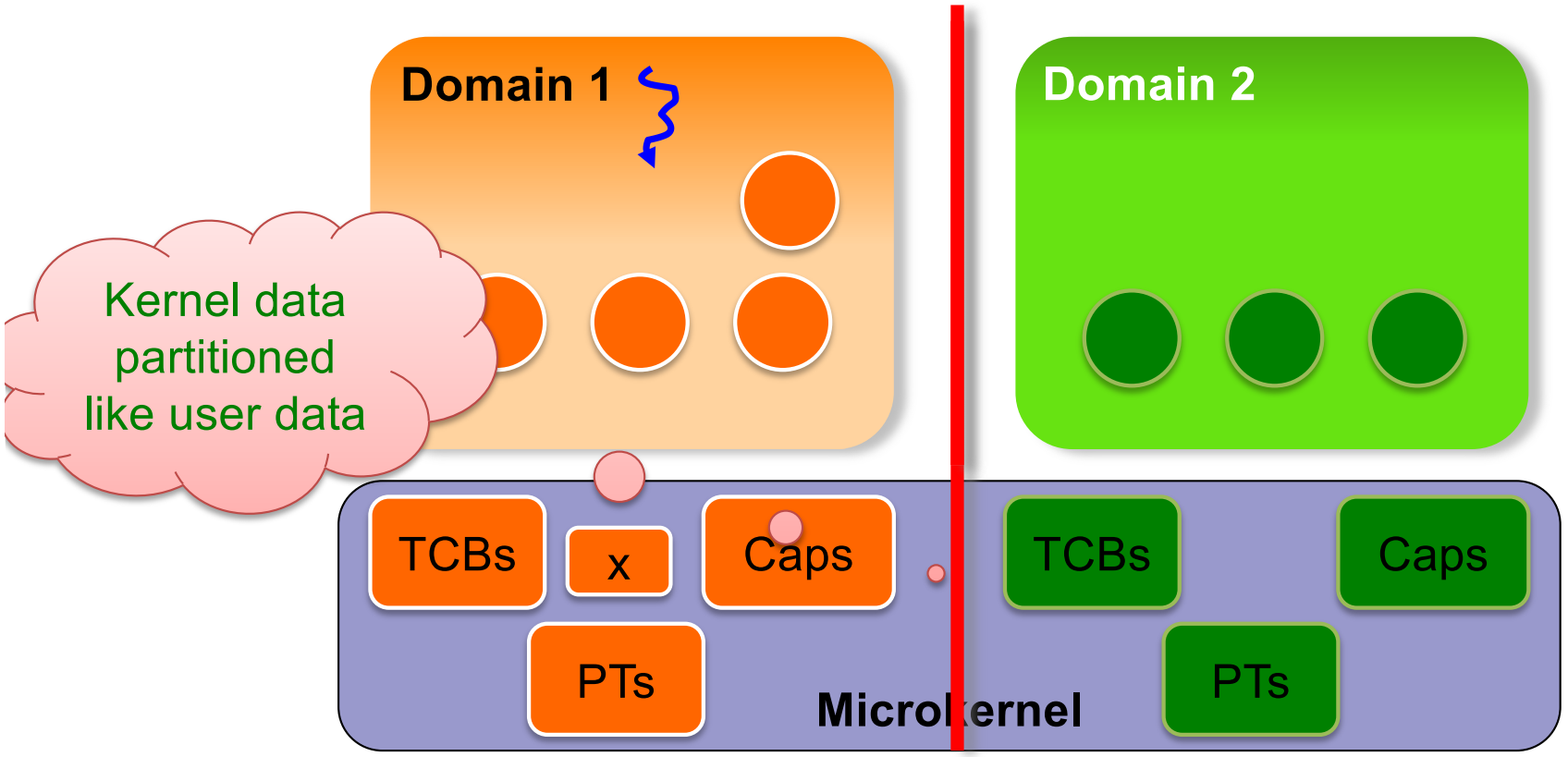
seL4 Design for Isolation

NICTA

User-level control of memory management

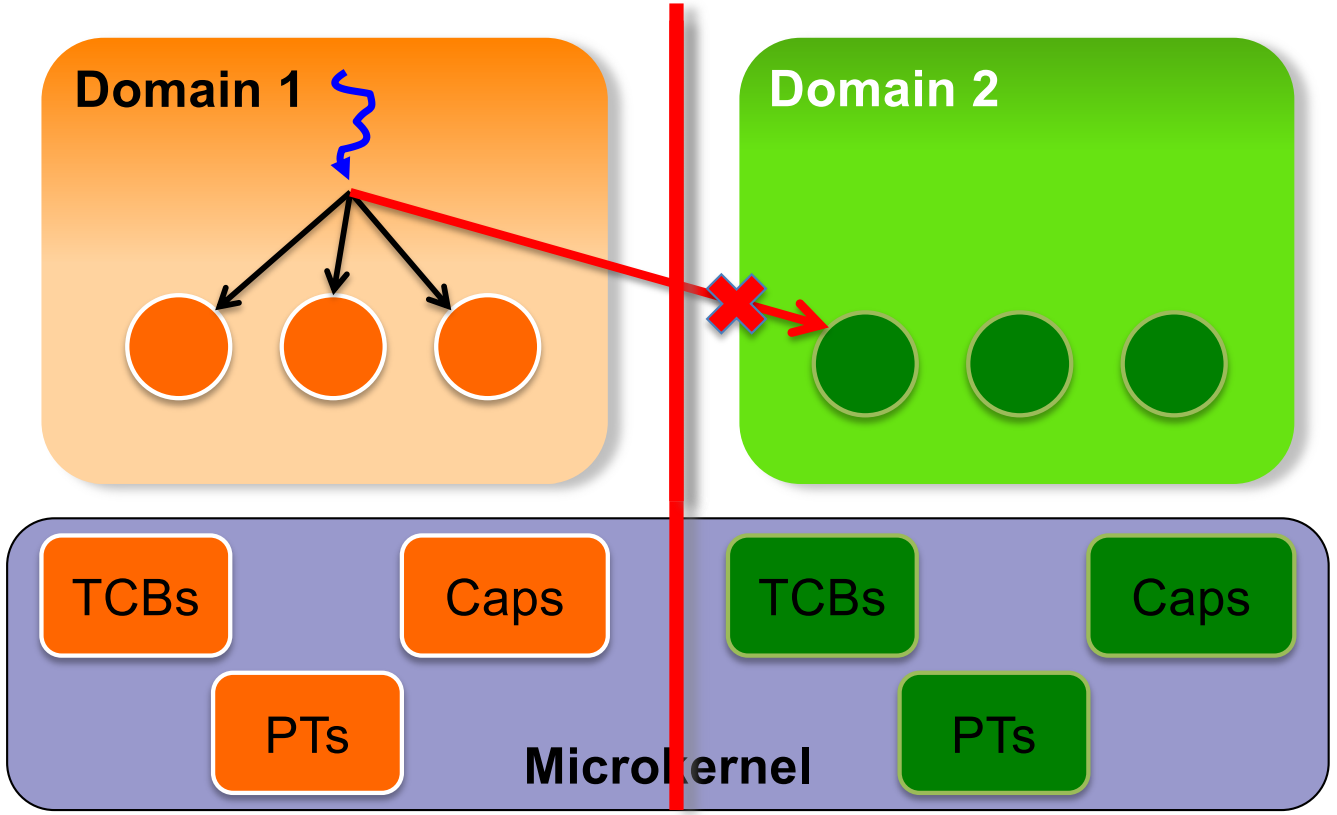


Availability: Ensuring Resource Access



- Strict separation of kernel resources (provided by user)
  ⇒ agent cannot deny access to another domain's resources

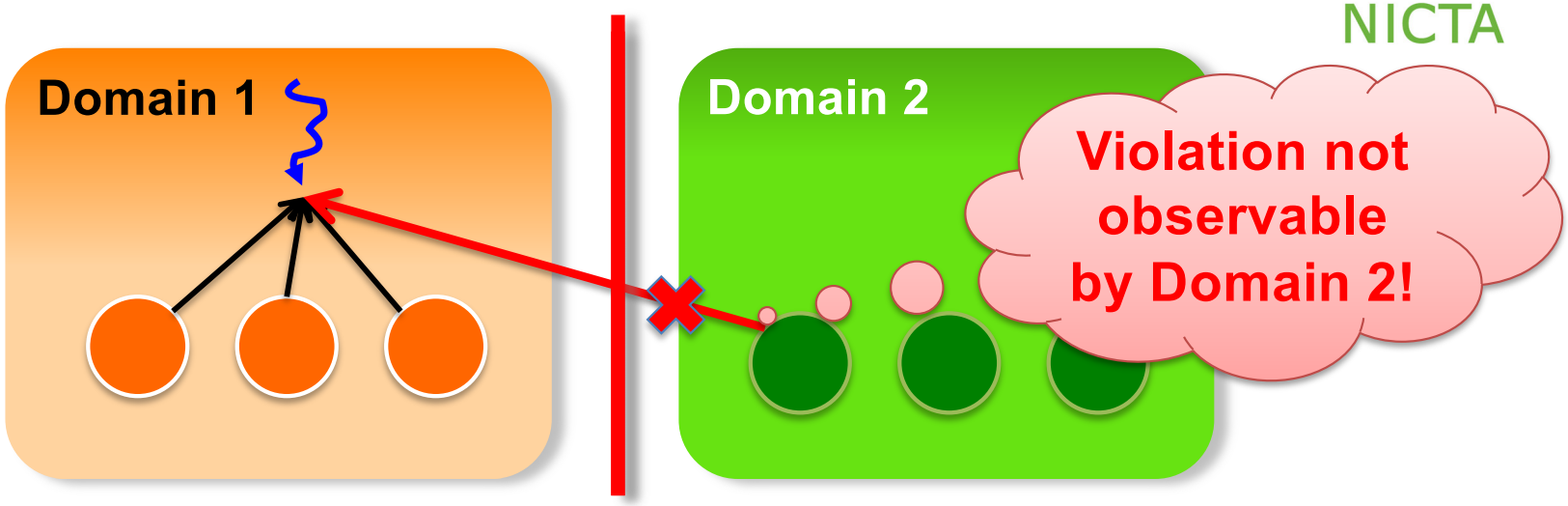
Integrity: Limiting Write Access



To prove:

