NATIONAL ICT AUSTRALIA LIMITED

TOWARDS TRUSTWORTHY EMBEDDED SOFTWARE

Gernot Heiser

gernot@nicta.com.au

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

NICTA Members

THE AUSTRALIAN NATIONAL UNIVERSITY
UNSW
First for Business
Department of State and Regional Development
BusinessACT ACT GOVERNMENT

Victoria The Place To Be
THE UNIVERSITY OF MELBOURNE
The University of Sydney
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA

NICTA Partners

Embedded System are Ubiquitous

The are becoming an essential part of everyday life

Lessons from Desktop Systems

Desktop computer systems suck:

- They crash
- They get hacked
- They get infected

Lessons from Desktop Systems

Desktop computer systems suck:

- They crash
- They get hacked
- They get infected

How about embedded systems?

NATIONAL ICT AUSTRALIA LIMITED

WHY TRUSTWORTHY COMPUTER SYSTEMS?

Scary Examples:

- Internet banking
  - ➜ how safe are your access keys?
  - ➜ how safe is your money?

WHY TRUSTWORTHY COMPUTER SYSTEMS?

Scary Examples:

- Internet banking
  - ➜ how safe are your access keys?
  - ➜ how safe is your money?
- Health cards
  - ➜ how private is your health data?
  - ➜ is someone changing your medication?

WHY TRUSTWORTHY COMPUTER SYSTEMS?

Scary Examples:

- Internet banking
  - ➜ how safe are your access keys?
  - ➜ how safe is your money?
- Health cards
  - ➜ how private is your health data?
  - ➜ is someone changing your medication?
- Property protection
  - ➜ is someone disabling your house/car alarm?

WHY TRUSTWORTHY COMPUTER SYSTEMS?

Scary Examples:

- Internet banking
  - ➜ how safe are your access keys?
  - ➜ how safe is your money?
- Health cards
  - ➜ how private is your health data?
  - ➜ is someone changing your medication?
- Property protection
  - ➜ is someone disabling your house/car alarm?
- Personal safety
  - ➜ is some hoon hacking their steering/breaks?

WHY TRUSTWORTHY COMPUTER SYSTEMS?

Scary Examples:

- Internet banking
  - ➜ how safe are your access keys?
  - ➜ how safe is your money?
- Health cards
  - ➜ how private is your health data?
  - ➜ is someone changing your medication?
- Property protection
  - ➜ is someone disabling your house/car alarm?
- Personal safety
  - ➜ is some hoon hacking their steering/breaks?
  - ➜ is someone hacking *your* steering/breaks?

ARE EMBEDDED SYSTEMS TRUSTWORTHY?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Are Embedded Systems Trustworthy?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
- Wireless connectivity
- Downloaded contents (entertainment etc)

ARE EMBEDDED SYSTEMS TRUSTWORTHY?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
  - ➜ increasing software complexity (Mloc embedded systems)
- Wireless connectivity
- Downloaded contents (entertainment etc)

ARE EMBEDDED SYSTEMS TRUSTWORTHY?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
  - ➜ increasing software complexity (Mloc embedded systems)
  - ➜ increased number of faults (1000's)
  - ➜ increased likelihood of security faults
- Wireless connectivity
- Downloaded contents (entertainment etc)

ARE EMBEDDED SYSTEMS TRUSTWORTHY?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
  - ➜ increasing software complexity (Mloc embedded systems)
  - ➜ increased number of faults (1000's)
  - ➜ increased likelihood of security faults
- Wireless connectivity
  - ➜ systems subject to attacks from outside (crackers)
- Downloaded contents (entertainment etc)

Are Embedded Systems Trustworthy?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
  - ➜ increasing software complexity (Mloc embedded systems)
  - ➜ increased number of faults (1000's)
  - ➜ increased likelihood of security faults
- Wireless connectivity
  - ➜ systems subject to attacks from outside (crackers)
- Downloaded contents (entertainment etc)
  - ➜ systems subject to attacks from inside (viruses, worms)

Are Embedded Systems Trustworthy?

Presently: Maybe:

- Embedded systems tend to be *simple* and *closed*

Security/Safety Challenges:

- Growing functionality
  - ➜ increasing software complexity (Mloc embedded systems)
  - ➜ increased number of faults (1000's)
  - ➜ increased likelihood of security faults
- Wireless connectivity
  - ➜ systems subject to attacks from outside (crackers)
- Downloaded contents (entertainment etc)
  - ➜ systems subject to attacks from inside (viruses, worms)

Future systems highly vulnerable!

- ➜ challenges at least as bad as in PC world
- ➜ defences aren't better

TRUSTWORTHINESS

Why is it a problem?

- Computer systems are too big, too complex
  - ➜ several millions loc in a mobile-phone handset
  - ➜ Gigabytes of software in top-range automobiles
- Software is inherently buggy
- Large and complex software is *very* buggy

TRUSTWORTHINESS

Why is it a problem?

- Computer systems are too big, too complex
  - ➜ several millions loc in a mobile-phone handset
  - ➜ Gigabytes of software in top-range automobiles
- Software is inherently buggy
- Large and complex software is *very* buggy
  - ➜ High-quality software has ≈1 bug / 1000loc (OS code: 1–10/1000loc)
  - ⇒ 1 Mloc has ≈1000 bugs (or ≈10,000)
  - ➜ each one is potentially fatal

TRUSTWORTHINESS

Why is it a problem?

- Computer systems are too big, too complex
  - ➜ several millions loc in a mobile-phone handset
  - ➜ Gigabytes of software in top-range automobiles
- Software is inherently buggy
- Large and complex software is *very* buggy
  - ➜ High-quality software has ≈1 bug / 1000loc (OS code: 1–10/1000loc)
  - ⇒ 1 Mloc has ≈1000 bugs (or ≈10,000)
  - ➜ each one is potentially fatal

Can such a large system ever be made trustworthy?

TRUSTWORTHINESS

Why is it a problem?

