Operating Systems

For Secure and Safe Embedded Systems

Part 1: Fundamentals

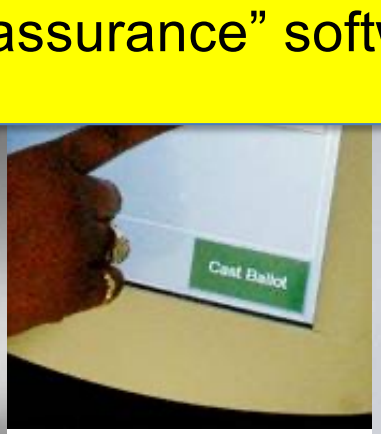
@GernotHeiser

Copyright Notice

Present Systems are *NOT* Trustworthy!

Yet they are expensive:

- $1,000 per line of code for "high-assurance" software!

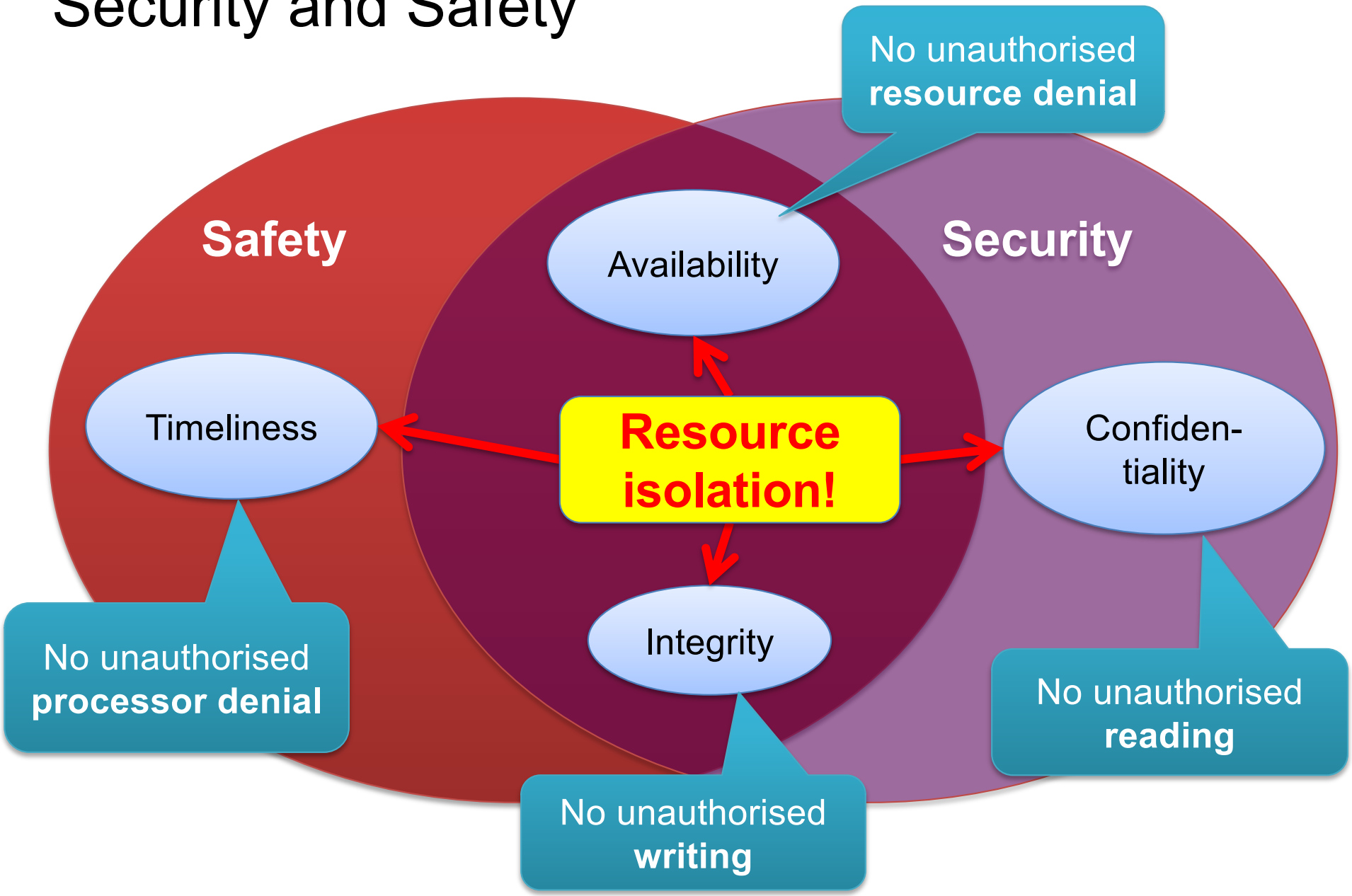


OS Fundamentals

Purpose of the OS / OS Functions

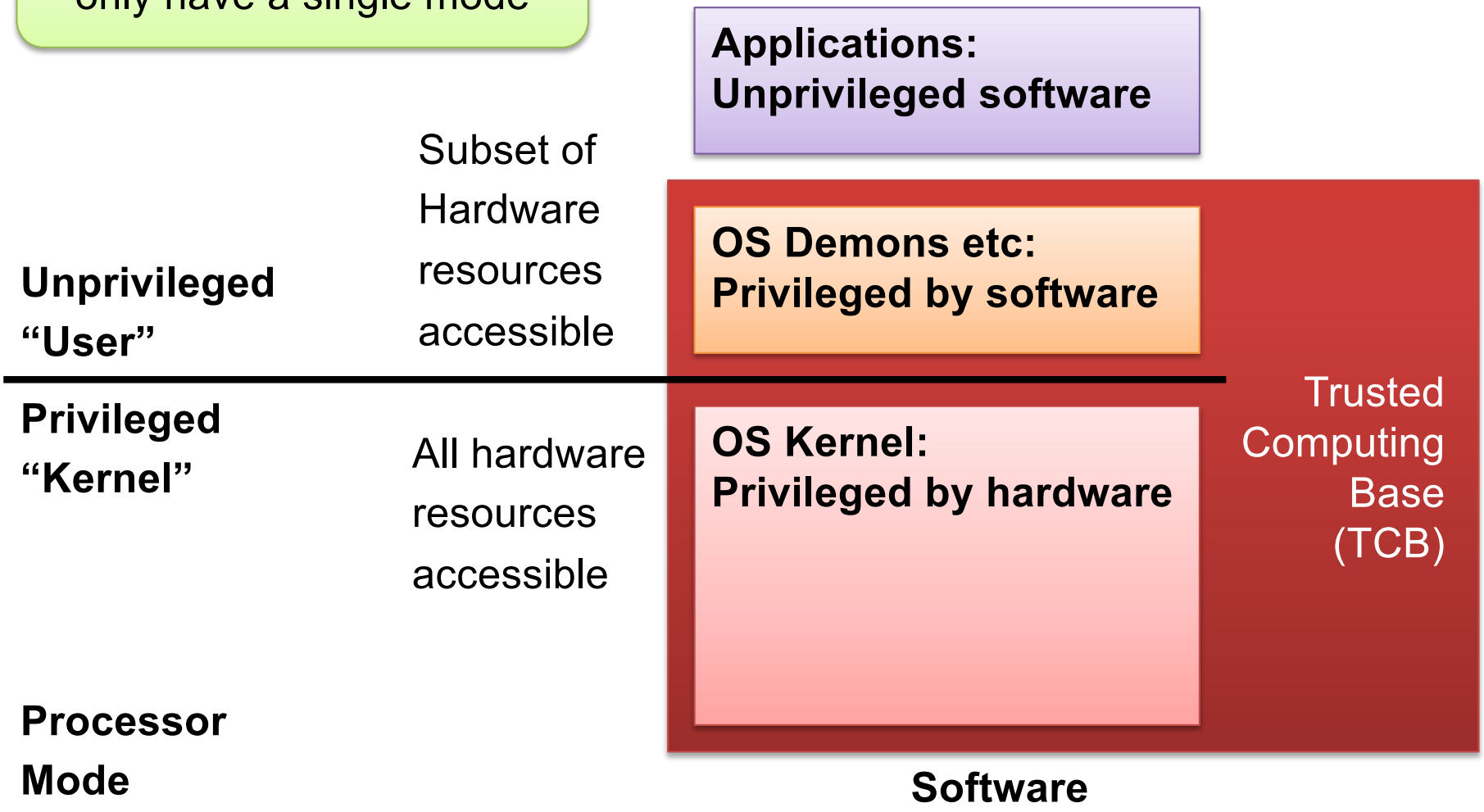
1. OS is an abstract machine
   - Extends basic hardware with added functionality
   - Provides high-level abstractions
     - More programmer-friendly
     - Common core of functionality for applications (eg file systems)
   - Abstracts hardware details irrelevant to programs
     - Portability
2. OS is a resource manager
   - Partition/multiplex limited resources
   - Ensure efficient resource usage
   - Ensure fairness/progress
   - Ensure security & safety

Security and Safety



Hardware Execution Modes and Privilege

Low-end microcontrollers only have a single mode



Memory Protection 1: None

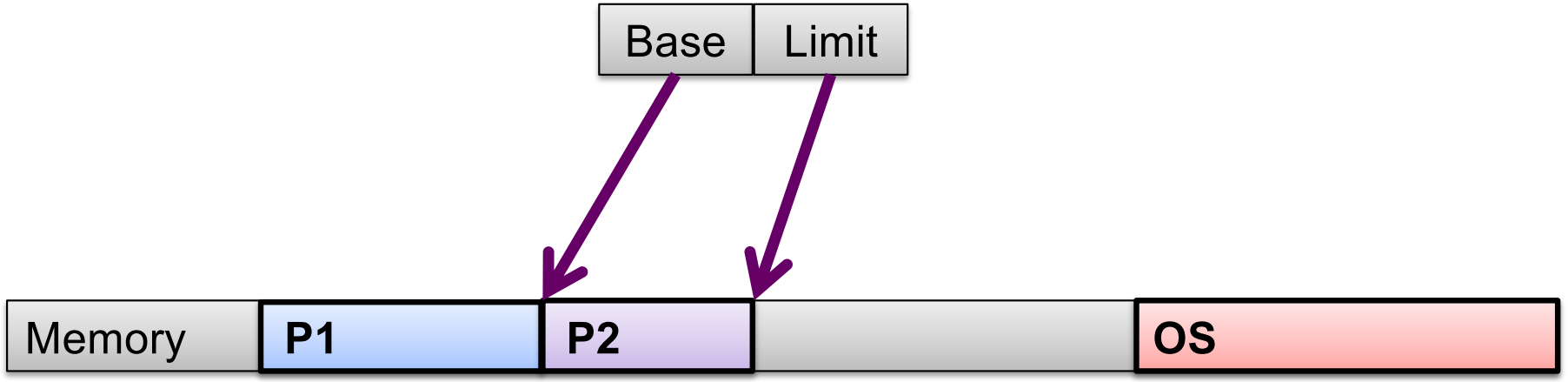
Low-end microcontrollers
- **Eg AVR**
- **ARM Cortex-M0**

- Software issues memory addresses
- No way to limit access
- Processes can overwrite each other and the OS
- OS has no special privilege – ”real-time executive”

Memory Protection 2: Bounds Registers

- Software issues memory addresses
- Processes know their memory location
- Bounds registers limit access
- Privileged OS controls and switches bounds registers

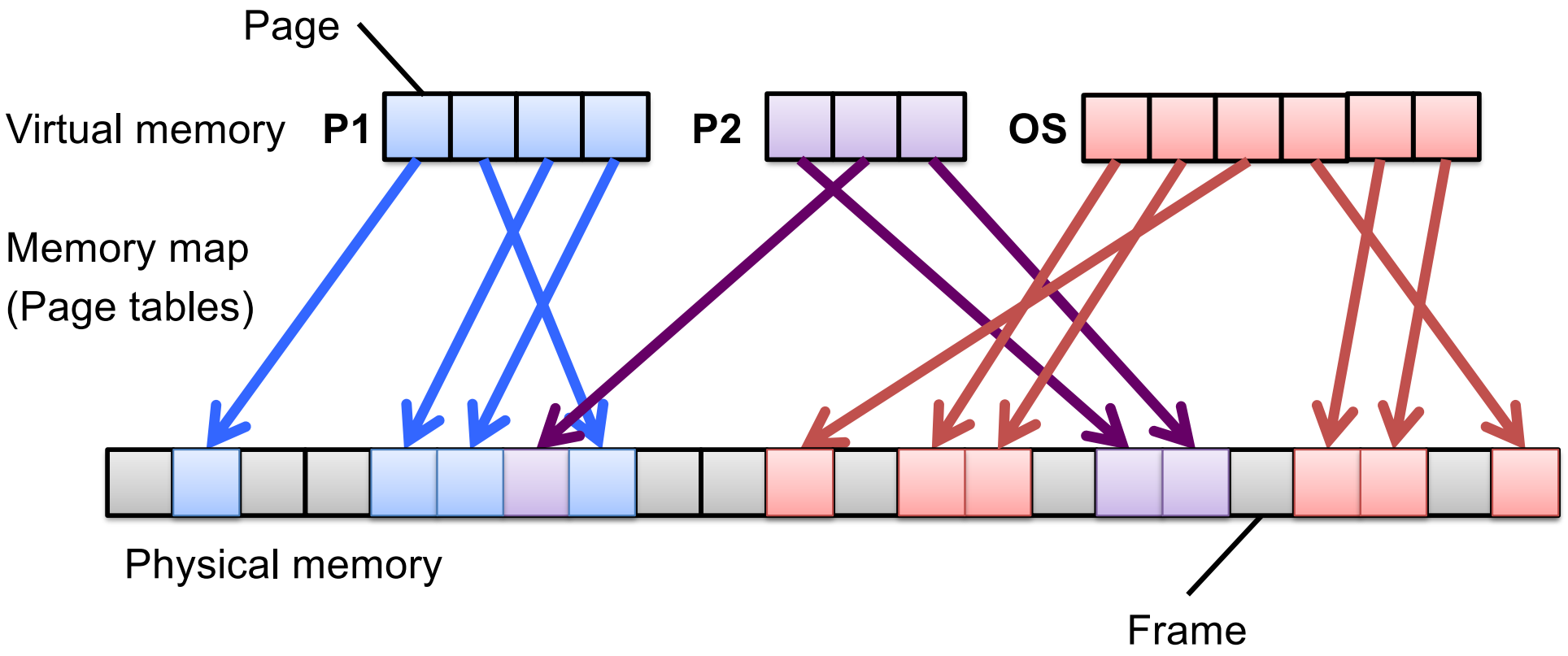
High-end microcontroller
- **Eg ARM Cortex-M4**



Memory Protection 3: Virtual Memory (Paging)

- Software issues virtual addresses
- Unmapped memory not addressable
- Physical memory completely hidden
- Privileged OS controls memory map

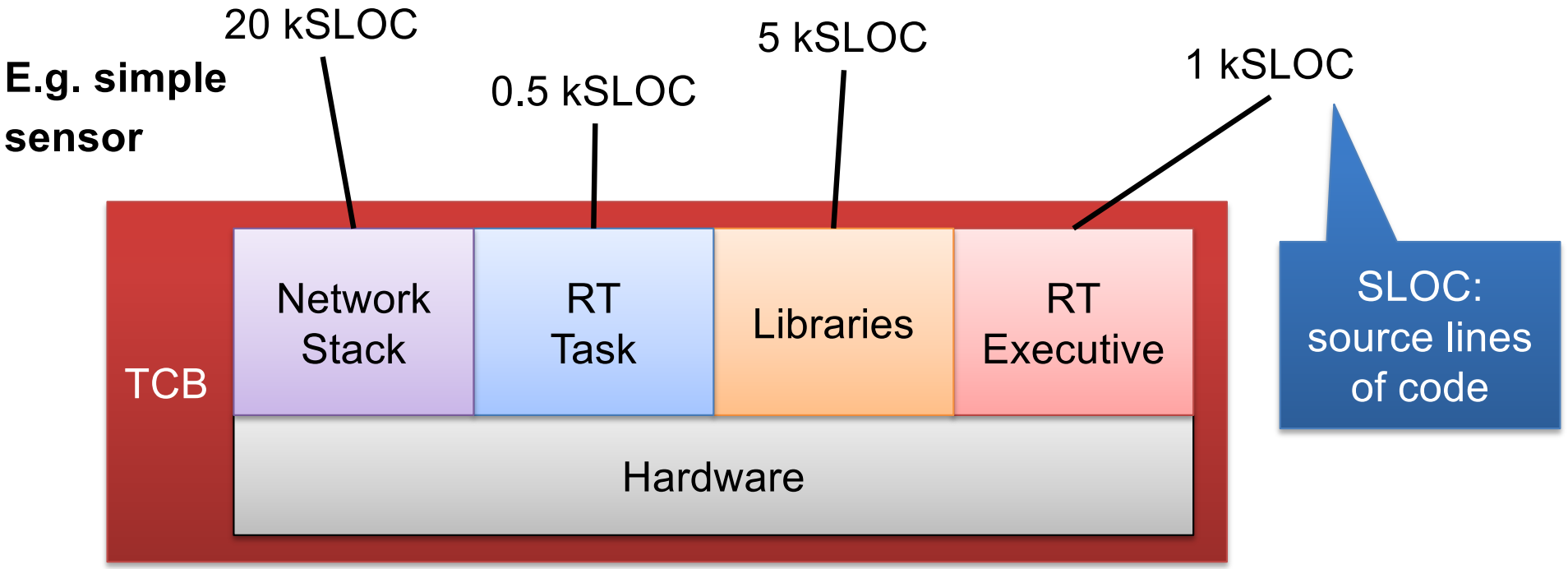
Typical microprocessor
- **x86**
- **ARM Cortex-A**



Real-Time Executives vs Security/Safety

- Cooperative system
- Everything trusts everything else
- Any bug anywhere can be an exploit

Totally defenceless, unsuitable for IoT



The Register®
Biting the hand that feeds IT

Security

81

IoT worm can hack Philips Hue lightbulbs, spread across cities

Easy chain reaction hack would spread across Paris, boffins say

By Darren Pauli 10 Nov 2016 at 06:02

SHARE ▼

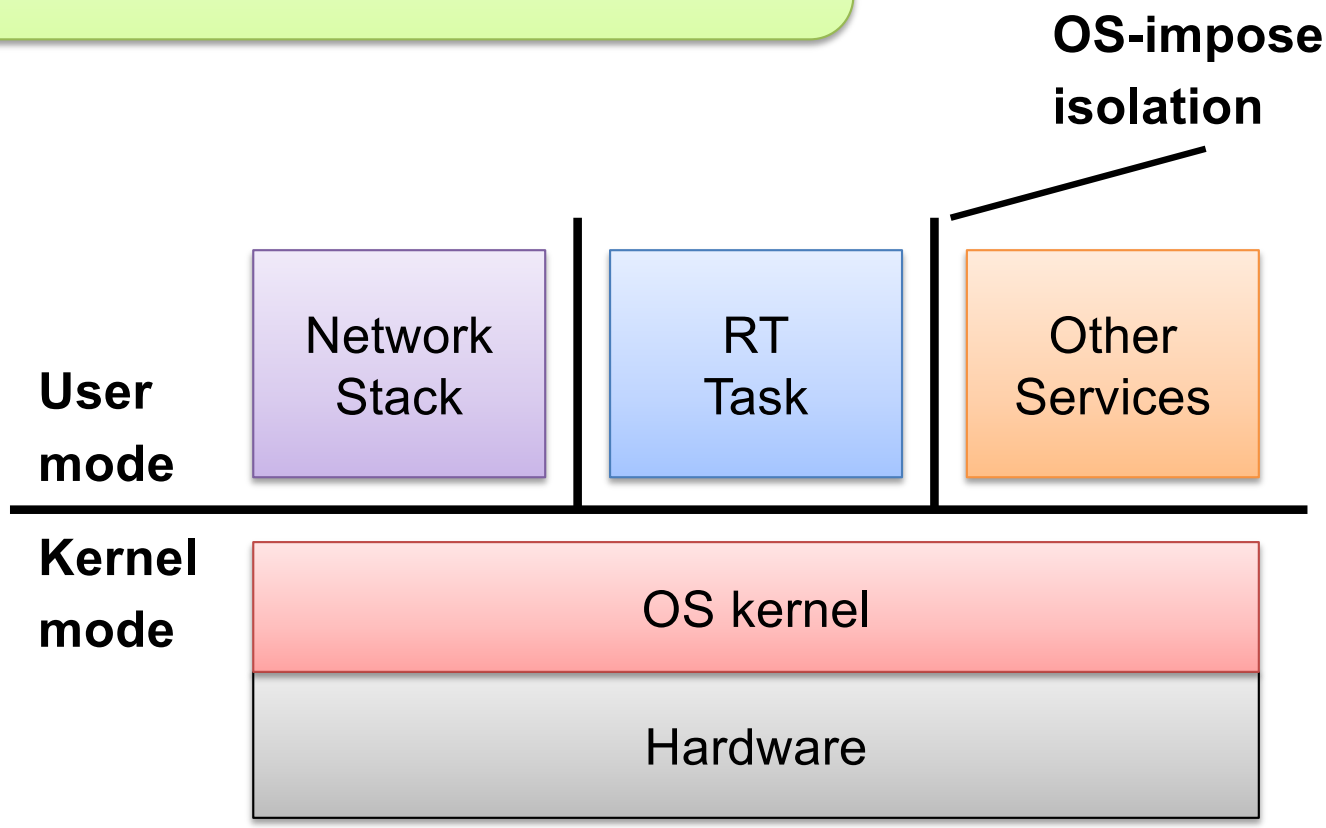
Researchers have developed a proof-of-concept worm they say can rip through Philips Hue lightbulbs across entire cities – causing the insecure web-connected globes to flick on and off.

The software nasty, detailed in a paper titled *IoT Goes Nuclear: Creating a ZigBee Chain Reaction* [PDF], exploits hardcoded symmetric encryption keys to control devices over Zigbee wireless networks. This allows the malware to compromise a single light globe from

Protected-Mode OS

- Misbehaving process cannot directly hurt OS or other process
- Potential to contain faults

Only sensible approach for non-trivial systems



CPS Challenge: SWaP

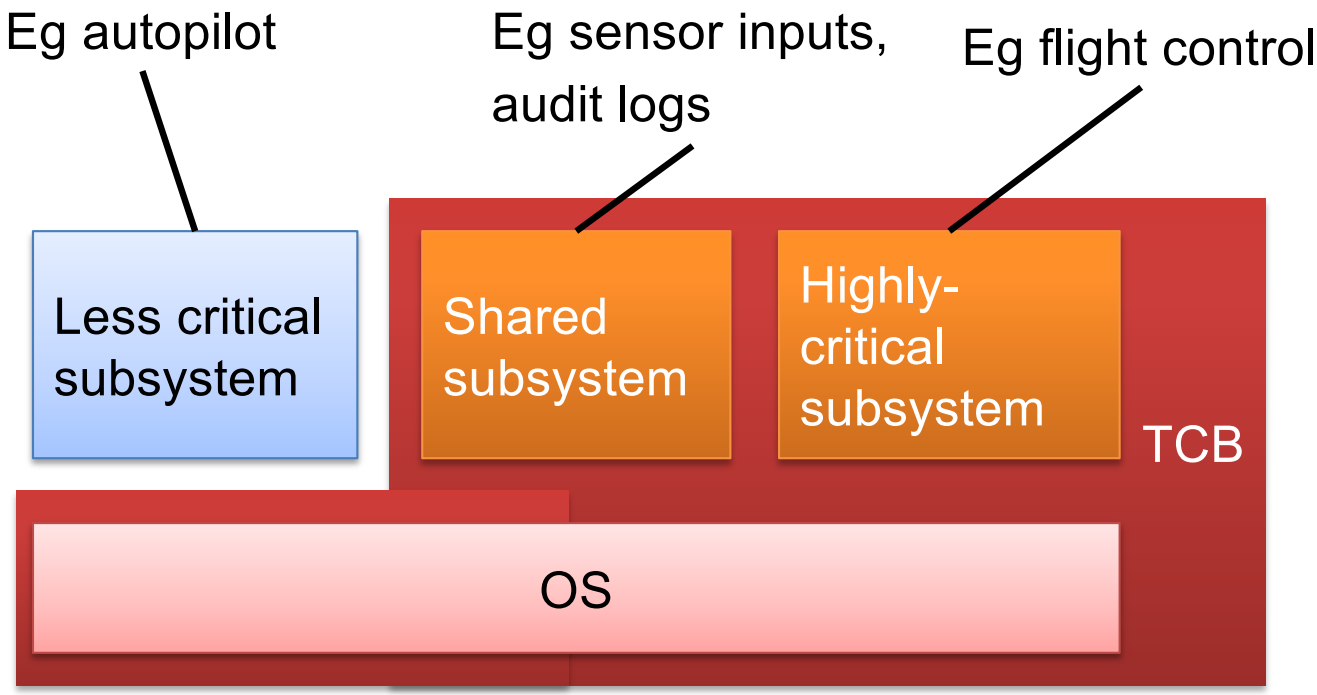
Traditional embedded-systems approach: one µ-controller per function

- Automotive reached 100 ECUs in top-of-line cars 10 years ago
- ECUs must be robust – expensive
  - Tolerant to wide temperature range
  - Resistant to dust, water, grease, acid
  - Resistant to Vibrations
- Packaging and cabling adds significant weight, consumes space & energy
- **SWaP: space, weight and power**
- Autonomous vehicles require far more functions than traditional
- General challenge for cyber-physical systems (CPS)
  - Robots, autonomous aircraft, smart factories

Way out: Consolidation of multiple functions on single processor

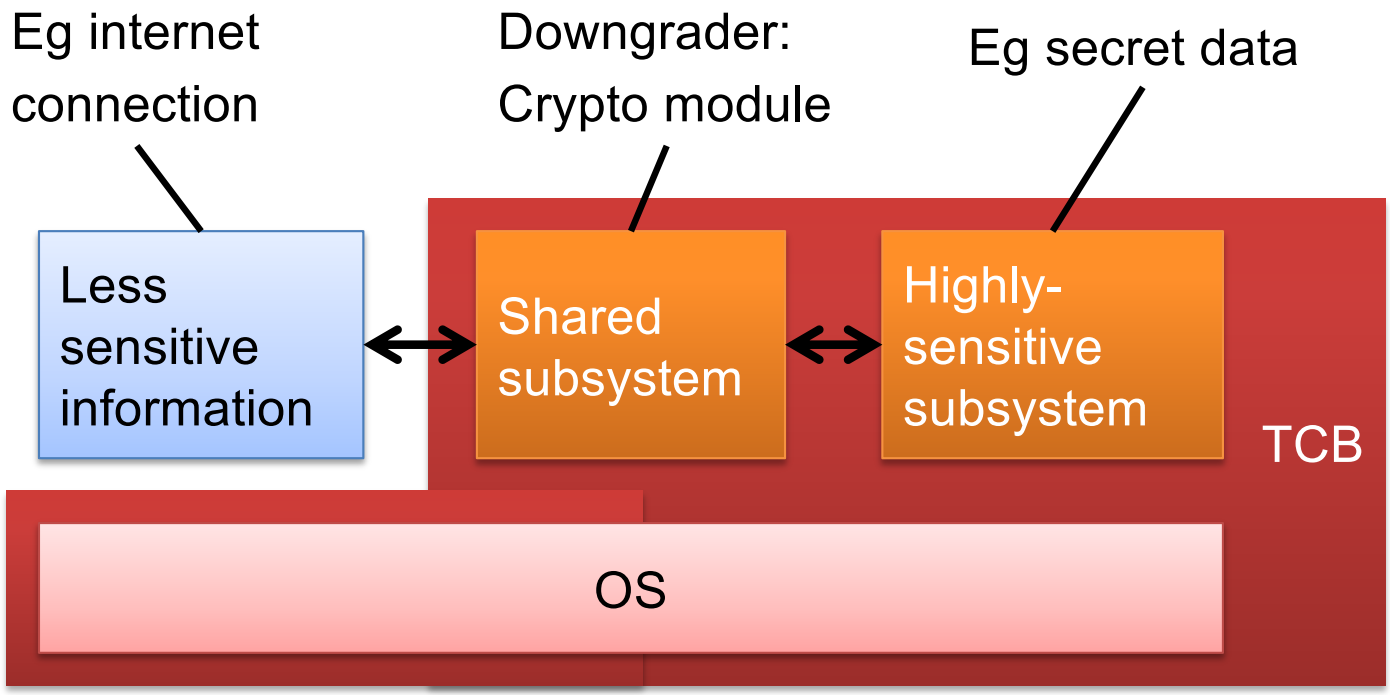
Consolidation: Mixed-Criticality Systems (MCS)

Certification requirement [ARINC-653]: More critical components must *not* depend on any less critical ones!



Security Equivalent: Cross-Domain Systems

Multiple classification levels on same device



OS Requirements for Security & Safety

An operating system for safety/security-critical systems must:

- Support functionalities of different criticalities
- Prevent low-crit functions from interfering with high-crit ones
- Prevent low-crit subsystems from inferring classified info
- Support certification of high-crit parts independent of low-crit
- Itself be certifiable at highest criticality

Enforce strong, certifiable isolation, spatial and temporal!

Operating Systems

For Secure and Safe Embedded Systems

Part 2: Security and OS Structure

@GernotHeiser

Copyright Notice

Security Design Principles

- Saltzer & Schroeder [SOSP ’73, CACM ’74]
  - **Economy of mechanism** – KISS
  - **Fail-safe defaults** – as in good engineering
  - **Complete mediation** – check everything
  - **Open design** – not security by obscurity
  - **Separation of privilege** – defence in depth
  - **Least privilege** – aka *principle of least authority* (POLA)
  - **Least common mechanism** – minimise sharing
  - **Psychological acceptability** – if it’s hard to use it won’t be

Security:
Access Control

Access Control

- **Who** can access **what** in which ways
  - The “who” are called **subjects**
    - e.g. users, processes etc.
  - The “what” are called **objects**
    - e.g. individual files, sockets, processes etc.
    - includes all subjects
  - The “ways” are called **permissions**
    - e.g. read, write, execute etc.
    - are usually specific to each kind of object
    - include those meta-permissions that allow modification of the protection state
      - e.g. own

Protection State

Access control matrix defines the **protection state** at particular time [Lampson'71]

	Obj1	**Obj2**	**Obj3**	**Subj2**
Subj1	R	RW		send
Subj2		RX		control
Subj3	RW		RWX own	recv

Note: All subjects are also objects!

Storing Protection State

- Not usually as access control matrix
  - too sparse, inefficient, dynamic
- Two obvious choices:
  - store individual **columns with each object**
    - defines the subjects that can access each object
    - each such column is called the object's **access control list**
  - store individual **rows with each subject**
    - defines the objects each subject can access
      aka subject's **protection domain**
    - each such row is called the subject's **capability list**

Access Control Lists (ACLs)

- Subjects usually aggregated into classes
  - e.g. UNIX: owner, group, everyone
  - more general lists in Windows
  - Can have negative rights
    eg. to overwrite group rights
- Meta-permissions (e.g. own)
  - control class membership
  - allow modifying the ACL
- Implemented in almost all commercial OSes

Obj1

Subj1	R
Subj2	
Subj3	RW

Capabilities

- A **capability** [Dennis & Van Horn, 1966] is a capability-list element

	Obj1	Obj2	Obj3	Subj2
Subj1	R	RW		send

  - **Names** an object to which the capability refers
  - **Confers** permissions over that object
- Capability is **prima facie authority** to perform an operation
  - System will perform operation iff appropriate capability is presented
- Less common in commercial systems
  - IBM System-38 → AS/400 → i-Series
  - KeyKOS (Visa transaction processing) [Bromberger et al, 1992]
- More common in research: EROS [Shapiro'99], Cheri, seL4

Capability-Based Access Control

Cap = Access Token:
Prima-facie evidence of privilege

Obj reference

Access rights

Object

Eg. thread, address space

Eg. read, write, send, execute…

Capabilities provide
- Fine-grained access control
- Reasoning about information flow

Any system call is invoking a capability:

err = method(cap, args);

Capabilities: Implementations

- Capabilities must be unforgeable
  - Traditionally protected by hardware (tagged memory), eg System-38
  - Can be copied etc like data
- On conventional hardware, either:
  - Stored as ordinary user-level data, but unguessable due to sparseness
    - contains password or secure hash: PCS [Anderson'86], Mungi
    - **"sparse" capabilies**
  - Stored separately (in-kernel), referred to by user programs by index/address, eg Mach [Accetta'86], EROS
    - **"partitioned" or "segregated" capabilities**
    - like UNIX file descriptors
- Sparse capabilities can be leaked more easily
  - Huge amplification of covert channels!