- Domain-1 doesn't have write *privileges* 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!*
  - Prove kernel only allows write if properly authorised

Confidentiality: Limiting Read Accesses



To prove:

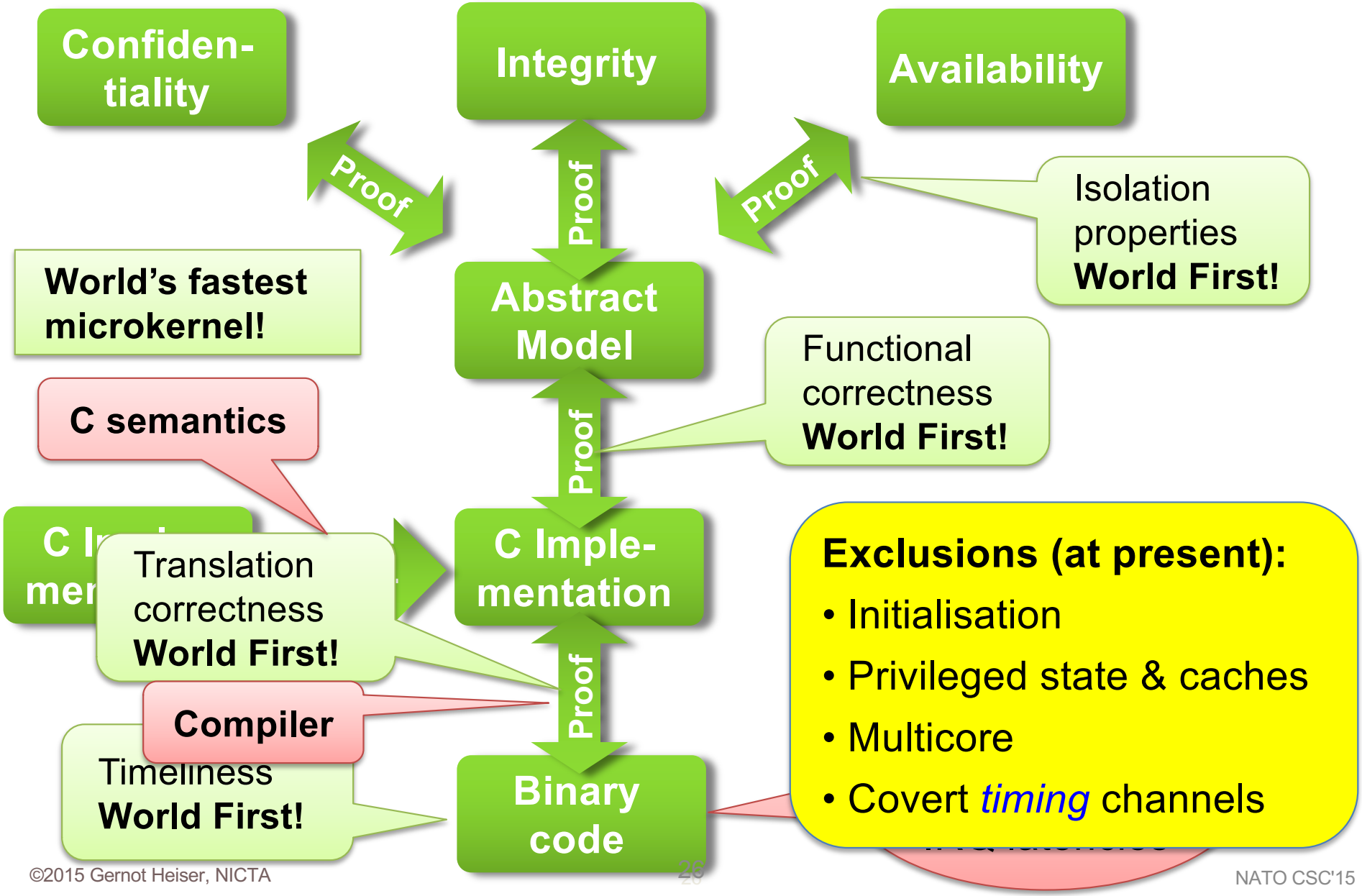
- Domain-1 doesn't have read capabilities to Domain-2 objects
  ⇒ no action of any agents will reveal Domain-2 state to Domain-1

Non-interference proof :

- Evolution of Domain 1 does not depend on Domain-2 state
- Includes absence of covert *storage* channels

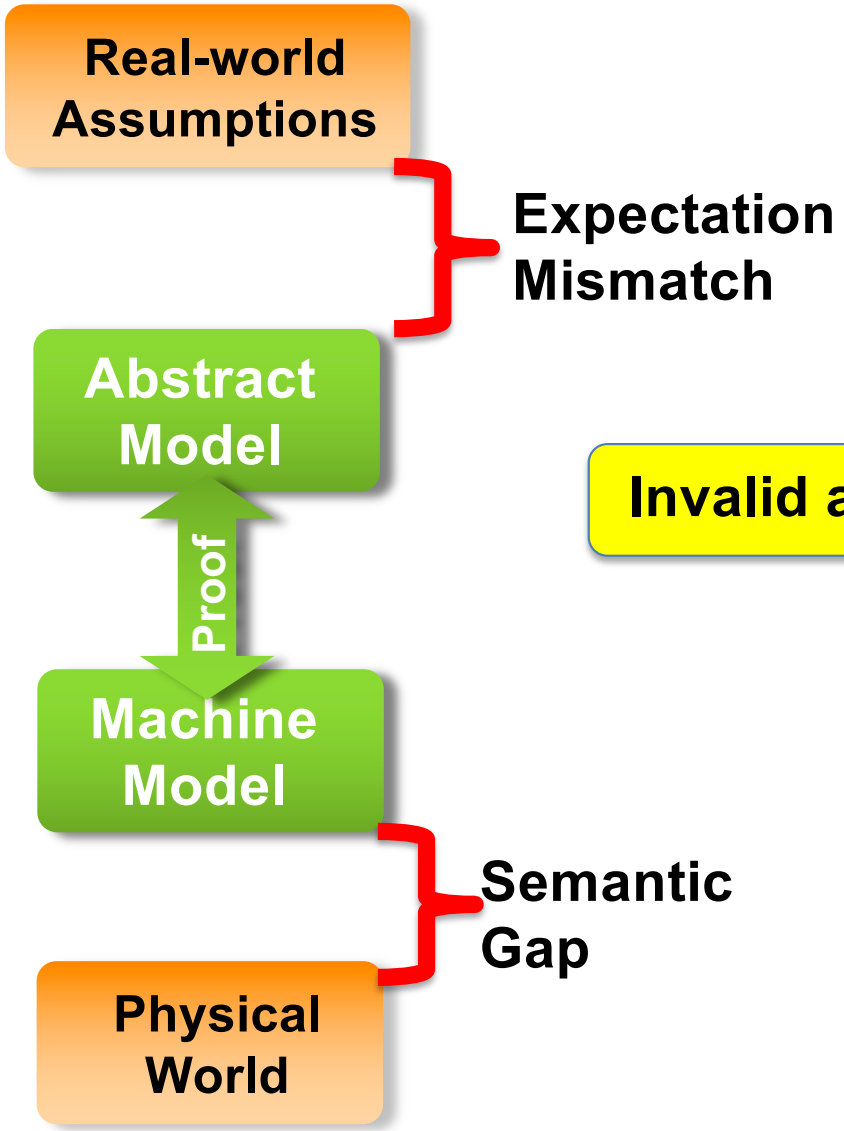
seL4: Proof of Correctness and Security

NICTA



What Are The Limitations?