- Computer systems are too big, too complex
  - ➜ several millions loc in a mobile-phone handset
  - ➜ Gigabytes of software in top-range automobiles
- Software is inherently buggy
- Large and complex software is *very* buggy
  - ➜ High-quality software has ≈1 bug / 1000loc (OS code: 1–10/1000loc)
  - ⇒ 1 Mloc has ≈1000 bugs (or ≈10,000)
  - ➜ each one is potentially fatal

Can such a large system ever be made trustworthy?

- Yes, if its *trusted computing base* (TCB) is trustworthy!
  - ➜ **TCB:** The part of the system that can circumvent security

TRUSTED COMPUTING BASE — TRUSTWORTHY?

How to make a TCB trustworthy?

- TCB must be *known to be functionally correct*
  ➜ How????

TRUSTED COMPUTING BASE — TRUSTWORTHY?

How to make a TCB trustworthy?

- TCB must be *known to be functionally correct*
  ➜ How????
- Testing?
- Proof?
- Modularity?

Trusted Computing Base — Trustworthy?

How to make a TCB trustworthy?

- TCB must be *known to be functionally correct*
  ➜ How????
- Testing?
  ➜ exhaustive testing doesn't scale beyond 100's loc
  ➜ non-exhaustive testing cannot show absence of bugs
- Proof?
- Modularity?

Trusted Computing Base — Trustworthy?

How to make a TCB trustworthy?

- TCB must be *known to be functionally correct*
  ➜ How????
- Testing?
  ➜ exhaustive testing doesn't scale beyond 100's loc
  ➜ non-exhaustive testing cannot show absence of bugs
- Proof?
  ➜ software proofs don't scale beyond 1000's loc
- Modularity?

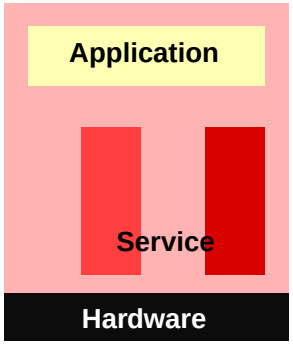
Trusted Computing Base — Trustworthy?

How to make a TCB trustworthy?

- TCB must be *known to be functionally correct*
  - ➜ How????
- Testing?
  - ➜ exhaustive testing doesn't scale beyond 100's loc
  - ➜ non-exhaustive testing cannot show absence of bugs
- Proof?
  - ➜ software proofs don't scale beyond 1000's loc
- Modularity?
  - ➜ can reduce complexity of verification problem
  - ➜ doesn't help if interfaces aren't enforced
  - ➜ can't guarantee if not type-safe or privileged-mode execution

EMBEDDED SYSTEMS TCB

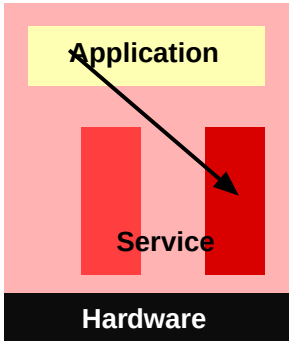
Traditional approach: Real-time executive



- Small, simple operating system
  - ➜ optimised for fast real-time response
  - ➜ suitable for systems with very limited functionality

EMBEDDED SYSTEMS TCB

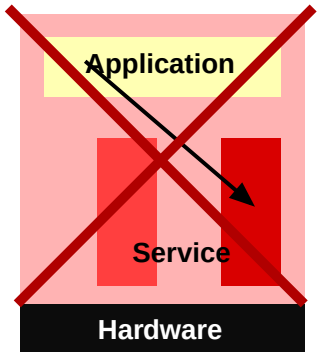
Traditional approach: Real-time executive



- Small, simple operating system
  - ➜ optimised for fast real-time response
  - ➜ suitable for systems with very limited functionality

EMBEDDED SYSTEMS TCB

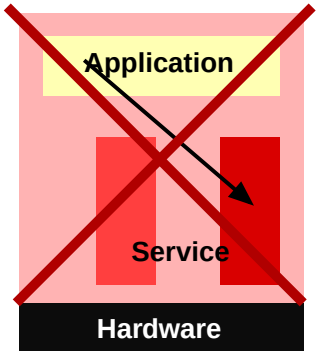
Traditional approach: Real-time executive



- Small, simple operating system
  - ➜ optimised for fast real-time response
  - ➜ suitable for systems with very limited functionality

EMBEDDED SYSTEMS TCB

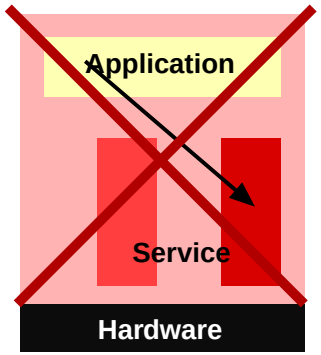
Traditional approach: Real-time executive



- Small, simple operating system
  - ➜ optimised for fast real-time response
  - ➜ suitable for systems with very limited functionality
- No internal protection
  - ➜ every small bug/failure is fatal
  - ➜ no defence against viruses, limited defence against crackers

EMBEDDED SYSTEMS TCB

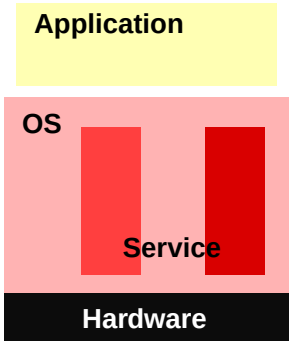
Traditional approach: Real-time executive



- Small, simple operating system
  - ➜ optimised for fast real-time response
  - ➜ suitable for systems with very limited functionality
- No internal protection
  - ➜ every small bug/failure is fatal
  - ➜ no defence against viruses, limited defence against crackers
- *TCB: all code* (millions of lines!)