ACLs and Capabilities: Duals?

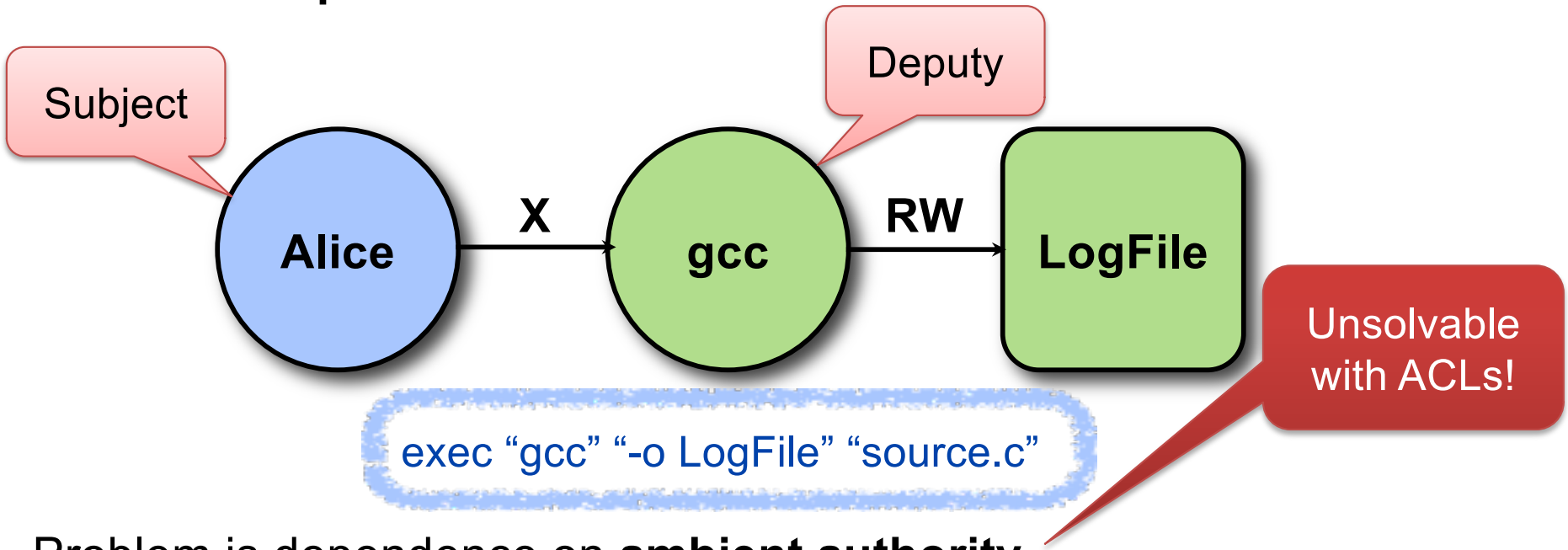
- In theory:
  - Dual representations of access control matrix
- Practical differences:
  - Naming and namespaces
    - Ambient authority
    - Deputies
  - Evolution of protection state
  - Forking
  - Auditing of protection state

Duals? Naming and Namespaces

- ACLs:
  - objects referenced by **name**
    - e.g. open(“/etc/passwd”,O_RDONLY)
  - require a subject (class) namespace
    - e.g. UNIX users and groups
- Capabilities:
  - objects referenced by **capability**
  - no further namespace required

Duals? Confused Deputies

- ACLs: separation of object naming and permission can lead to **confused deputies**



- Problem is dependence on **ambient authority**
  - Deputy uses its own authority when performing action on behalf of client
- Capabilities are both names and permissions, avoids confusion
  - You can’t name something without having permission to it
  - Presentation is **explicit** (not ambient)

Duals? Evolution of Protection State

- ACLs:
  - Protection state changes by modifying ACLs
    - Requires certain meta-permissions on the ACL
- Capabilities:
  - Protection state changes by delegating and revoking capabilities
  - Fundamental properties enable reasoning about **information flow**:
    - A can send message to B only if A holds cap to B
    - A can obtain access to C only if it receives message with cap to C
  - **Right to delegate** may also be controlled by capabilities
    - e.g. A can delegate to B only if A has a capability to B that carries appropriate permissions
    - A can delegate X to B only if it has **grant** authority on X

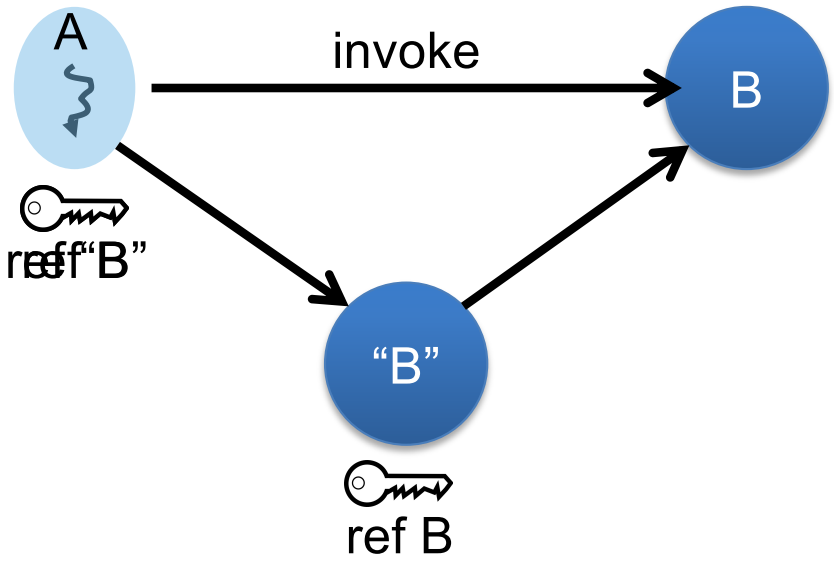
Duals? Forking

- What permissions should children get?
- ACLs: depends on the child's subject
  - UNIX etc.: child inherits parent's subject
    - Inherits **all** of the parent's permissions
    - Any program you run inherits all of your authority
    - Eg must trust web browser not to leak data
  - **Violation of least privilege**
- Capabilities: child has no caps by default
  - Parent gets a capability to the child upon fork
  - Used to delegate explicitly the necessary authority
  - **Defaults to least privilege**

Interposing Object Access

Caps are opaque object references (pure names)

- Holder cannot tell which object a cap references nor the authority
- Supports transparent interposition (virtualisation)



Usage:

- API virtualisation
- Security monitor
  - Security policy enforcement
  - Info flow tracing
  - Packet filtering…
- Secure logging
- Debugging
- Lazy object creation
  - Initial cap to constructor
  - Replace by proper object cap

Duals: Saltzer & Schroeder Principles

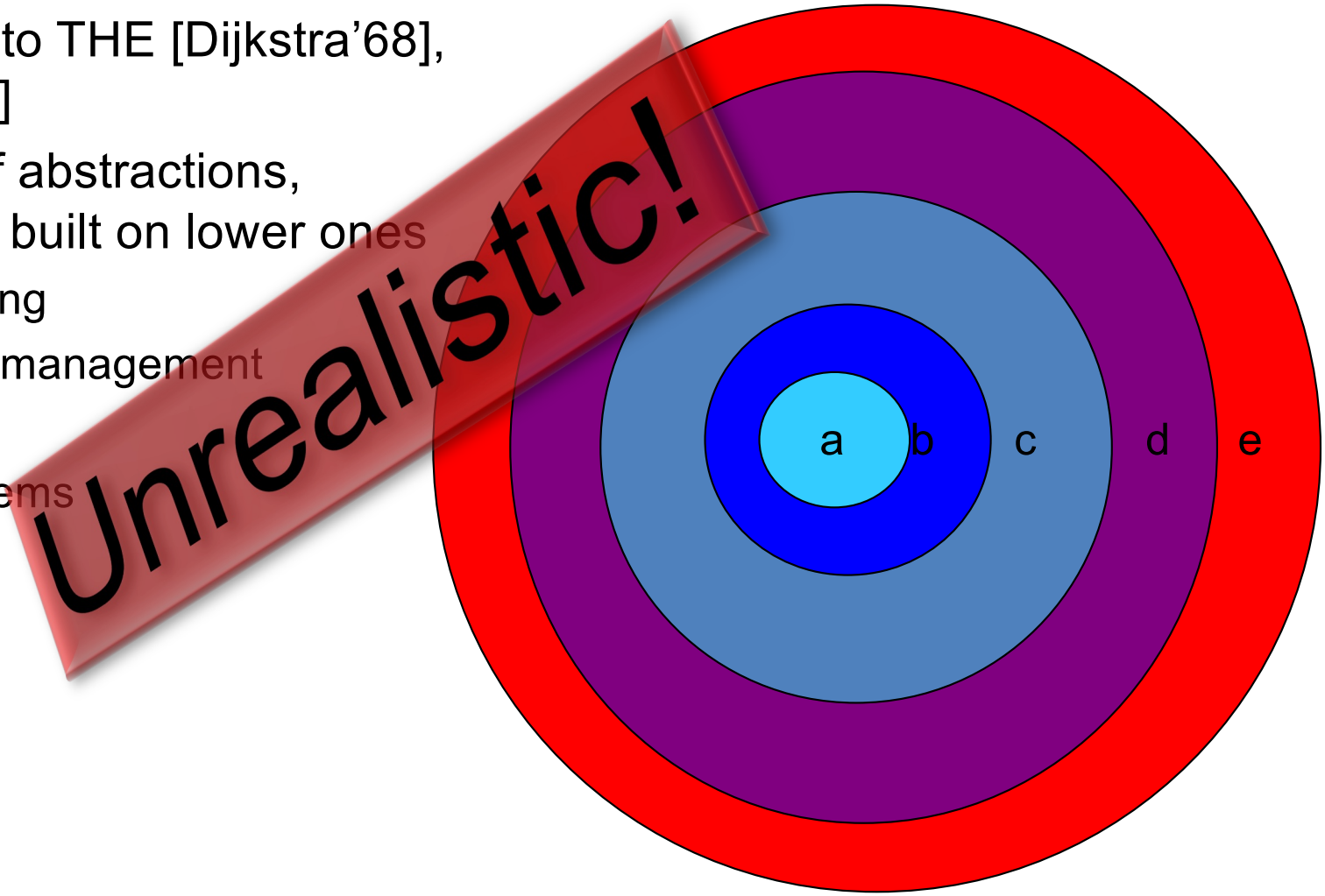
Security Principle	ACLs	Capabilities
Economy of Mechanism	Dubious	Yes!
Fail-safe defaults	Generally not	Yes!
Complete mediation	Yes (if properly done)	Yes (if properly done)
Open design	Neutral	Neutral
Separation of privilege	No	Doable
Least privilege	No	Yes
Least common mechanism	No	Yes
Psychological acceptability	Neutral	Neutral

OS Structure

OS Structure

Classic layered approach

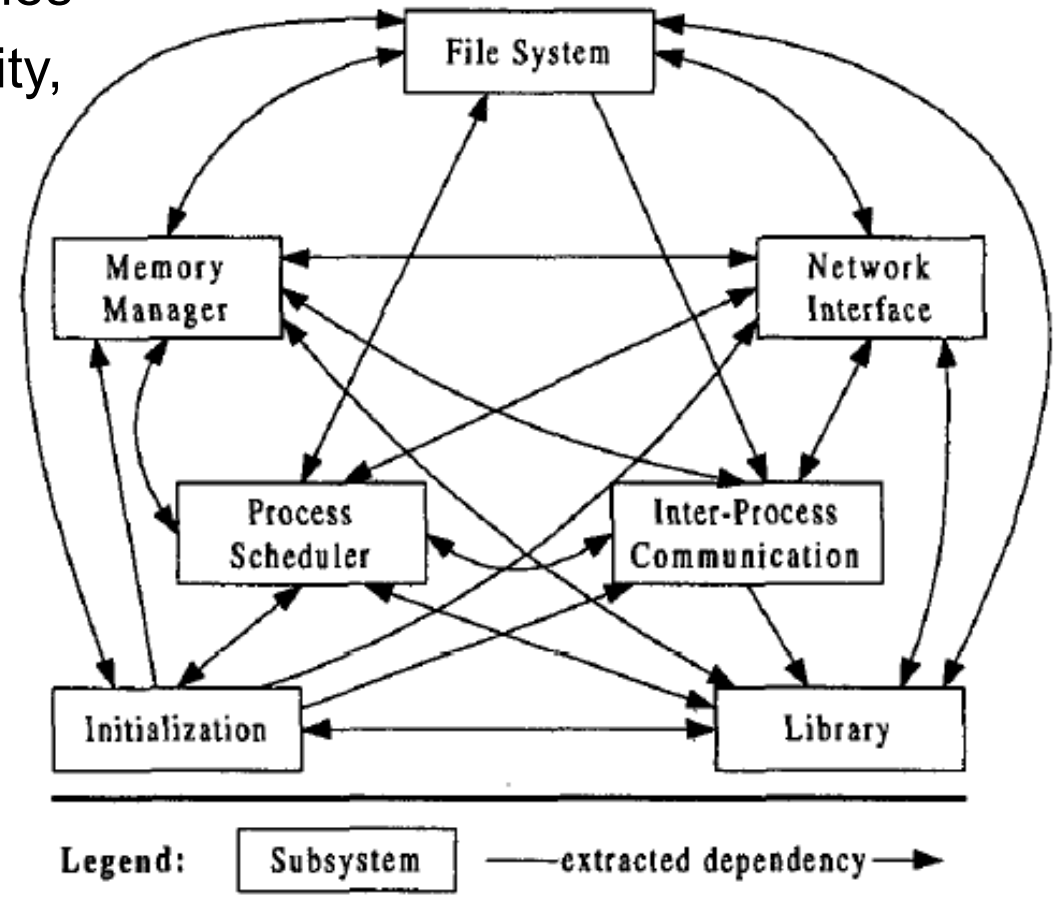
- Going back to THE [Dijkstra'68], Multics [60s]
- Hierarchy of abstractions, higher ones built on lower ones
  1. Scheduling
  2. Memory management
  3. Devices
  4. File systems
  5. Users



Courtesy Kevin Elphinstone

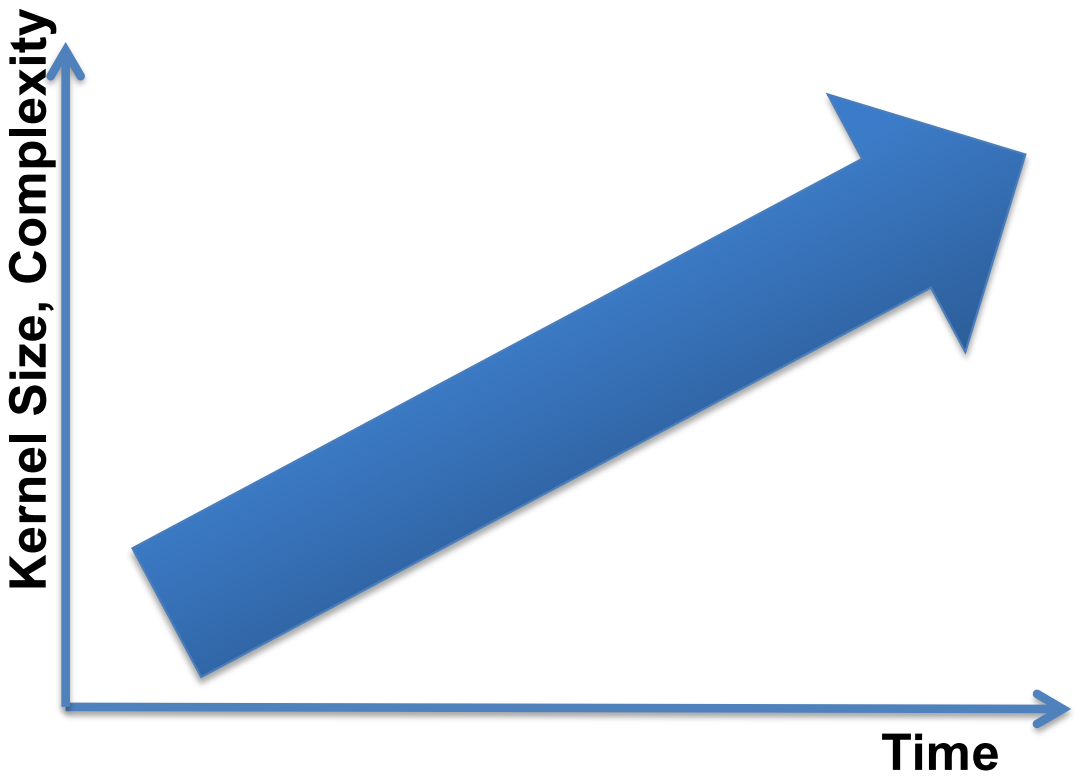
Problem with Layered Model

- Too many inter-dependencies
- Resulting in weak modularity, layer-cutting
- Complex interactions of functionality no-one understands
- Huge number of corner cases that are impractical to test



Courtesy Kevin Elphinstone

Trends in Operating Systems



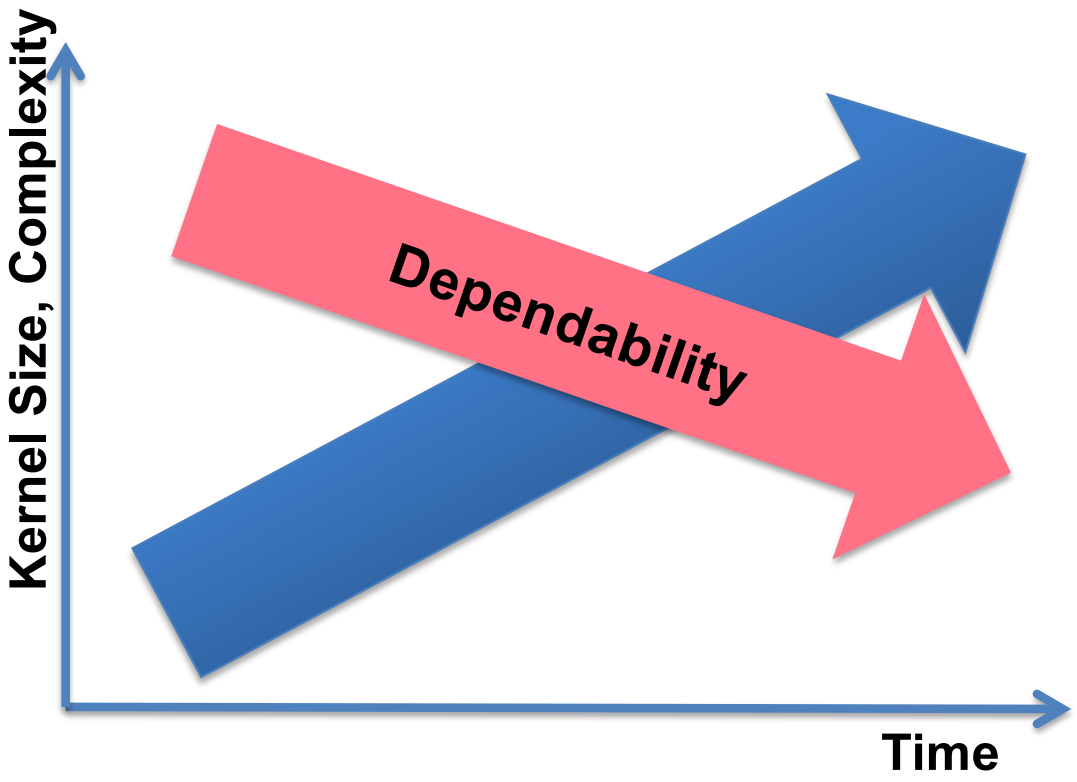
Complexity Drivers

- New hardware
  - New device drivers / driver classes
  - New file systems
  - Multicore scalability
- New usage domains
  - Better power management
  - New network protocols
  - Better real-time behaviour
- New security challenges
  - New crypto libs, protocols
  - Improved access control
- Etc …

Complexity: Enemy of Dependability

- Typical defect density of industry-standard code: 2–5 bugs per kSLOC
  - Linux might be somewhat better: ≈ 1 bug/kSLOC
- 10–25% of kernel bugs are security vulnerabilities
  - Conservatively, this means 0.1 vulnerability / kSLOC
- Linux kernel is 10s of MSLOC ⇒ **thousands of vulnerabilities!**
  - Plus system services (daemons) running with high privileges

Trends in Commodity Operating Systems



Complexity Drivers

- New hardware
  - New device drivers / driver classes
  - New file systems
  - Multicore scalability
- New usage domains
  - Better power management
  - New network protocols
  - Better real-time behaviour
- New security challenges
  - New crypto libs, protocols
  - Improved access control
- Etc …

Complexity: Enemy of Dependability

- Typical defect density of industry-standard code: 2–5 bugs per kSLOC
  - Linux might be somewhat better: ≈ 1 bug/kSLOC
- 10–25% of kernel bugs are security vulnerabilities
  - Conservatively, this means 0.1 vulnerability / kSLOC
- Linux kernel is 10s of MSLOC ⇒ thousands of vulnerabilities!

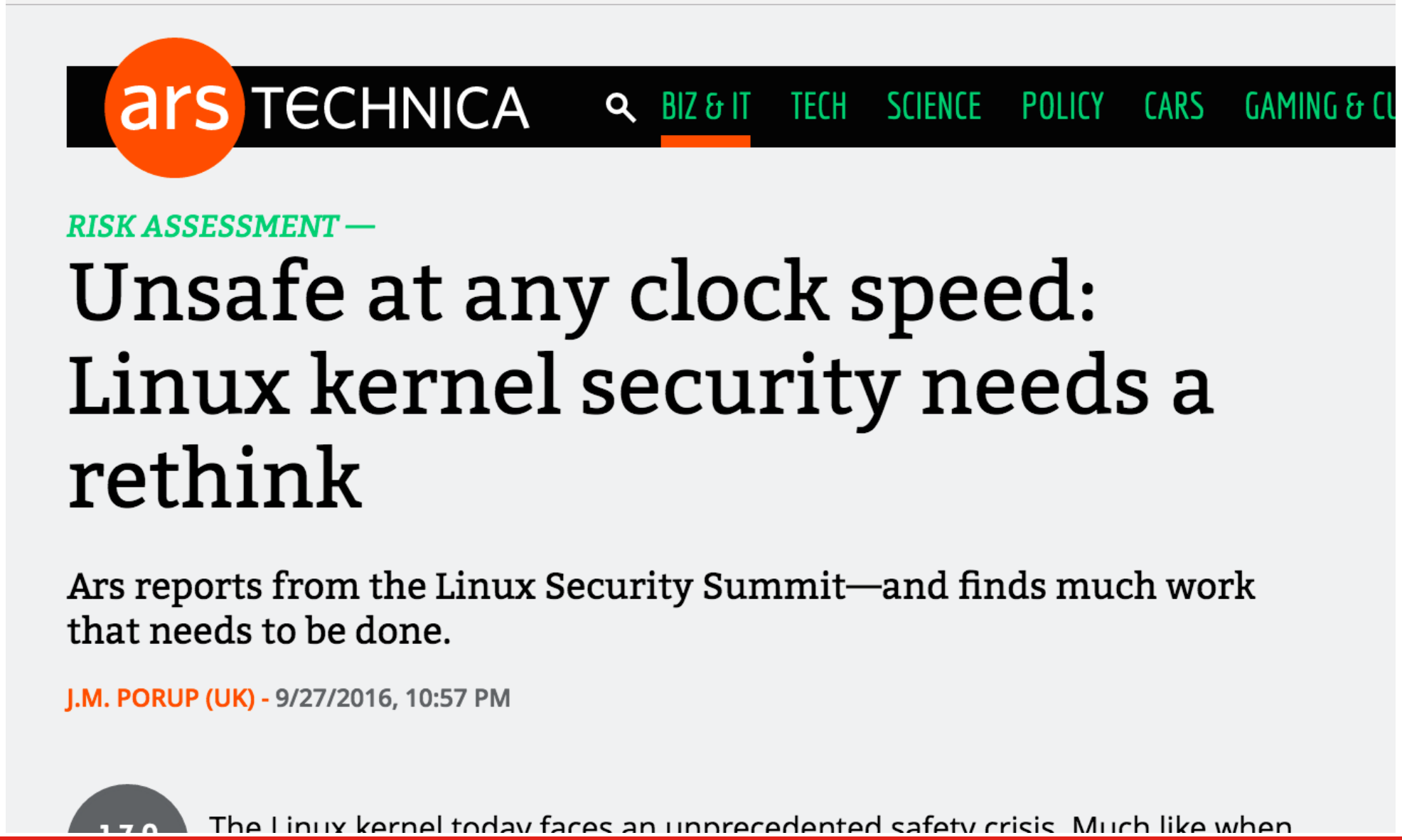
Core problem: New features increase kernel complexity

⇒ reduced dependability

- Impossible to assure ***security*** – too many bugs
- Impossible to assure ***safety*** – too complex to analyse timeliness

The monolithic OS model
Is fundamentally broken!

I’m not alone saying this…



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

The Linux kernel today faces an unprecedented safety crisis. Much like when

Operating Systems

For Secure and Safe Embedded Systems

Part 3: Microkernels and seL4

@GernotHeiser

Copyright Notice

Microkernels

What is Needed for Safety & Security?

Need certifiable argument for isolation:

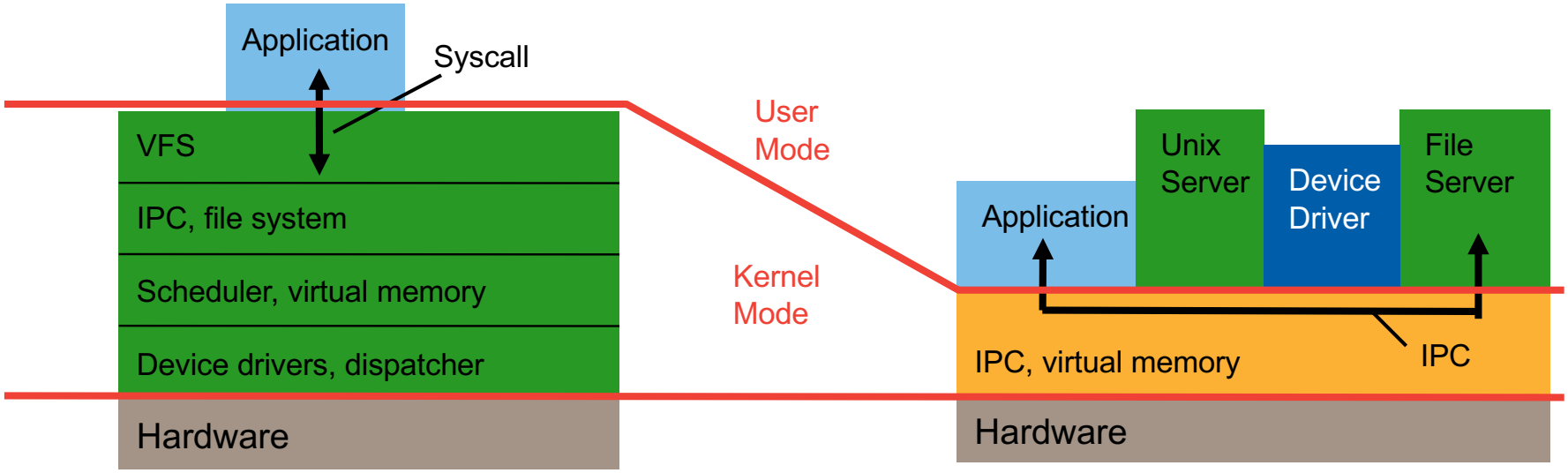
- Able to convince a skeptical certification authority
- Requires thorough analysis of trusted computing base
  - What can possibly go wrong?
  - Usually informal or semi-formal arguments
  - Ideally formal proof

Intractable unless
- **Small TCB**
- **Low conceptual complexity**
- **Well-defined interfaces/ interactions**

Reducing TCB: Microkernels

- Idea of microkernel:
  - Flexible, minimal platform
  - Mechanisms, not policies
  - Actual OS functionality provided by user-mode servers
  - Servers invoked by kernel-provided message-passing mechanism (IPC)
  - Goes back to Nucleus [Brinch Hansen'70]

IPC performance is critical!



