Secure Operating Systems

Gernot Heiser
NICTA and UNSW and Open Kernel Labs
Sydney, Australia

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

NICTA Members

NICTA Partners

Sydney

NICTA

Overview

- *Operating systems security overview*
- Types of secure systems
- Security policies
- Security mechanisms
- Design principles
- OS security verification
- OS design for security

Secure Operating System

NICTA

- Provides for secure execution of applications
- Must provide security policies that support the users' security requirements
- Must enforce those security policies
- Must be safe from tampering etc.

Security Policies

- Security policy
  - Specifies *allowed* and *disallowed states* of a system
  - OS needs to ensure that no disallowed state is ever entered
  - OS *mechanisms* prevent transitions from allowed to disallowed states
- Security policy needs to identify the *assets* to be secure
  - For computer security, assets are typically *data*
- Perfect security is generally unachievable
  - Need to be aware of *threats*
  - Need to understand what *risks* can be tolerated

Security Mechanisms

NICTA

- Used to enforce security policy
  - Computer access control (login authentication)
  - Operating system file access control system
  - Controls implemented in tools
- Example:
  - Policy: only accountant can access financial system
  - Mechanism: on un-networked computer in locked room with only one key
- A *secure system* provides mechanisms that ensure that violations are
  - Prevented
  - Detected
  - Recovered from

Trust

- Systems always have *trusted entities*
  - Hardware, operating system, sysadmin
- Totally of trusted entities is the *trusted computing base* (TCB)
  - The part of the system that can circumvent security
- A *trusted system* can be used to process security-critical assets
  - Gone through some process ("*assurance*") to establish its trustworthiness
  - Should really be called *trustworthy system*
- *Trusted computing*:
  - Provides mechanisms and procedures for trusted systems
  - In practice usually refers to TCG mechanisms for secure boot, encryption etc

Trusted Computing Base

- TCB: *The totality of protection mechanisms within a computer system — including hardware, firmware and software — the combination of which is responsible for enforcing a security policy*

  [RFC 2828]

  A TCB consists of one or more components that together enforce a unified security policy over a product or system

  The ability of the TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct inputs by system administrative personnel or parameters related to the security policy

Covert Channels (Side Channels)

NICTA

- Information flow that is not controlled by a security mechanism
  - Security requires *absence of covert channels*
- Two types of covert channels
  - Covert *storage* channel uses an attribute of a shared resource
    - Typically meta data, like existence or accessibility of an object
    - Global names create covert storage channels
    - In principle subject to access control
    - A sound access-control system should be *free* of covert channels
  - Covert *timing* channel uses temporal order of accesses to shared resource
    - Outside access-control system
    - Difficult to reason about
    - Difficult to prevent

Covert Timing Channels

NICTA

- Created via shared resource whose behaviour can be monitored
  - Network bandwidth
  - CPU load
  - Response time
  - Locks
- Requires access to a time source
  - Real-time clock
  - Anything else that allows unrelated processes to synchronise
  - Preventable by perfect virtualisation?
- Critical issue is bandwidth
  - In practice, the damage is limited if the bandwidth is low
    - e.g DRM doesn't care about low-bandwidth channels
  - Beware of amplification
    - e.g leaking of passwords

Establishing Trustworthiness

NICTA

- Process to show *TCB is trustworthy*
- Two approaches
  - *Assurance* (systematic evaluation and testing)
  - *Formal verification* (mathematical proof)
- *Certification* confirms process was successfully concluded

Assurance

- Process for *bolstering* (substantiating or specifying) trust
  - Specifications
    - Unambiguous description of system behaviour
    - Can be formal (mathematical model) or informal
  - Design
    - Justification that it meets specification
    - Mathematical translation of specification or compelling argument
  - Implementation
    - Justification that it is consistent with the design
    - Mathematical proof or code inspection and rigorous testing
    - By implication must also satisfy specification
  - Operation and maintenance
    - Justification that system is used as per assumption in specification
- Assurance does not *guarantee* correctness or security!

Assurance: Orange Book

NICTA

US Department of Defence "Orange Book" [DoD 86]:

- Officially the *Trusted Computing Systems Evaluation Criteria* (TCSEC)
- Defines security classes
  - D: minimal protection
  - C1-2: discretionary access control (DAC)
  - B1-B3: mandatory access control (MAC)
  - A1: verified design
- Designed for military use
- Systems can be certified to a certain class
  - Very costly, hence only available for big companies
  - Most systems only certified C2 (essentially Unix-style security)
- Superseded by *Common Criteria*
  - Orange book no longer has any official standing
  - However, still an excellent reference for security terminology and rationale

Assurance: Common Criteria

NICTA

Common Criteria for IT Security Evaluation [ISO/IEC 15408]:

- ISO standard, developed out of Orange Book and other approaches
  - US, Canada, UK, Germany, France, Netherlands
  - For general use (not just military, not just operating systems)
- Unlike Orange Book, doesn't prescribe specific security requirements
  - Evaluates quality assurance used to ensure requirements are met
- *Target of evaluation* (TOE) evaluated against *security target* (ST)
  - ST is statement of desired security properties
  - Based on *protection profiles* (PPs) — generic sets of requirements
    - Defined by “users” (typically governments)
- Seven *evaluation assurance levels* (EALs)
  - Higher levels imply more thorough evaluation (and higher cost)
  - *Not* necessarily better security
- Details later

Formal Verification

- Process of mathematical proof of security properties
- Mased on a mathematical *model* of the system
- Two Parts:
  - Proof that *model satisfies security requirements*
    - Generally difficult, except for very simple models
  - Proof that *code implements model*
    - Proving theorems showing correspondence
    - Even harder, feasible only for few 1000 LOC
    - Hardly ever done
- Note: *model checking* (static analysis) is not sufficient
  - Shows presence or absence of certain properties of code
    - Uninitialised variables, array-bounds, null-pointer de-ref.
  - Does not prove implementation correctness

Summary

- Computer security is complex
  - Depends on many aspects of computer system
- Policy defines security, mechanisms enforce security
- Important to consider:
  - What are the assumptions about threats and trustworthiness?
  - Incorrect assumptions ⇒ no security
- Security is never absolute
  - Given enough resources, mechanisms can be defeated
  - Important to understand limitations
  - Inherent tradeoffs between security and usability
- Human factors are important
  - People make mistakes
  - People may not understand security impact of actions
  - People may be less trustworthy than thought

Overview

- Operating systems security overview
- *Types of secure systems*
- Security policies
- Security mechanisms
- Design principles
- OS security verification
- OS design for security