EMBEDDED SYSTEMS TCB

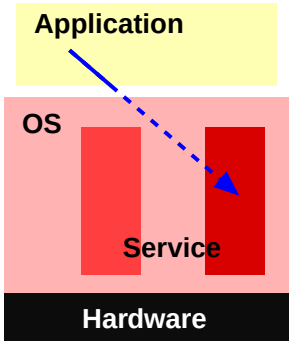
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  - ➜ operating system protected from application misbehaviour

EMBEDDED SYSTEMS TCB

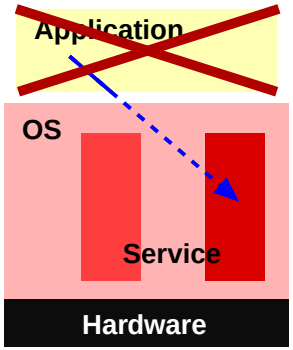
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  ➜ operating system protected from application misbehaviour

EMBEDDED SYSTEMS TCB

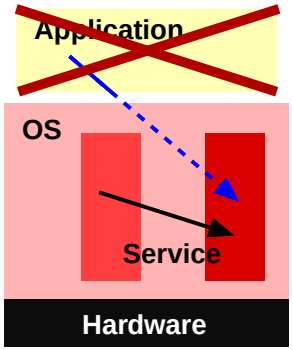
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  ➜ operating system protected from application misbehaviour

EMBEDDED SYSTEMS TCB

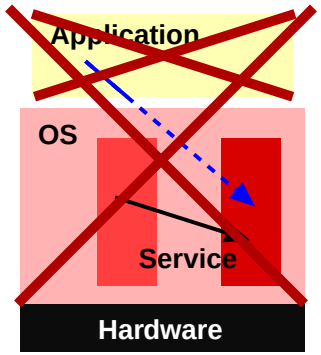
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  - ➜ operating system protected from application misbehaviour

EMBEDDED SYSTEMS TCB

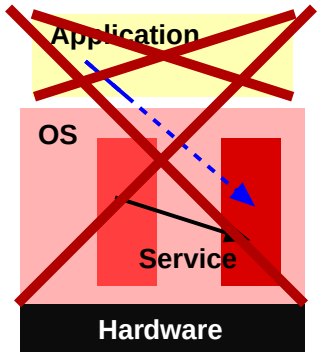
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  ➜ operating system protected from application misbehaviour

EMBEDDED SYSTEMS TCB

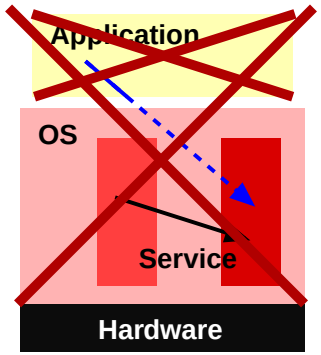
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  - ➜ operating system protected from application misbehaviour
  - ➜ excessive code base for small embedded system
  - ➜ too much code on which security of system is dependent

EMBEDDED SYSTEMS TCB

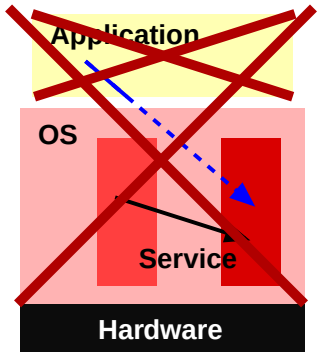
Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  - ➜ operating system protected from application misbehaviour
  - ➜ excessive code base for small embedded system
  - ➜ too much code on which security of system is dependent
- Dubious or non-existent real-time capabilities
  - ➜ unsuitable for hard real-time systems

EMBEDDED SYSTEMS TCB

Popular approach: Linux, Windows Embedded, ...



- Scaled-down version of desktop operating system
  - ➜ operating system protected from application misbehaviour
  - ➜ excessive code base for small embedded system
  - ➜ too much code on which security of system is dependent
- Dubious or non-existent real-time capabilities
  - ➜ unsuitable for hard real-time systems
- *TCB: kernel + some user-level code* (100,000s of lines!)

Trustworthiness Requires Componentisation

- Use modularity to manage complexity
  ➜ break into bite-sized pieces
- Use language or hardware mechanisms to enforce interfaces

Trustworthiness Requires Componentisation

- Use modularity to manage complexity
  - ➜ break into bite-sized pieces
- Use language or hardware mechanisms to enforce interfaces
- Verify components individually
  - ➜ using exhaustive testing or formal methods
  - ➜ requires components that are small enough
  - ➜ really depends on interfaces being explicit and enforced

Trustworthiness Requires Componentisation

- Use modularity to manage complexity
  - ➜ break into bite-sized pieces
- Use language or hardware mechanisms to enforce interfaces
- Verify components individually
  - ➜ using exhaustive testing or formal methods
  - ➜ requires components that are small enough
  - ➜ really depends on interfaces being explicit and enforced
- Of course, this alone is *insufficient* for trustworthiness
  - ➜ no guarantee that whole system operates as intended
  - ➜ solution left as an exercise for the reader 

Kernel is a Big Headache

- Kernel = code that executes in privileged mode
  - ➜ any piece of kernel code has unrestricted access
  - ⇒ no hardware encapsulation possible

Kernel is a Big Headache

- Kernel = code that executes in privileged mode
  - ➜ any piece of kernel code has unrestricted access
  - ⇒ no hardware encapsulation possible
  - ➜ type safety is of limited value
    - ➜ kernel contains many operations that are inherently type-unsafe
    - ➜ type safety introduces significant overhead
    - ➜ type-safe languages tend to increase TCB (run-time system)

      or are too weak to be useful

Kernel is a Big Headache

- Kernel = code that executes in privileged mode
  - ➜ any piece of kernel code has unrestricted access
  - ⇒ no hardware encapsulation possible
  - ➜ type safety is of limited value
    - ➜ kernel contains many operations that are inherently type-unsafe
    - ➜ type safety introduces significant overhead
    - ➜ type-safe languages tend to increase TCB (run-time system)

      or are too weak to be useful
- Kernel is indivisible module
  - ➜ can only be verified in one piece
  - ⇒ kernel *must* be small — 1000's loc