NICTA



System Architecture:

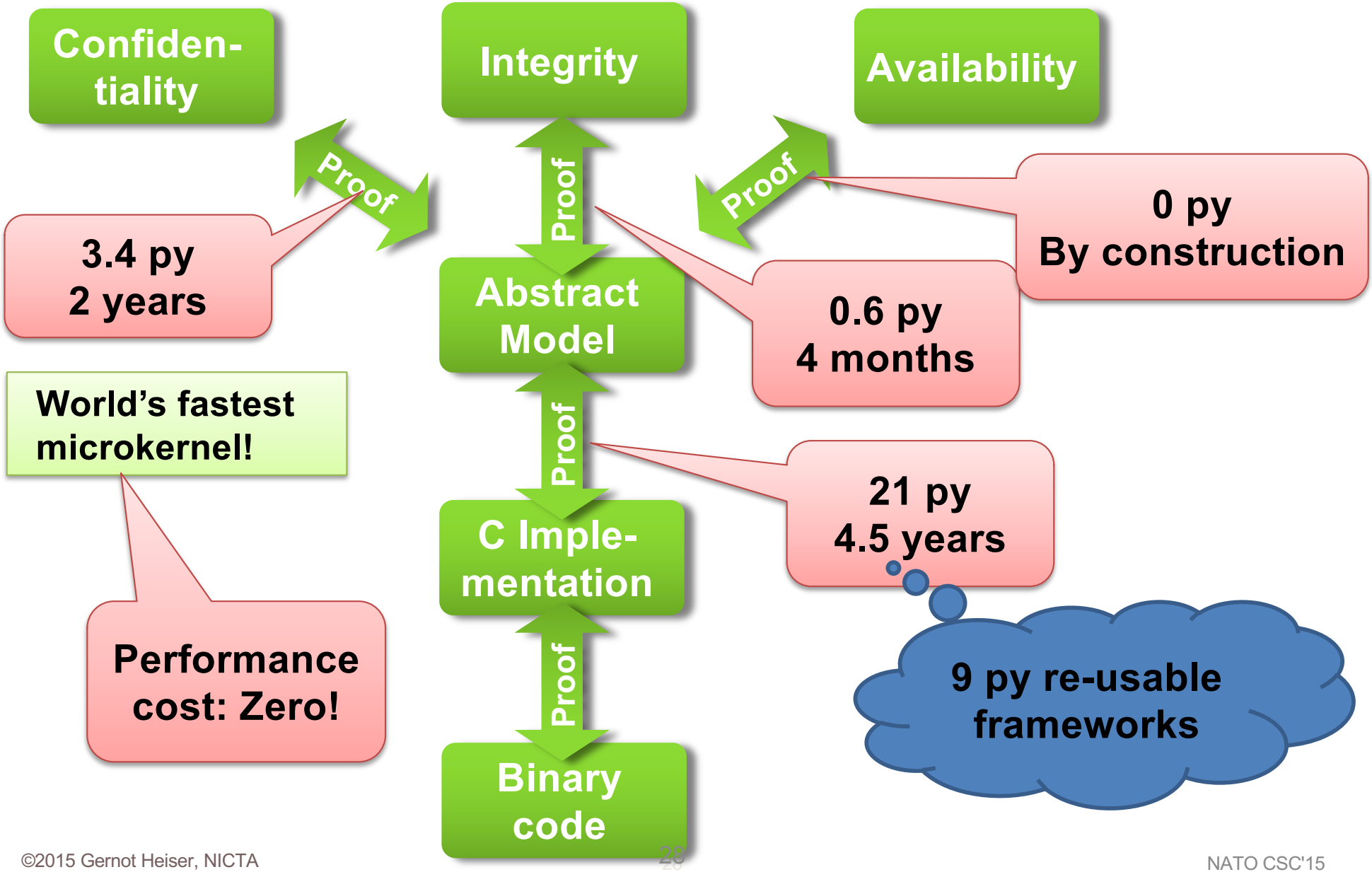
- Secure kernel does not guarantee secure system!

Invalid assumptions!

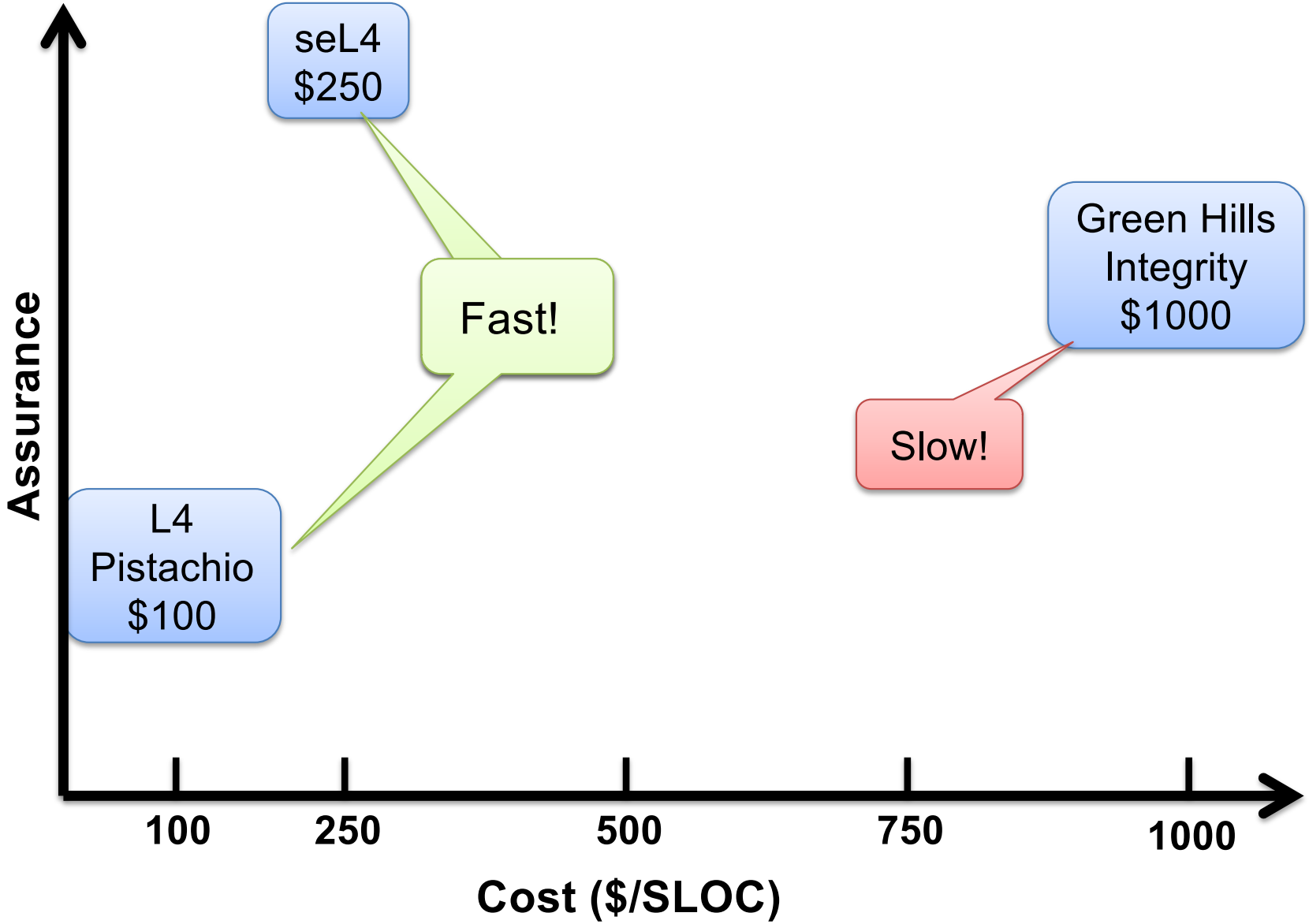
Verification limitations:

- Initialisation
- Privileged state & caches
- Multicore
- Covert *timing* channels