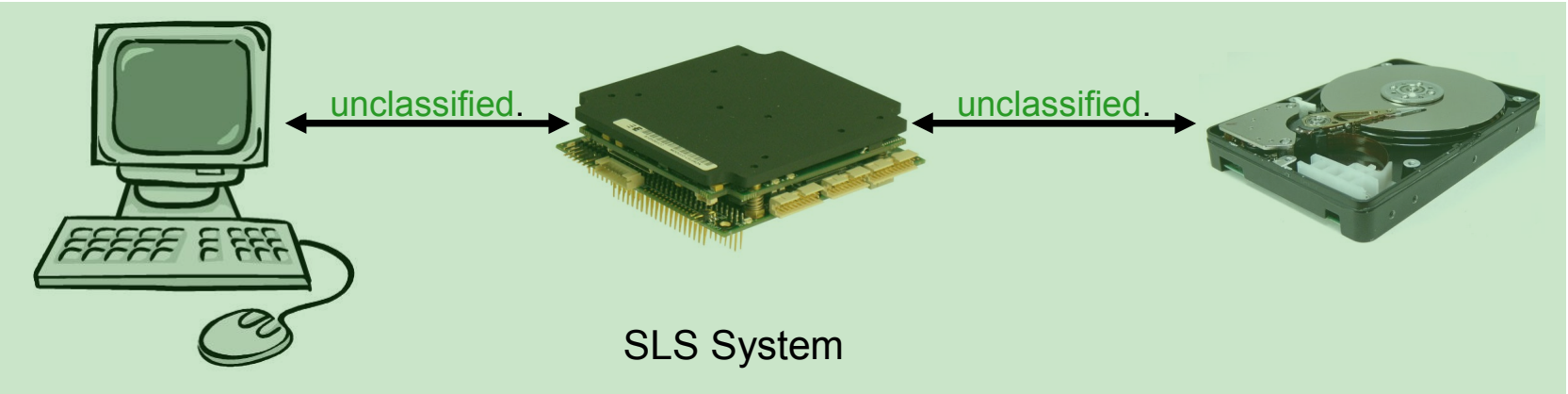
Secure Systems Classification

NICTA

- Based on Orange Book terminology
  - Assumes military-style security problem
  - Data of different security classifications
  - System must ensure that classification is enforced
- Classifies systems based on the kind of data they can deal with
  - *Single-level secure* (SLS) system
  - *Multiple single-level secure* (MSL) system
  - *Multi-level secure* (MLS) system
- Basis of *multiple-independent levels of security* (MILS) architecture

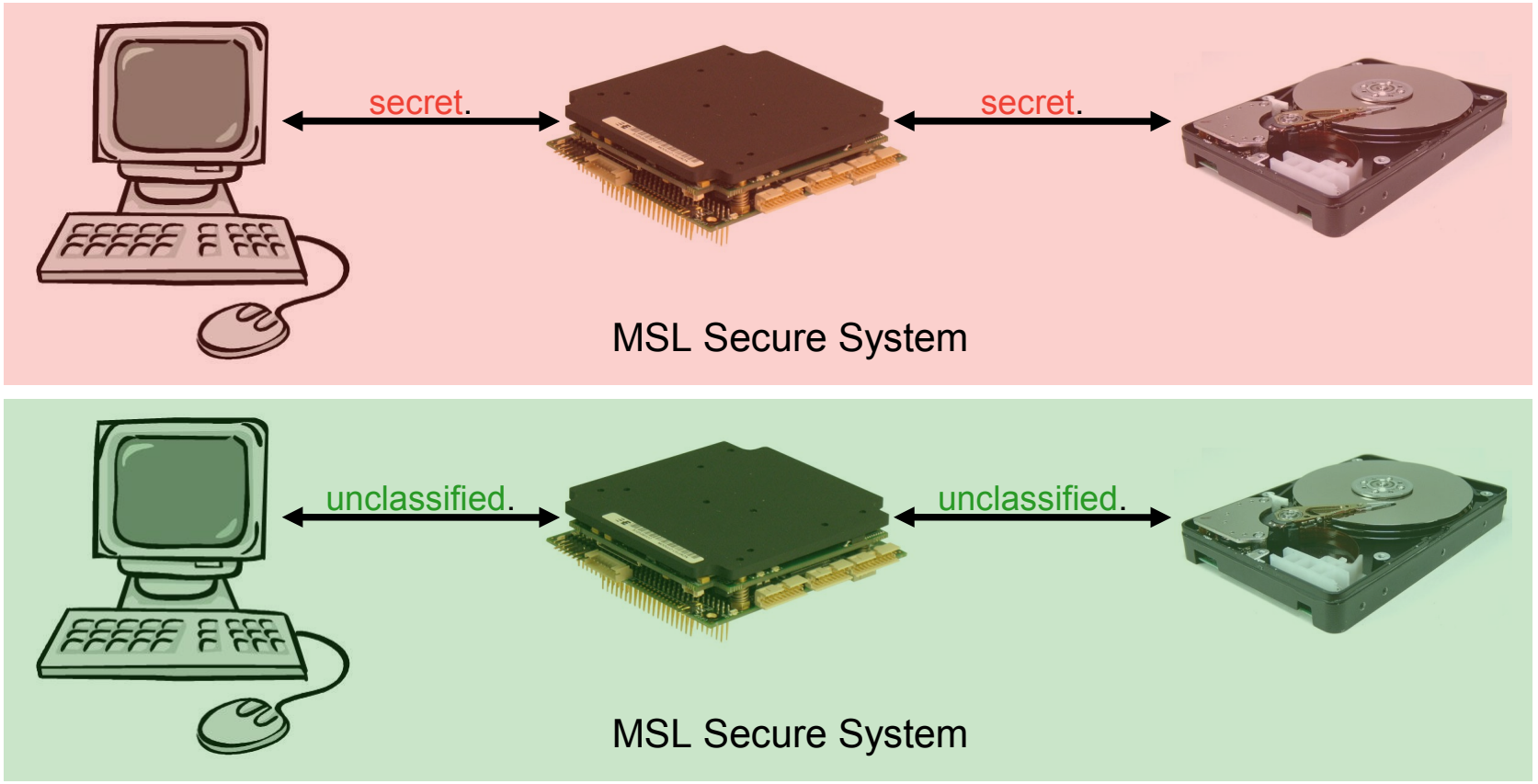
Single-Level Secure (SLS) System

- Suitable only for processing data of one particular security level
  - generally the lowest, i.e. unclassified