Kernel is a Big Headache

- Kernel = code that executes in privileged mode
  - ➜ any piece of kernel code has unrestricted access
  - ⇒ no hardware encapsulation possible
  - ➜ type safety is of limited value
    - ➜ kernel contains many operations that are inherently type-unsafe
    - ➜ type safety introduces significant overhead
    - ➜ type-safe languages tend to increase TCB (run-time system)

      or are too weak to be useful
- Kernel is indivisible module
  - ➜ can only be verified in one piece
  - ⇒ kernel *must* be small — 1000's loc
- Kernel must be *minimal*
  - ➜ contain only code that *must* be privileged

Kernel is a Big Headache

- Kernel = code that executes in privileged mode
  - ➜ any piece of kernel code has unrestricted access
  - ⇒ no hardware encapsulation possible
  - ➜ type safety is of limited value
    - ➜ kernel contains many operations that are inherently type-unsafe
    - ➜ type safety introduces significant overhead
    - ➜ type-safe languages tend to increase TCB (run-time system)

      or are too weak to be useful
- Kernel is indivisible module
  - ➜ can only be verified in one piece
  - ⇒ kernel *must* be small — 1000's loc
- Kernel must be *minimal*
  - ➜ contain only code that *must* be privileged
- Kernel *must be a microkernel*

What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services

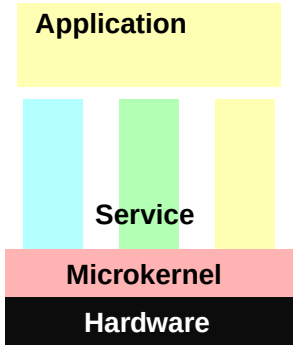
NATIONAL ICT AUSTRALIA LIMITED

What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical

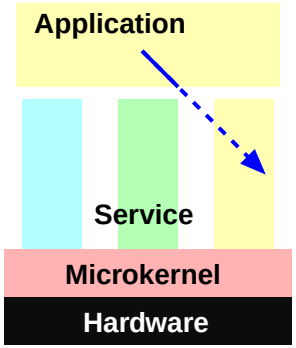
What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical



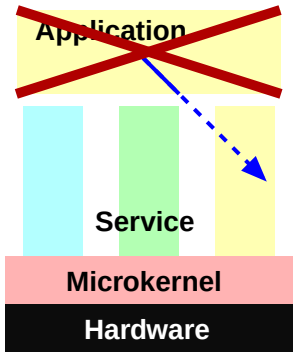
What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical



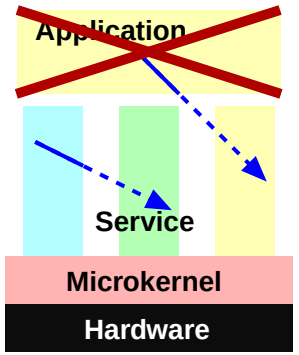
What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical



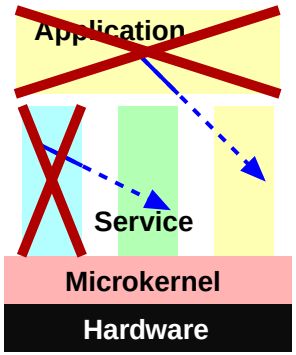
What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical

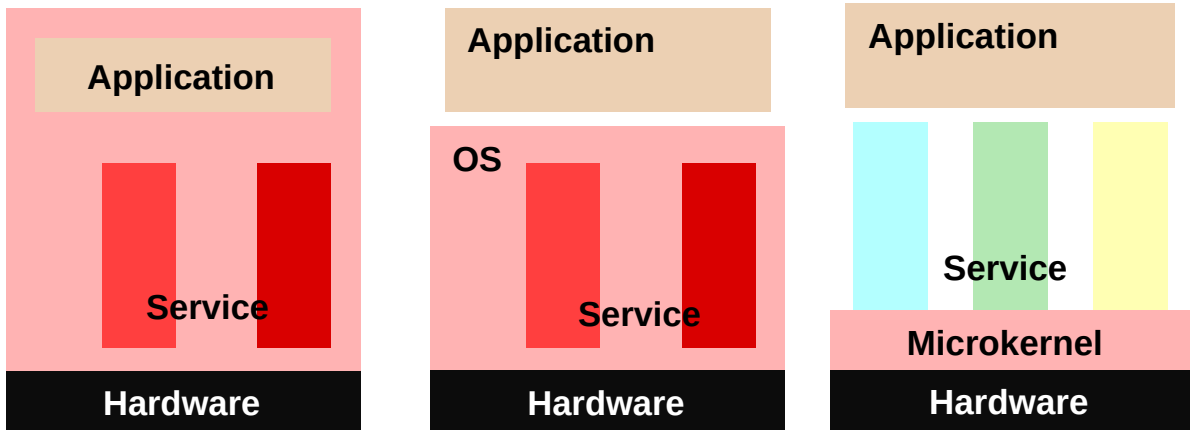


What is a Microkernel?

- Microkernel *only* contains code that *must* execute in privileged mode
  - ➜ e.g., code that manipulates processor state, MMU, ...
- Microkernel contains no code that can safely run in user mode
  - ➜ no device drivers
  - ➜ no resource-management policies
  - ➜ no actual operating-system services
- Services and policies implemented as user-level servers
  - ➜ applications obtain services by sending messages to servers
  - ⇒ performance of message-passing IPC is critical
- Faults can be contained
  - ➜ removes more code from TCB
  - ➜ kernel size: ≈ 10,000 loc