Monolithic vs Microkernel OS Evolution

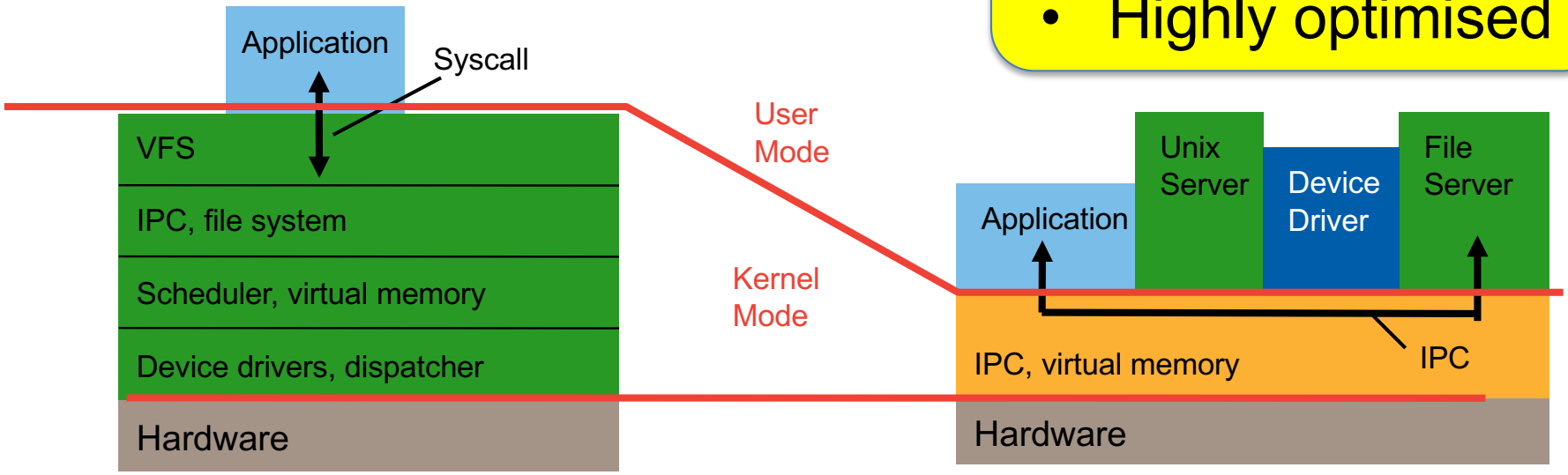
Monolithic OS

- New features add code kernel
- New policies add code kernel
- Kernel complexity grows

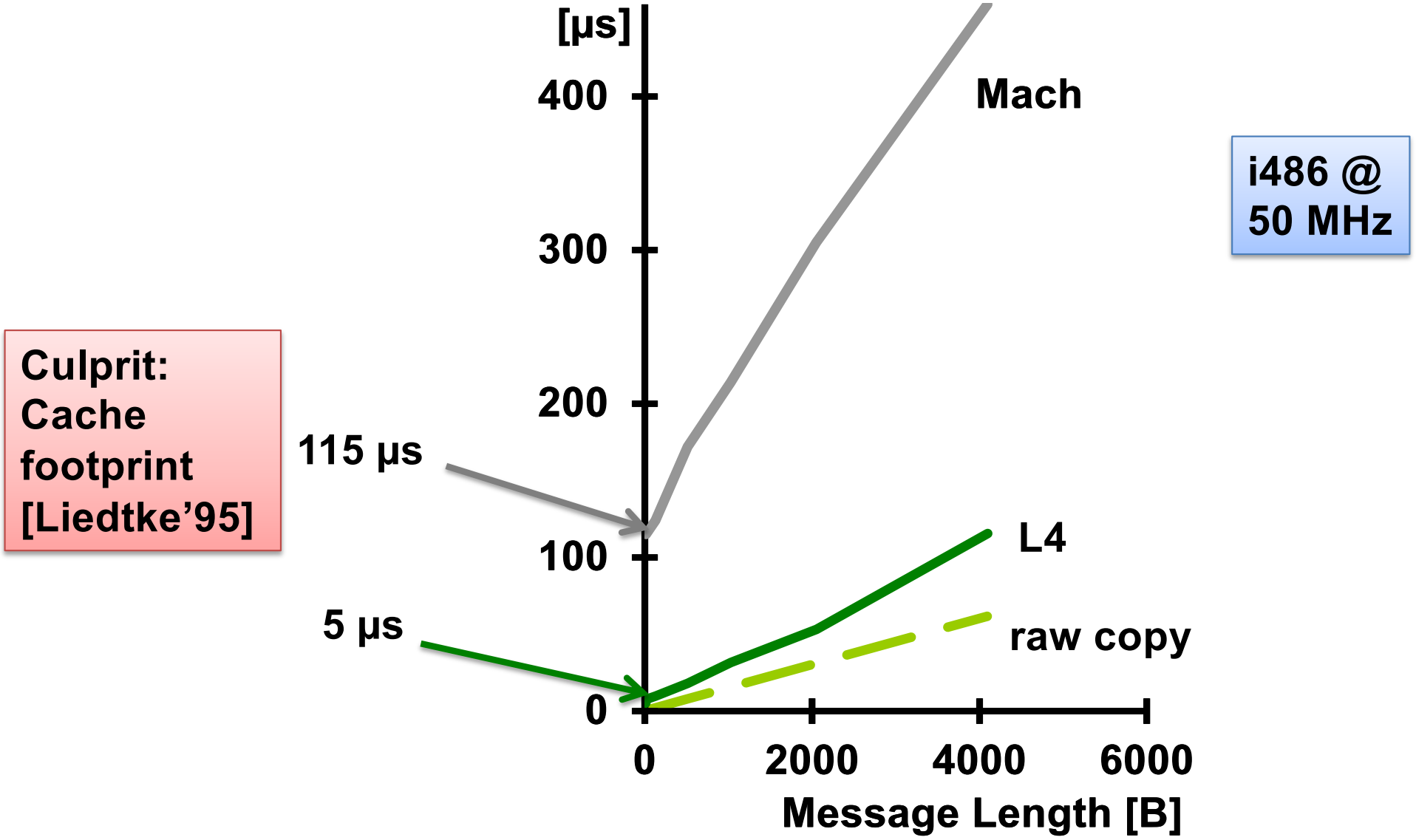
Microkernel OS

- Features add usermode code
- Policies replace usermode code
- Kernel complexity is stable

- Adaptable
- Dependable
- Highly optimised



1993 “Microkernel”: IPC Performance



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. [SOSP'95]

- Advantages of resulting small kernel:
  - Easy to implement, port?
  - Easier to optimise
  - Hopefully enables a minimal *trusted computing base*
  - Easier debug, maybe even *prove* correct?
- Challenges:
  - API design: generality despite small code base
  - Kernel design and implementation for high performance

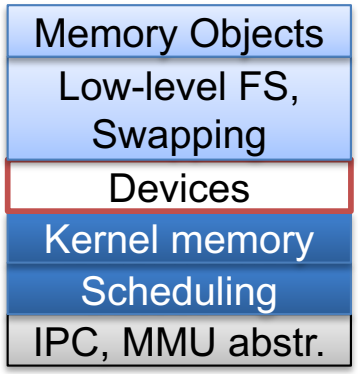
Limited by arch-specific micro-optimisations

Small attack surface, fewer failure modes

Microkernel Evolution

First generation

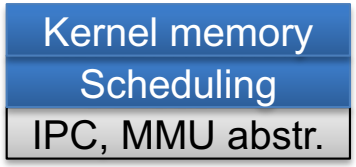
- Eg Mach [’87] (QNX, Chorus)



- 180 syscalls
- 100 kSLOC
- 100 µs IPC

Second generation

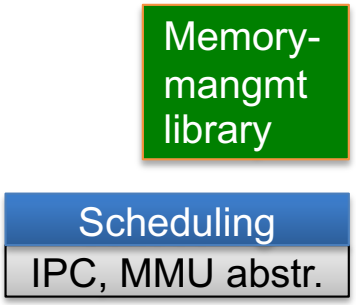
- L4 [’95] (PikeOS, Integrity)



- ~7 syscalls
- ~10 kSLOC
- ~ 1 µs IPC

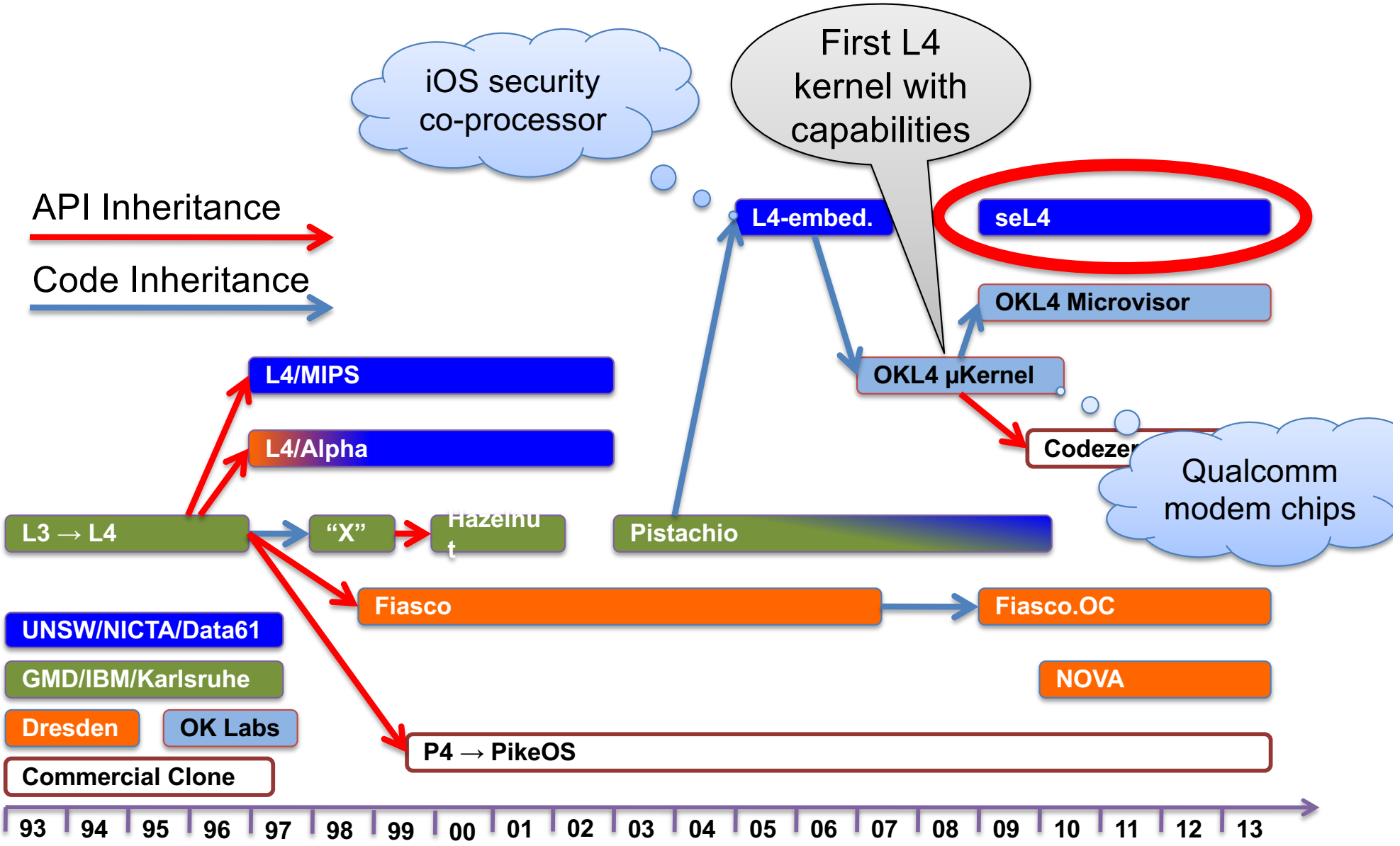
Third generation

- seL4 [’09]



- ~3 syscalls
- 9 kSLOC
- 0.1 µs IPC
- *capabilities*
- *design for isolation*

L4: A Family of High-Performance Microkernels



L4 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	**2017**	**i7 Skylake (32-bit)**	**3,400**	**203**	**0.06**
seL4	**2017**	**I7 Skylake (64-bit)**	**3,400**	**138**	**0.04**
seL4	**2017**	**Cortex A53**	**1,200**	**225**	**0.19**

Minimality: Source Code Size

Name	Architecture	C/C++	asm	total kSLOC
Original	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**

What Mechanisms?

- Fundamentally, the microkernel must abstract
  - *Physical memory:* Address spaces
  - *CPU:* Threads
  - *Interrupts/Exceptions*
- Unfettered access to any of these bypasses security
  - No further abstraction needed for devices
    - memory-mapping device registers and interrupt abstraction suffices
    - …but some generalised memory abstraction needed for I/O space
- Above isolates execution units, hence microkernel must also provide
  - *Communication* (traditionally referred to as *IPC*)
  - *Synchronization*

Design subject to performance goals:

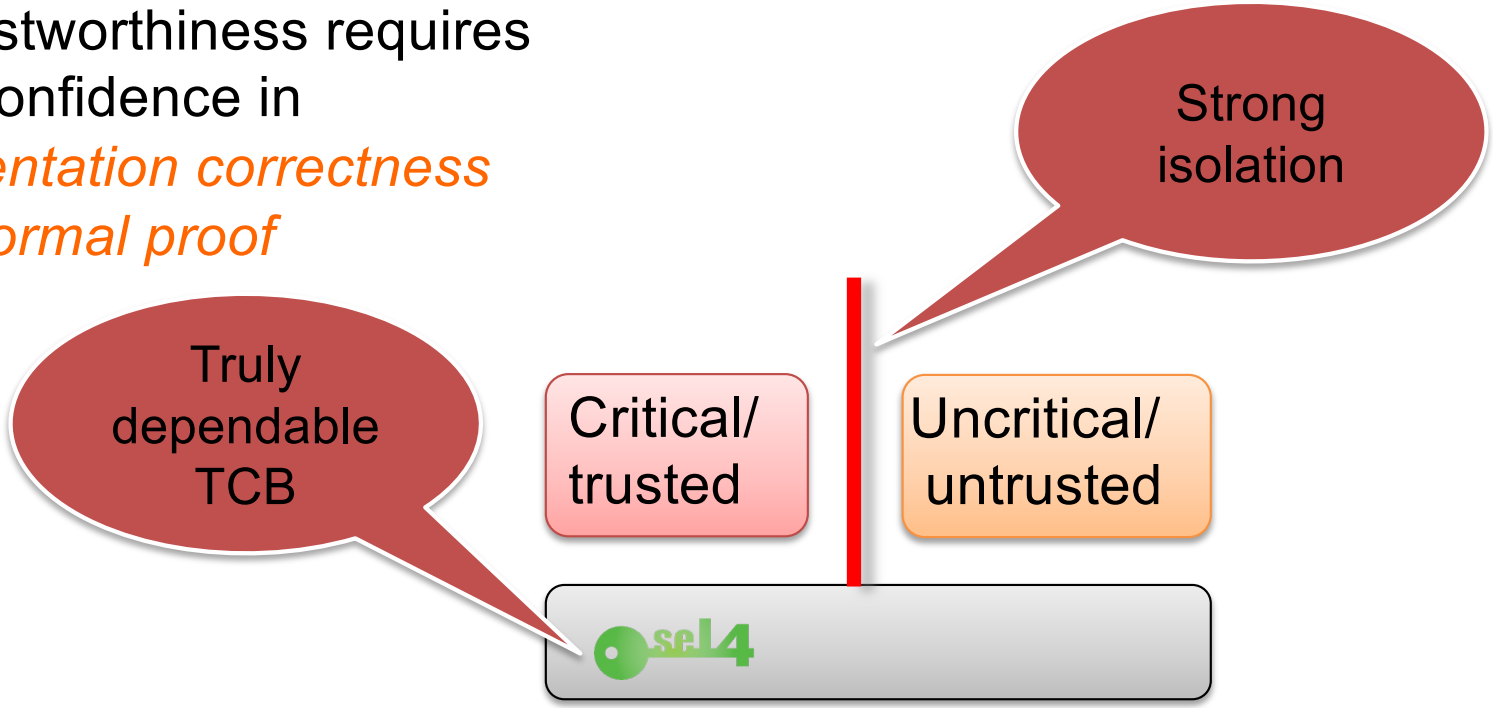
- Frequent operations as fast as possible (near hardware limit)
- Don't pay for what you don't need

The seL4 Microkernel

Design Motivation

1. *Object capabilities* are good for reasoning about usermode access
   - Just retro-fitting them to traditional L4 is insufficient:
     - **Availability** – need strong control over kernel resources
     - **Confidentiality** – reason about information flow through kernel data

2. Real trustworthiness requires strong confidence in *implementation correctness* ideally *formal proof*



Fundamental Design Decisions

1. Memory management is user-level responsibility
   - Kernel never allocates memory (post-boot)
   - Kernel objects controlled by user-mode servers

Isolation

Perfor-mance

2. Memory management is fully delegatable
   - Supports hierarchical system design
   - Enabled by *capability-based access control*

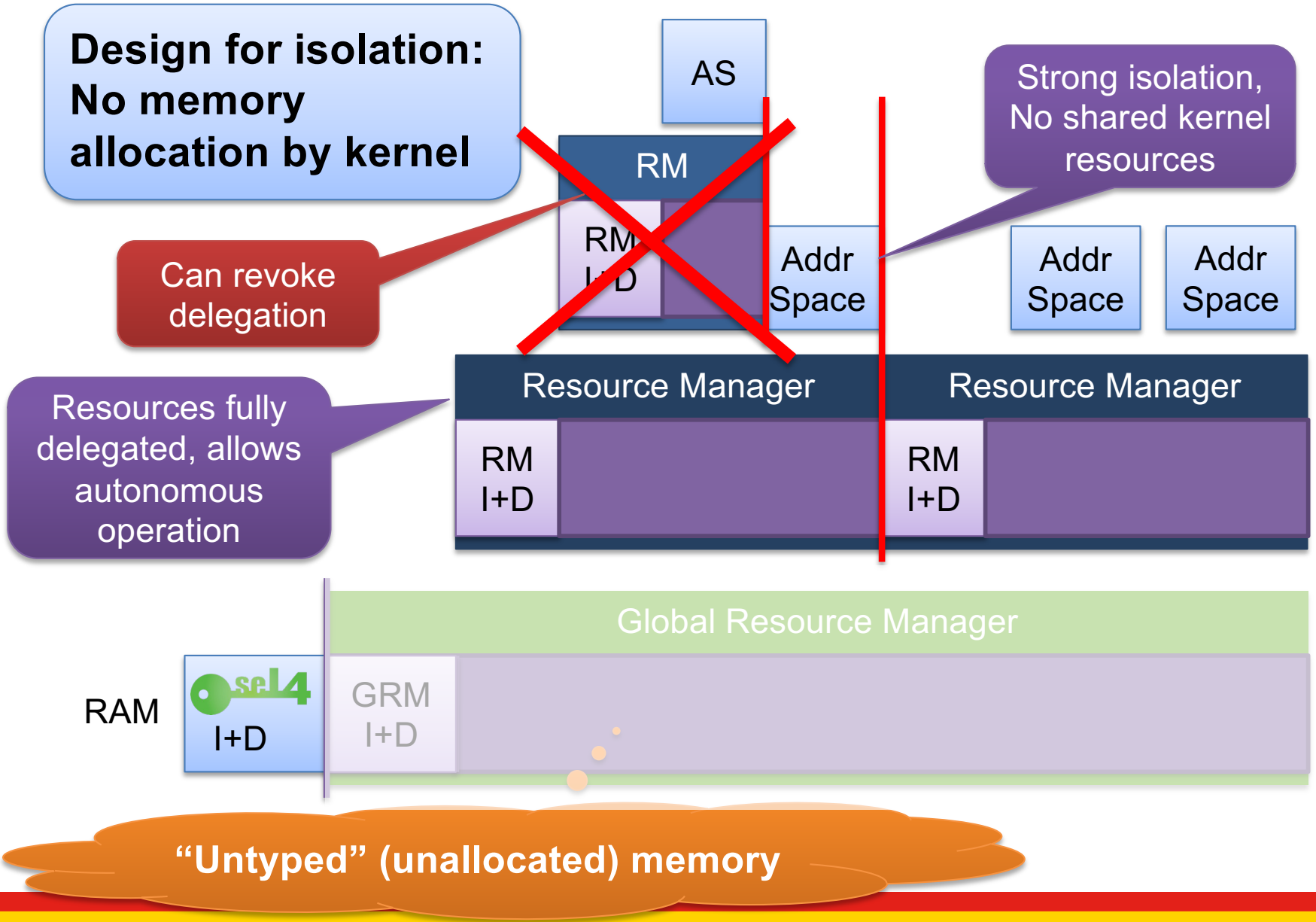
3. "Incremental consistency" design pattern
   - Fast transitions between consistent states
   - Restartable operations with progress guarantee

Real-time

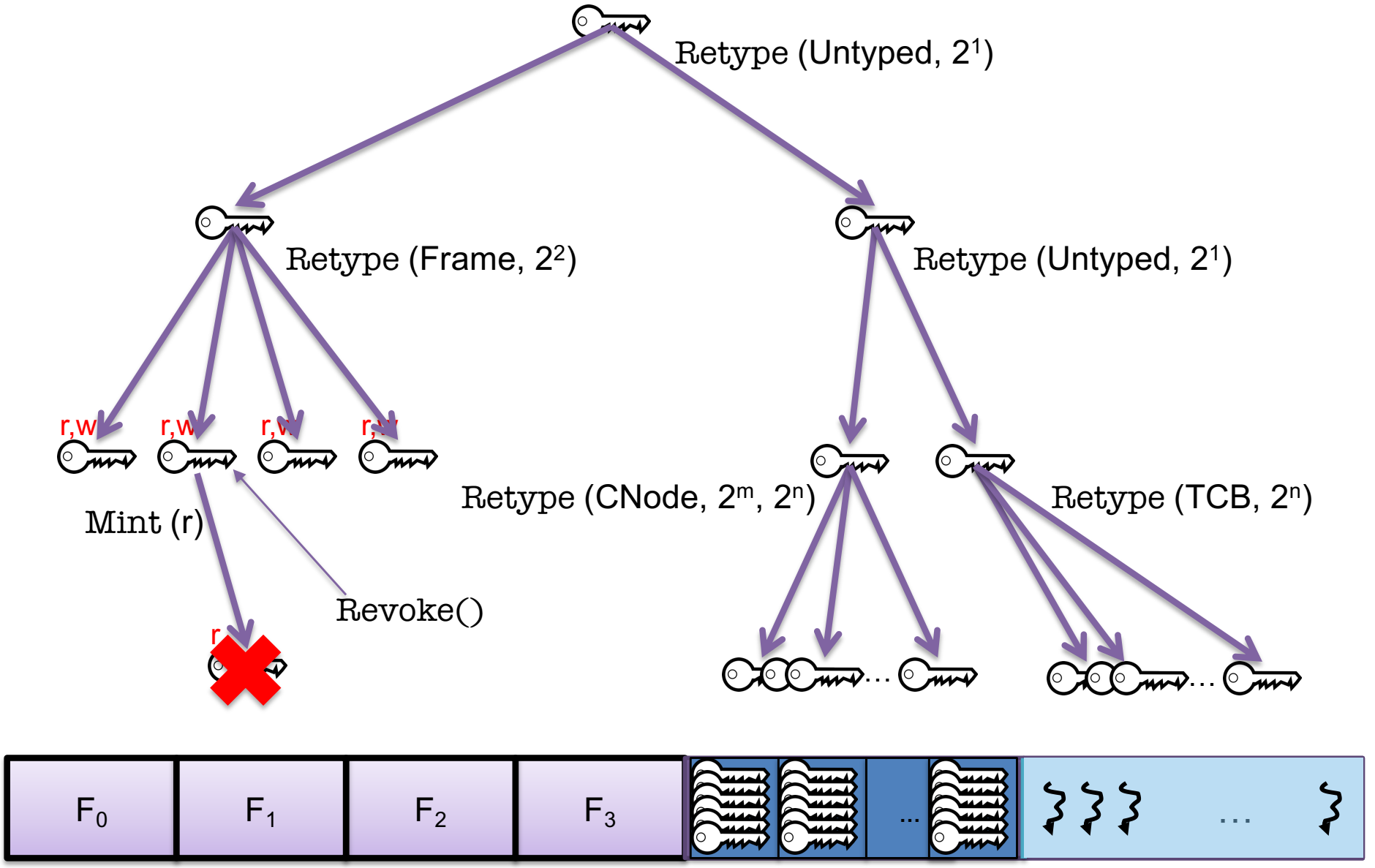
4. No concurrency in the kernel
   - *Interrupts never enabled in kernel*
   - Interruption points to bound latencies
   - Clustered multikernel design for multicores

Verification, Performance

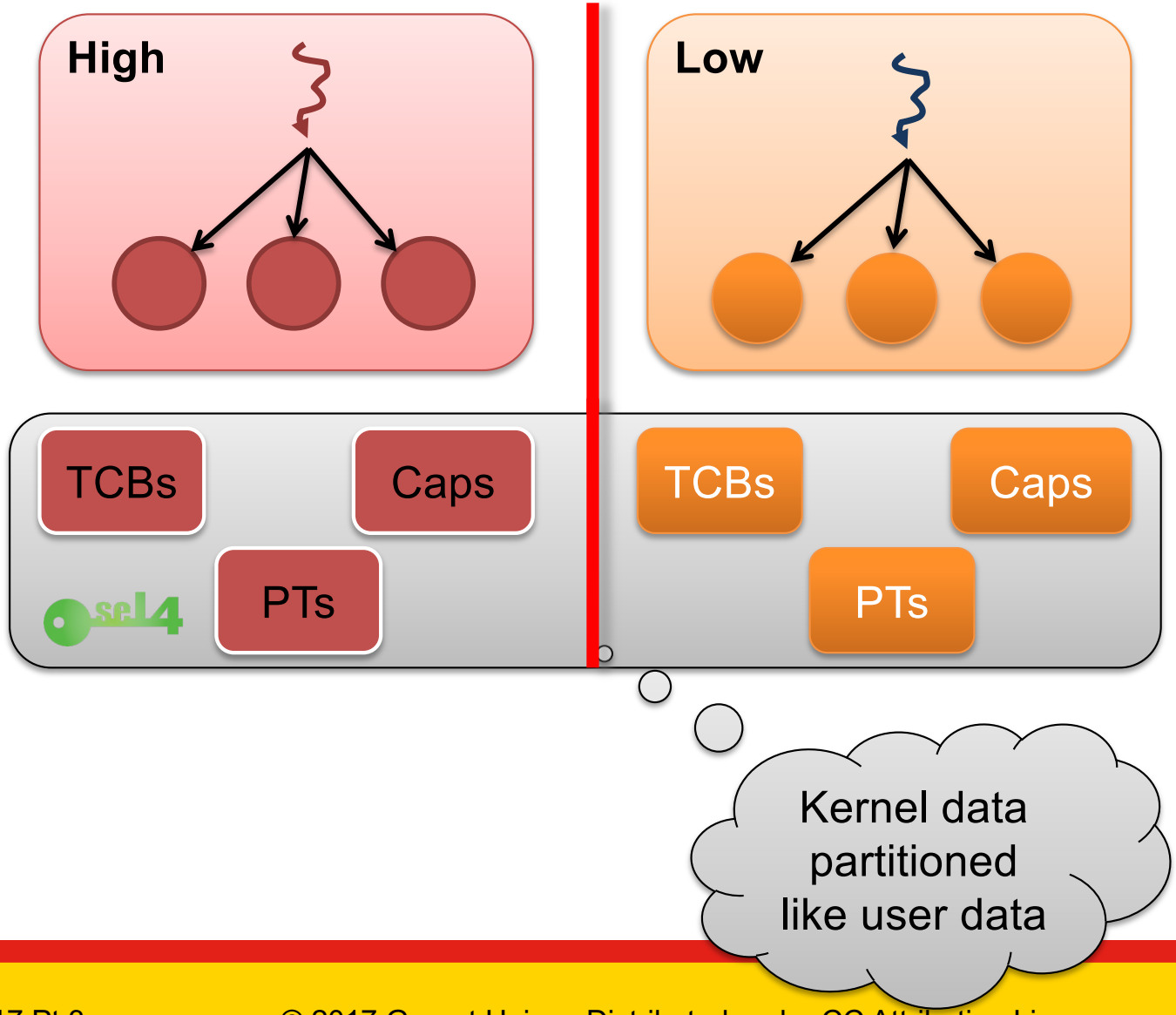
What's Different to Other Microkernels?



Core Mechanism: Retype of "Untyped" Memory



seL4 Isolation Goes Deep



How About Real Time?

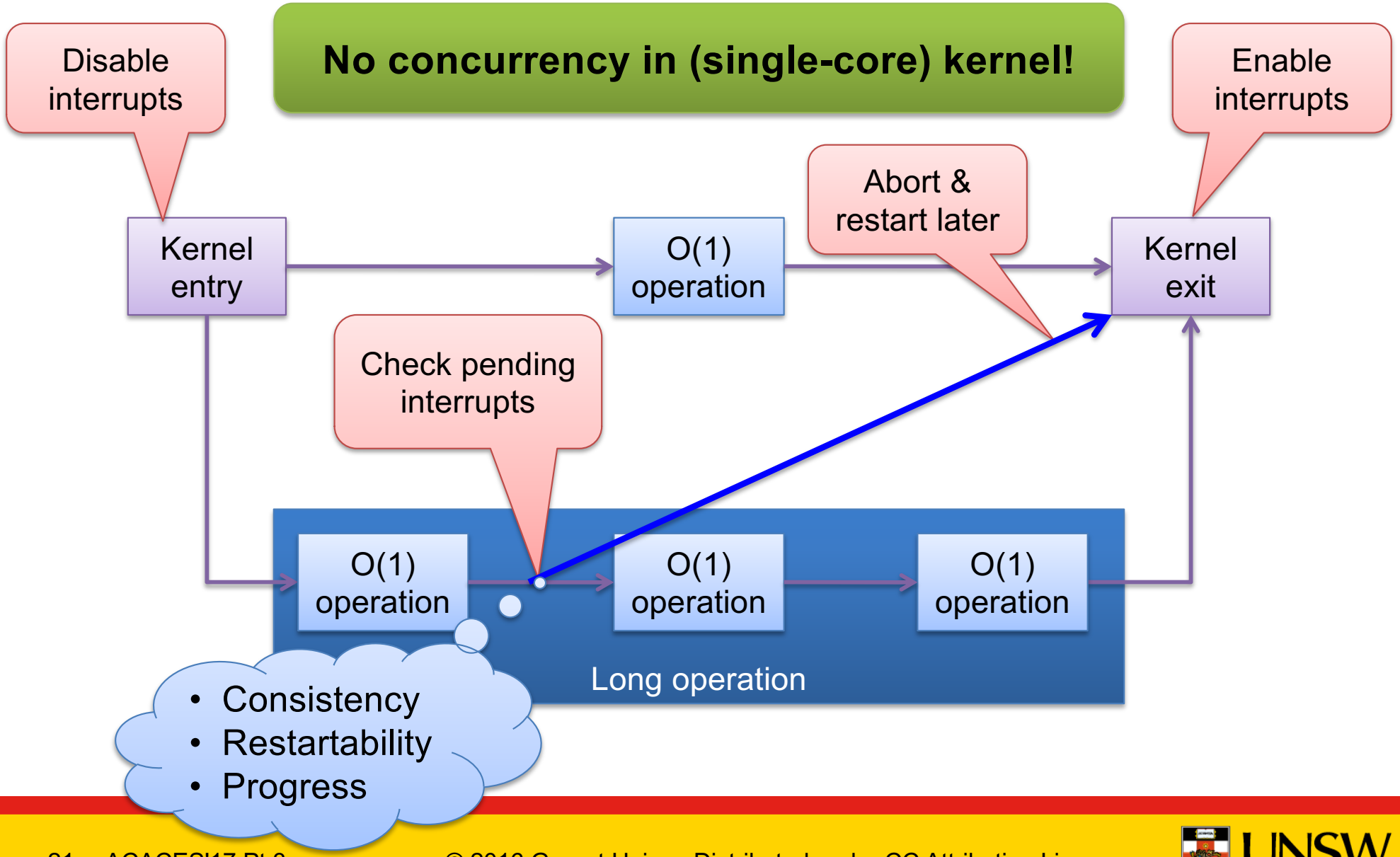
- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
    - Easier reasoning about correctness
    - Better average-case performance
- How about long-running system calls?
  - Use strategic *premption points*
  - (Original) Fiasco has fully preemptible kernel
    - Like commercial microkernels (QNX, Green Hills INTEGRITY)

Limited concurrency in kernel!

```
while (!done) {
  process_stuff();
  PSW.IRQ_disable=1;
  PSW.IRQ_disable=0;
}
```

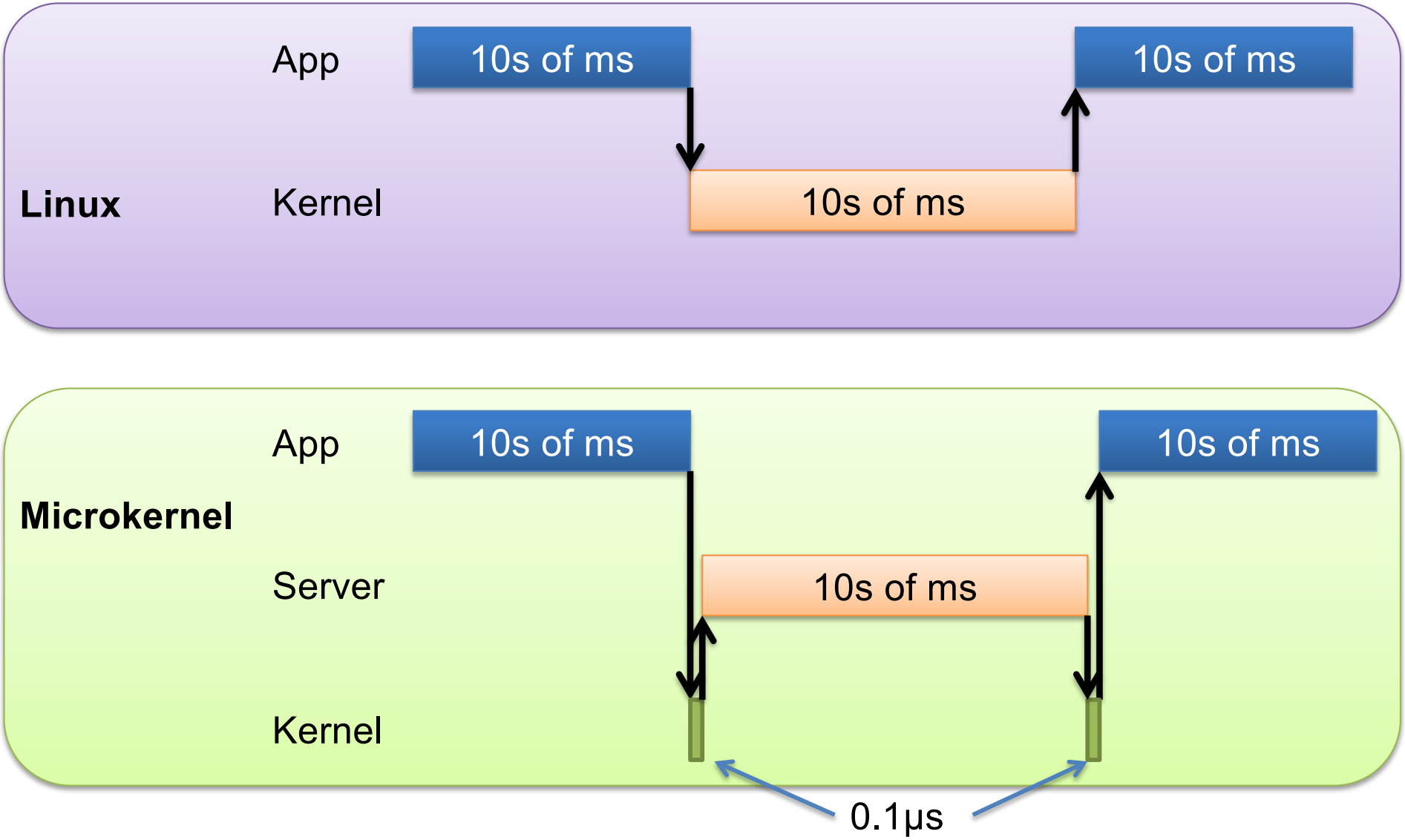
Lots of concurrency in kernel!