Multiple Single-Level (MSL) Secure System

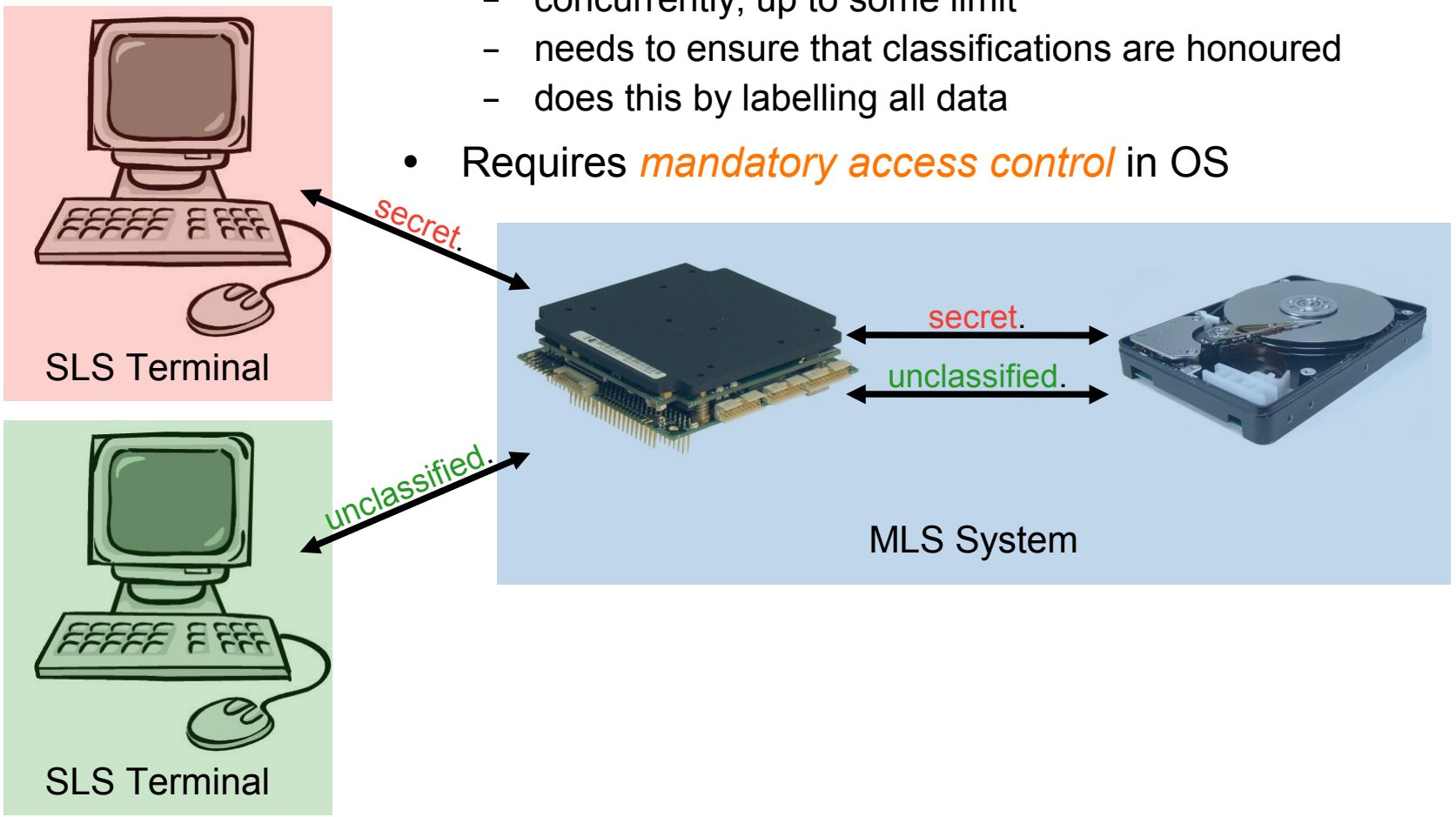
- System suitable for processing data of several security levels
  - only one security level at a time, up to some limit
- Multiple instances used, each one as a SLS system



Multi-Level Secure (MLS) System

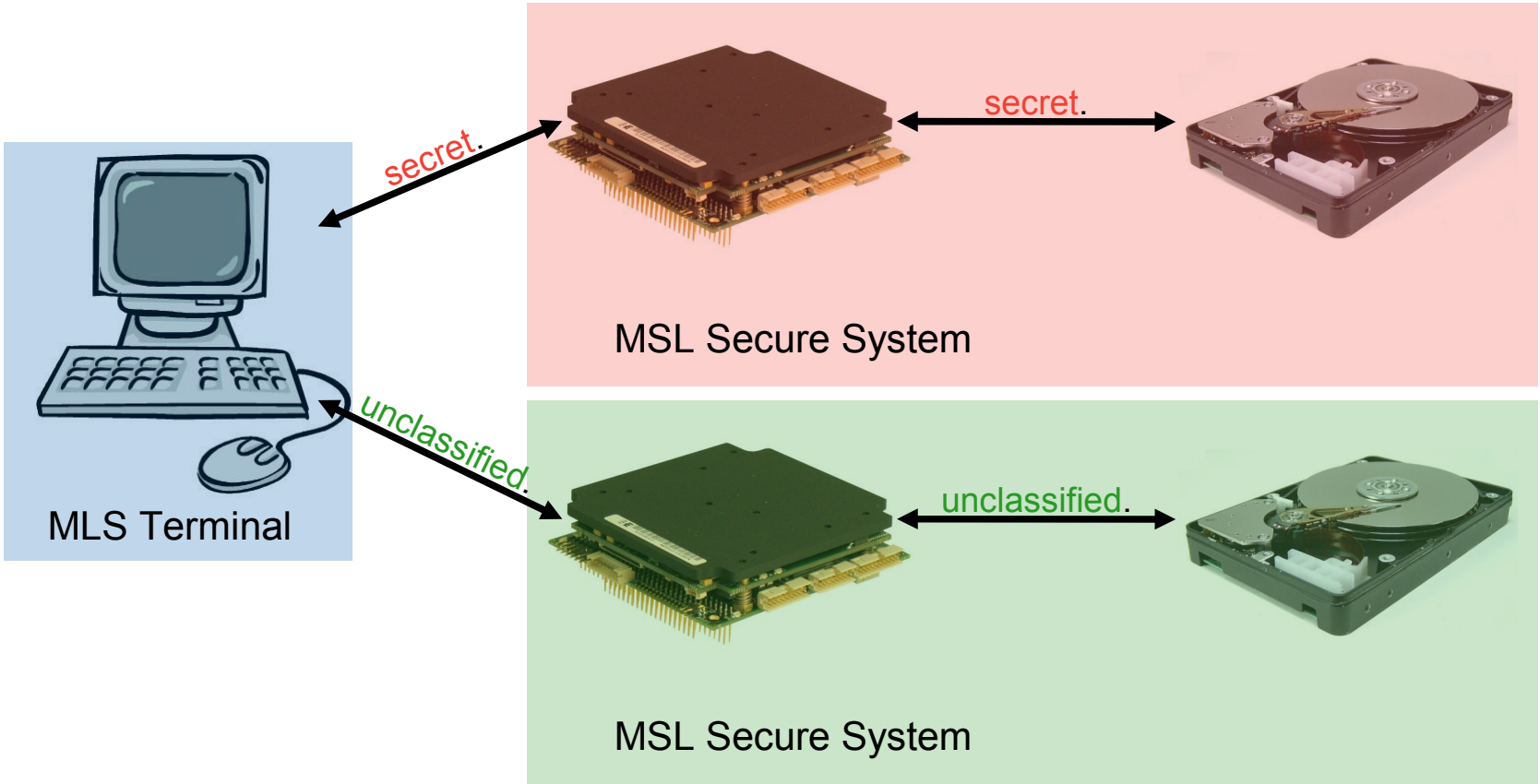
NICTA

- Suitable for processing data of several security levels
  - concurrently, up to some limit
  - needs to ensure that classifications are honoured
  - does this by labelling all data
- Requires *mandatory access control* in OS