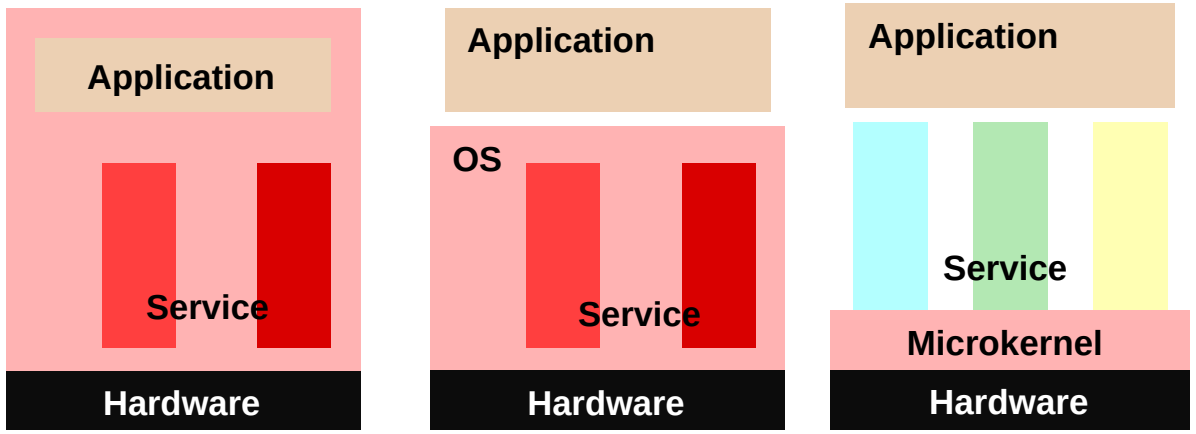


EMBEDDED SYSTEMS TCB COMPARISON



System:	traditional embedded	Linux/ Windows	Microkernel-based (L4)

Embedded Systems TCB Comparison

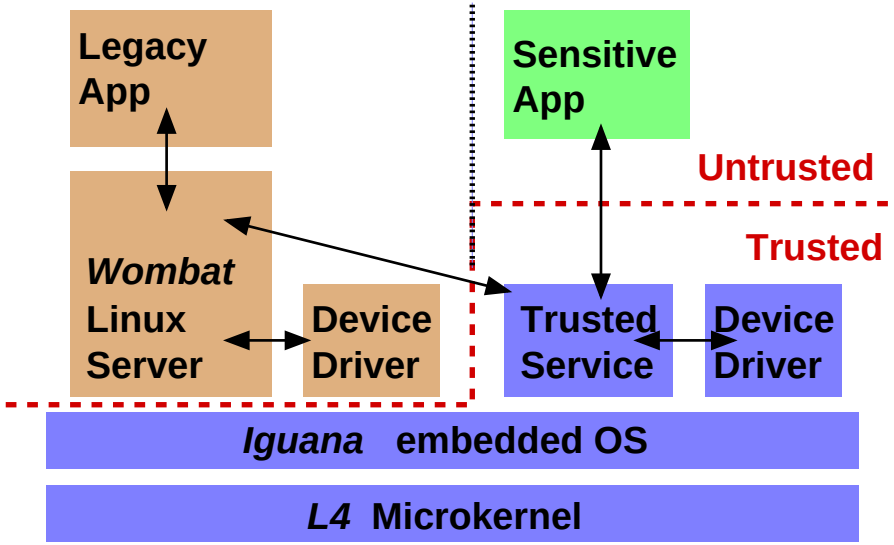


System:	traditional embedded	Linux/ Windows	Microkernel-based (L4)
TCB:	**all** code	$\geq$ 200,000 loc	$\geq$ 20,000 loc

Note: size (TCB) $\geq$ size (kernel)

VIRTUALISATION: BEST OF BOTH WORLDS

- Minimal secure environment (L4 + Iguana)
  - ➜ small TCB
  - ➜ fully real-time capable
- Virtualised legacy OS for application support
  - ➜ binary compatibility with native legacy code
  - ➜ performance close to native



- Secure subsystem can provide services to legacy system
  - ➜ e.g. device drivers

Requirements for Trusted Systems

0. Very small, high-performance microkernel: L4
1. Microkernel mechanisms for secure systems
2. Verification of microkernel implementation
3. Trustworthy temporal model of microkernel
4. Techniques for building high-performance microkernel-based systems

seL4: Microkernel Mechanisms for Secure Systems

seL4: Microkernel for secure embedded systems:

- Security requirements for embedded systems:
  - ➜ Integrity: protecting data from damage
  - ➜ Availability: ensuring system operation
  - ➜ Privacy: protecting sensitive data from loss
  - ➜ IP Protection: controlling propagation of valuable data

seL4: Microkernel Mechanisms for Secure Systems

seL4: Microkernel for secure embedded systems:

- Security requirements for embedded systems:
  - ➜ Integrity: protecting data from damage
  - ➜ Availability: ensuring system operation
  - ➜ Privacy: protecting sensitive data from loss
  - ➜ IP Protection: controlling propagation of valuable data
- Issue: Present L4 API unsuitable for highly-secure systems
  - inefficient information flow control mechanisms
    - ➜ present mechanisms double or triple IPC costs
  - insufficient resource isolation (kernel memory pool)
    - ➜ applications can force kernel to run out of memory
    - ➜ present countermeasures are inflexible

seL4: Microkernel Mechanisms for Secure Systems

Project status:

- Semi-formal API specification in literal Haskell
  - ➜ automatic generation of API documentation from source
  - ➜ draft available from `http://ertos.nicta.com.au/research/sel4`

seL4: Microkernel Mechanisms for Secure Systems

Project status:

- Semi-formal API specification in literal Haskell
  - ➜ automatic generation of API documentation from source
  - ➜ draft available from `http://ertos.nicta.com.au/research/sel4`
- Proof of separation properties
  - ➜ suitable for confinement, DRM

seL4: Microkernel Mechanisms for Secure Systems

Project status:

- Semi-formal API specification in literal Haskell
  - ➜ automatic generation of API documentation from source
  - ➜ draft available from `http://ertos.nicta.com.au/research/sel4`
- Proof of separation properties
  - ➜ suitable for confinement, DRM
- Prototype implementation in Haskell, integrated with ISA simulator
  - ➜ rapid prototyping: API changes implemented in hours/days
  - ➜ can build and execute apps using standard build tools
  - ➜ used for porting user-level software

seL4: Microkernel Mechanisms for Secure Systems

Project status:

- Semi-formal API specification in literal Haskell
  - ➜ automatic generation of API documentation from source
  - ➜ draft available from `http://ertos.nicta.com.au/research/sel4`
- Proof of separation properties
  - ➜ suitable for confinement, DRM
- Prototype implementation in Haskell, integrated with ISA simulator
  - ➜ rapid prototyping: API changes implemented in hours/days
  - ➜ can build and execute apps using standard build tools
  - ➜ used for porting user-level software
- C implementation: Dec '06