The Cost of Strong Security



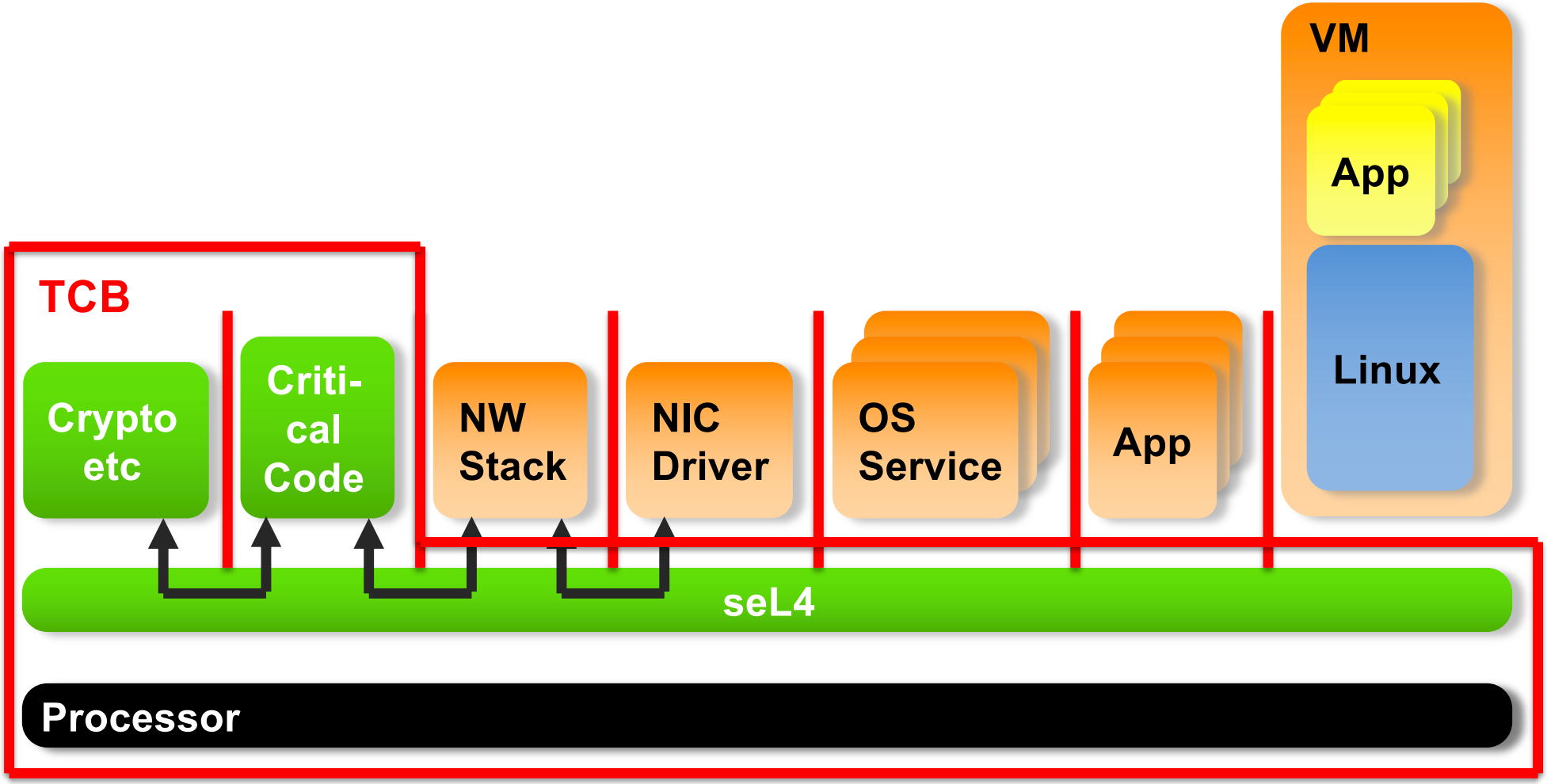
Microkernel Life-Cycle Cost in Context



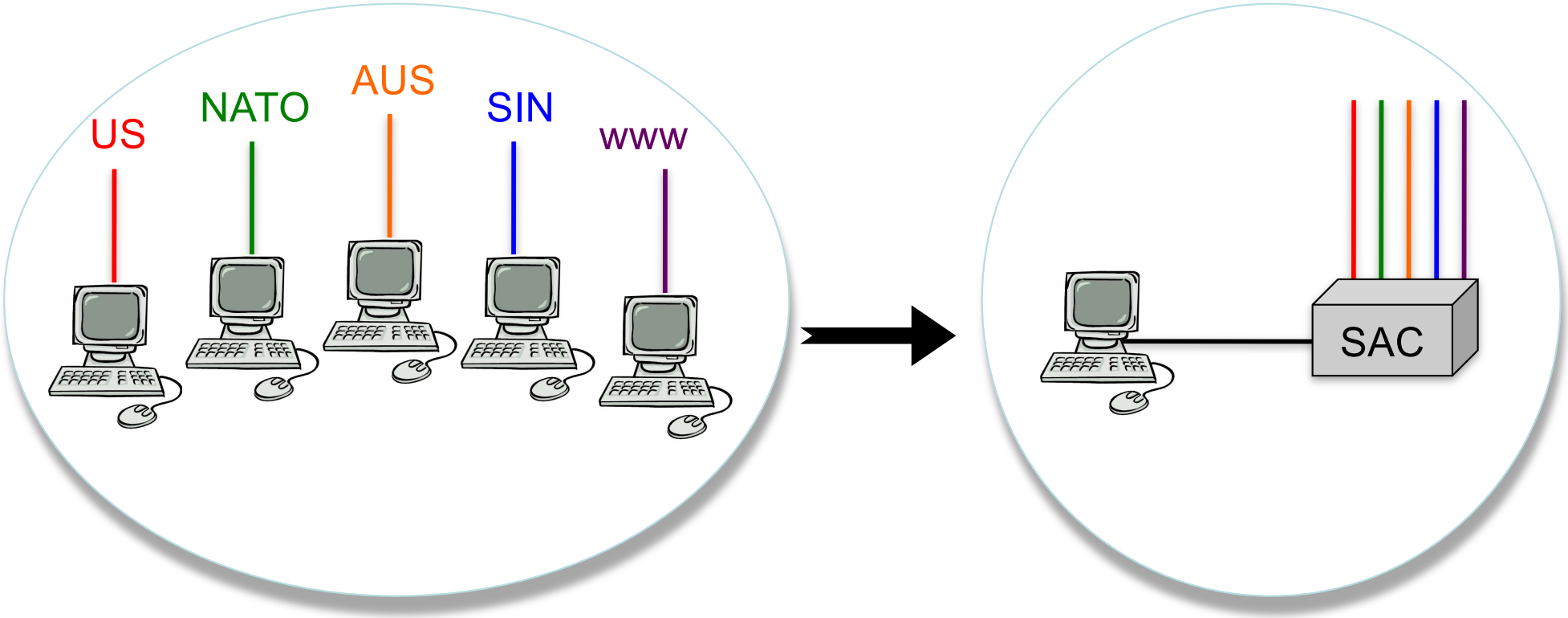
Overview

- What is seL4, what are Microkernels?
- What is formal verification, what does it achieve?
- **What can you do with seL4?**