Incremental Consistency

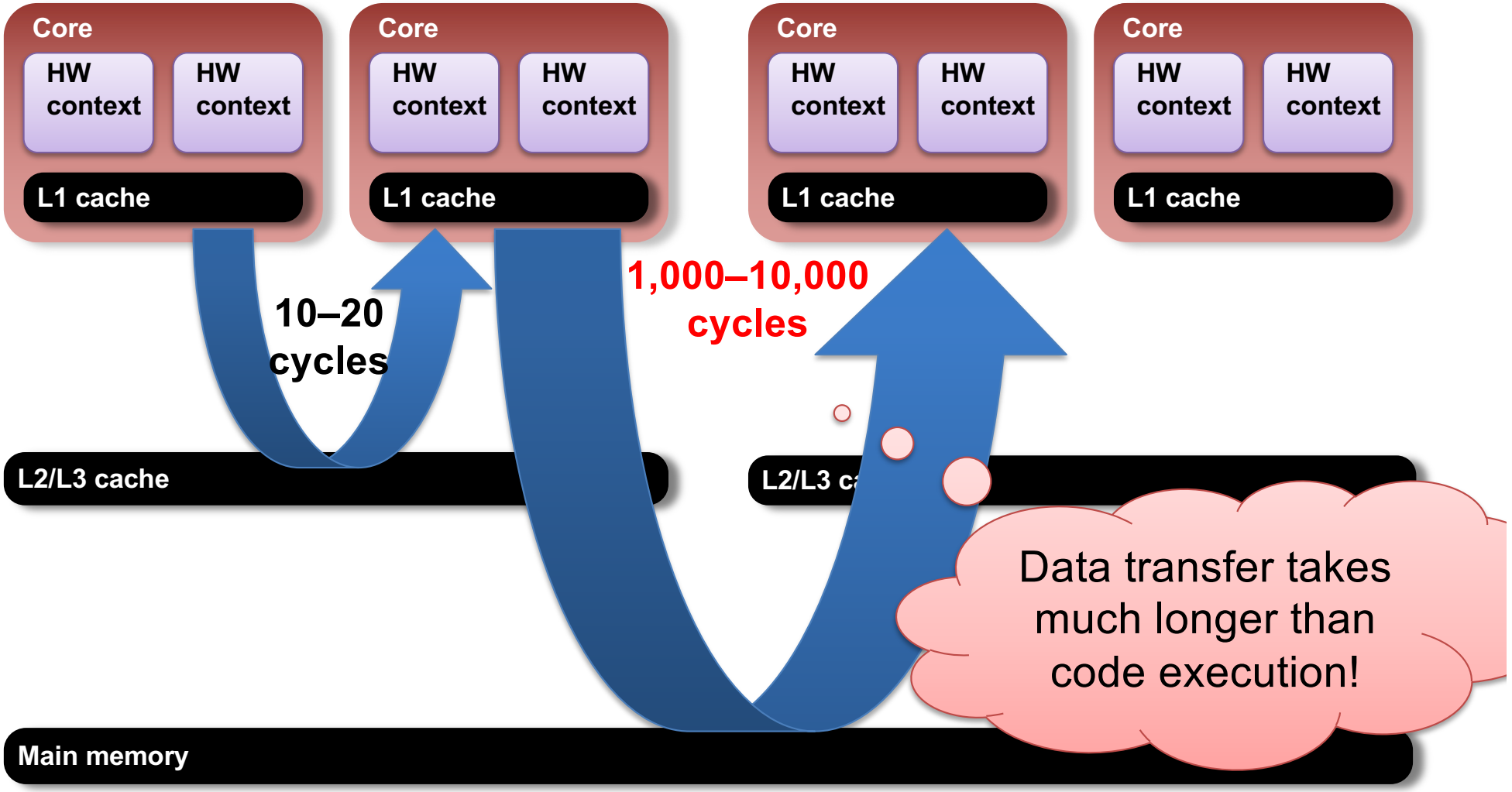


Multicore seL4

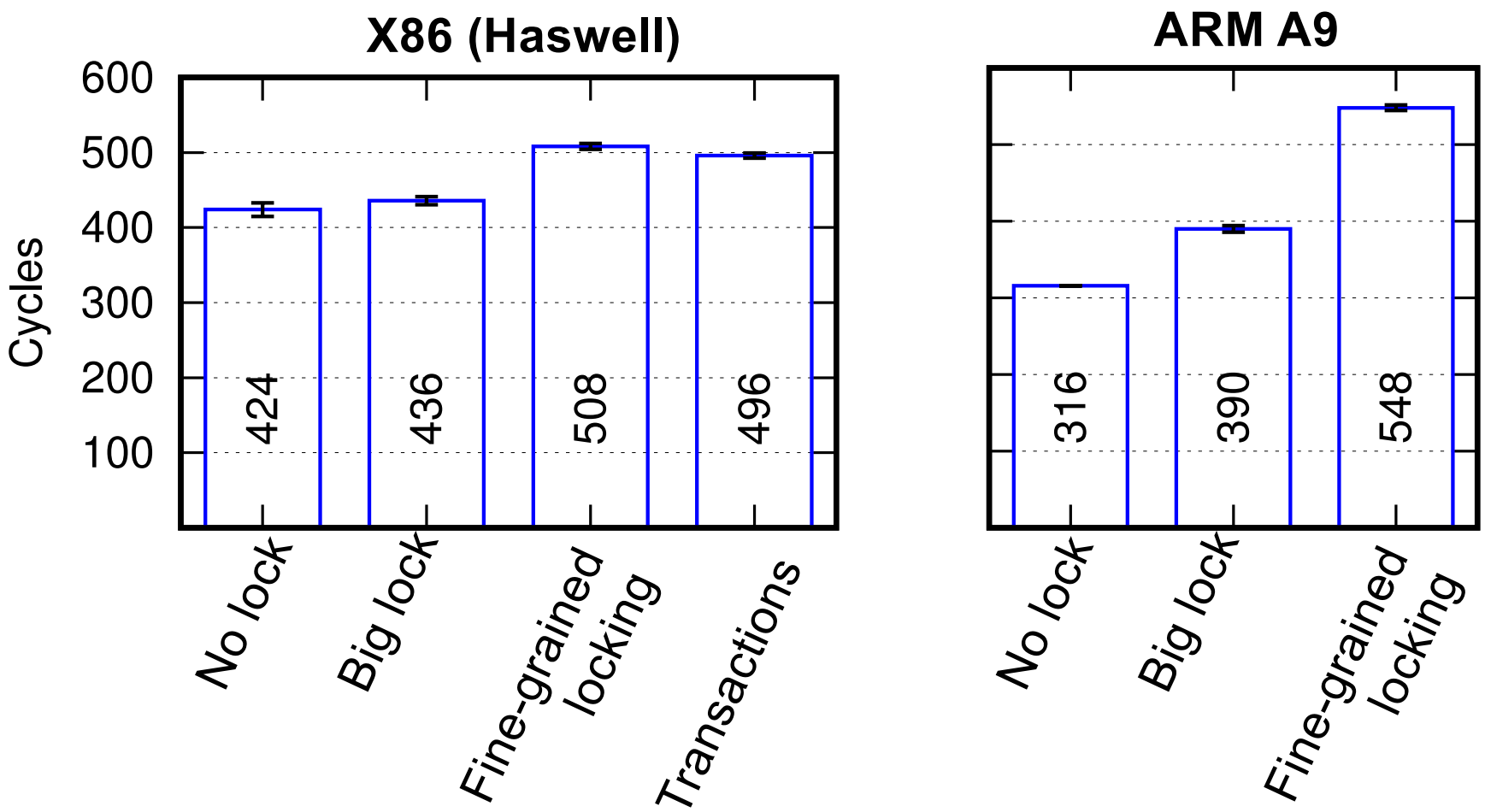
Microkernel vs Monolithic OS Execution



Cache Line Migration Latencies

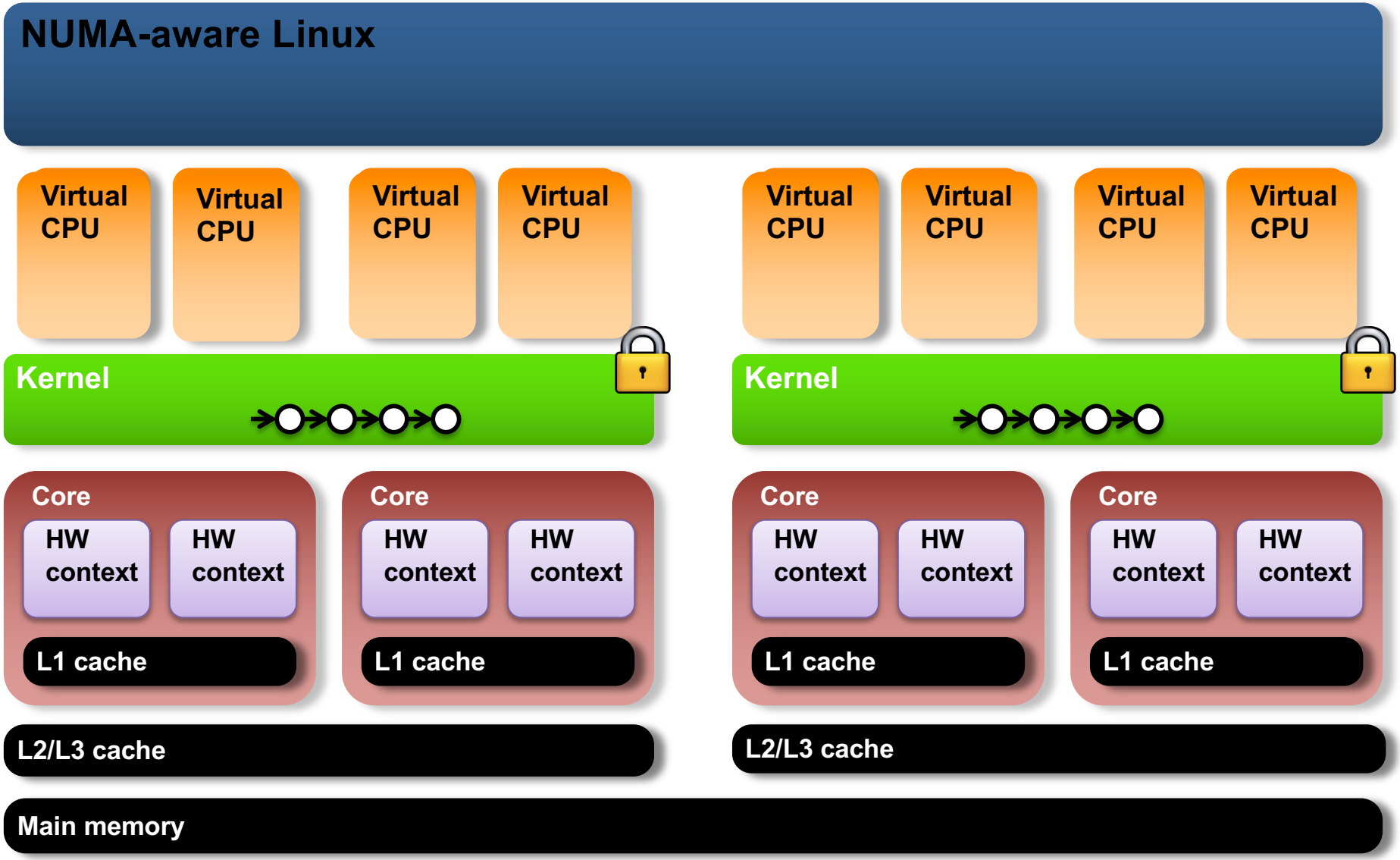


Cost of Locking

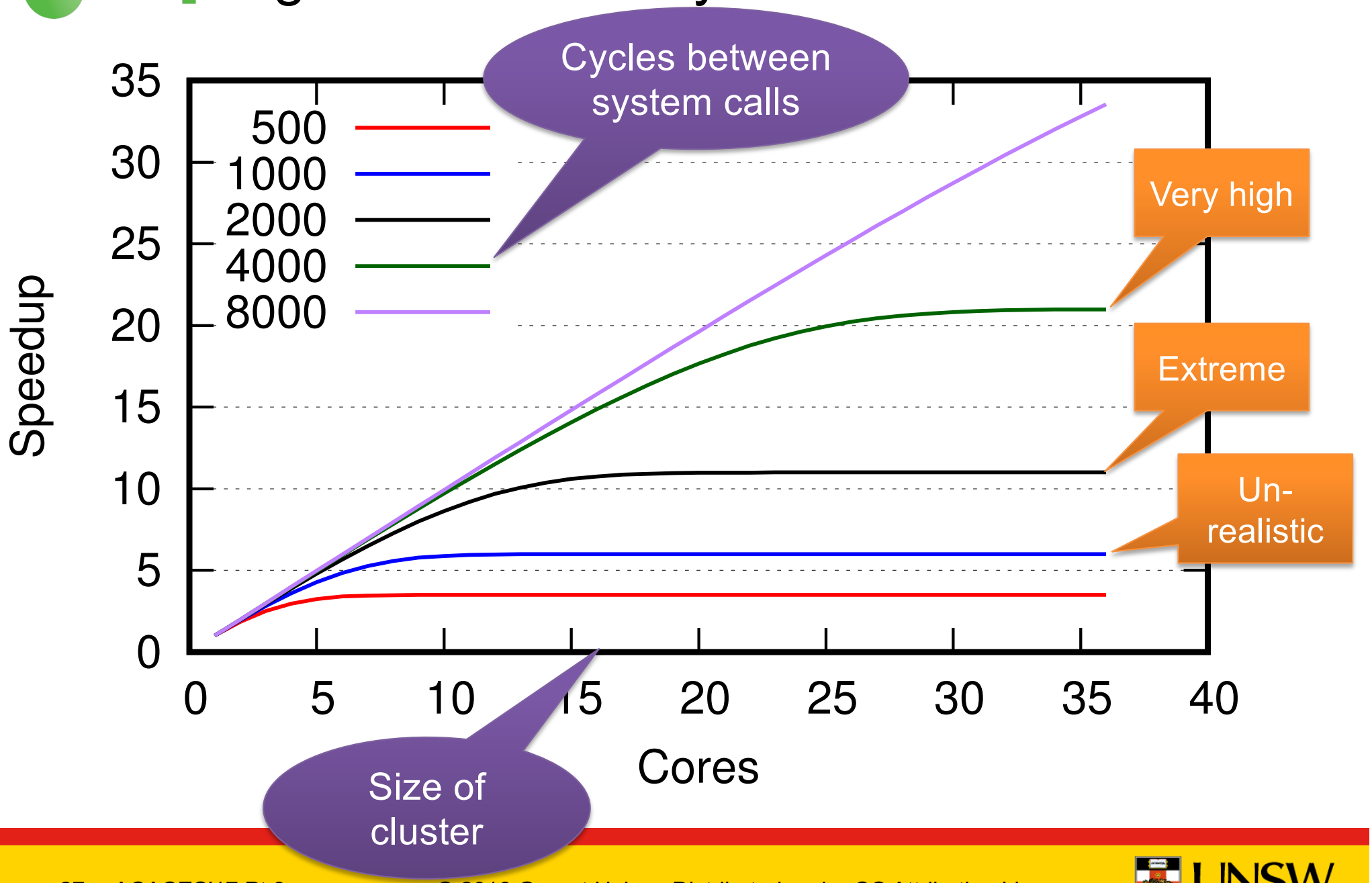


Locks have a cost –
significant in a fast microkernel!

Multicore Design: Clustered Multikernel



Big-Lock Scalability



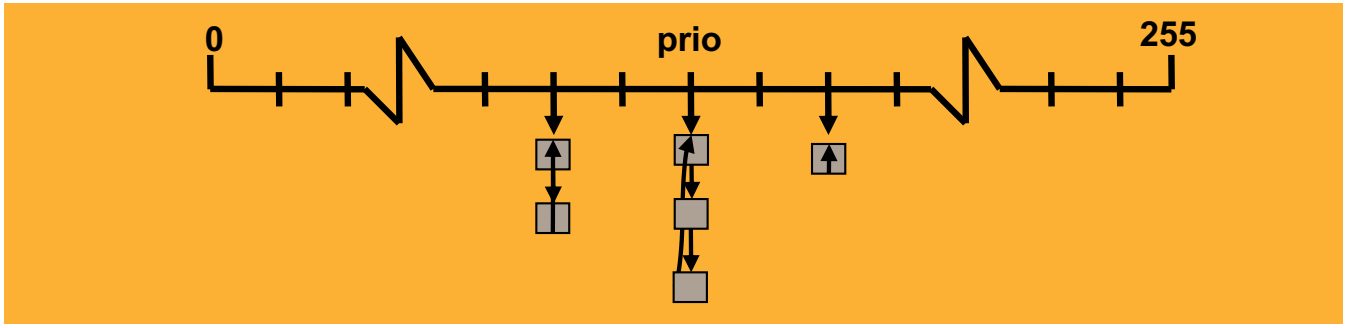
Mixed Criticality: Temporal Integrity

Classical L4 Scheduling

- 256 hard priorities (0–255)
  - Priorities are strictly observed
  - The scheduler will always pick the highest-prio runnable thread
  - Round-robin scheduling within prio level
- Thread scheduling parameters:
  - Priority
  - Time slice

Issue:

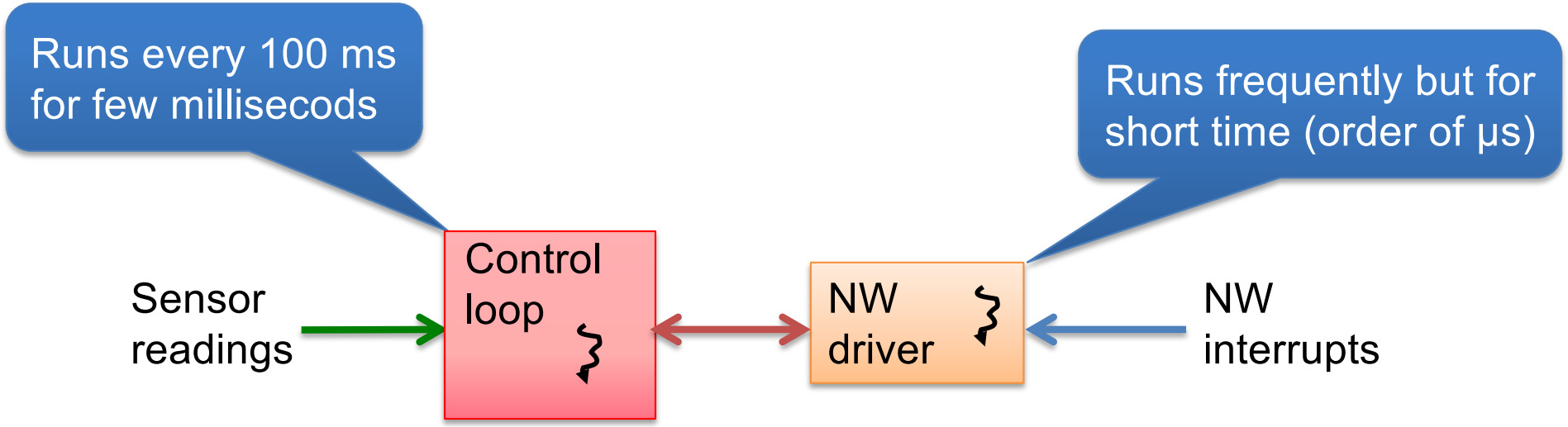
- highest-prio can monopolise CPU
- Priority = “importance”



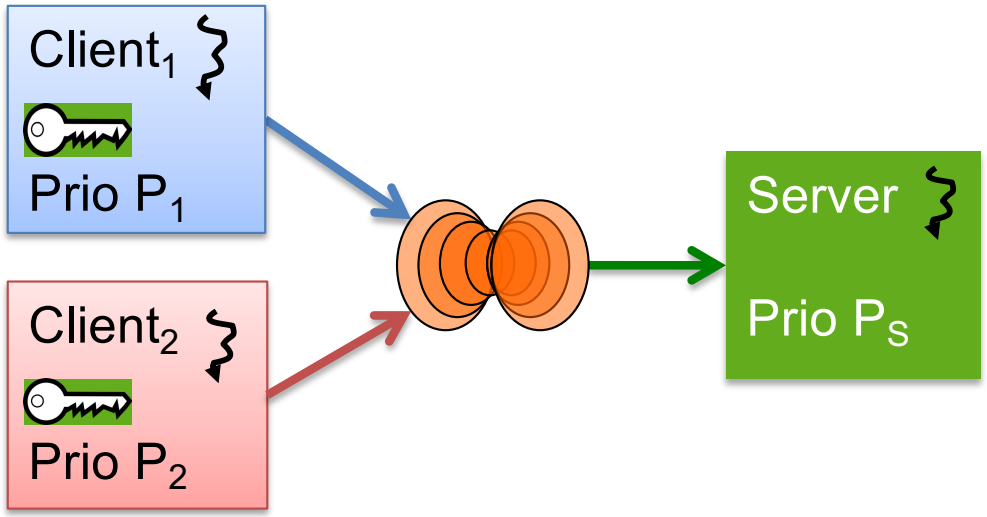
Issue with Priority = Importance

NW driver must preempt control loop

- … to avoid packet loss
- Driver must run at high prio
- Driver must be trusted not to monopolise CPU

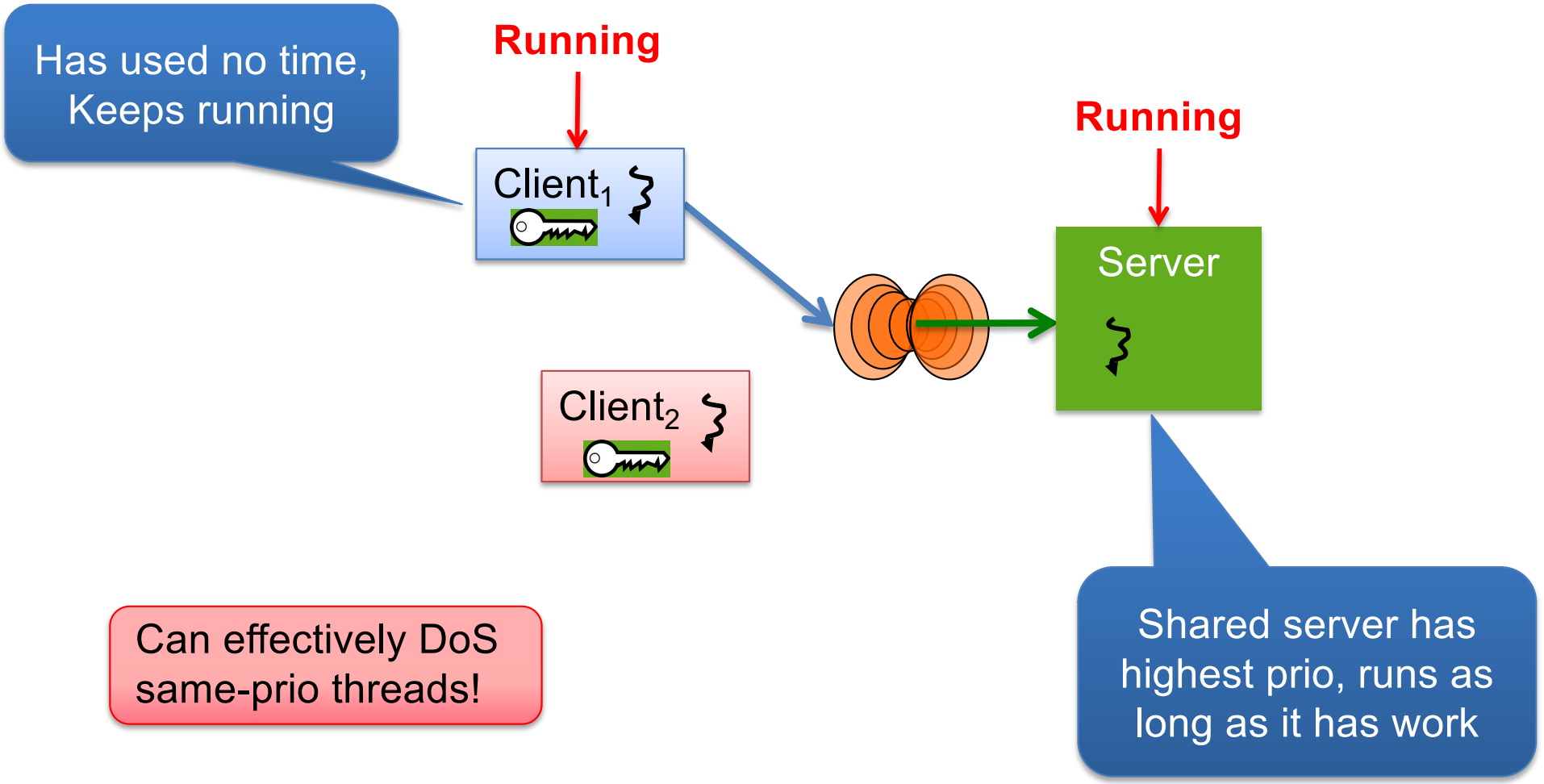


Shared Intra-Core Servers



$P_S > \max(P_1, P_2)$

Problem With Shared Servers



Separate Scheduling & Threads

Classical thread attributes

- Priority
- Time slice

Not runnable if null

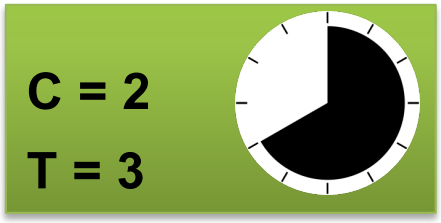
New thread attributes

- Priority
- Scheduling context capability

Limits CPU access!