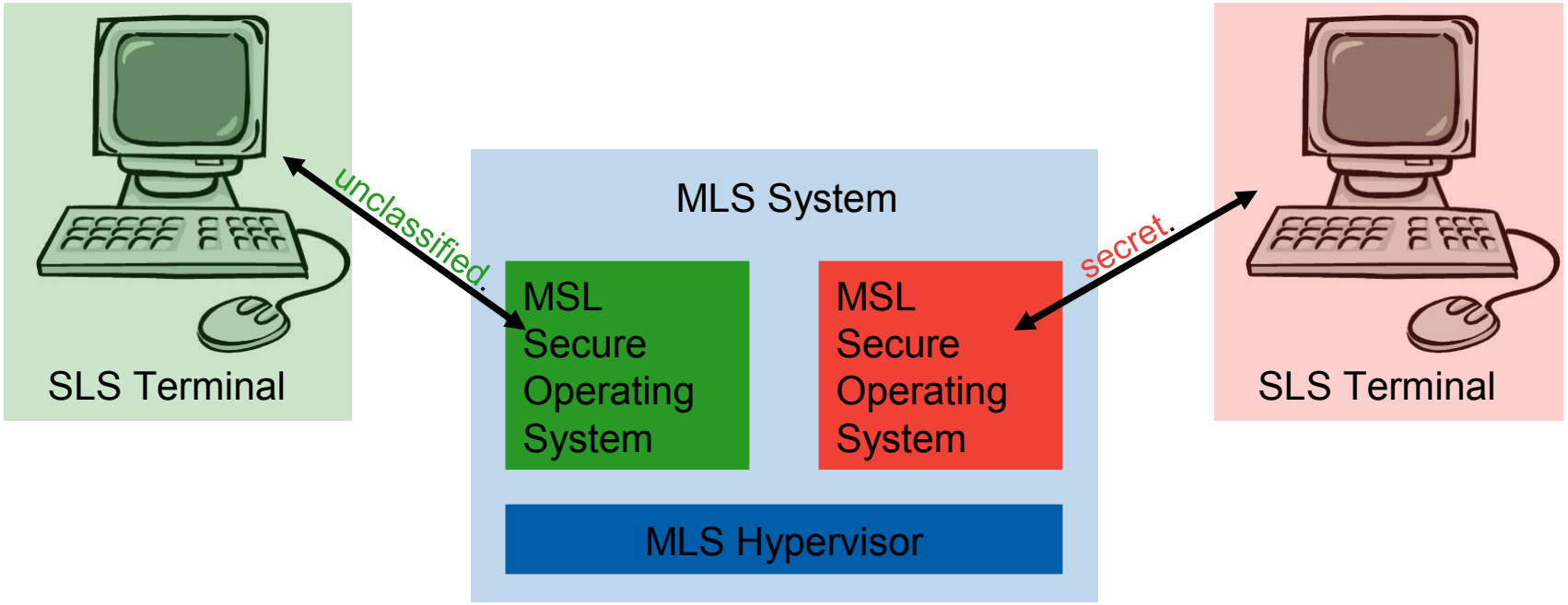
MLS + MSL System

- MLS component handles multiple levels of data
- Only a single level of data goes to each of the MSL secure systems



MLS System Using Virtualization

- MLS hypervisor runs several MSL secure OSes in individual virtual machines
- Result is MLS system
- An example of a *multiple independent levels of security* (MILS) architecture
  - Hypervisor here operates as a *separation kernel*
  - Separates (isolates) different *security domains*



Overview

- Operating systems security overview
- Types of secure systems
- *Security policies*
- Security mechanisms
- Design principles
- OS security verification
- OS design for security

Security Policies: Categories

- *Discretionary* (user-controlled) policies (DAC)
  - e.g A can read B's objects only with A's permission
  - User decides about access (at their discretion)
  - Classical example: Unix permissions
- *Mandatory* (system-controlled) policies (MAC)
  - e.g certain users cannot ever access certain objects
  - No user can change these
  - Focus on restricting *information flow*
  - Inherent requirement for MLS systems, MILS
- *Role-based* policies (RBAC)
  - Agents can take on specific pre-defined roles
    - Well-defined set of roles for each agent
    - e.g normal user, sysadmin, database admin
  - Access rights depend on role

Models for Security Policies

NICTA

- Represent a whole class of security policies
- Most system-wide policies focus on confidentiality
  - e.g military-style multi-level security models
- Classical example is *Bell-LaPadula* model [BL76]
  - Example of a *labelled security model*
  - Most others developed from this
  - Orange Book based on this model
- Other models
  - *Chinese-wall* policy focuses on conflict of interest
  - *Clark-Wilson* model focuses on separation of duty

Bell-LaPadula Model

- Each object a has a security *classification* $L(a)$
- Each agent o has a security *clearance* $L(o)$
- Classifications
  - e.g top secret > secret > confidential > unclassified
- Rule 1 (*no read up*):
  - A can *read* o only if $L(a) \geq L(o)$
  - Standard confidentiality
- Rule 2 (✸ *Property — no write down*)
  - A can *write* o only if $L(a) \leq L(o)$
  - Prevents *leakage* (accidental of by conspiracy)

Bell-LaPadula Model

NICTA

- Mother of all military-style security models
- Inherently requires implementation as MAC
  - All subjects must be bound to policy
- If implemented inside a single system, requires MLS system
- Major limitation: cannot deal with *declassification*
  - Needed to pass any information from high- to low-security domain
    - Logging
    - Command chain
    - Documents where sensitive portions have been censored
    - Encrypted data
- Typically dealt with by special *privileged functions*
  - Outside security policy
  - Outside systematic reasoning
  - Part of TCB
  - Likely source of security holes

Overview

- Operating systems security overview
- Types of secure systems
- Security policies
- *Security mechanisms*
- Design principles
- OS security verification
- OS design for security

Security Mechanisms

NICTA

- Used to implement security policies
- Based on access control
  - Discretionary access control (DAC)
  - Mandatory access control (MAC)
  - Role-based access control (RBAC)
- Access rights
  - *Simple rights*
    - Read, write, execute/invoke, send, receive
  - *Meta rights* (DAC only)
    - Copy
      - Propagate own rights to another agent
    - Own
      - Change rights of an object or agent

Access Control Matrix

	Objects			
Agents	S_1	S_2	O_3	O_4
S_1	terminate	wait, signal, send	read	
S_2	wait, signal, terminate			read, execute, write
S_3		wait, signal, receive		
S_4	control		execute	write

Defines each agent's rights on any object

Note: agents are objects too

Properties of the Access Control Matrix

NICTA

- Rows define agents' *protection domains (PDs)*
- *Columns* define objects' *accessibility*
- Dynamic data structure:
  - Frequent permanent changes (e.g. object creation, `chmod`)
  - Frequent temporary changes (e.g. `setuid`)
- Very *sparse* with many repeated entries
- Impractical to store explicitly

Protection-Matrix Implementation: ACLs

NICTA

Represent column-wise: access control list (ALC):

- *ACL* associated with *object*
- Usually condensed via *domain classes* (UNIX, NT groups)
- Full ACLs used by Multics, Apollo Domain, Andrew FS, NTFS
- Can have *negative rights* to:
  - Reduce window of vulnerability
  - Simplify exclusion from groups
- Sometimes implicit (Unix process hierarchy)
- Implemented in almost all commercial systems