L4.verified: Formal verification of Kernel

- Leverage small size of kernel to *prove* correctness

L4.verified: Formal verification of Kernel

- Leverage small size of kernel to *prove* correctness
- Part 1: Pilot project (Jan '04 – Mar '05)
  - Verified thin "slice" of API all the way to source code
    - ➜ memory-management functions
    - ➜ $>$ 10% of kernel code, $>$ 20% of kernel complexity
    - ➜ 1.5 person years
  - Did almost complete formalisation of present L4 API

L4.verified: Formal verification of Kernel

- Leverage small size of kernel to *prove* correctness
- Part 1: Pilot project (Jan '04 – Mar '05)
  - Verified thin "slice" of API all the way to source code
    - ➜ memory-management functions
    - ➜ $>$ 10% of kernel code, $>$ 20% of kernel complexity
    - ➜ 1.5 person years
  - Did almost complete formalisation of present L4 API
- Part 2: Main project (Apr '05 – Mar '08)
  - Refinement approach using the Isabelle/HOL theorem prover
    - ➜ Importing seL4 API specification (Haskell)
    - ➜ Importing C/assembler implementation to be proved

L4.verified: Formal verification of Kernel

- Leverage small size of kernel to *prove* correctness
- Part 1: Pilot project (Jan '04 – Mar '05)
  - Verified thin "slice" of API all the way to source code
    - ➜ memory-management functions
    - ➜ > 10% of kernel code, > 20% of kernel complexity
    - ➜ 1.5 person years
  - Did almost complete formalisation of present L4 API
- Part 2: Main project (Apr '05 – Mar '08)
  - Refinement approach using the Isabelle/HOL theorem prover
    - ➜ Importing seL4 API specification (Haskell)
    - ➜ Importing C/assembler implementation to be proved

    3
  - Result to be usable in existing deployments
    - ➜ no sacrificing of performance for verifiability
  - On track...

POTOROO: COMPLETE TEMPORAL MODEL OF KERNEL

- Prerequisite for complete real-time analysis of whole system
  - ➜ strict worst-case execution times (WCET)
  - ➜ probabilistic WCET

POTOROO: COMPLETE TEMPORAL MODEL OF KERNEL

- Prerequisite for complete real-time analysis of whole system
  - ➜ strict worst-case execution times (WCET)
  - ➜ probabilistic WCET
- Essential for trustworthy real-time systems
  - ➜ RT analysis of applications pointless without timing model of kernel

POTOROO: COMPLETE TEMPORAL MODEL OF KERNEL

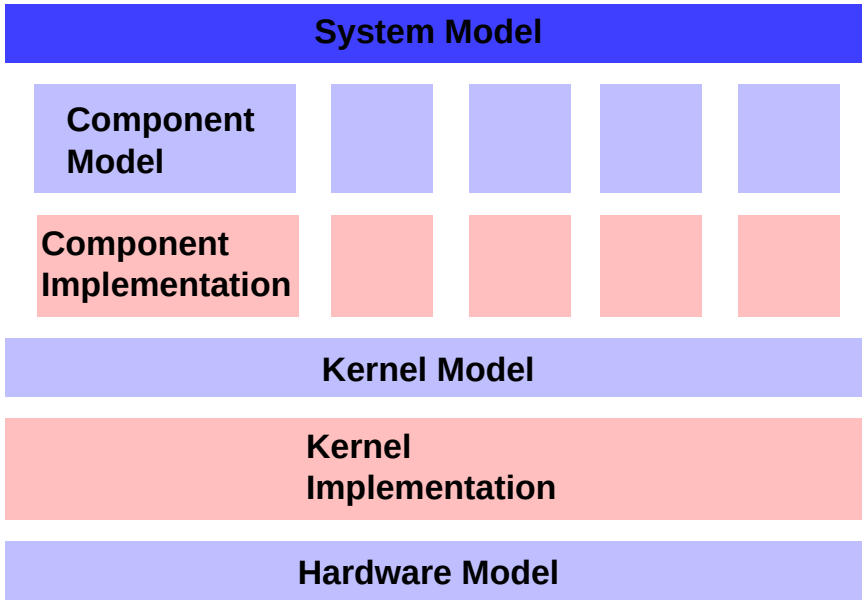
- Prerequisite for complete real-time analysis of whole system
  - ➜ strict worst-case execution times (WCET)
  - ➜ probabilistic WCET
- Essential for trustworthy real-time systems
  - ➜ RT analysis of applications pointless without timing model of kernel
- Measurement-based approach augmented by static analysis
  - ➜ *measure* execution-time profiles of basic blocks
  - ➜ convolute into overall execution-time profile
  - ➜ static analysis to ensure worst case observed
  - ➜ static analysis to reduce pessimism

CAmkES: Component Architecture for Microkernel-Based Embedded Systems

- Aim: approach for highly-componentised embedded software
  - ☑ reduce software cost by enforcing modularity
  - ☑ deliver on fault isolation, hot upgrades, security enforcement, ...