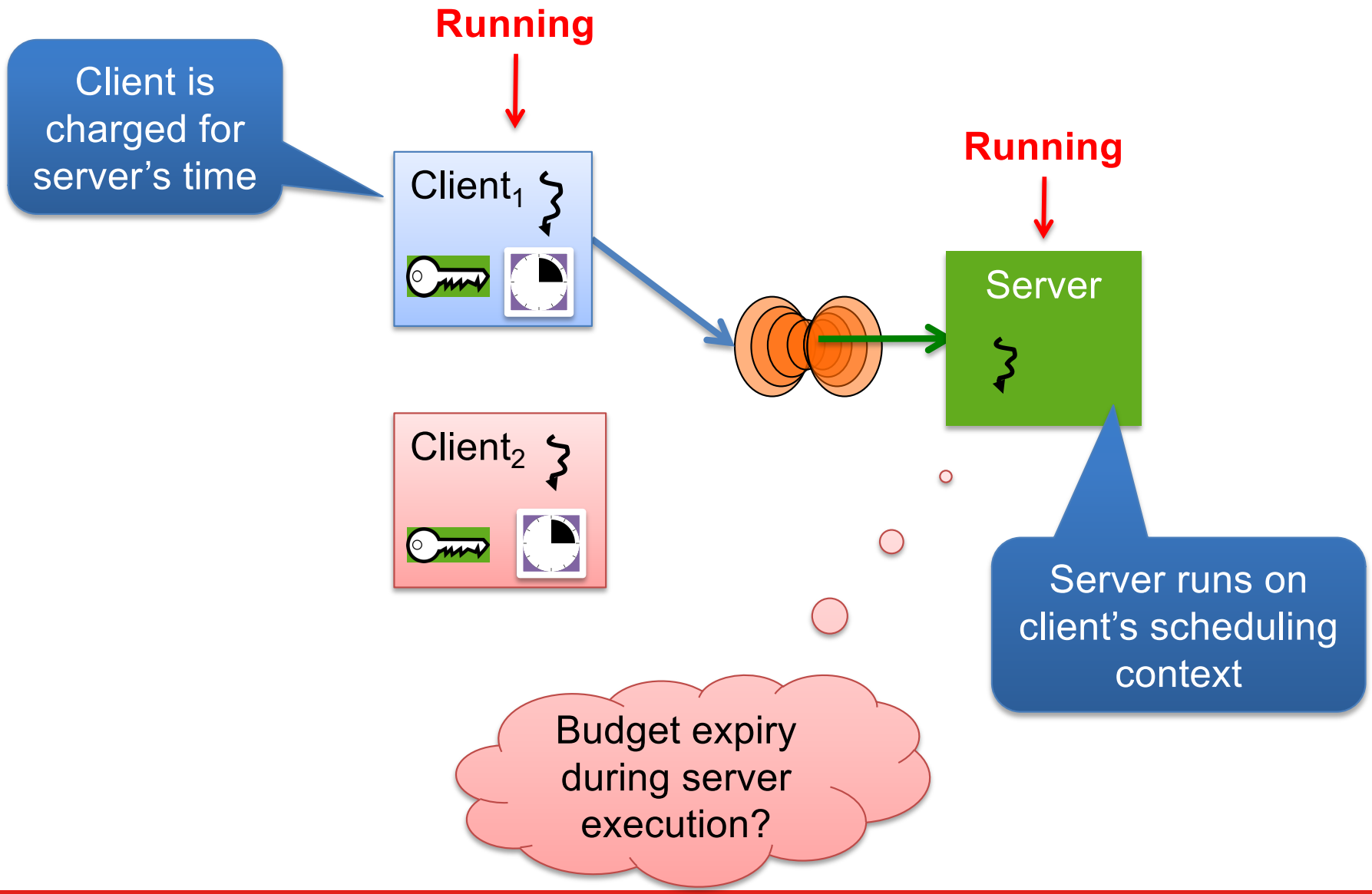
Scheduling context object
- T: period
- C: budget (≤ T)

High-prio thread cannot monopolise



`SchedControl` capability conveys right to assign budgets (i.e. perform admission control)

Shared Server w. Scheduling Contexts



Budget Expiry Options

- Multi-threaded servers (COMPOSITE [Parmer '10])
  - Model allows this
  - Forcing all servers to be thread-safe is policy 😢
- Bandwidth inheritance with "helping" (Fiasco [Steinberg '10])
  - Ugly dependency chains 😢
  - Wrong thread charged for recovery cost 😢
- Use *timeout exceptions* to trigger one of several possible actions:
  - Provide emergency budget
  - Cancel operation & roll-back server
  - Change criticality
  - Implement priority inheritance (if you must…)

Operating Systems

For Secure and Safe Embedded Systems

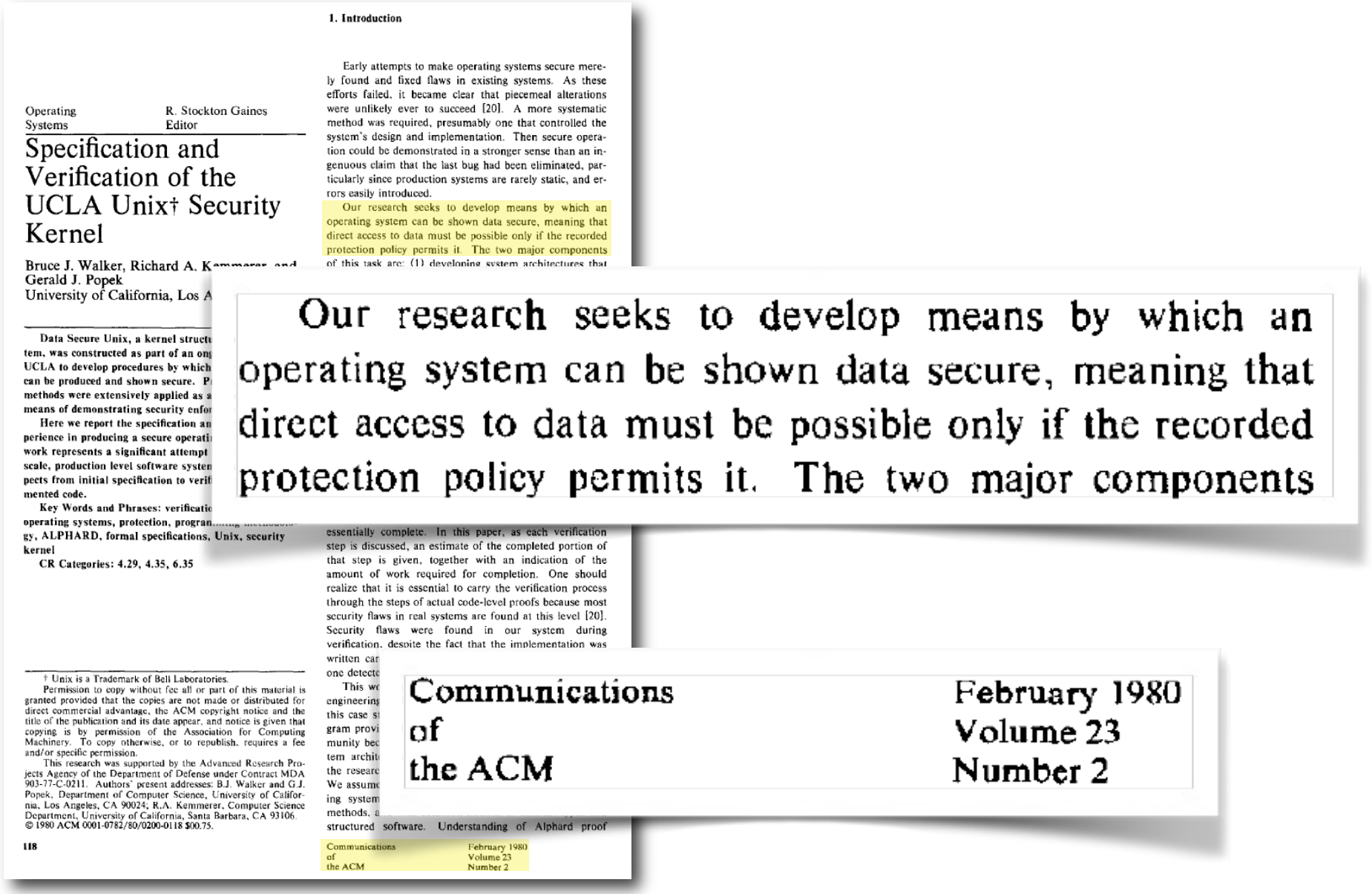
Part 4: Formal Verification

@GernotHeiser

Copyright Notice

Proving Security

A 30-Year Dream



Operating Systems — R. Stockton Gaines, Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los A

Data Secure Unix, a kernel structu
tem, was constructed as part of an ong
UCLA to develop procedures by which
can be produced and shown secure. P
methods were extensively applied as a
means of demonstrating security enfor
Here we report the specification an
perience in producing a secure operati
work represents a significant attempt
scale, production level software systen
pects from initial specification to verif
mented code.

Key Words and Phrases: verificatio
operating systems, protection, progran
gy, ALPHARD, formal specifications, Unix, security kernel
CR Categories: 4.29, 4.35, 6.35

† Unix is a Trademark of Bell Laboratories.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. Authors' present addresses: B.J. Walker and G.J. Popek, Department of Computer Science, University of California, Los Angeles, CA 90024; R.A. Kemmerer, Computer Science Department, University of California, Santa Barbara, CA 93106.

1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous claim that the last bug had been eliminated, particularly since production systems are rarely static, and errors easily introduced.

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components of this task are: (1) developing system architectures that

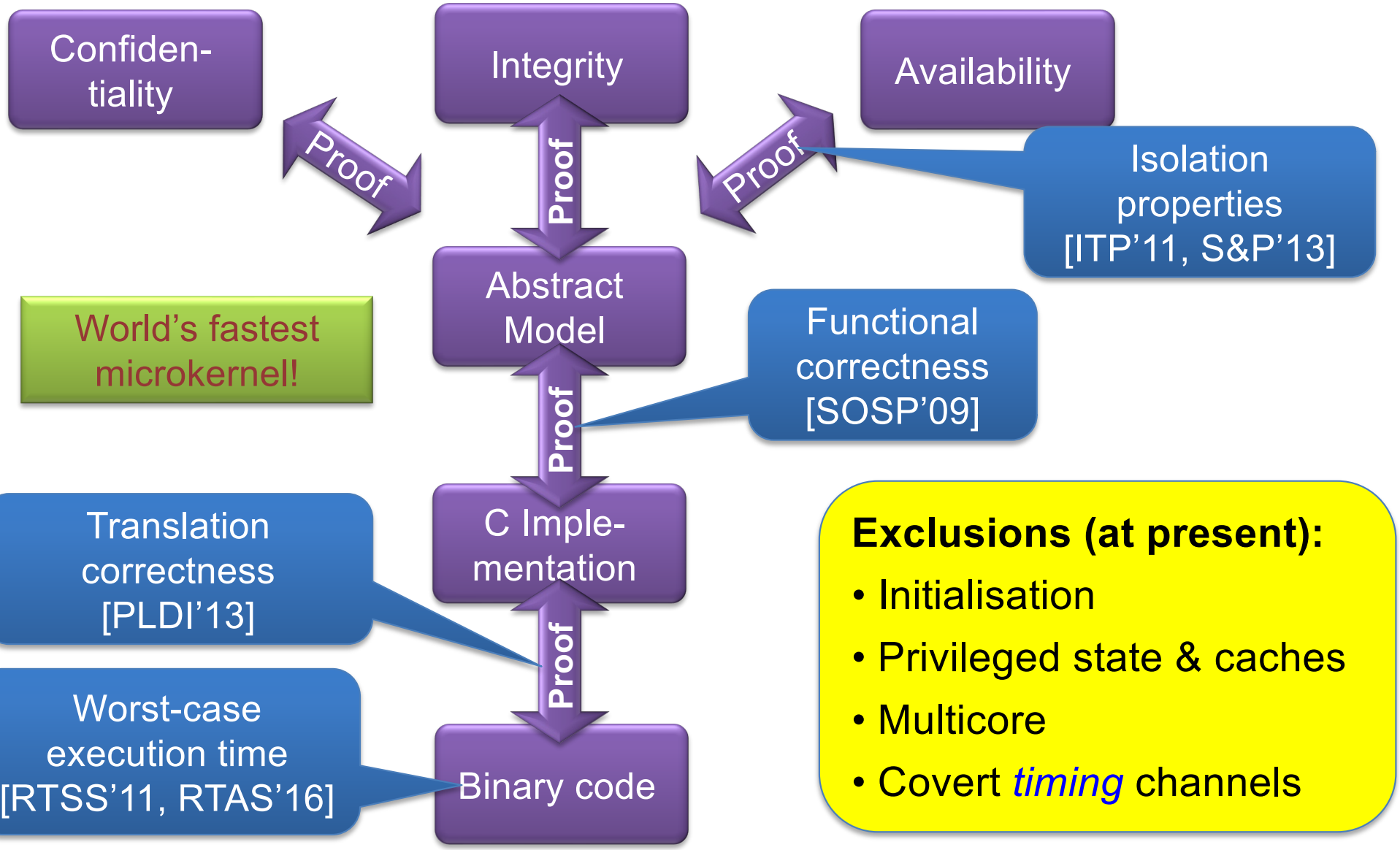
essentially complete. In this paper, as each verification step is discussed, an estimate of the completed portion of that step is given, together with an indication of the amount of work required for completion. One should realize that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level [20]. Security flaws were found in our system during verification, despite the fact that the implementation was written car
one detecte

This w
engineering
this case s
gram provi
munity bec
tem archit
the researc
We assume
ing system
methods, a
structured software. Understanding of Alphard proof

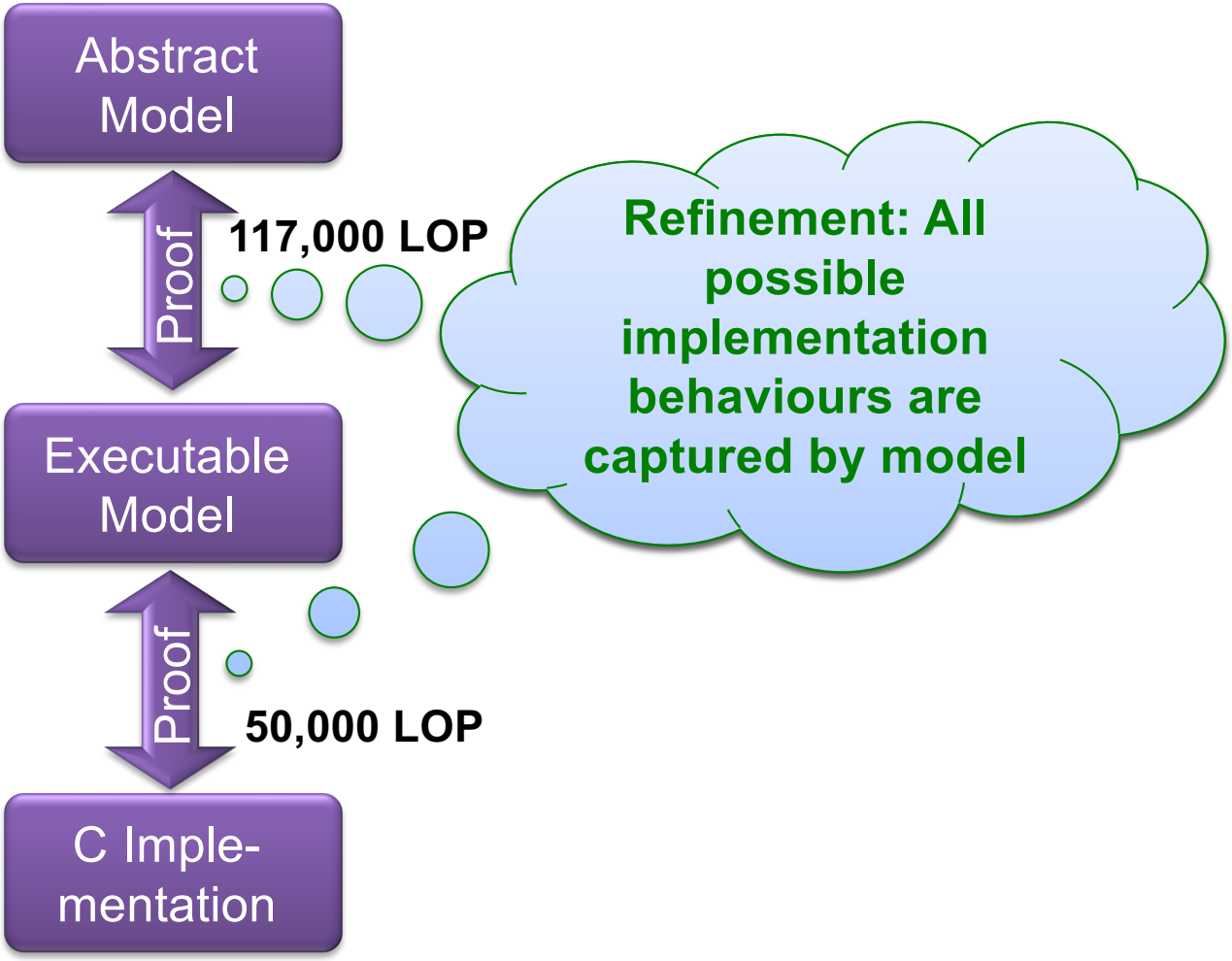
Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