Protection-Matrix Implementation: Capabilities

NICTA

Represent row-wise: capabilities [DV 66]:

- *Capability List* associated with agent
  - Each capability confers a certain right to its holder
- Can have *negative rights* to:
  - Reduce window of vulnerability
  - Simplify management of groups of capabilities
- Caps have been popular in research for a long time
- Few successful commercial systems until recently:
  - main one is IBM System/38 / AS400 / i-Series
  - increasingly appearing in commercial systems (usually add-on)

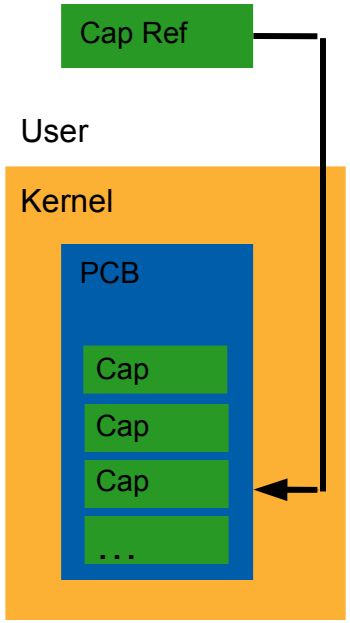
Capabilities

NICTA

- Main advantage of capabilities is the *fine-grained access control*:
  - Easy to provide specific agents access to individual objects
- Capability presets *prima facie* evidence of the *right to access*
  - Capability ⇒ *object identifier* (implies naming)
  - Capability ⇒ (set of) *access rights*
    - Any representation must contain object ID and access rights
    - Any representation must protect capability from forgery
- How are caps implemented and protected?
  - *Tagged* — protected by hardware
    - Popular in the past, rarely today (exception: IBM i-Series)
  - *Sparse* (or *user-mode*) — protected by sparsity
    - probabilistically secure, like encryption
    - propagation outside system control — hard to enforce security policies
  - *Partitioned*/*segregated* — protected by software (kernel)
    - main version of caps used in modern systems

Segregated Capabilities

- System maintains *capability list* (Clist) with each agent (process)
  - User code uses indirect references to caps (clist index)
    - c.f Unix file descriptors
  - System validates permissions on access
    - syscall or page-fault time
- Many research systems
  - Hydra, Mach, EROS, and many others
- Increasingly commercial systems
  - KeyKOS (92), OKL4 (08)
  - Add-on to Linux, Solaris



Confinement

- Problem 1: Executing untrusted code
  - You downloaded a game from the internet
  - How can you be sure it doesn't steal/corrupts your data?
- Problem 2: Digital rights management (DRM)
  - You own copyrighted material (e.g. entertainment media content)
  - You want to let others use it (for a fee)
  - How can you prevent them from making unauthorised copies?
- You need to *confine* the program (game, viewer) so it cannot leak
- Cannot be done with most protection schemes!
  - Not with Unix or most other ACL-based schemes
  - Not with most tagged or sparse capability schemes
  - Multi-level security has some inherent confinement (but can't do DRM)
- Some protection models can confine in principle
  - e.g segregated caps system, can instruct system not to accept any
  - EROS has formal proof of confinement of a model of the system [SW00]
  - Similar seL4 (machine-checked proof)
- In practice difficult to achieve due to *covert channels*

Overview

- Operating systems security overview
- Types of secure systems
- Security policies
- Security mechanisms
- *Design principles*
- OS security verification
- OS design for security

Design Principles for Secure OS

NICTA

- Least privilege (POLA)
- Economy of mechanisms
- Fail-safe defaults
- Complete mediation
- Open design
- Separation of privilege
- Least common mechanisms
- Psychological acceptability

Least Privilege

NICTA

- Also called the *principle of least authority* (POLA)
- Agent should only be given the minimal rights needed for task
  - Minimal protection domain
  - PD determined by *function*, not *identity*
    - Unix *root* is evil
    - Aim of role-based access control (RBAC)
  - Rights added as needed, removed when no longer needed
  - Violated by all mainstream OSes
- Example: executing web applet
  - Should not have all of user's privileges, only minimal access
  - Hard to do with ACL-based systems
  - Main motivation for using caps

Least Privilege: Implications for OS

NICTA

- OS kernel executes in privileged mode of hardware
  - Kernel has unlimited privilege!
- POLA implies keeping kernel code to an absolute minimum
  - This means a secure OS must be based on a microkernel!
- Trusted computing base can bypass security
- POLA requires that TCB is minimal
  - Microkernel plus minimal security manager

Economy of Mechanisms

NICTA

- KISS principle of engineering
  - “keep it simple, stupid!”
- Less code/features/stuff ⇒ less to get wrong
  - Makes it easier to fix if something does go wrong
  - Complexity is the natural enemy of security
- Also applies to interfaces, interactions, protocols, ...
- Specifically applies to TCB

Fail-Safe Defaults

- Default action is no-access
  - If action fails, system remains secure
  - If security administrator forgets to add rule, system remains secure
  - “better safe than sorry”

Complete Mediation

NICTA

- Check every access
  - Violated in Unix file access:
    - Access rights checked at open(), then cached
    - Access remains enabled until `close()`, even if attributes change
  - Also implies that any rights propagation must be controlled
    - Not done with tagged or sparse capability systems
- In practice conflicts with performance!
  - Caching of buffers, file descriptors etc
  - Without caching unacceptable performance
- Should at least limit window of opportunity
  - e.g guarantee caches are flushed after some fixed period
  - Guarantee no cached access after revoking access

Open Design

- Security must not depend on secrecy of design or implementation
  - TCB must be open to scrutiny
  - *Security by obscurity is poor security*
    - Not all security/certification agencies seem to understand this
- Note that this doesn't rule out passwords or secret keys
  - But their creation requires careful *cryptoanalysis*

Separation of Privilege

NICTA

- Require a combination of conditions for granting access
  - e.g user is in group wheel *and* knows the root password
  - Take-grant model for capability-based protection:
    - Sender needs *grant* right on capability
    - Receiver needs *take* right to accept capability
  - In reality, the security benefit of a separate *take* right is minimal
    - Practical cap implementations only provide *grant* as a privilege
- Closely related to least privilege

Least Common Mechanisms

NICTA

- Avoid sharing mechanisms
  - Shared mechanism ⇒ shared channel
  - Potential covert channel
- Inherent conflict with other design imperatives
  - Simplicity ⇒ shared mechanisms
  - Classical tradeoff...