CAmkES: Component Architecture for Microkernel-Based Embedded Systems

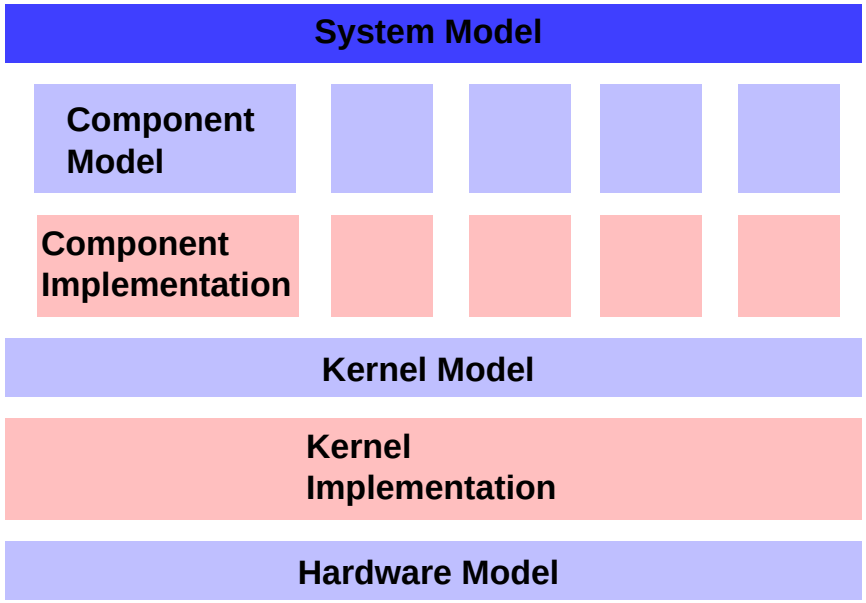
- Aim: approach for highly-componentised embedded software
  - ☑ reduce software cost by enforcing modularity
  - ☑ deliver on fault isolation, hot upgrades, security enforcement, ...
- Ultimate goal:
  Full system verification



CAmkES: Component Architecture for Microkernel-Based Embedded Systems

- Aim: approach for highly-componentised embedded software
  - ☑ reduce software cost by enforcing modularity
  - ☑ deliver on fault isolation, hot upgrades, security enforcement, ...
- Ultimate goal:

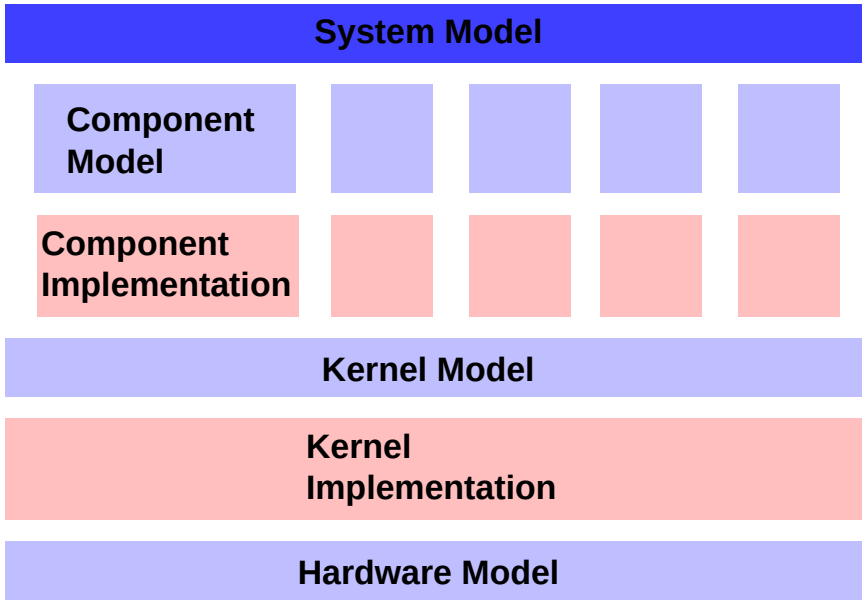
  Full system verification
  - ➜ kernel-enforced component boundaries
  - ☑ can verify components individually
  - ☒ model composition?
  - ➜ Distant future...



CAmkES: Component Architecture for Microkernel-Based Embedded Systems

- Aim: approach for highly-componentised embedded software
  - ☑ reduce software cost by enforcing modularity
  - ☑ deliver on fault isolation, hot upgrades, security enforcement, ...
- Ultimate goal:

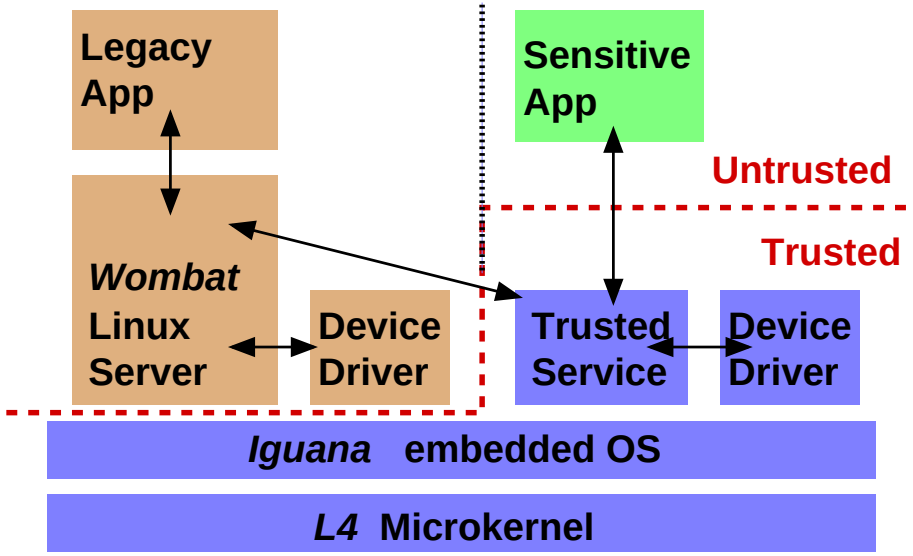
  Full system verification
  - ➜ kernel-enforced component boundaries
  - ☑ can verify components individually
  - ☒ model composition?
  - ➜ Distant future...
- Status: static prototype