Communications of the ACM — February 1980 Volume 23 Number 2

Provable Security Enforcement



Proving Functional Correctness



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
```

```
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;
```

MIT Technology Review

10 BREAKTHROUGH TECHNOLOGIES 2011

Crash-Proof Code

Making critical software safer

WILLIAM BULKELEY

May/June 2011

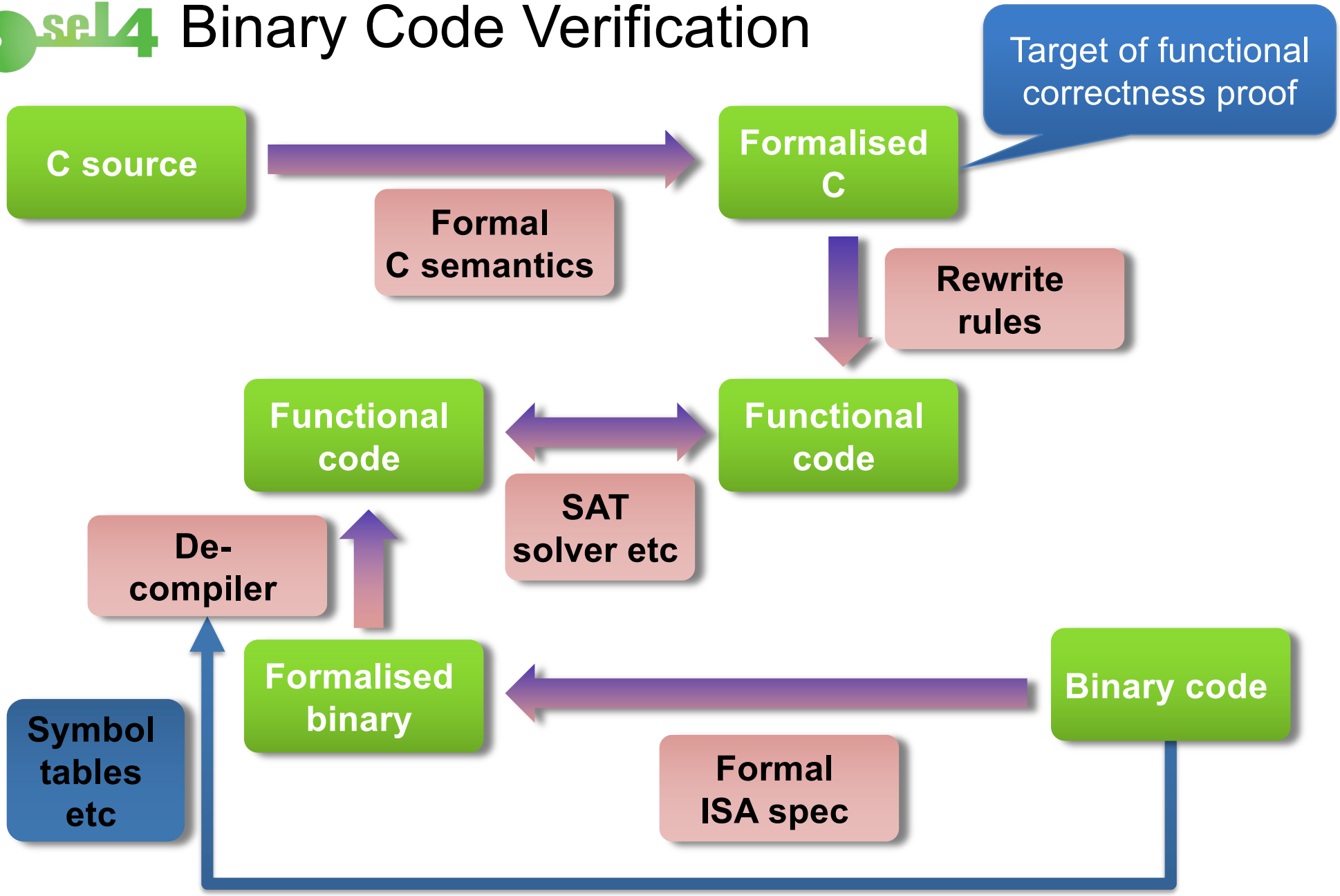
seL4 Formal Verification Summary

Can prove further properties on abstract level!

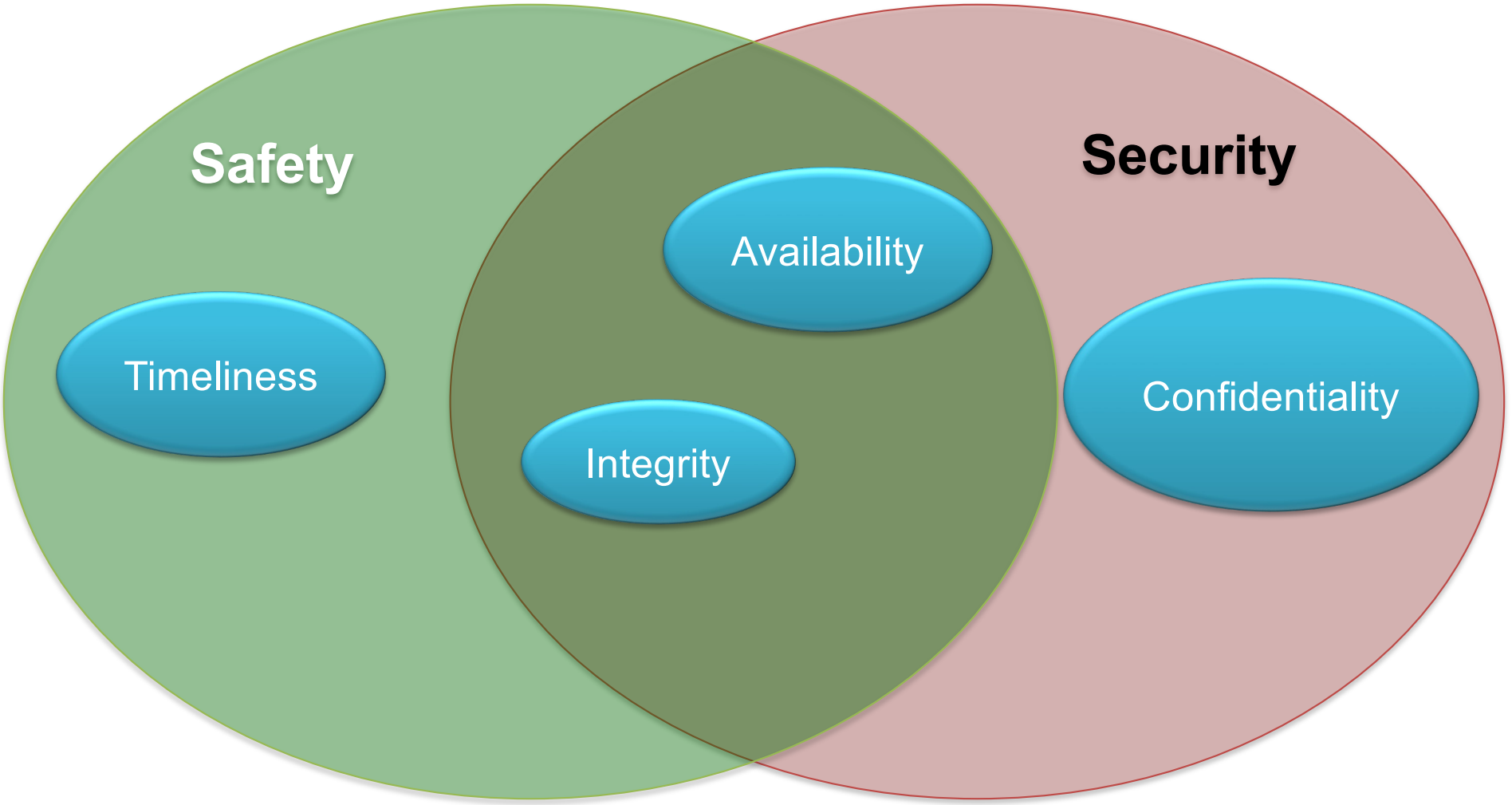
Kinds of properties proved

- 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
  - 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

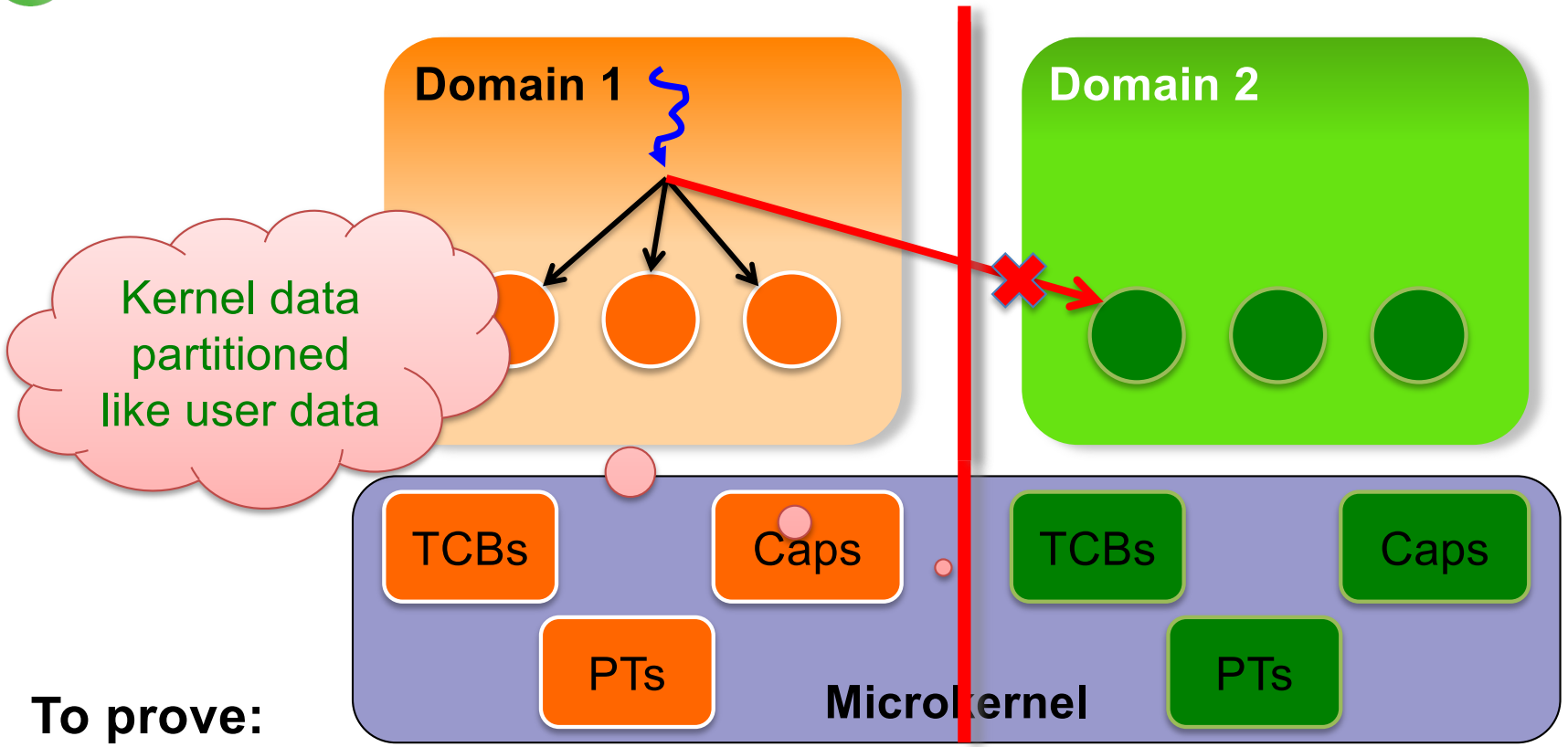
Binary Code Verification



Security vs Safety



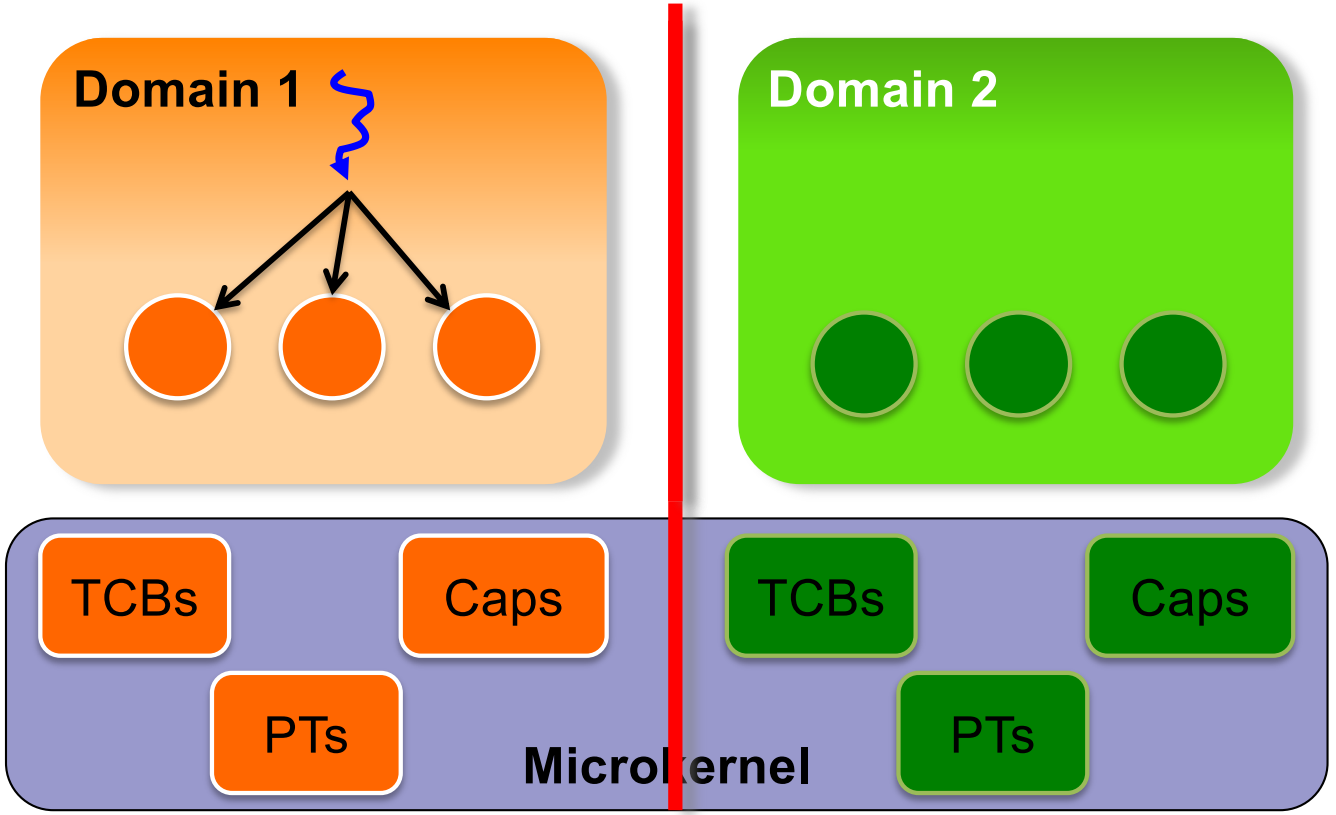
Integrity: Limiting Write Access



To prove:

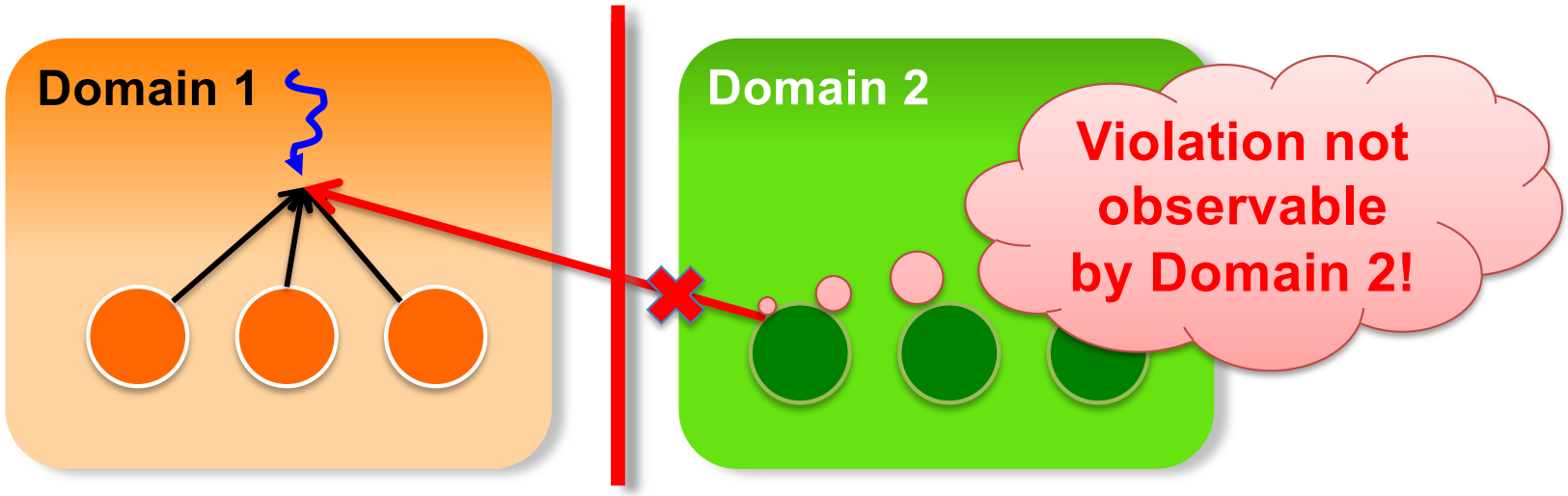
- Domain-1 doesn't have write *capabilities* to Domain-2 objects
  ⇒ no action of Domain-1 agents will modify Domain-2 state
- Specifically, *kernel does not modify on Domain-1's behalf!*
  - Event-based kernel operates on behalf of well-defined user thread
  - Prove kernel only allows write upon capability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

Availability: Ensuring Resource Access



- Strict separation of kernel resources
  ⇒ agent cannot deny access to another domain's resources
- Nothing to do: implied by other properties

Confidentiality: Limiting Read Accesses



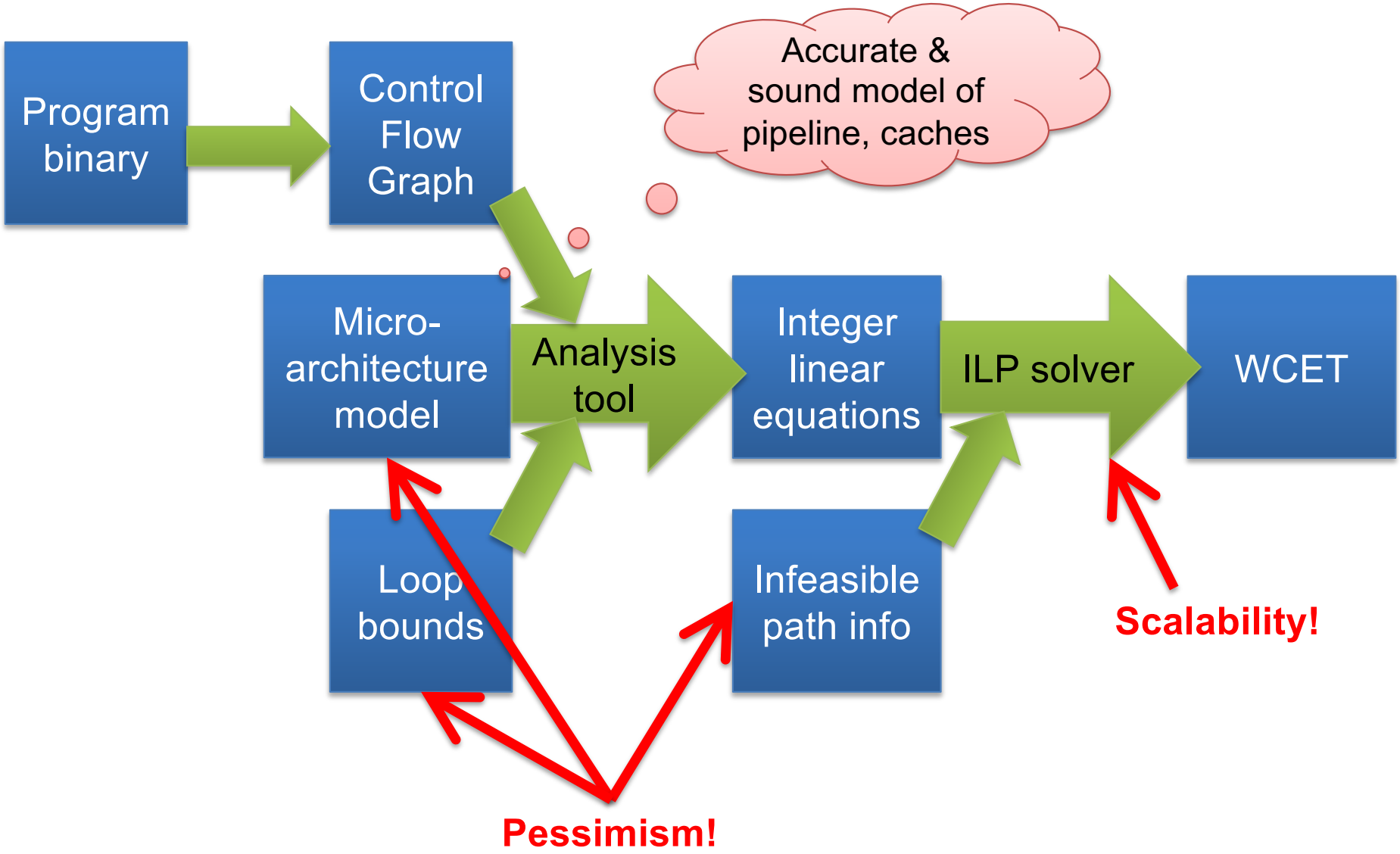
To prove:

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

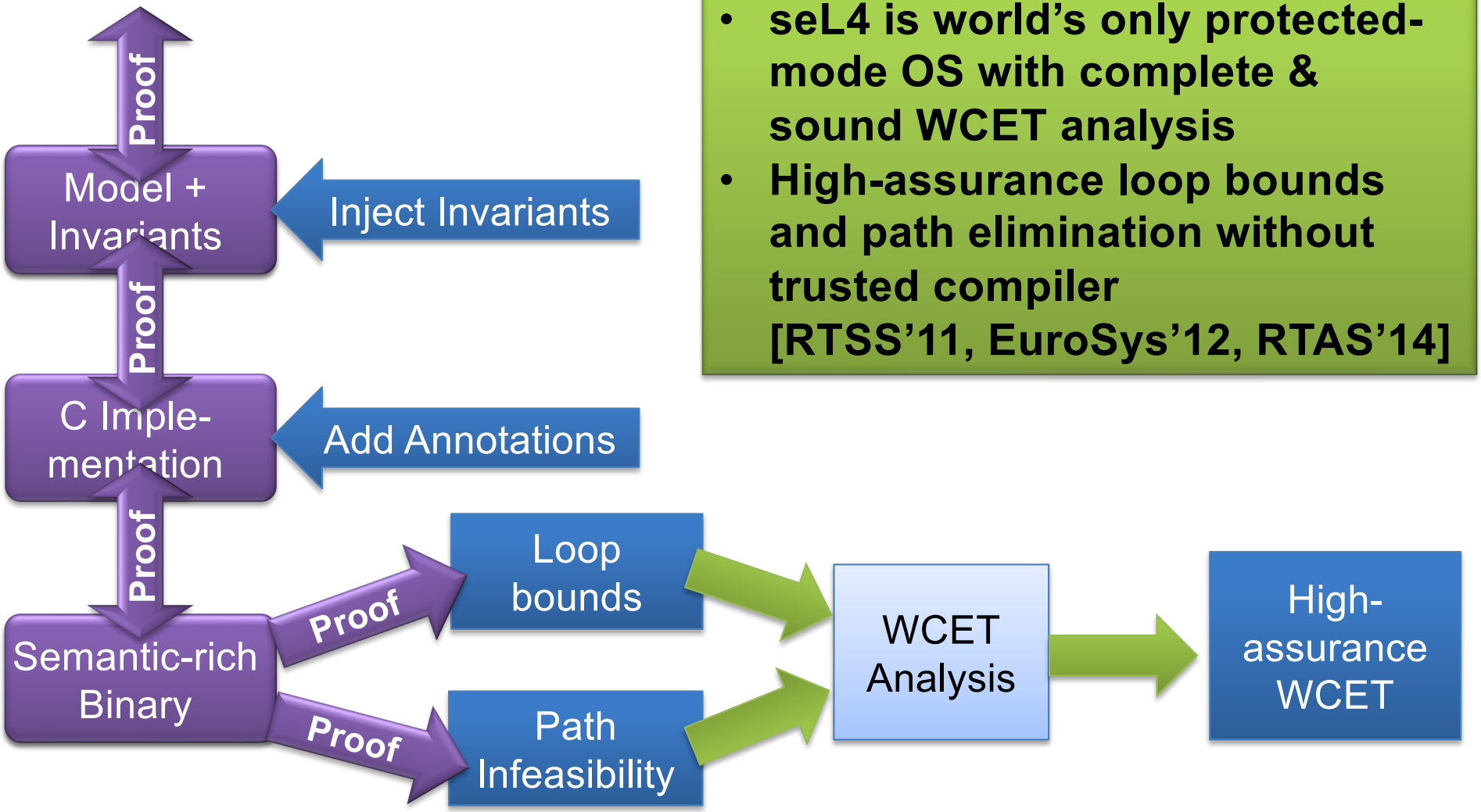
Non-interference proof :

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

Worst-Case Execution Time (WCET) Analysis



Proving Loop Bounds & Infeasible Paths



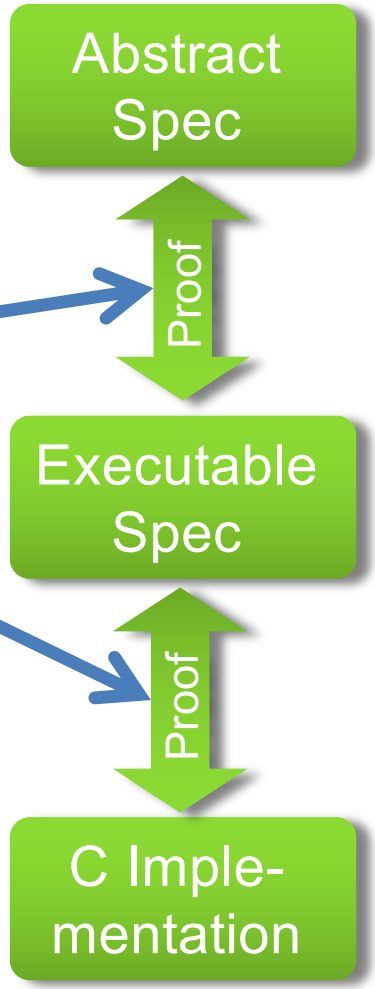
- **seL4 is world's only protected-mode OS with complete & sound WCET analysis**
- **High-assurance loop bounds and path elimination without trusted compiler [RTSS'11, EuroSys'12, RTAS'14]**

Verification Cost

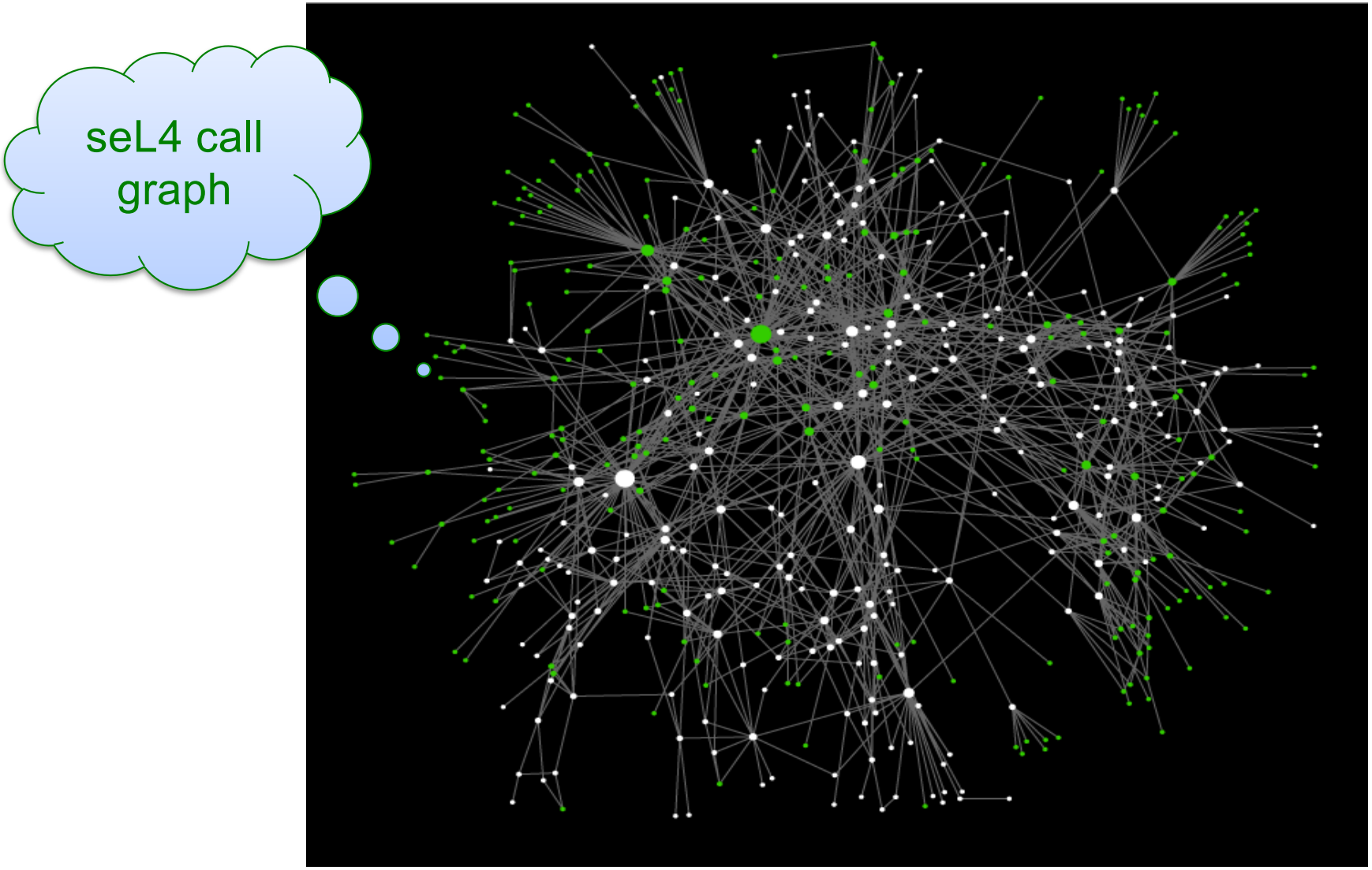
Verification Cost Breakdown

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	3–4 py

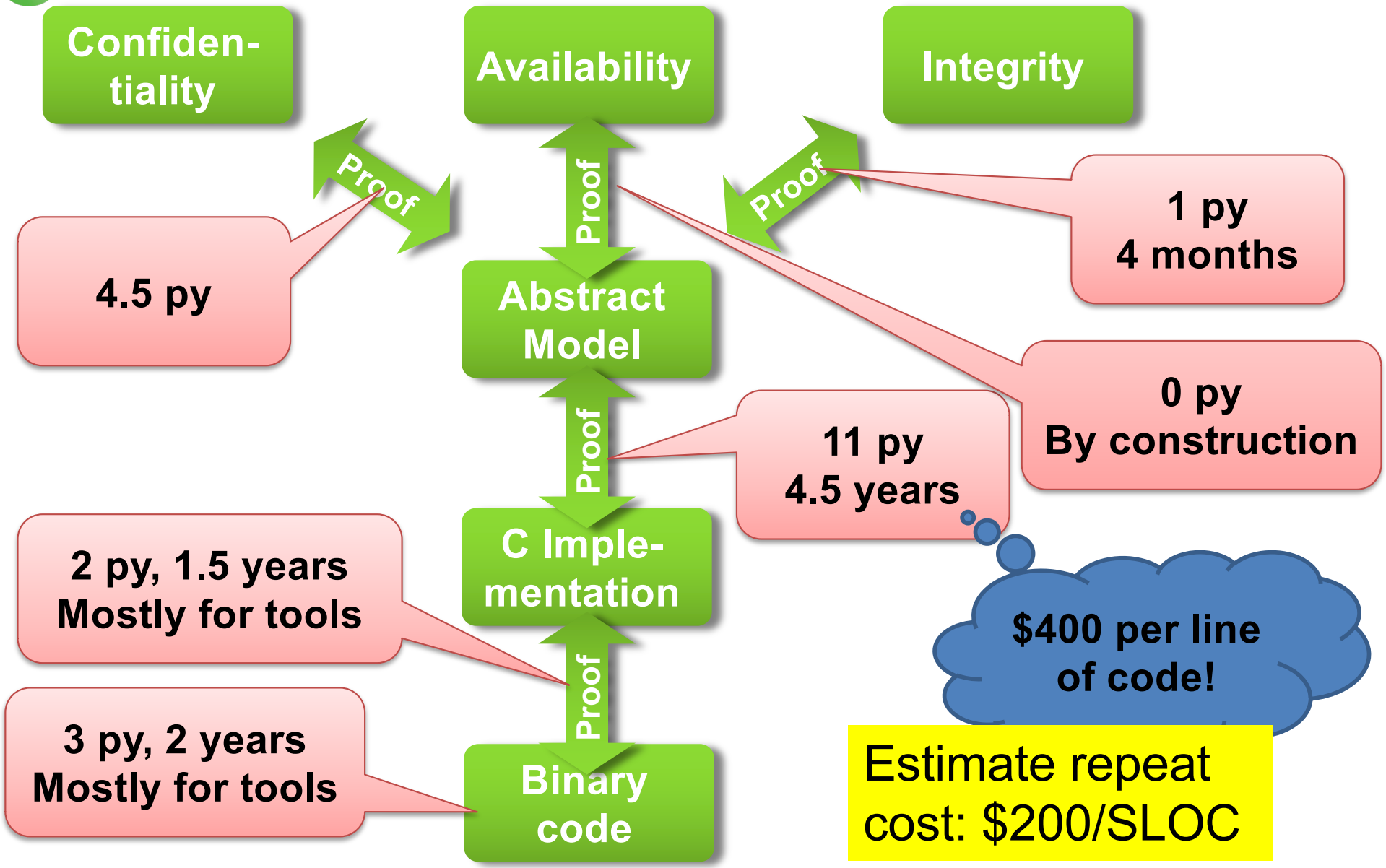
Reusable!



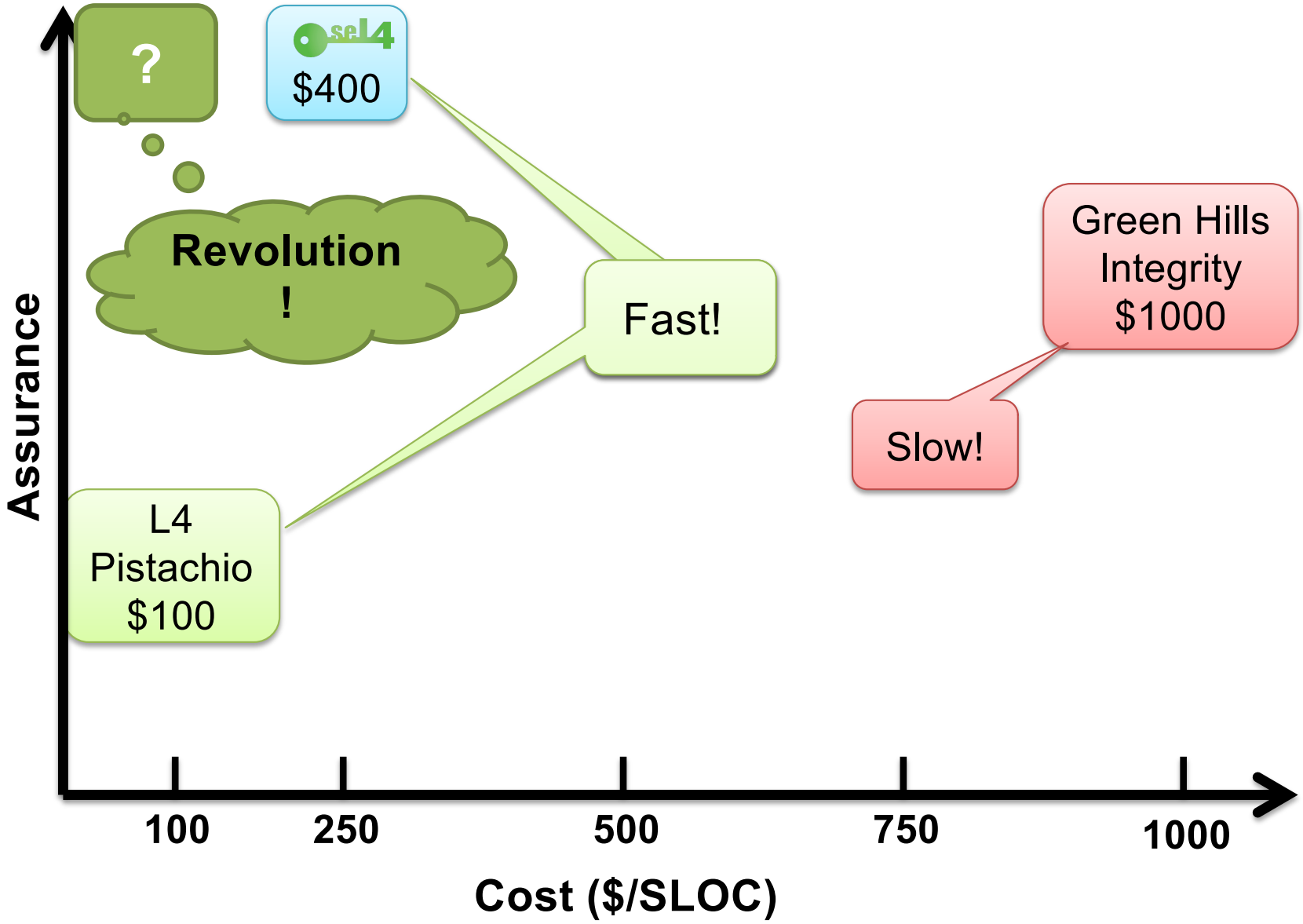
Why So Hard for 9,000 LOC?



Cost of Assurance



Microkernel Life-Cycle Cost in Context



Cost of Assurance

Industry Best Practice:

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

State of the Art – seL4:

  - $400/LOC, 0 faults/kSLOC, *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

Operating Systems

For Secure and Safe Embedded Systems

Part 5: Using seL4 for Trustworthy Systems

@GernotHeiser

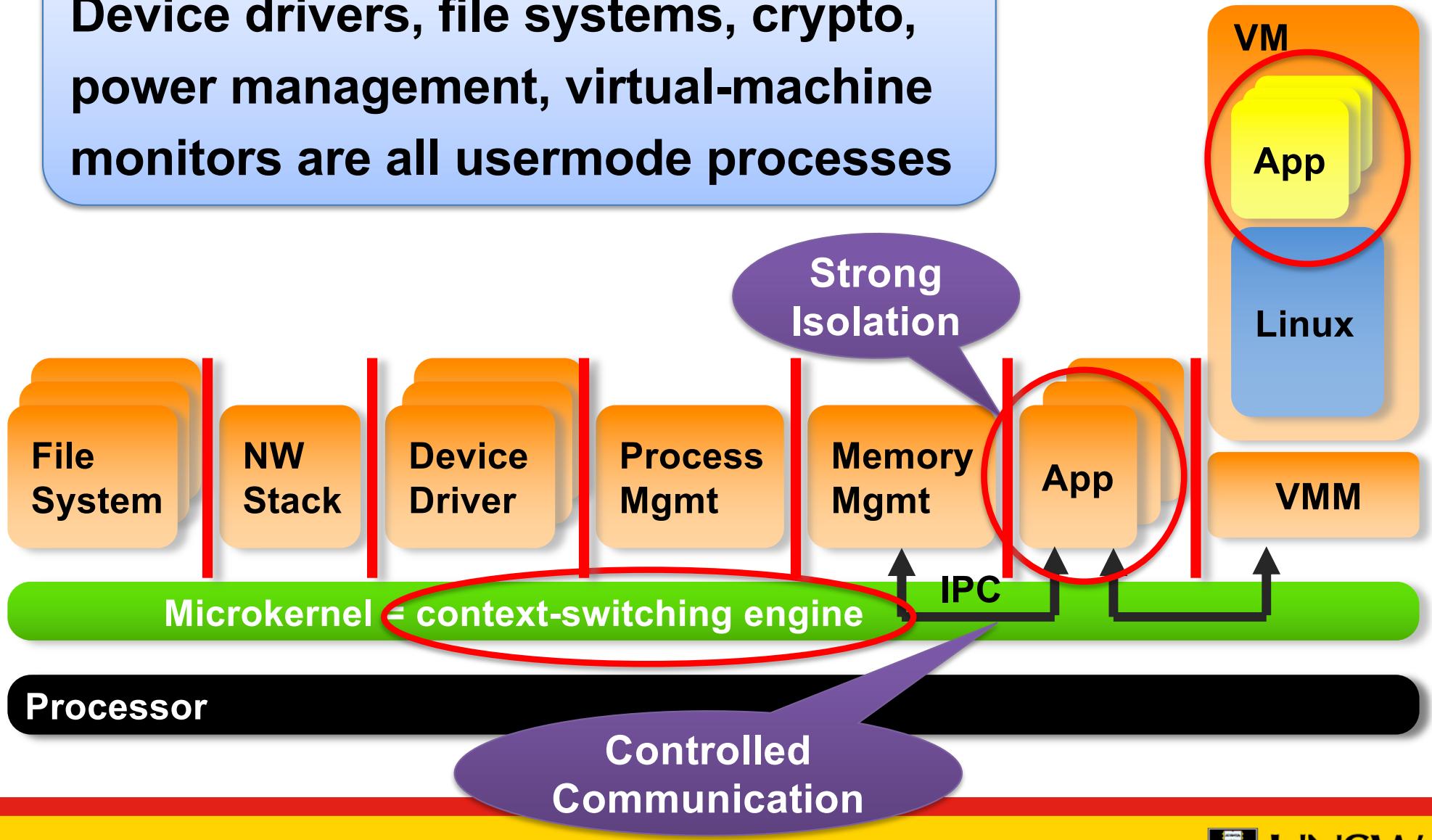
Never Stand Still | Engineering | Computer Science and Engineering

Copyright Notice

seL4 Concepts

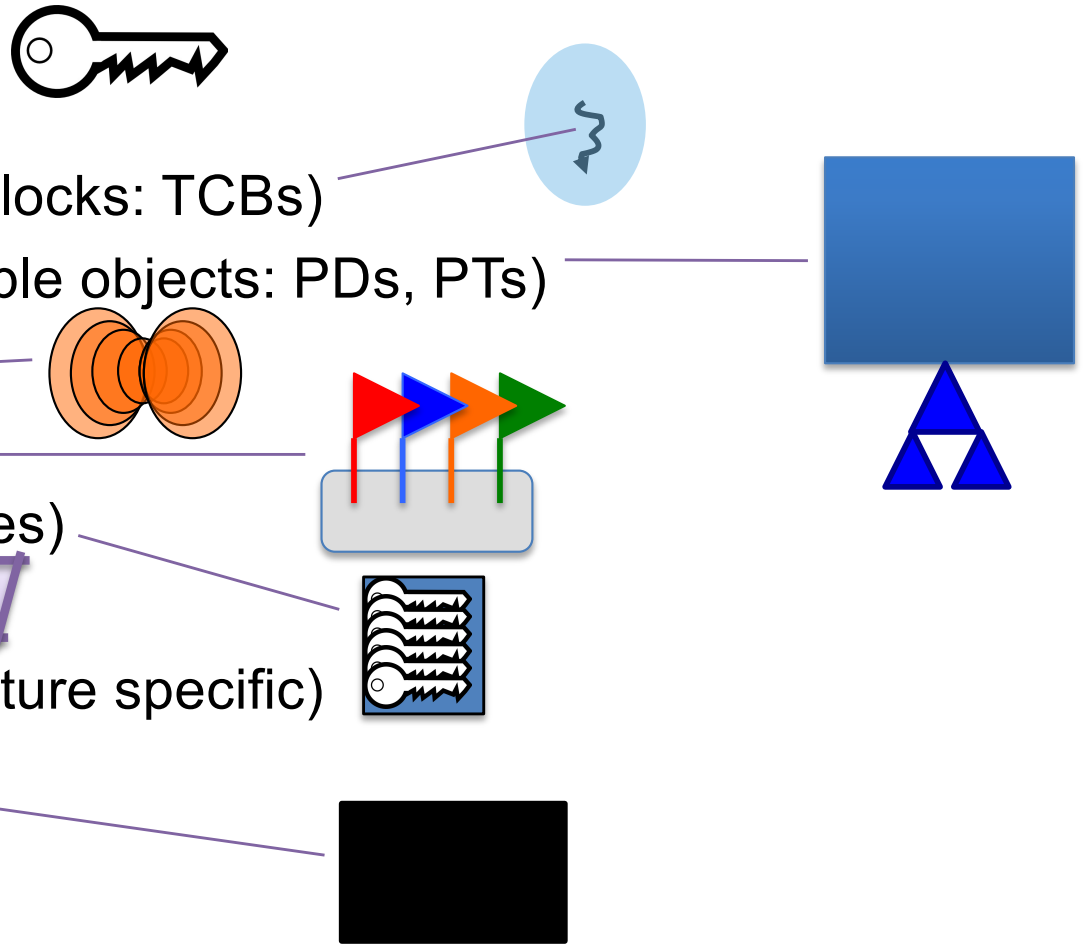
Remember: Microkernel ≠ Operating System

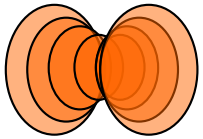
Device drivers, file systems, crypto, power management, virtual-machine monitors are all usermode processes



seL4 Concepts

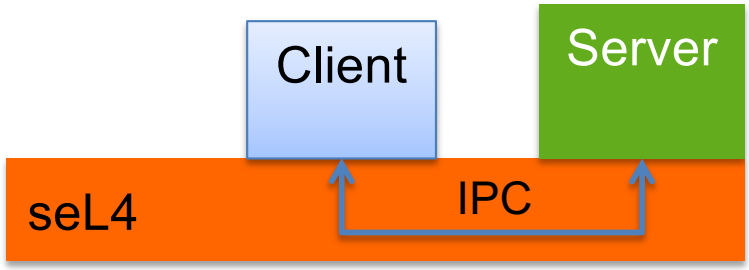
- Capabilities (Caps)
  - mediate access
- Kernel objects:
  - Threads (thread-control blocks: TCBs)
  - Address spaces (page table objects: PDs, PTs)