Psychological Acceptability

- Security mechanisms should not add to difficulty of use
  - Hide complexity introduced by security mechanisms
  - Ensure ease of installation, configurations, use
  - Systems are used by humans!
- Inherently problematic:
  - Security inherently inhibits ease of use
  - Idea is to minimise impact
- Security-usability tradeoff is to a degree unavoidable

Overview

- Operating systems security overview
- Types of secure systems
- Security policies
- Security mechanisms
- Design principles
- *OS security verification*
- OS design for security

Common-Criteria Protection Profiles

NICTA

- Controlled Access Protection Profile (CAPP)
  - standard OS security, derived from Orange Book C2
  - certified up to level EAL3
- Single-level Operating System Protection Profile
  - superset of CAPP
  - certified up to EAL4+
- Labeled Security Protection Profile (LSPP)
  - mandatory access control for COTS OSes
  - similar to Orange Book B1
- Role-based Access Control Protection Profile
- Multi-level Operating System Protection Profile
  - superset of CAPP, LSPP
  - certified up to EAL4+
- Separation Kernel Protection Profile (SKPP)
  - strict partitioning
  - certifications aiming for EAL6–7

Common Criteria Assurance Levels

NICTA

- EAL1: functionally tested
  - simple to do, can be done without help from developer
- EAL2: structurally tested
  - functional and interface spec
  - black- and white-box testing
  - vulnerability analysis
- EAL3: methodically tested and checked
  - improved test coverage
  - procedures to avoid tampering during development
  - highest assurance level achieved for Mac OS X

Common Criteria Assurance Levels

NICTA

- EAL4: methodically designed, tested and reviewed
  - Design docs used for testing, avoid tampering during delivery
  - Independent vulnerability analysis
  - Highest level feasible on existing product (not developed for CC certific.)
  - Achieved by a number of main-stream OSes
    - Windows 2000: EAL4 in 2003
    - SuSe Enterprise Linux: EAL4 in 2005
    - Solaris-10: EAL4+ in 2006
      - Controlled access protection profile (CAPP) — *Note: EAL3 profile!*
      - Role-based access control PP — *example of non-NSA PP?*
    - RedHat Linux EAL4+ in 2007
  - They still get broken!
    - Certification is based on assumptions about environment, etc...
    - Most use is outside those assumptions
      - Certification means nothing in such a case
      - Presumably there were no compromises were assumptions held

Common Criteria Assurance Levels

NICTA

- EAL5: semi-formally designed and tested
  - Formal model of TEO security policy
  - Semi-formal model of functional spec & high-level design
  - Semi-formal arguments about correspondence
  - Covert-channel analysis
  - IBM z-Series hypervisor EAL5 in 2003 (partitioning)
  - Attempted by Mandrake for Linux with French Government support
- EAL6: semiformally verified design and tested
  - Semiformal low-level design
  - Structured representation of implementation
  - Modular and layered TOE design
  - Systematic covert-channel identification
  - Green Hills Integrity microkernel presently undergoing EAL6+ certification
    - Separation kernel protection profile

Common Criteria Assurance Levels

NICTA

- EAL7: formally verified design and tested
  - Formal functional spec and high-level design
  - Formal and semiformal demonstration of correspondence
    - Between specification and low-level design
  - Simple TOE
  - Complete independent confirmation of developer tests
  - LynuxWorks claims LynxSecure separation kernel EAL7 “certifiable”
    - But not *certified*
  - Green Hills also aiming for EAL7

Note:

- *Even EAL7 relies on testing!*
- EAL7 requires proof of correspondence between formal descriptions
- However, no requirement of formalising LLD, implementation
- Hence no requirement for formal proof of implementation correctness

Common Criteria Limitations

NICTA

- Little (if any) use in commercial space outside national security
  - This was one of the intentions, by all indications, CC failed here
- Very expensive
  - Industry rule-of-thumb: EAL6+ costs $10k per LOC
  - Dominated by documentation requirements
  - No “credit” for doing things better
    - Eg formal methods instead of excessive documentation
- Lower EALs of limited practical use
  - Windows is EAL4+ certified!
  - Marketing seems to be main driver behind EAL3–4 certification
- Over-evaluation abuses system
  - Eg.CAPP (EAL3 profile) certification to EAL4
  - In reality a pointless exercise

Formal Verification

- Based on mathematical model of the system
- Complete verification requires two parts:
  - Proof that model satisfies requirements of security policies
    - Typically prove generic properties that actual policies map to
    - Required by CC EAL5–7
  - Proof that implementation has same properties as model
    - Proof of correspondence between model and implementation
    - Not required by CC even at EAL7
    - Done by some kernels with very limited functionality
    - Never done for any general-purpose OS!
- Model-checking (static analysis) is *incomplete* formal verification
  - Shows presence or absence of certain properties
    - e.g uninitialised variables, array-bounds overflows
  - Nevertheless useful for assurance

Common Criteria and Formal Verification

NICTA

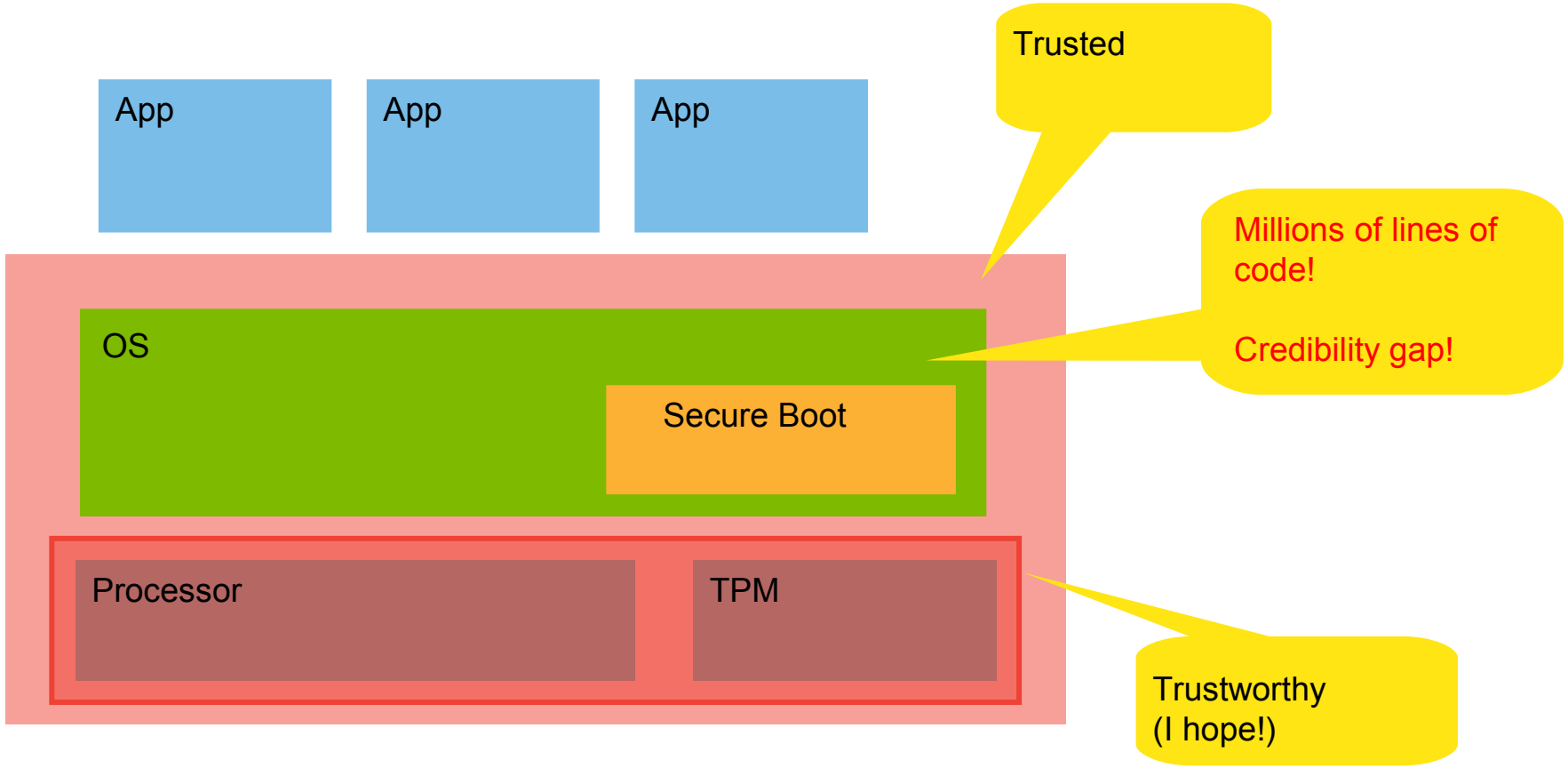
EAL	Requirem.	Funct Spec	HLD	LLD	Implem.
EAL 1	Informal	Informal	Informal	Informal	Informal
EAL 2	Informal	Informal	Informal	Informal	Informal
EAL 3	Informal	Informal	Informal	Informal	Informal
EAL 4	Informal	Informal	Informal	Informal	Informal
EAL 5	Formal	Semiformal	Semiformal	Informal	Informal
EAL 6	Formal	Semiformal	Semiformal	Semiformal	Informal
EAL 7	Formal	Formal	Formal	Semiformal	Informal