- Working on dynamic system, performance, non-functional properties

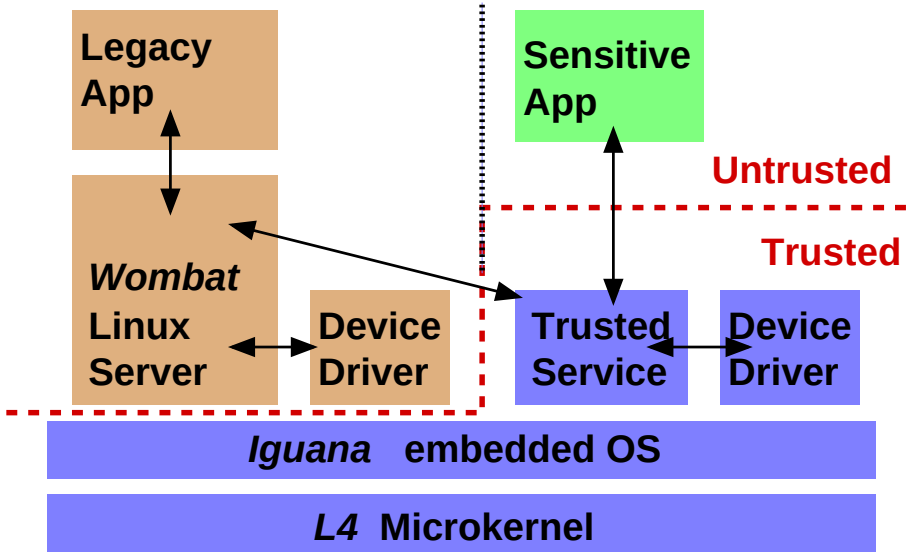
Present State

- Pistachio: Mature microkernel
  - ➜ 10,000 lines of code (shrinking)
  - ➜ highly efficient
- Iguana: Core OS services
  - ➜ naming, protection, memory...
  - ➜ device drivers
  - ➜ optional Linux server



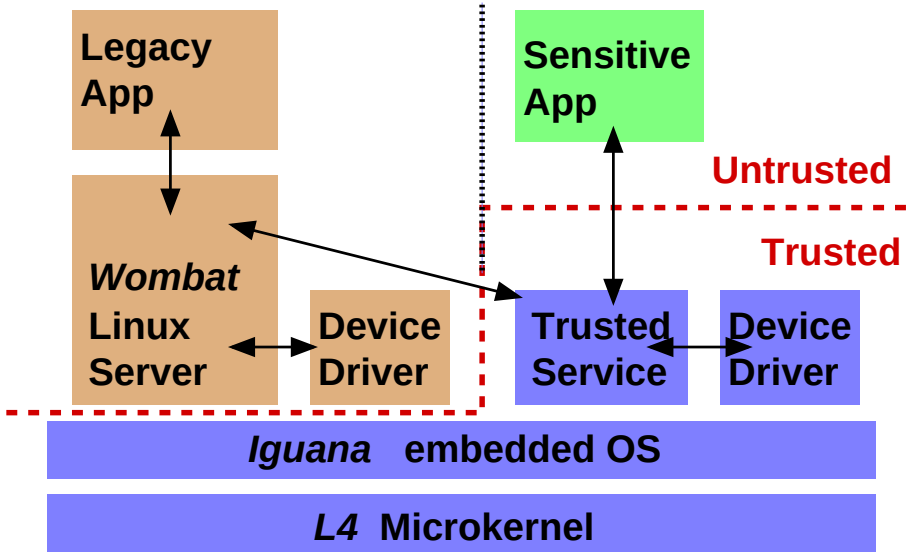
Present State

- Pistachio: Mature microkernel
  - ➜ 10,000 lines of code (shrinking)
  - ➜ highly efficient
- Iguana: Core OS services
  - ➜ naming, protection, memory...
  - ➜ device drivers
  - ➜ optional Linux server
- Multiple architectures
  - ➜ on ARM, MIPS, x86



Present State

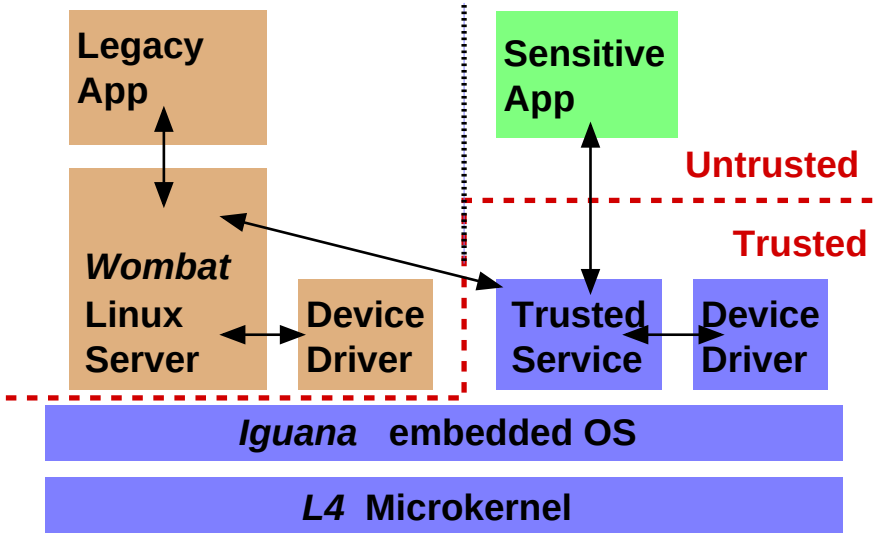
- Pistachio: Mature microkernel
  - ➜ 10,000 lines of code (shrinking)
  - ➜ highly efficient
- Iguana: Core OS services
  - ➜ naming, protection, memory...
  - ➜ device drivers
  - ➜ optional Linux server
- Multiple architectures
  - ➜ on ARM, MIPS, x86
- Commercially deployed
  - ➜ new base of Qualcomm CDMA chip firmware
  - ➜ other deployments in pipeline



System Performance

Benchmark	Native Linux	Wombat
1 Task	47.5	46.3
2 Tasks	24.8	24.1
3 Tasks	16.7	16.3

- AIM7 "compute server mix", jobs/min/task
- All on PLEB2 (Intel PXA 255 XScale, 200MHz CPU, 100MHz RAM)



TRUSTWORTHY COMPUTER SYSTEMS — A DREAM?

- Maybe

TRUSTWORTHY COMPUTER SYSTEMS — A DREAM?

- Maybe, but
- A trustworthy TCB is a starting point and seems doable
- Prerequisite: small TCB, small kernel

Trustworthy Computer Systems — A Dream?

- Maybe, but
- A trustworthy TCB is a starting point and seems doable
- Prerequisite: small TCB, small kernel
- We are on track to deliver a trustworthy TCB
- ... without sacrificing performance
- ... usable in real systems