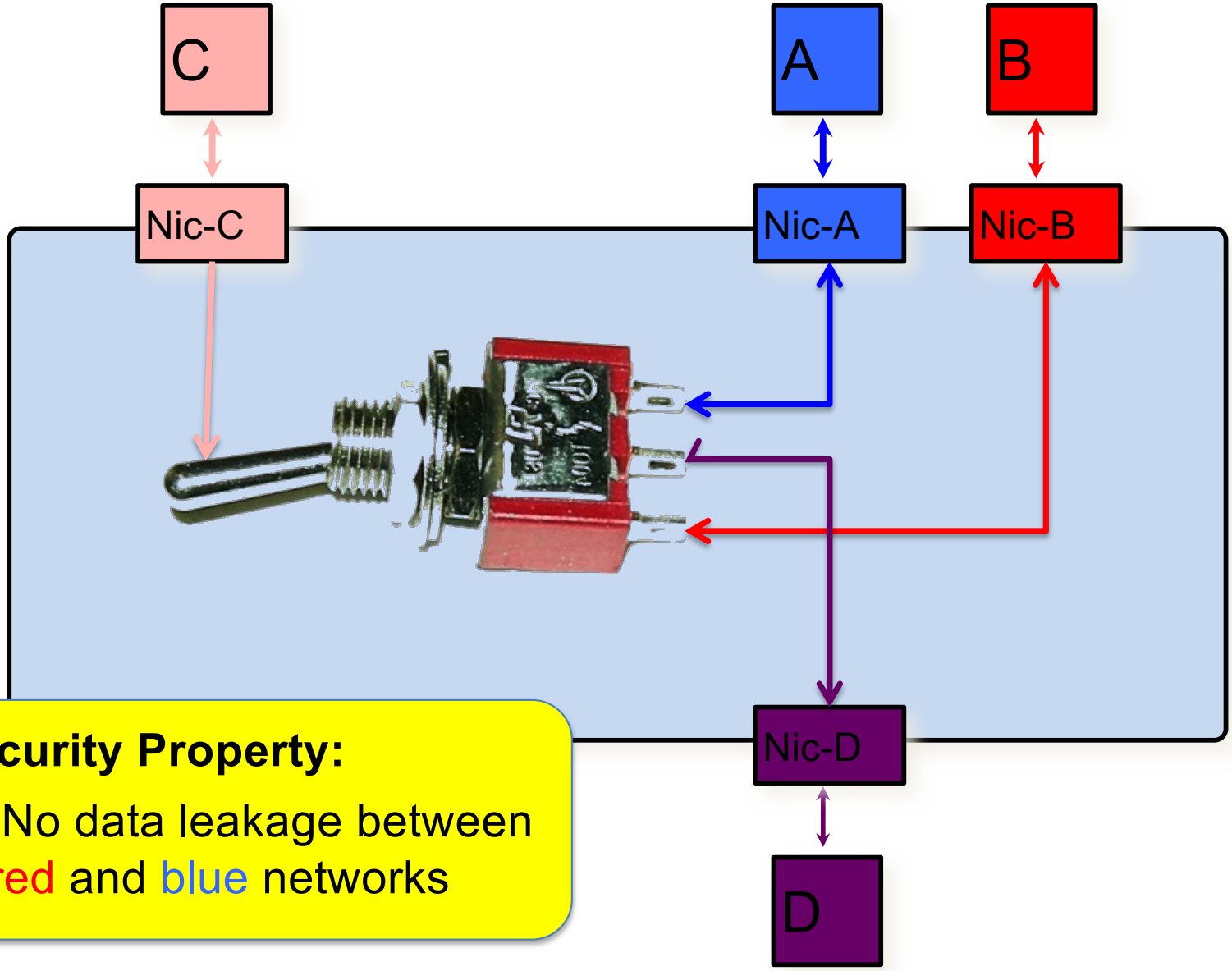
Key: Architecture for Minimal TCB



Proof of Concept: Secure Access Controller



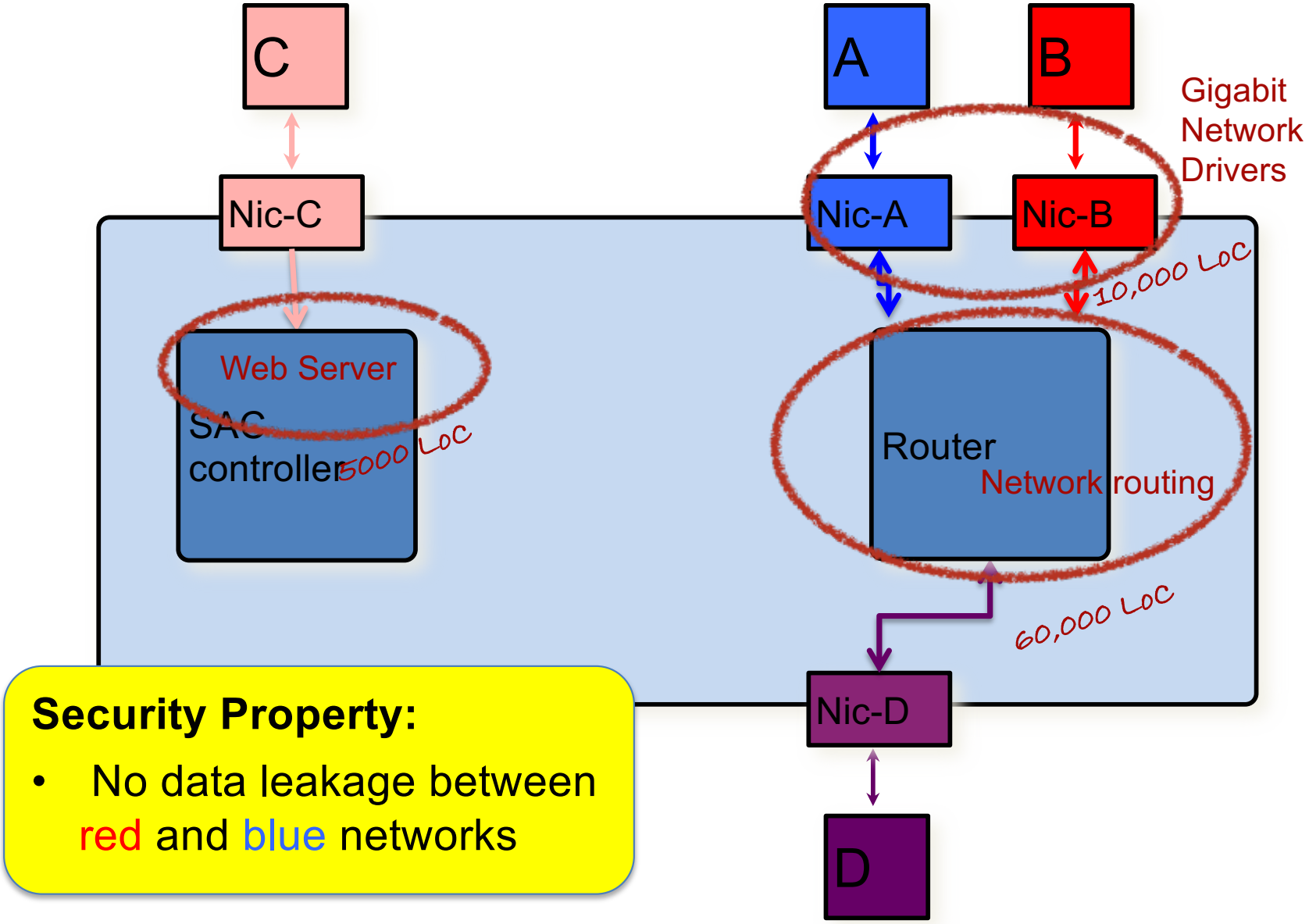
Logical Function



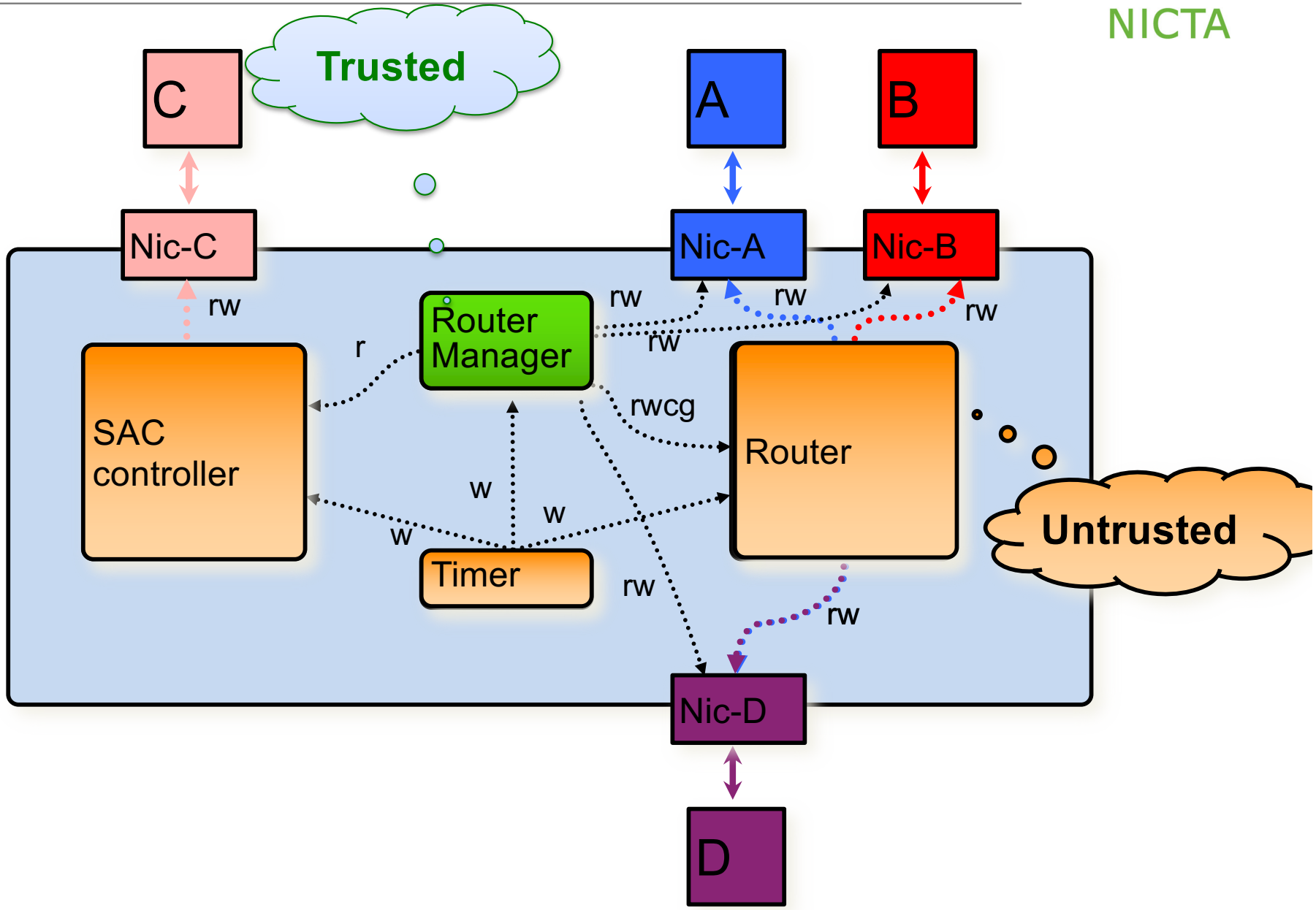
Security Property:

- No data leakage between red and blue networks

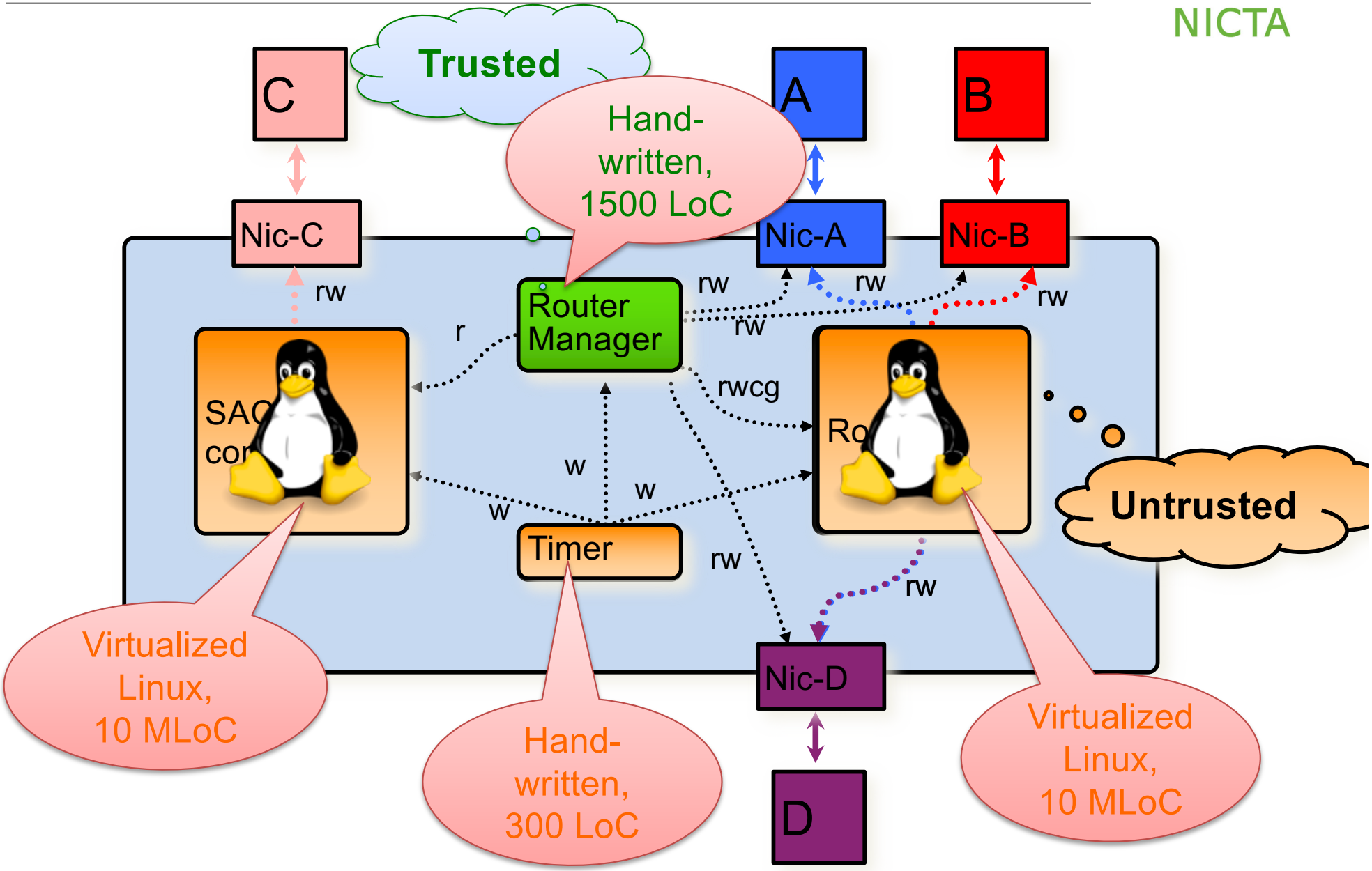
Logical Function



Minimal Trusted Computing Base



Minimal Trusted Computing Base



Complete High-Assurance System

NICTA

DARPA HACMS Program:

- Provable vehicle safety
- "Red Team" must not be able to divert vehicle

SMACCMcopter
Research Vehicle

Boeing Unmanned
Little Bird (AH-6)
Deployment Vehicle

NICTA

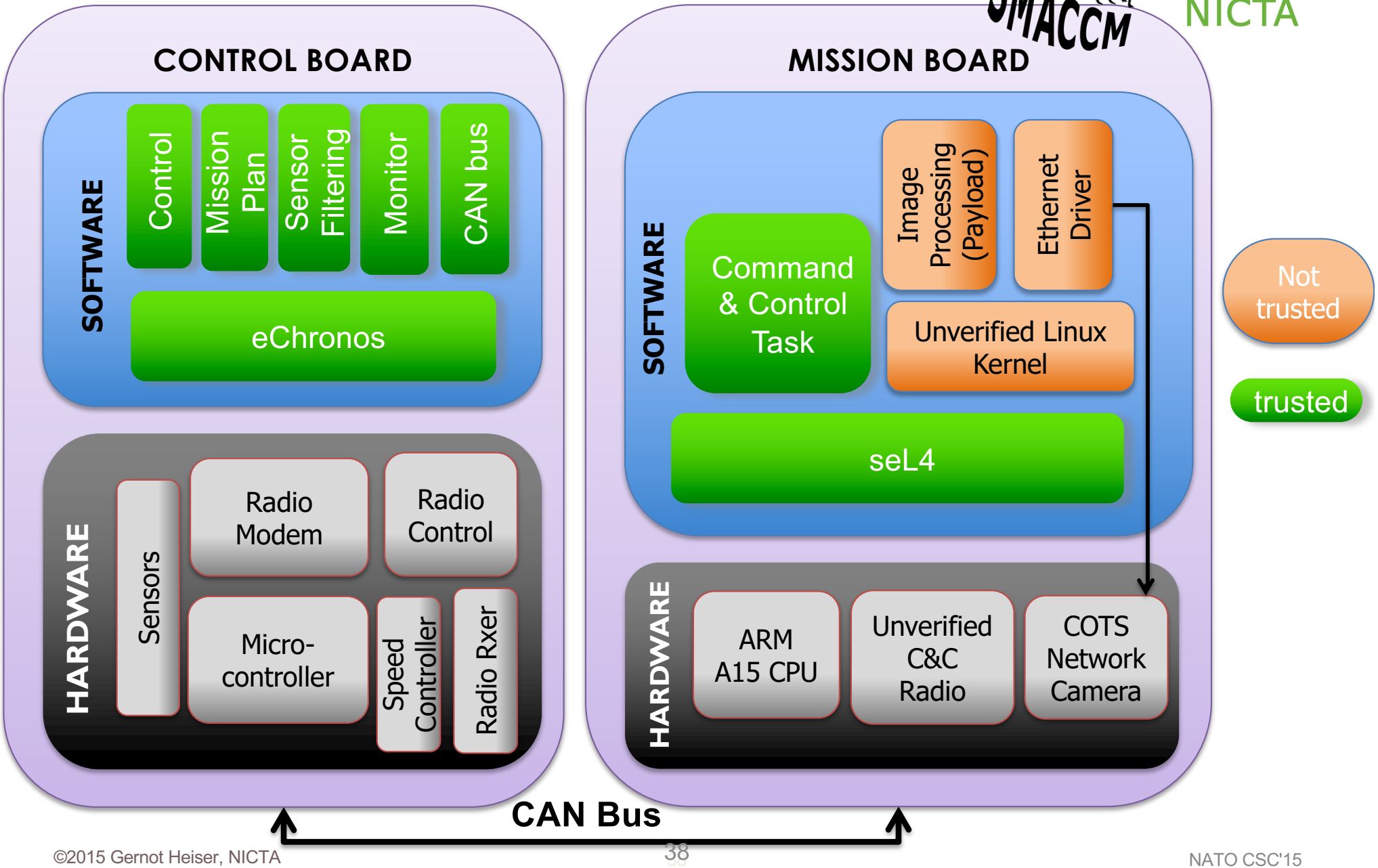
Rockwell Collins

BOEING

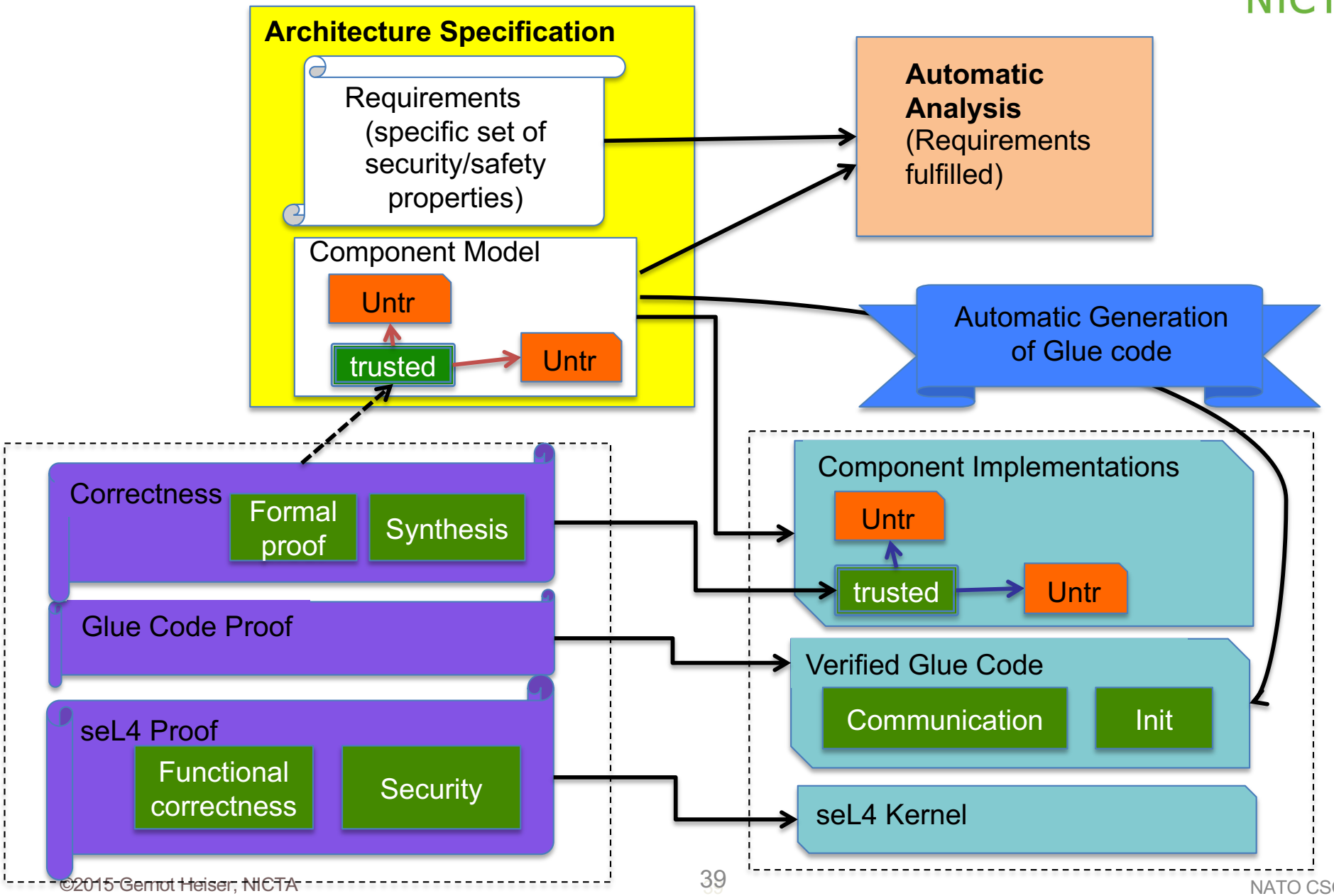
galois

University of Minnesota

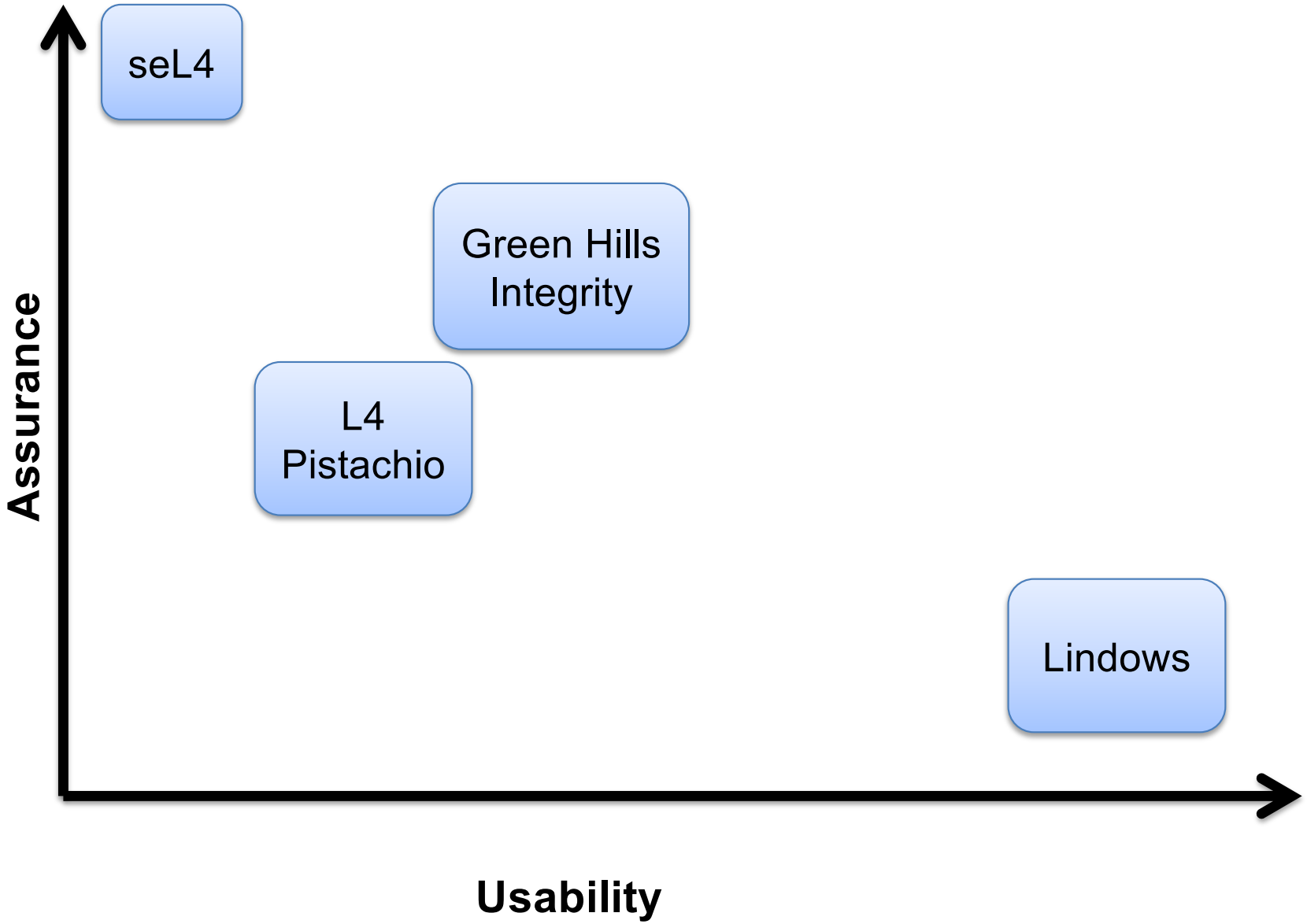
SMACCMcopter Architecture



Architecting System-Level Security/Safety



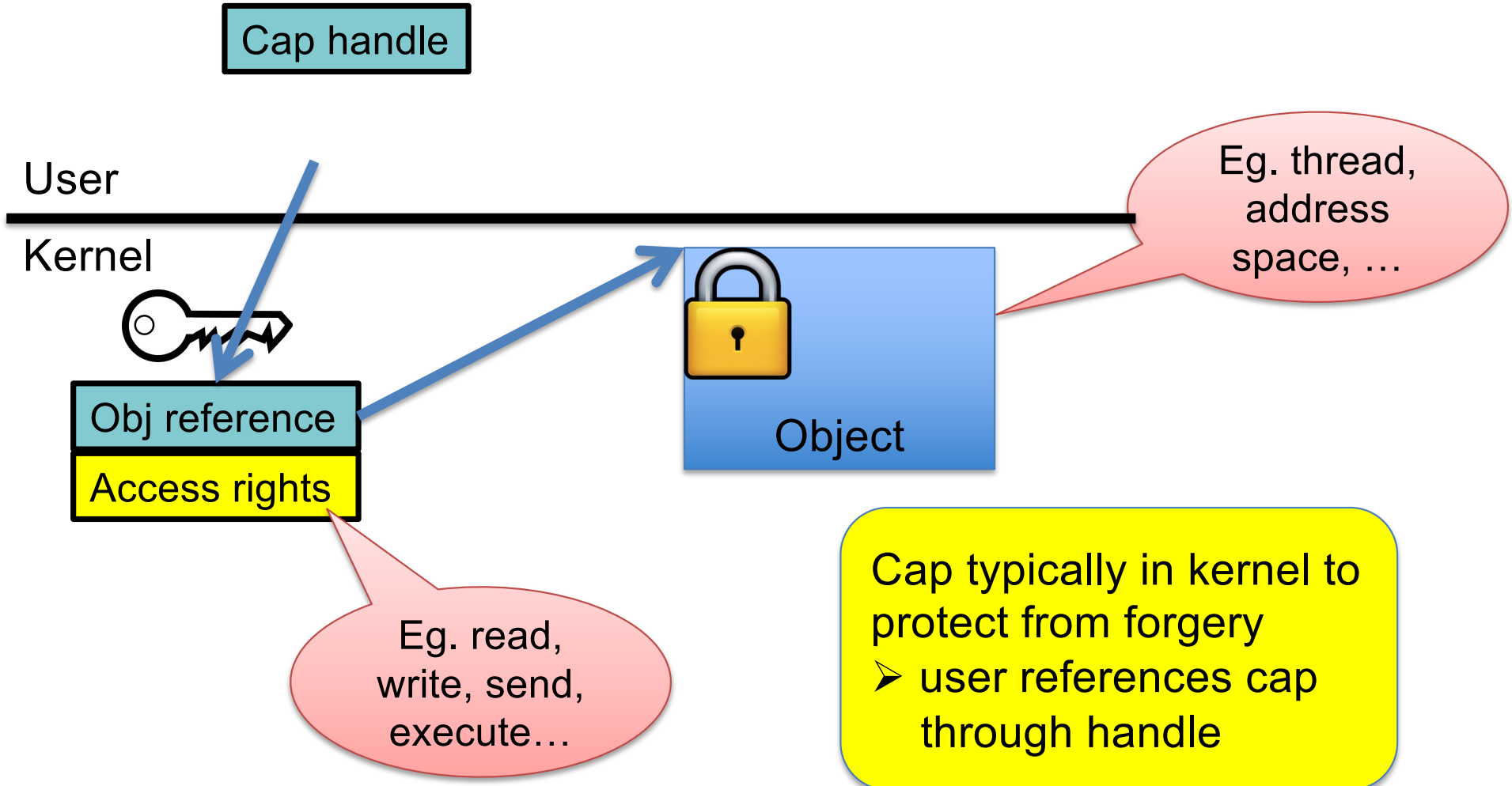
Security Comes at a Cost



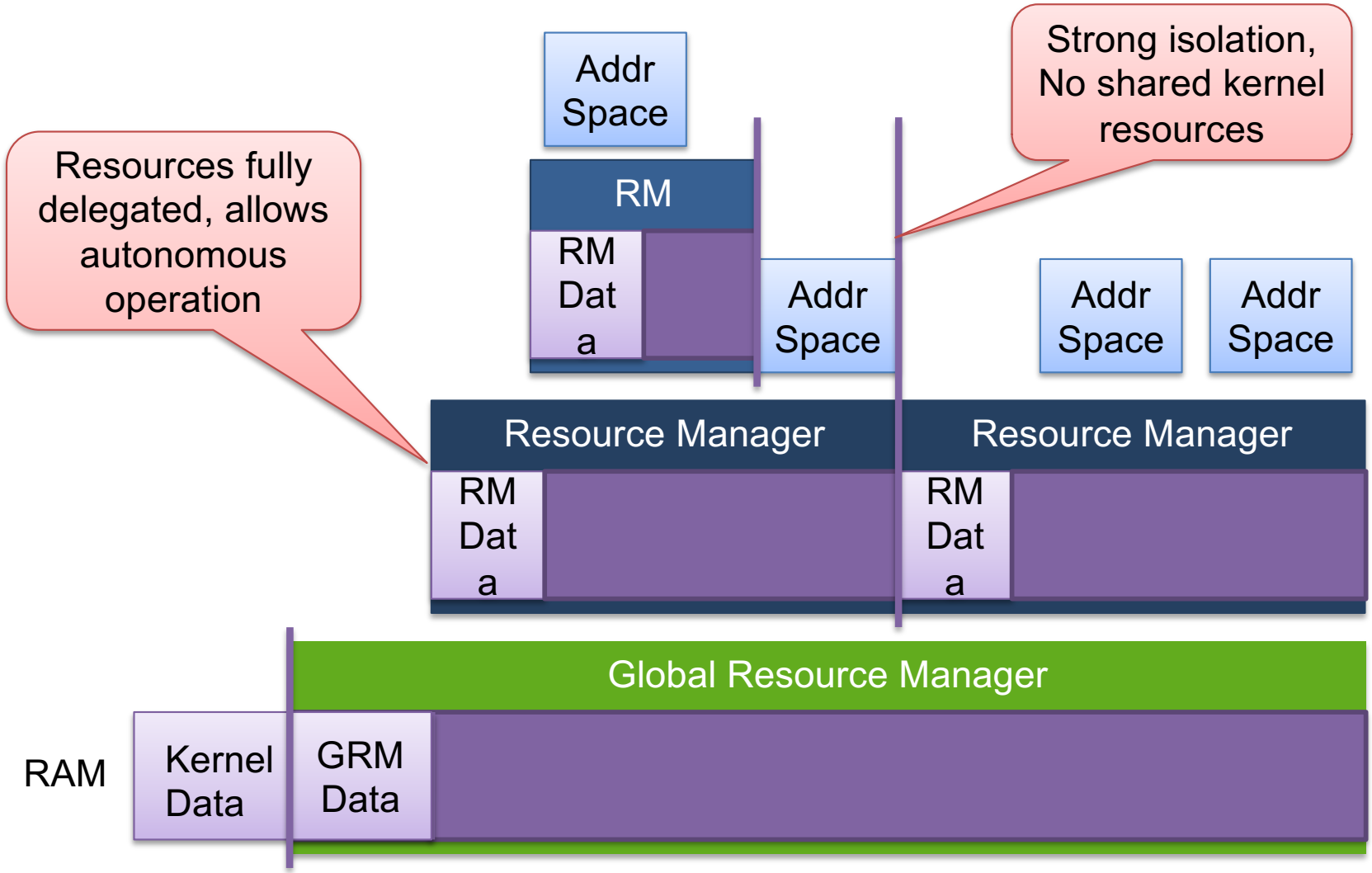
What are Capabilities?

NICTA

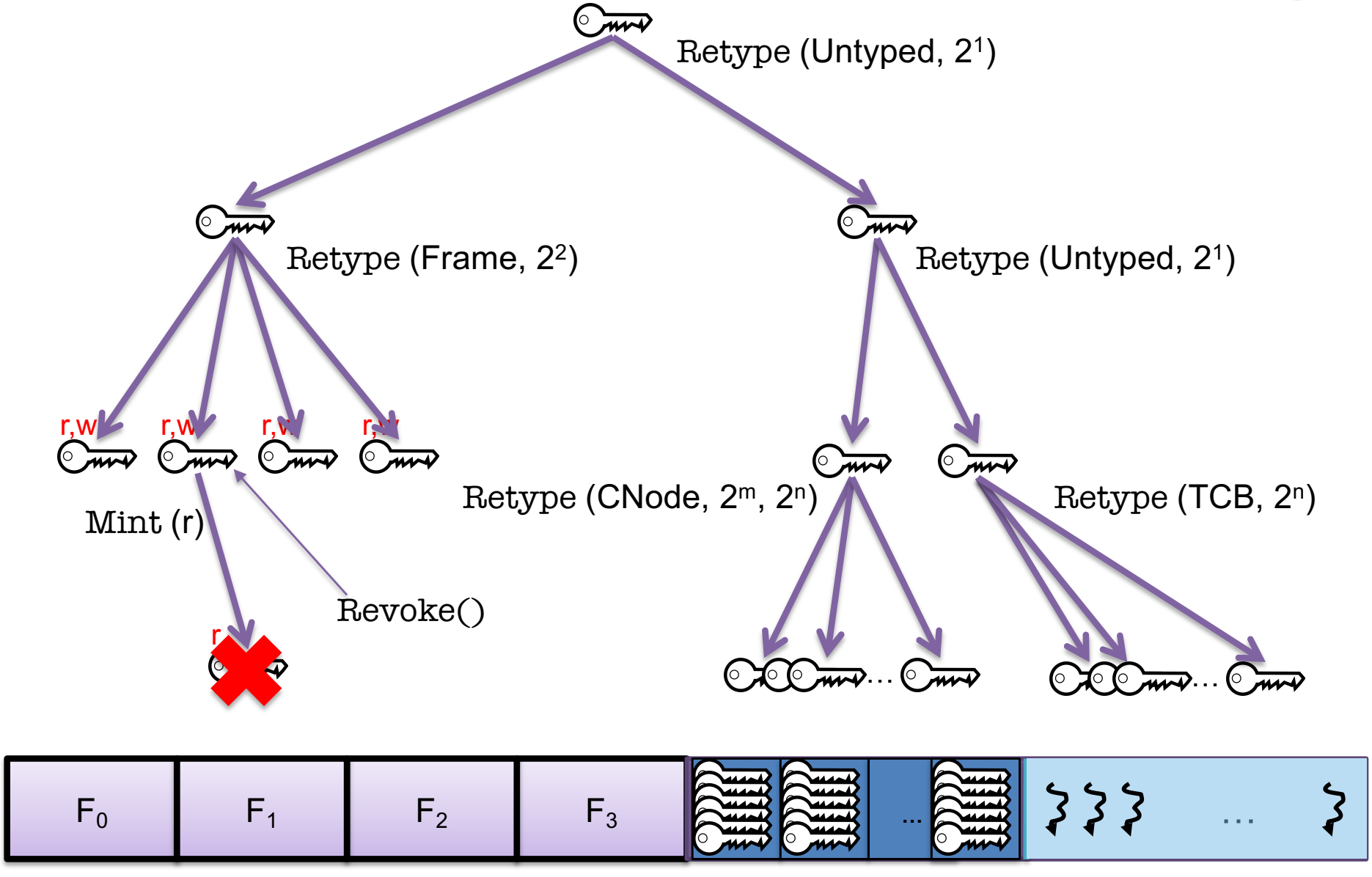
Capability = Access Token



seL4 Memory Management Approach



Memory Management Mechanics: Retype



Typical System Setup

1. Partition Untyped into pools

2. In each pool:
    1. Create Address-Space, Cnode, TCB, EP, AEP objects
    2. Maybe create IRQ objects (if partition is allowed to handle IRQs)
    3. Convert most (or all) of remaining Untyped into Frames
    4. Map some Frames into Address space(s)
    5. Load code into frames
    6. Associate Thread(s) with code
    7. Start initial thread

- Partitions are completely isolated
- Partition without free Untyped cannot do resource allocation
- Partition without Untyped caps cannot do resource revocation

Mechanism vs Policy: Process

- No concept of a “process” in seL4 – user-level abstraction

- Process concept implies policy:
  - Single- or multi-threaded?
  - How much stack space, fixed or growable?
  - How much heap space, fixed or growable?
  - Limit on virtual or physical memory use?
  - Shared libraries, shared buffers?

- Actual system will define policies (and embed in libraries etc)

Why NOT Use seL4?

- Very rudimentary programming environment!
  - Fair enough, but it's improving!
  - DARPA SBIR call for building seL4 ecosystem
- I like unsafe/insecure systems!
  - Ok, go shoot yourself
- I like the thrill of danger!
  - Like getting sued for building a critical system on outdated technology
- Actually, I want to use seL4!
  - Right answer ;-)

http://seL4.systems

gernot@nicta.com.au

http://microkerneldude.wordpress.com

@GernotHeiser