Trusted Computing vs Secure OS

NICTA

- TPM-based trusted-computing approach is based on
  - Hardware root of trust
  - Mechanisms to provide a chain of trust
- Objective is to guarantee that system boots into a well-defined configuration
  - Guarantees that a particular OS binary is running
  - What does this mean about security/trustworthiness?

Trusted Computing vs Secure OS



→ TPM-based trusted-computing approach is of limited use

- As long as the OS isn't trustworthy

Overview

- Operating systems security overview
- Types of secure systems
- Security policies
- Security mechanisms
- Design principles
- OS security verification
- *OS design for security*

OS Design for Security

NICTA

- Minimize kernel code
  - Kernel = code that executes in privileged mode
  - Kernel can bypass any security
  - Kernel is inherently part of TCB
  - Kernel can only be verified as a whole (not in components)
    - It's hard enough to verify a minimal kernel
- How?
  - Generic mechanisms (economy of mechanisms)
  - No policies, only mechanisms
  - Mechanisms as simple as possible
  - Only code that must be privileged in order to support secure systems
  - Free of covert channels:
    - No global names, absolute time
- Formally specify API

OS Design for Security

NICTA

- Minimize mandatory TCB
  - Unless formally verified, TCB must be assumed imperfect
  - The smaller, the fewer defects
  - POLA requires, economy of mechanisms leads to minimal TCB
- Ensure TCB is well defined and understood
  - Make security policy explicit
  - Make granting of authority explicit
- Flexibility to support various uses
  - Make authority delegatable
  - Ensure mechanisms allow high-performance implementation
- Design for verifiability
  - Minimize implementation complexity