  - Endpoints (IPC)
  - Notifications
  - Capability spaces (CNodes)
  - Frames
  - Interrupt objects (architecture specific)
  - Untyped memory
- System calls
  - Send, Wait (and variants)
  - Yield



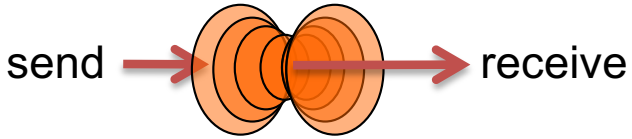


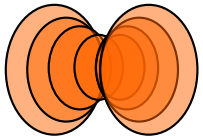
Inter-Process Communication (IPC)

- Fundamental microkernel operation
  - Kernel provides no services, only mechanisms
  - OS services provided by (protected) user-level server processes
  - invoked by IPC

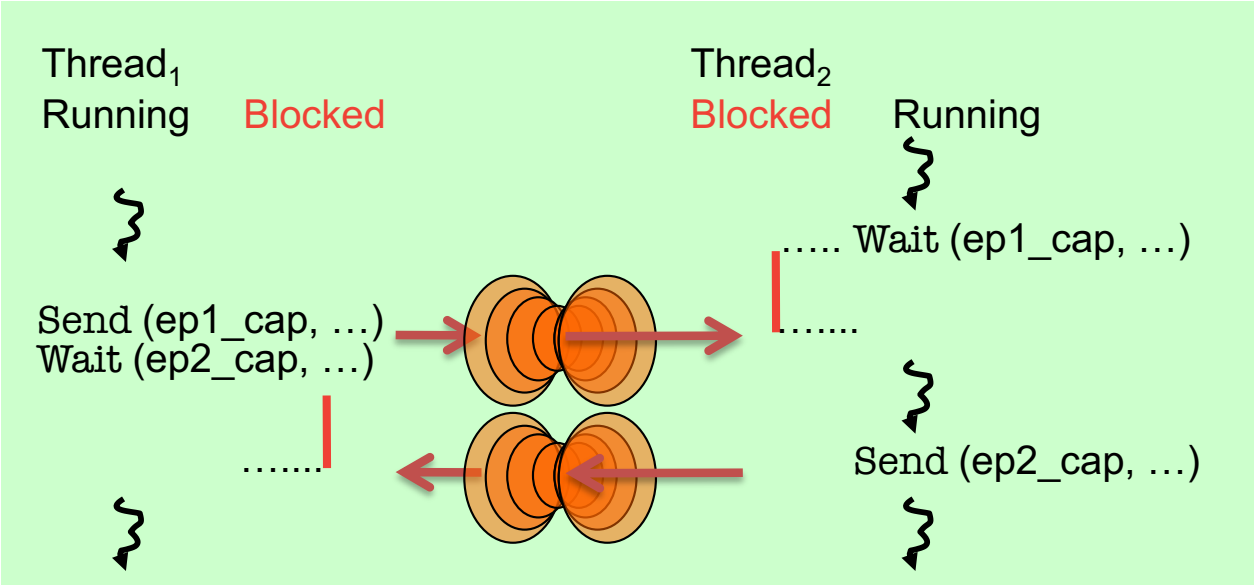


- seL4 IPC uses a handshake through *endpoints*:
  - Transfer points without storage capacity
  - Message must be transferred instantly
    - Single-copy user ➞ user by kernel

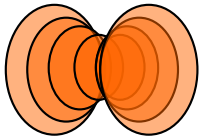




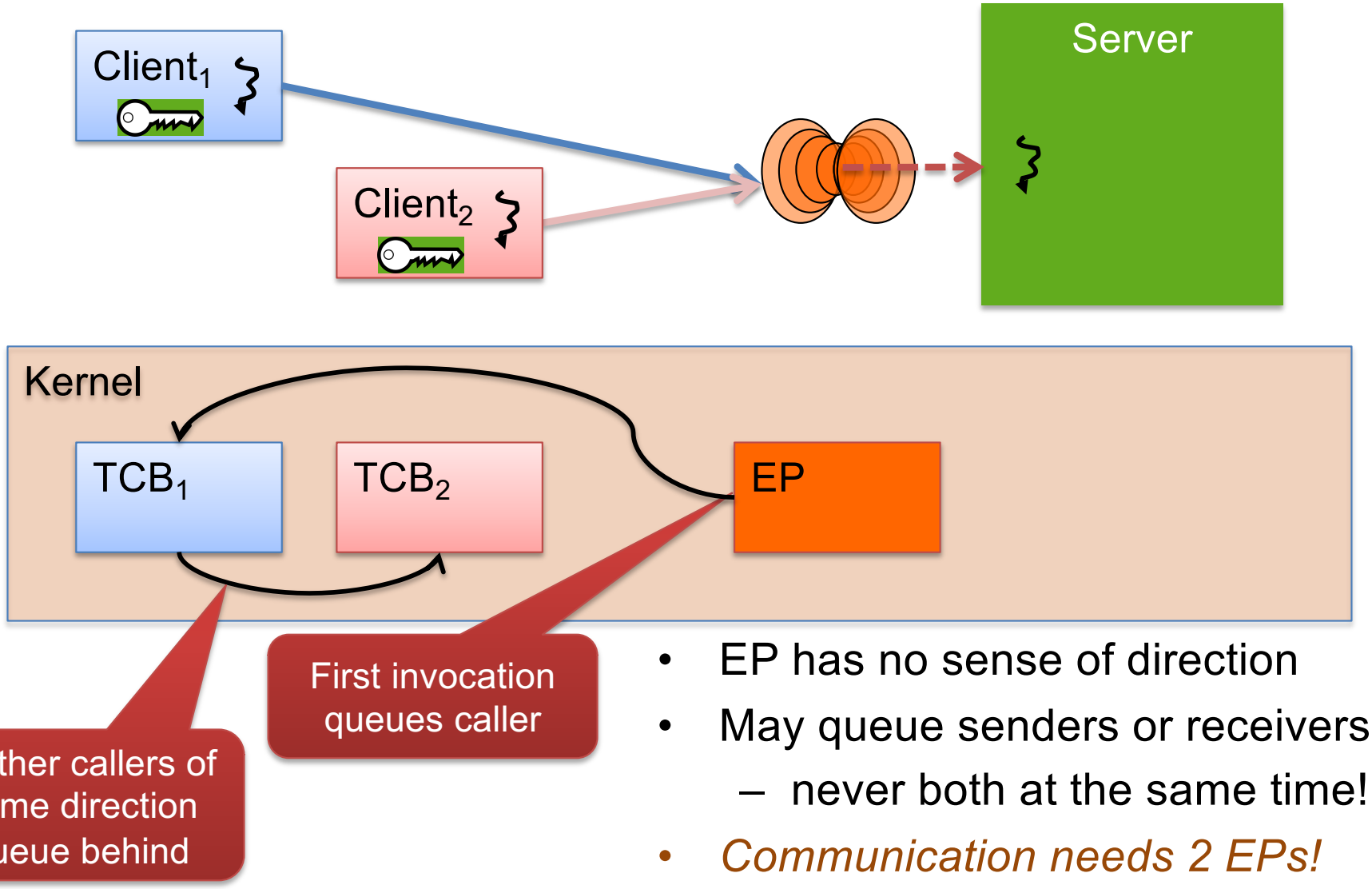
IPC: Endpoints



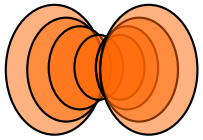
- Threads must rendez-vous for message transfer
  - One side blocks until the other is ready
  - Implicit synchronisation
- Message copied from sender's to receiver's *message registers*
  - Message is combination of caps and data words



IPC Endpoints are Message Queues

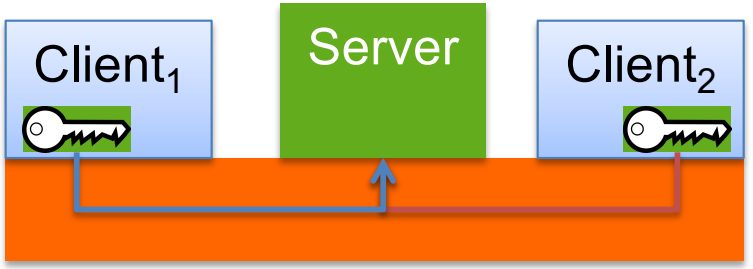


- EP has no sense of direction
- May queue senders or receivers
  - never both at the same time!
- *Communication needs 2 EPs!*



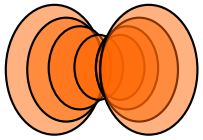
Client-Server Communication

- Asymmetric relationship:
  - Server widely accessible, clients not
  - How can server reply back to correct client?

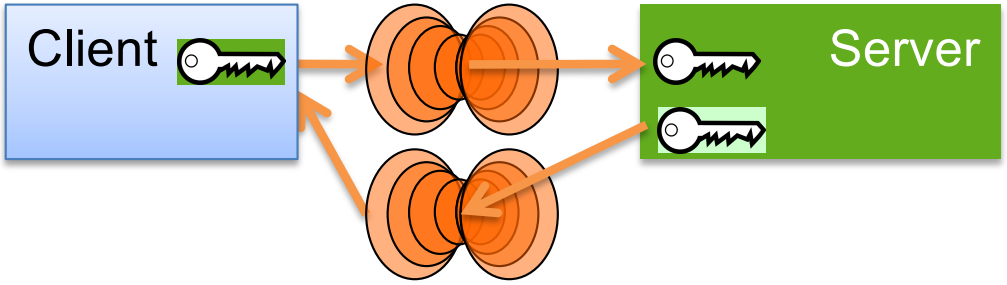


- Client can pass (session) reply cap in first request
  - Server needs to maintain session state
  - Forces stateful server design

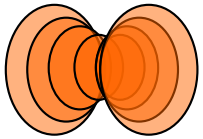
- seL4 solution: Kernel provides single-use *reply cap*
  - Only for `Call` operation (`Send+Wait`)
  - Allows server to reply to client
  - One-shot (automatically destroyed after first use)
  - Supports stateless servers



Call RPC Semantics



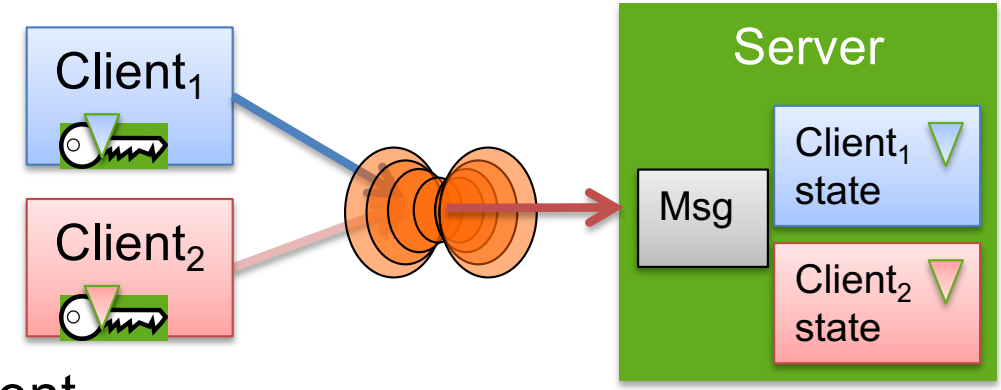
Client	**Kernel**	**Server**
		Wait(ep,&rep)
Call(ep,…)		
	mint rep	
	deliver to server	
		process
		Send(rep,…)
	deliver to client	
	destroy rep	
process		*process*



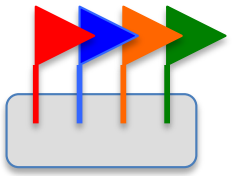
Stateful Servers: Identifying Clients

Stateful server serving multiple clients

- Must respond to correct client
  - Ensured by reply cap
- Must associate request with correct state



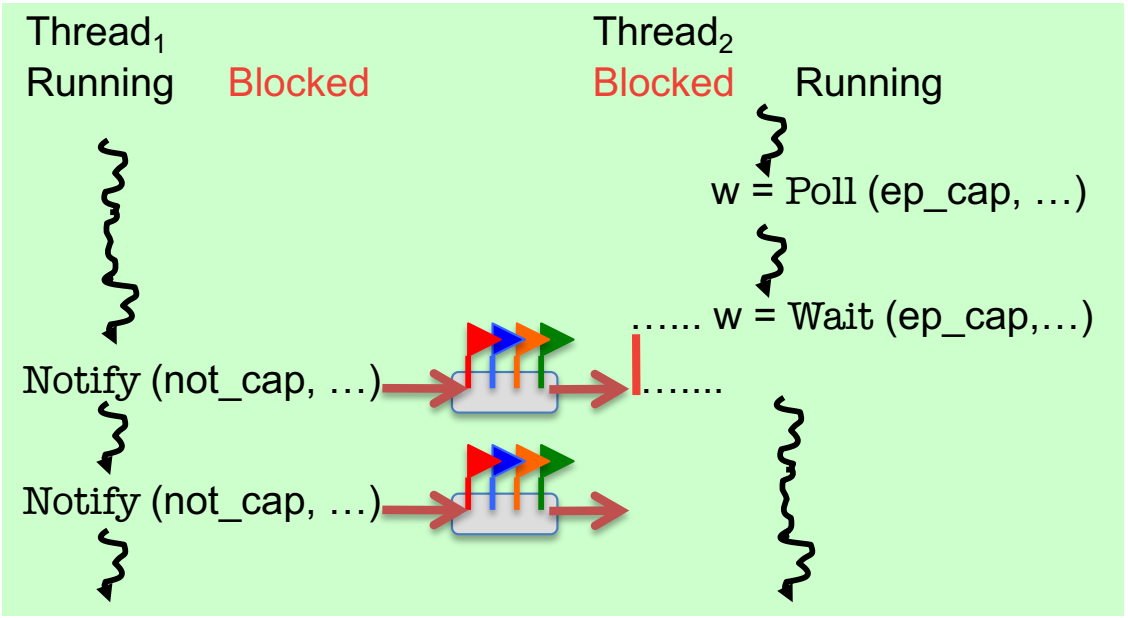
- Could use separate EP per client
  - endpoints are lightweight (16 B)
  - but requires mechanism to wait on a set of EPs (like select)
- Instead, seL4 allows to individually mark ("badge") caps to same EP
  - server provides individually badged caps to clients
  - server tags client state with badge
  - kernel delivers badge to receiver on invocation of badged caps



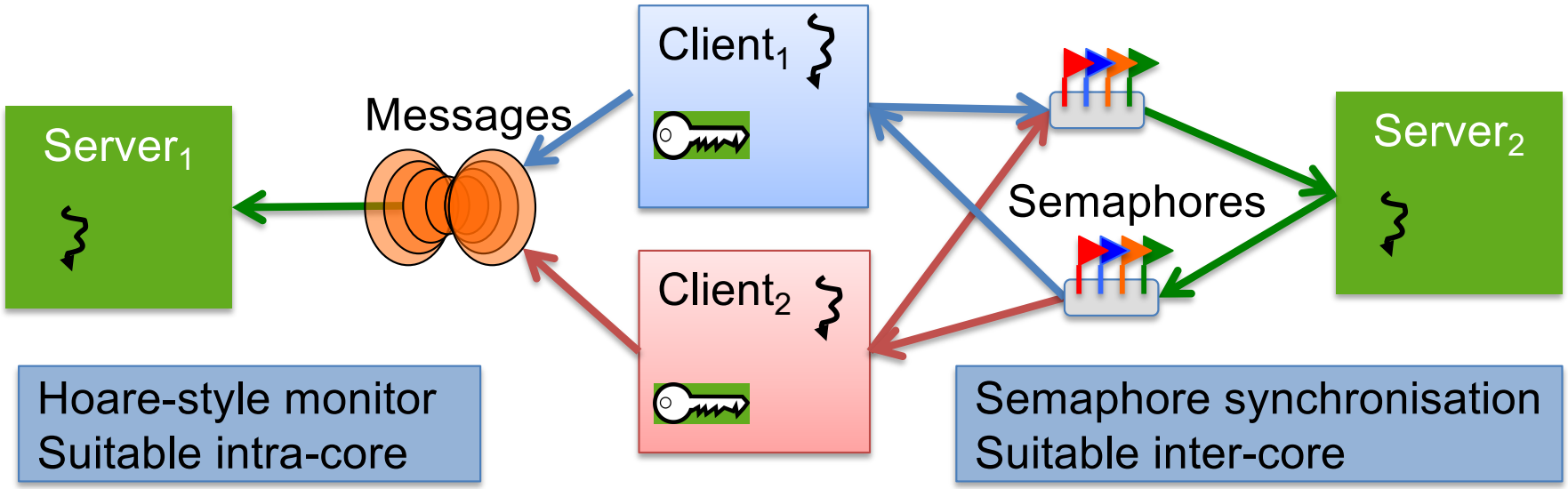
Notifications: Semaphore Synchronisation

- Logically, a Notification is an array of binary semaphores
  - Multiple signalling, select-like wait
  - Not a message-passing IPC operation!

- Implemented by *data word* in Notification
  - Send OR-s sender's *cap badge* to data word
  - Receiver can poll or wait
    - waiting returns and clears data word
    - polling just returns data word



Shared Servers for Critical Sections

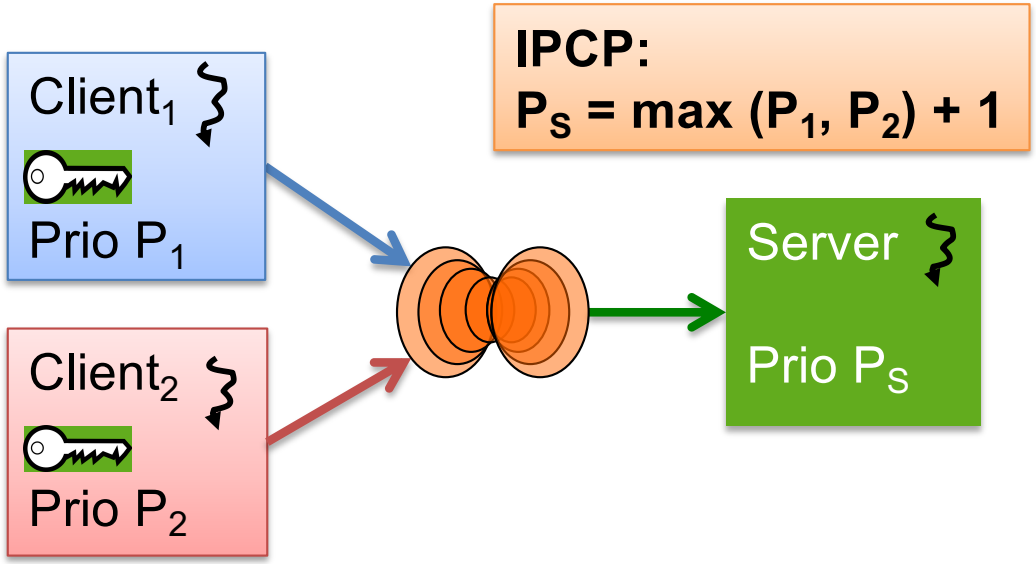