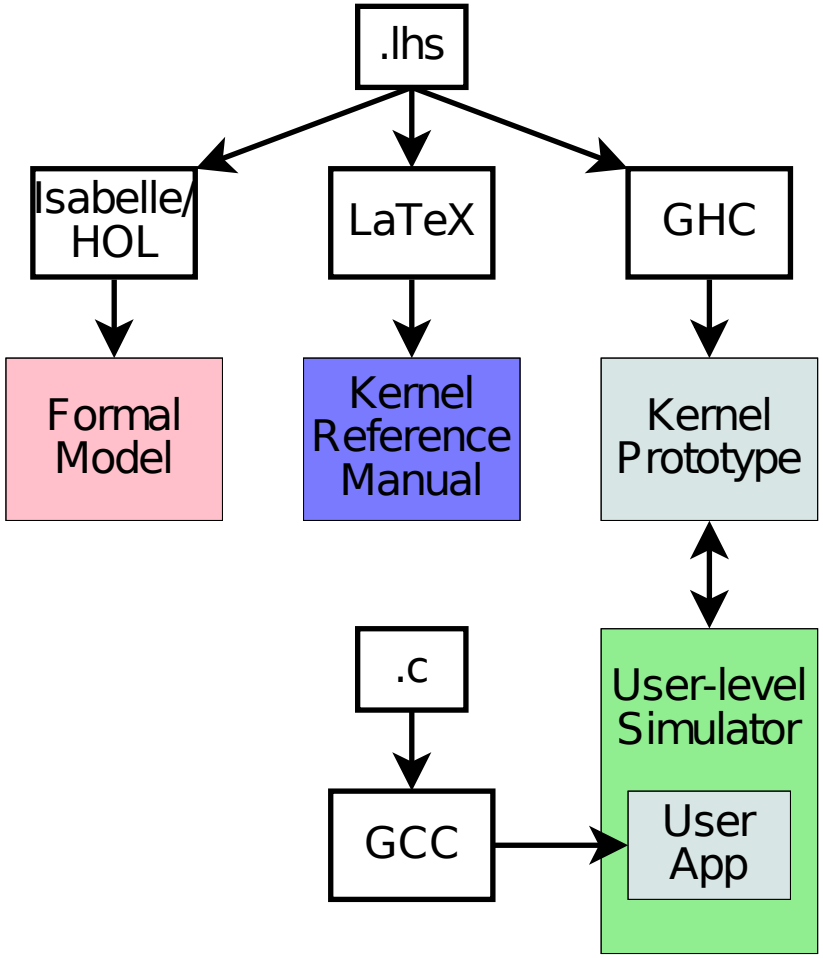
Example: NICTA's seL4

NICTA

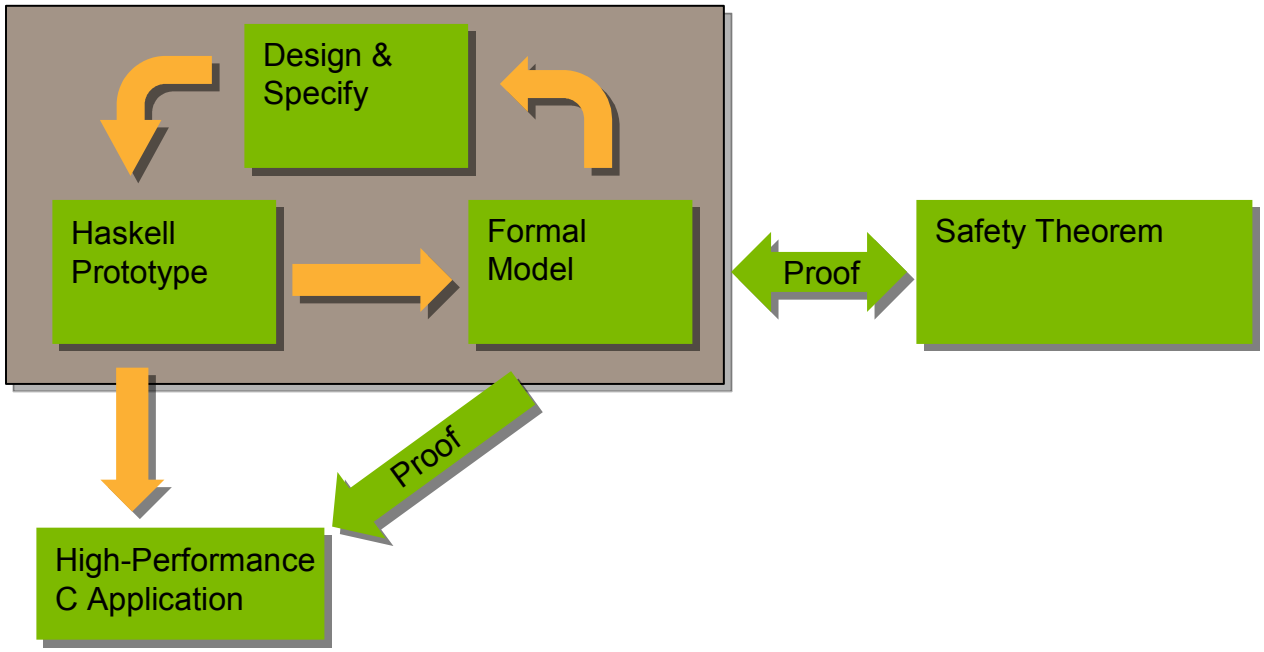
- High-security version of L4 microkernel API
  - All authority granted by capabilities
  - Only four system calls: read, write, create, derive
  - Kernel memory explicitly managed by user-level resource manager
  - 7,000–10,000 lines of kernel code
- Semi-formal API spec in Haskell
  - Easily formalised in theorem prover
  - Machine-checked proofs of security properties
  - Designed for formal verification, to be finished mid-2008

Kernel Prototyping in Haskell

- Model the kernel in detail
- Literate Haskell to model
  - Pure functional programming language
  - Embedded documentation
  - Close to Isabelle/HOL
  - Formalized Haskell becomes intermediate representation for refinement proof
- Executable
  - Supports running user-level code

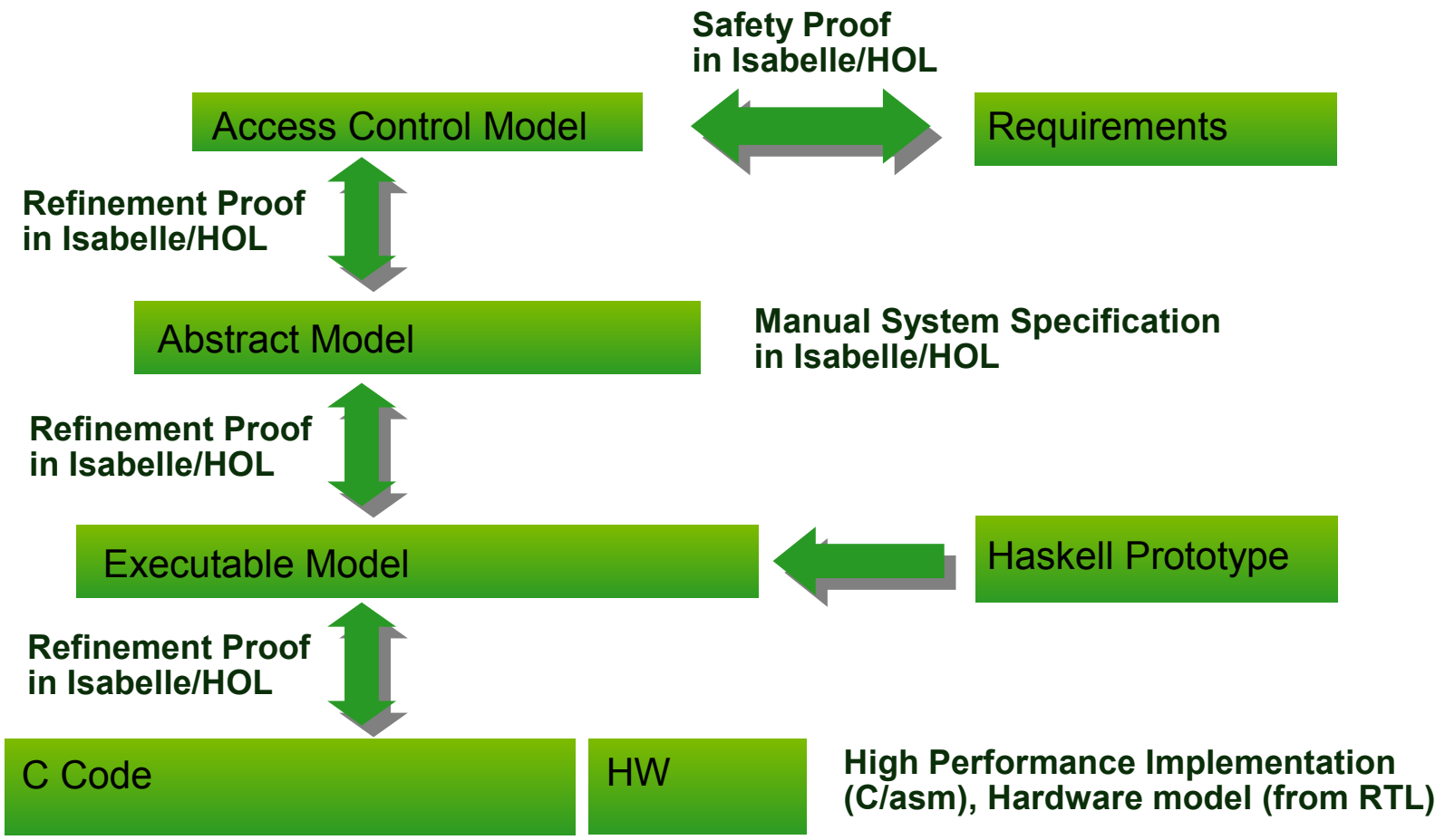


Iterative Design and Formalisation



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

seL4 Correctness Proof



seL4 Formal Verification

NICTA

Aim: Complete proof chain from security requirements to implementation

- Running since January 2004
- Achieved to date:
  - Formal, machine-checked proofs of safety properties (isolation)
  - Formal, machine-checked proof of concrete spec satisfying the abstract spec
    - In CC language: formally-verified high-level design
  - Formal, machine-checked proof that executable model refines spec
    - In CC language: formally-verified low-level design
  - Already most formally-analysed general-purpose OS ever
- In progress:
  - Formal, machine-checked proof that implementation refines spec
    - In CC language: formally-verified implementation
  - To be completed by December 2008
- *You want trusted virtualization — you've got it!*

NICTA

From **imagination** to **impact**