```
serv_1() {
   …
    wait(ep);
    while (1) {
       /* critical section */
       Reply&wait(ep);
     }
}
```

```
client() {
    while (1) {
      …
      call(ep);
      …
      signal(sem_ry);
      …
      wait(sem_rq);
      }
}
```

```
serv_2() {
   …
    while (1) {
      wait(sem_rq);
      /* critical section */
      signal(sem_ry);
      }
}
```

Shared Intra-Core Servers Implement Priority Ceiling Protocol (IPCP)

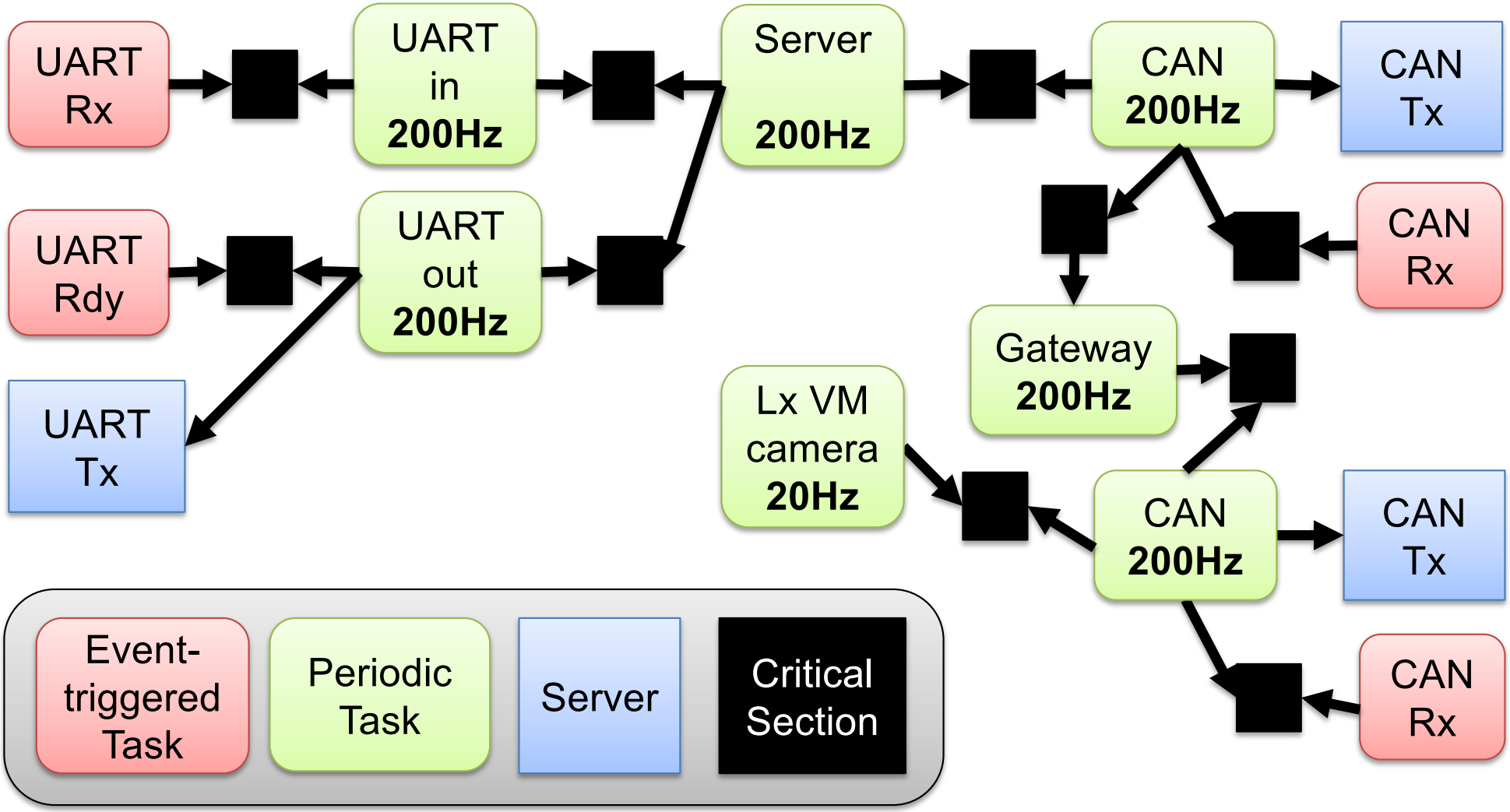


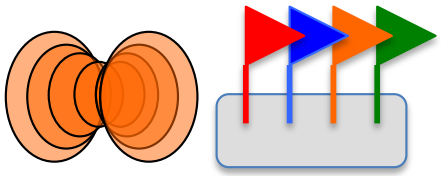
IPCP:
$P_S = \max(P_1, P_2) + 1$

Immediate Priority Ceiling:

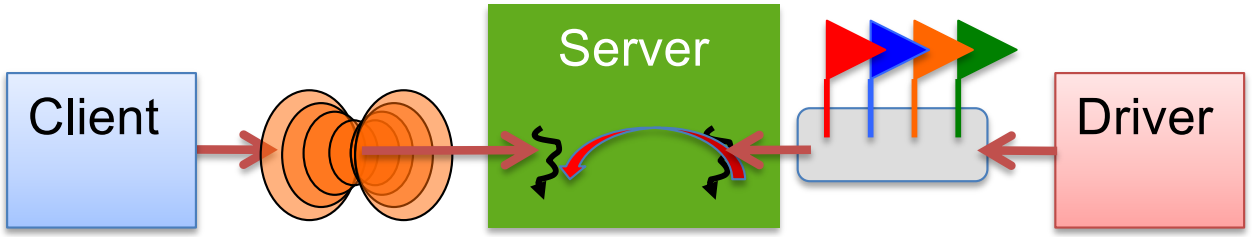
- Requires correct priority configuration
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

E.g. UAV (HACMS) Mission Computer



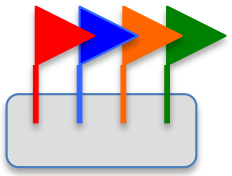


Waiting on EP *and* Notification

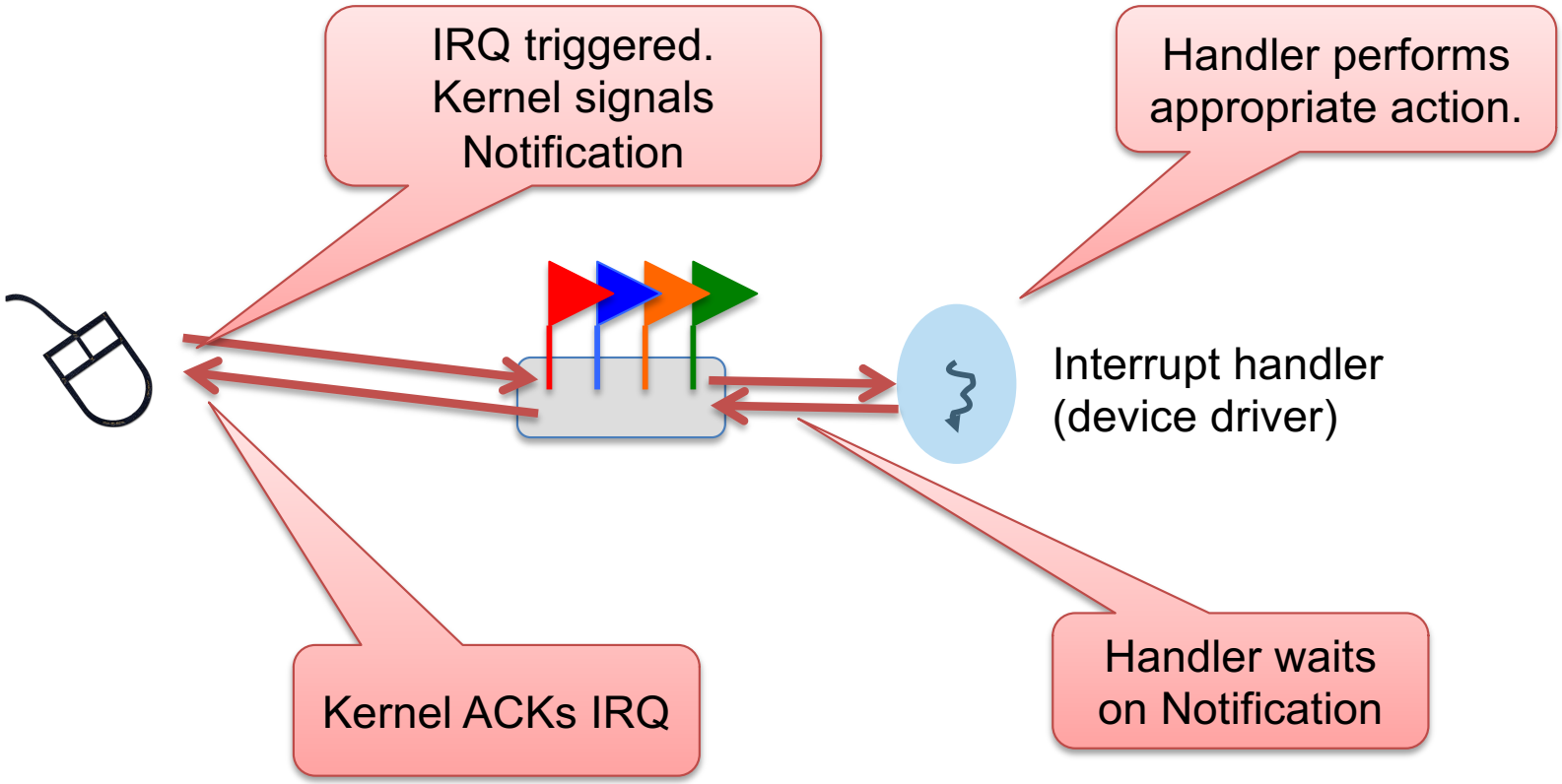


Server with synchronous and asynchronous interface

- Example: file system
  - synchronous (RPC-style) client protocol
  - asynchronous notifications from driver
- Could have separate threads waiting on endpoints
  - forces multi-threaded server, concurrency control
- Alternative: allow single thread to wait on both channels
  - Notification is *bound* to thread
  - thread waits on endpoint
  - Notification delivered as if caller had been waiting on it

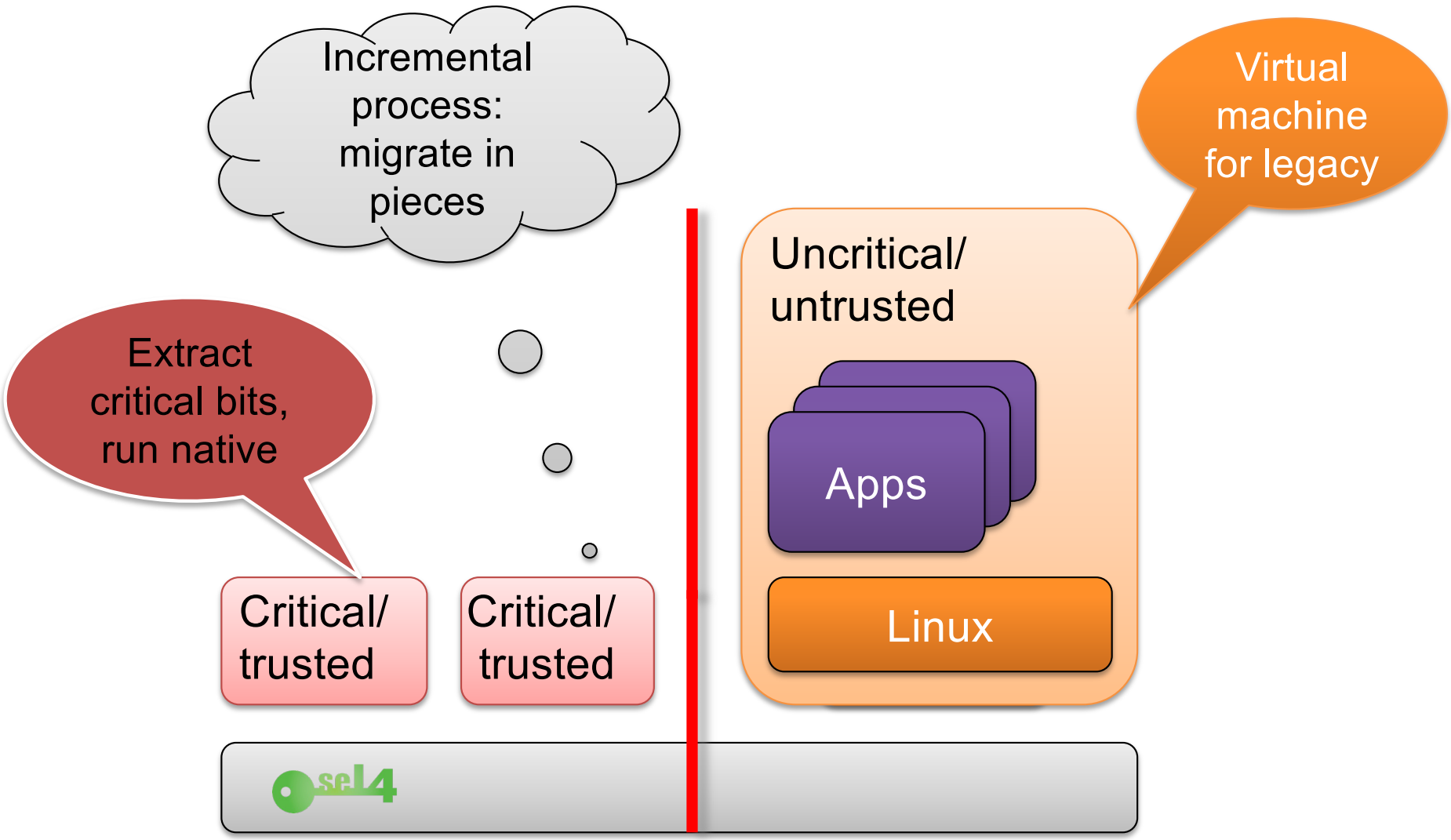


Interrupt Handling

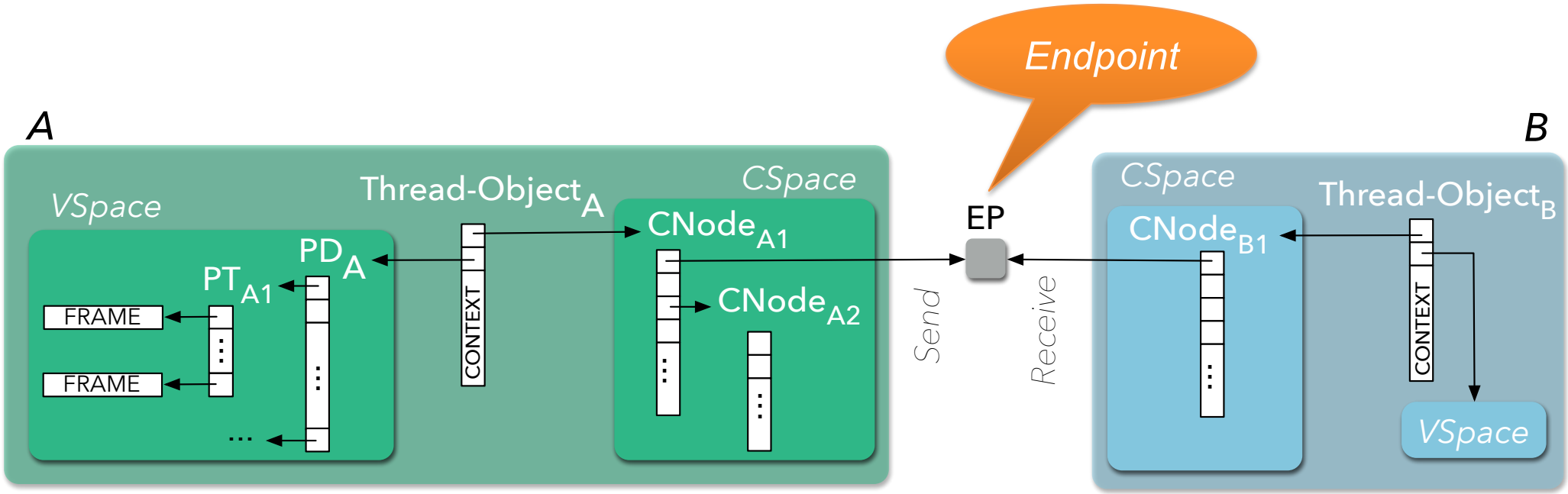


Building Trustworthy Systems

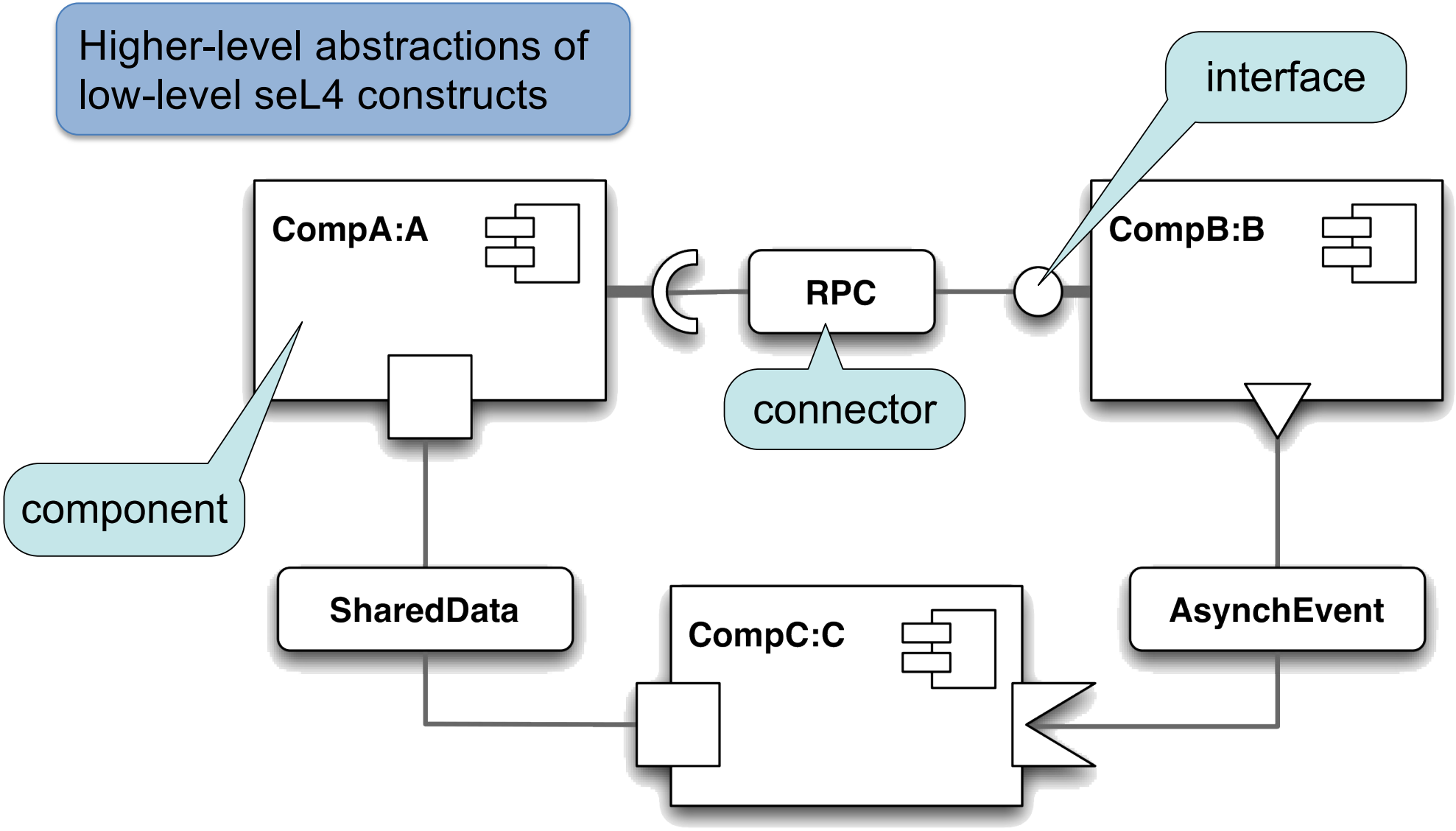
Security by Architecture



Example: Communicating Processes



Component Middleware: CAmkES



Case Study: DARPA HACMS

Boeing Unmanned Little Bird

Retrofit existing system!

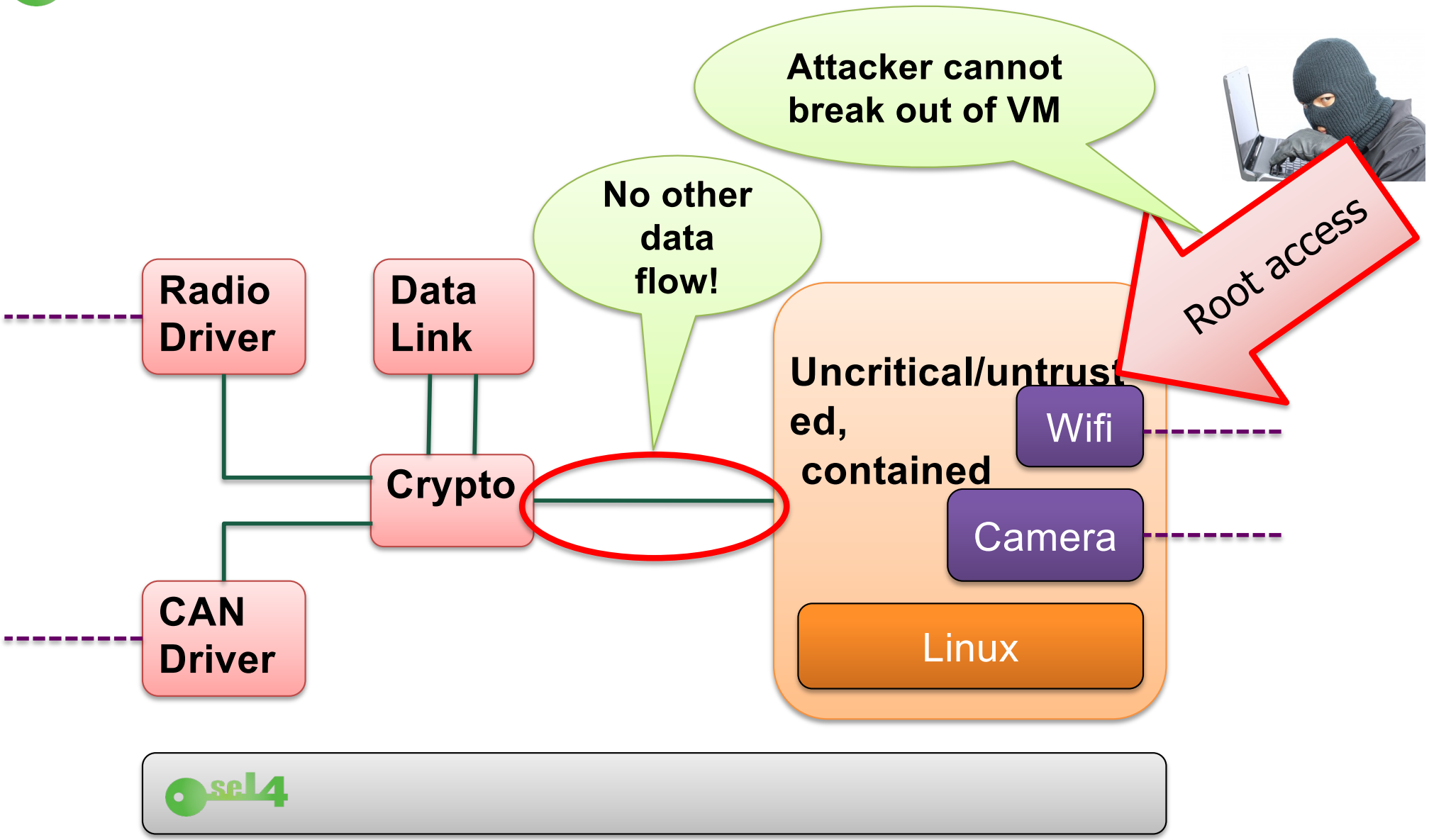
US Army Autonomous Trucks

SMACCMcopter Research Vehicle

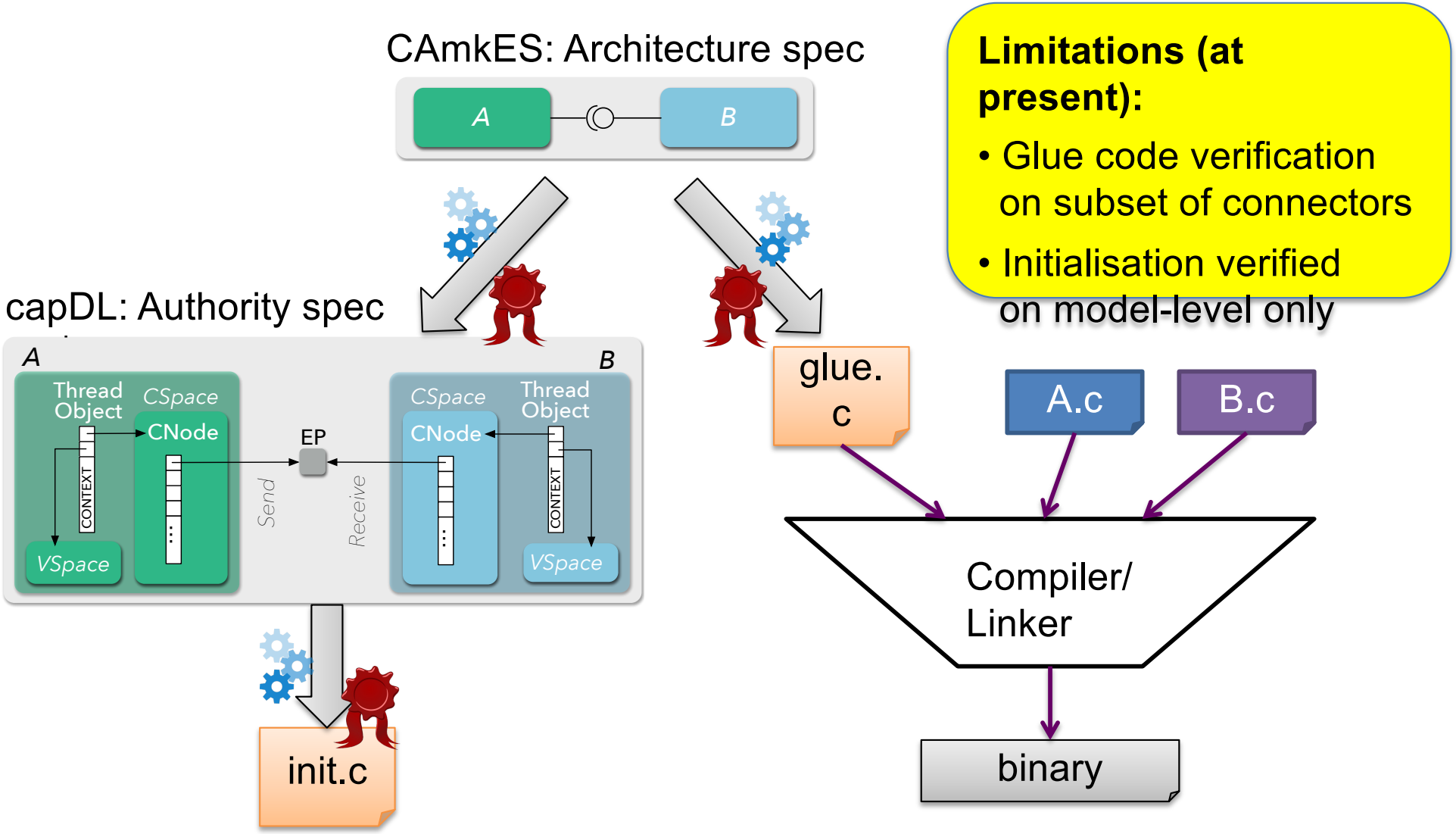
Develop technology

TARDEC GVRbot

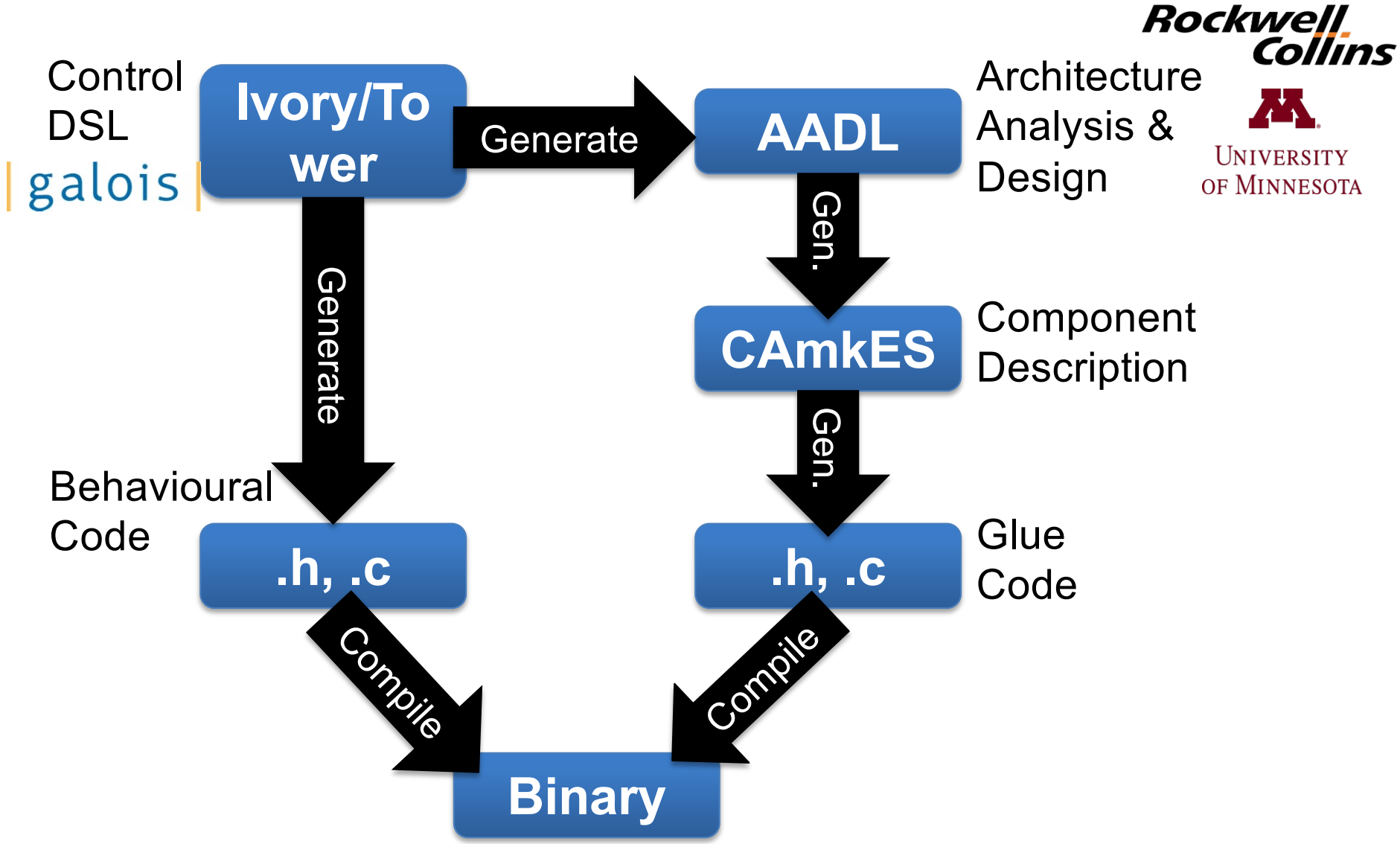
Case Study: Simplified HACMS UAV



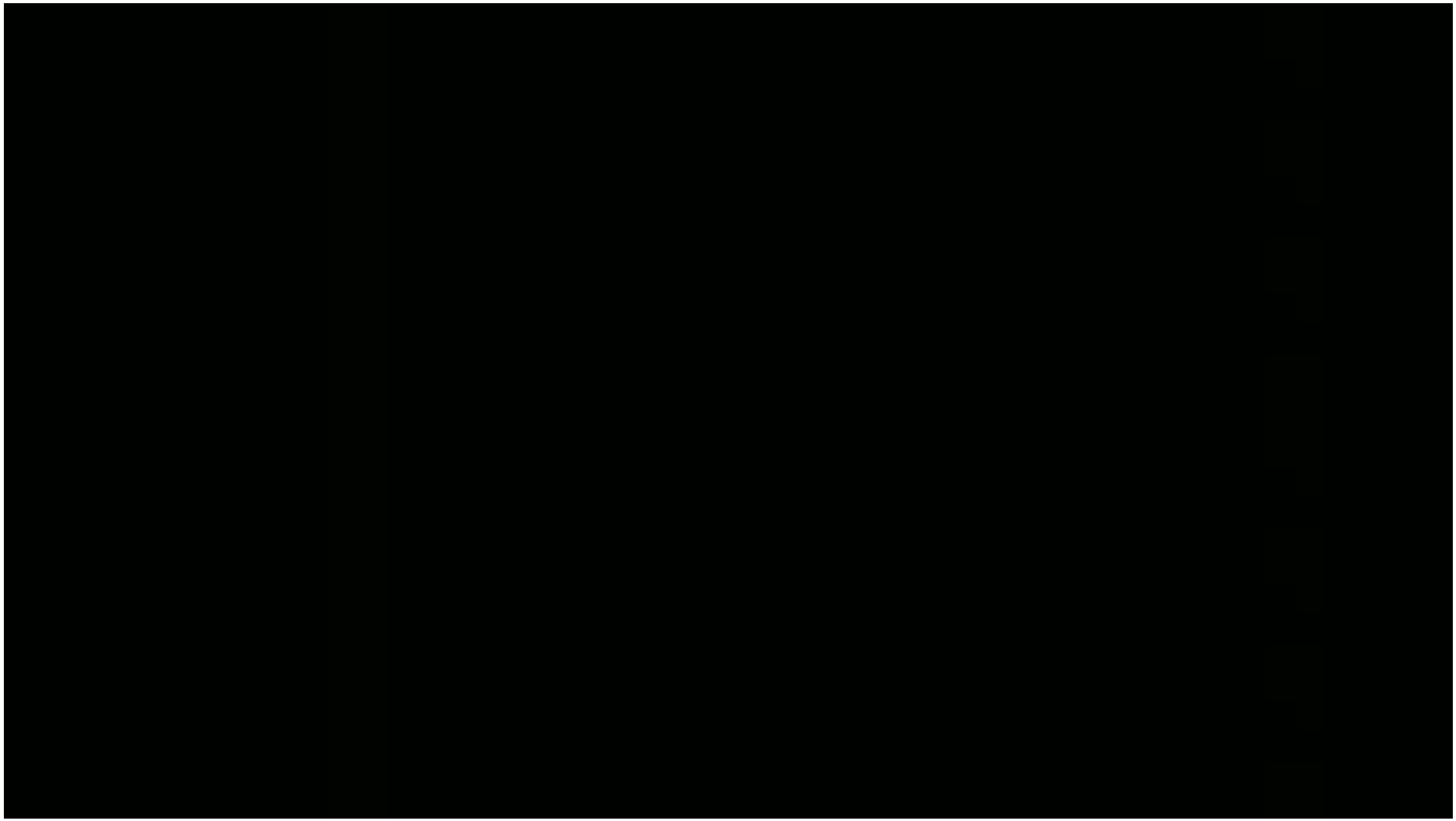
Automating the Abstraction



Model-Driven Design Using AADL

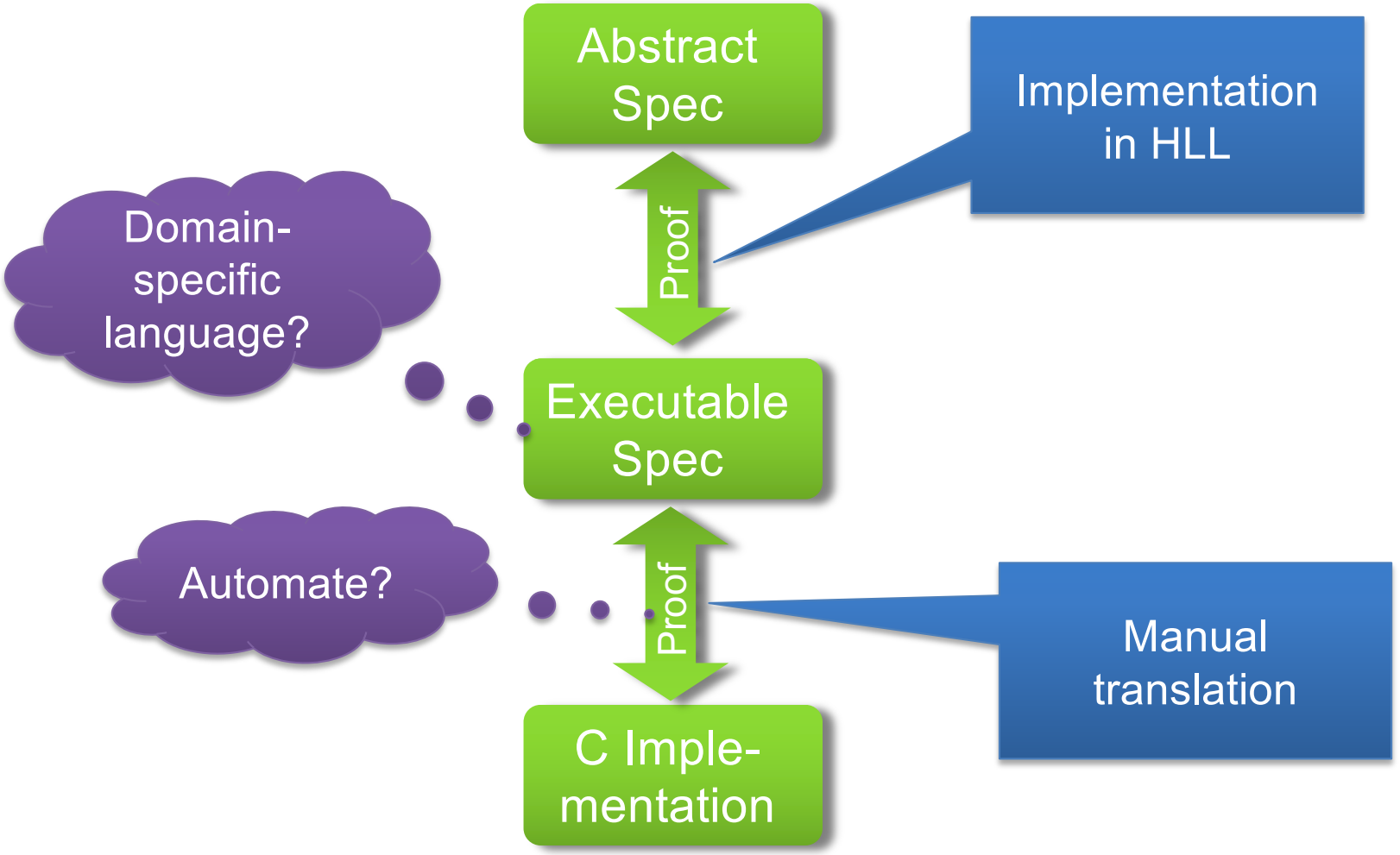


seL4 in the Real World (Courtesy Boeing, DARPA)



Work in Progress: Automating Verification

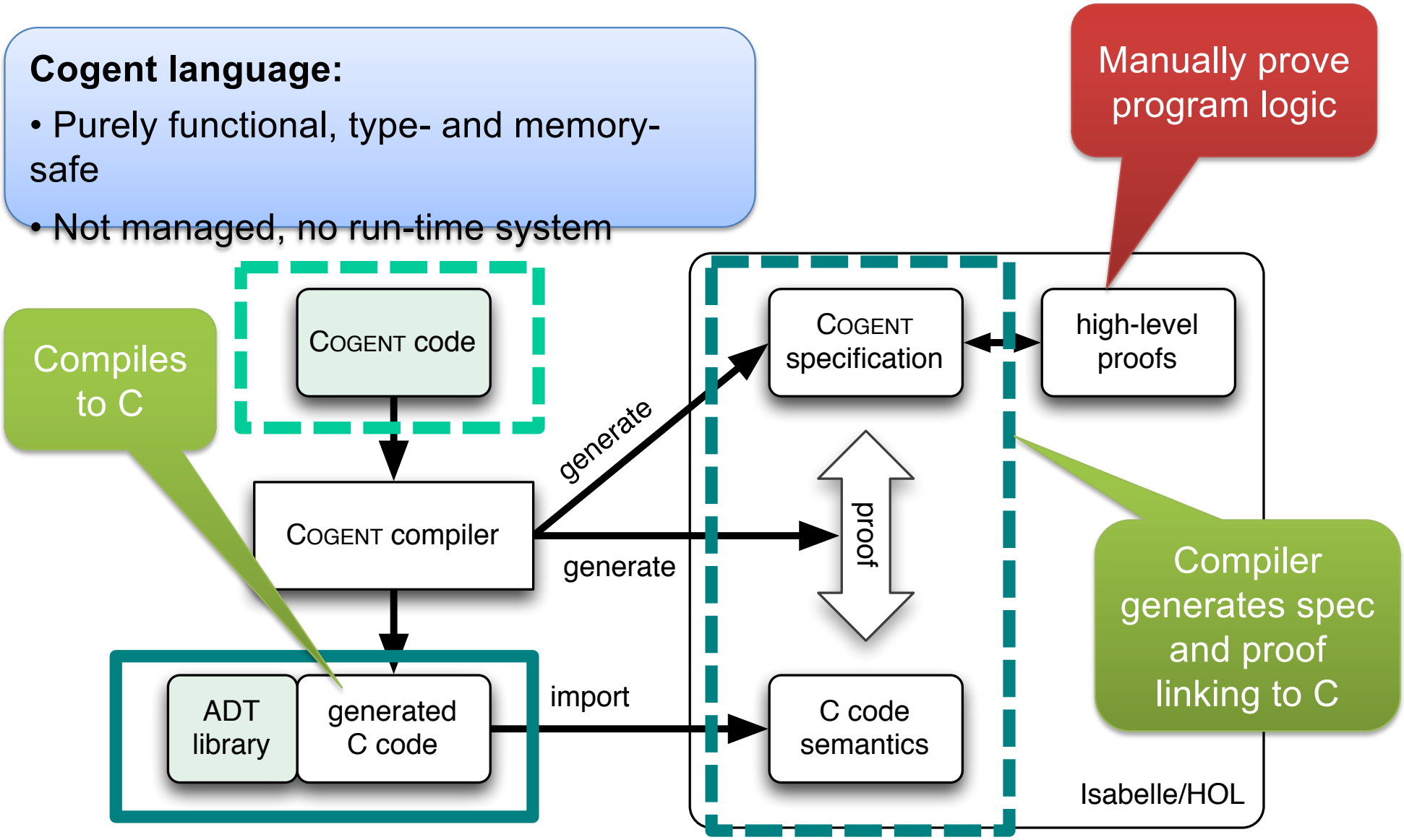
Remember: 2-Step Refinement



Cogent: Code and Proof Co-Generation

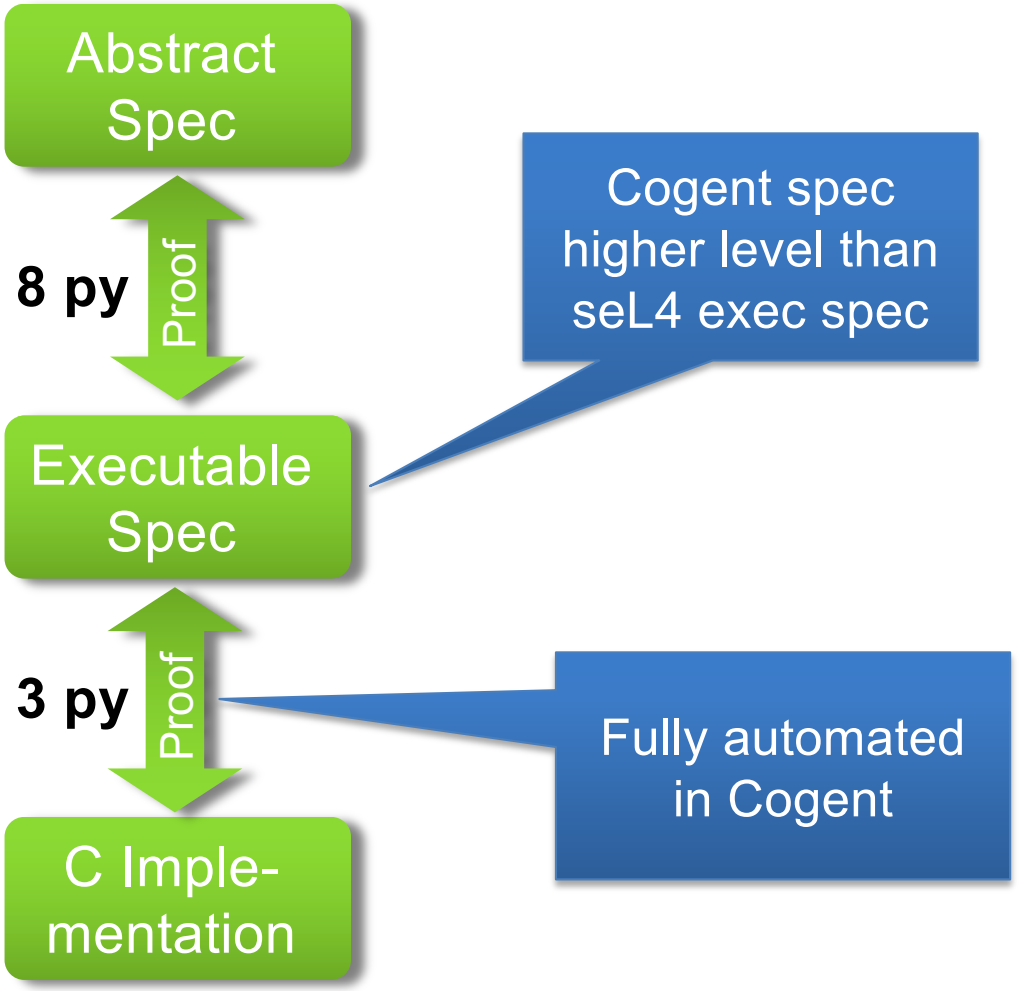
Cogent language:

- Purely functional, type- and memory-safe
- Not managed, no run-time system



Remember: Verification Cost Breakdown

- Successful file-system case study
- Extending to network stacks



Thank you!