NICTA

Secure Operating Systems

Gernot Heiser
NICTA & UNSW & Open Kernel Labs

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

NICTA Members
ANU THE AUSTRALIAN NATIONAL UNIVERSITY
UNSW THE UNIVERSITY OF NEW SOUTH WALES
Department of State and Regional Development
NICTA Partners
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

Overview

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

Secure Operating System

- 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 tranistions from allowed to disallowed states
- Security policy needs to identify the *assets* to be secured
  - 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

Data Security

NICTA

Three aspects:

- **Confidentiality**: prevent *theft* of data
  - concealing data from unauthorised agents
  - need-to-know principle
- **Integrity**: prevent *damage* of data
  - trustworthiness of data: data *correctness*
  - trustworthiness of origin of data: *authentication*
- **Availability**: prevent *denial* of service
  - ensuring data is usable when needed

Security Policy

- Partitions system into allowed and disallowed states
- Ideally mathematical model
- In practice, usually natural-language description
  - often imprecise, ambiguous, inconsistent, unenforceable
  - Example: transactions over $10k require manager approval
    - but transferring $10k into own account is no violation

Security Mechanisms

- 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
- Assumed to be *trustworthy*
  - is it???

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

- 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 exisstence 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

- 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 rigorous testing
    - by implication must also satisfy specification
  - Operation and maintenance
    - justification that system is used as per assumptions in specification
- Assurance does not *guarantee* correctness or security!

Assurance: Orange Book

NICTA

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

- 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

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

NICTA

- 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
    - uninitialilsed 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
- **Security policies**
- OS security verification
- Security mechanisms
- Design principles
- 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*
- *Role-based* policies (RBAC)
  - agents can take on specific pre-defined roles
    - well-defined set of roles for each agent
    - eg normal user, sysadmin, database admin
  - access rights depend on role

Security Policy Models

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]
  - most others developed from this
  - Orange Book based on this model
- Other models
  - *Chinese-wall* policy focusses on conflict of interest
  - *Clark-Wilson* model focusses on separation of duty

Bell-LaPadula Model

NICTA

- Each object a has a security *classification* $L(a)$
- Each agent o has a security *clearance* $L(o)$
- Classifications and clearances form hierachical *security levels*
  - 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 (*$\star$ Property — no write down*)
  - a can *write* o only if $L(a) \leq L(o)$
  - prevents *leakage* (accidental of by conspiracy)
  - problems:
    - logging
    - command chain
  - need way to *de-classify* data

Bell-LaPadula Extensions

- Can combine with discretionary access rights
  - read/write permissions on specific objects
  - e.g. SELinux
- Can add orthogonal security categories indicating types of data
  - restrict access to relevant categories
  - Denning's *lattice model* [Den76]

Chinese Wall Policy

NICTA

- Employed by investment banks to manage conflict of interest
- Idea: Consultant cannot talk to clients' competitors
  - single consultant can have multiple concurrent clients
- Define *conflict classes* (groups of potentially competing clients)
  - eg banks, oil companies, insurance companies, OS vendors
- Consultant dealing with client of class A cannot talk to others in A
  - but can continue talking to members of other classes
  - some data belongs to several conflict classes
- Public information is not restricted
  - consultant can read and write public info at any time
  - but must observe $\star$ property (cannot publish confidential info)
- Example of a *dynamic MAC policy*
  - allowed information flow changes over time
  - needs a rule for removing conflicts (after deal is done)

Overview

- Operating systems security overview
- Security policies
- **OS security verification**
- Security mechanisms
- Design principles
- 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+

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 main-stream Oses
    - Windows 2000: EAL4 in 2003
    - SuSe Enterprise Linux: EAL4 in 2005
    - Solaris-10 EAL4+ in 2006
      - controlled access protection profile (CAPP)
      - role-based access control 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: semiformally designed and tested
  - formal model of TEO security policy
  - semi-formal model of functional spec & high-level design
  - semi-formal argument 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
  - independent vulnerability analysis
  - 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 implementation
  - Hence no requirement for formal proof of implementation correctness

Formal Verification

NICTA

- 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 gereral-purpose OS!
- Model-checking (static analysis) is *incomplete* formal verification
  - shows presence or absence of certain properties
    - eg uninitialised variables, array-bounds overflows
  - nevertheless useful for assurance

Overview

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

Security Mechanisms

- 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 Access Control Matrix

NICTA

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

Issues for Protection System Design

NICTA

- Propagation of rights:
  - Can agent grant access to other?
- Restriction of rights:
  - Can agent propagate subset of own rights?
- Revocation of rights:
  - Can access, once granted, be revoked?
- Amplification of rights:
  - Can unprivileged agent perform restricted operations?
- Determination of object accessibility
  - Which agents have access to particular object?
  - Is object accessible at all (garbage collection)?
- Determination of agent's protection domain
  - Which objects are accessible?

Access Matrix Implementation: ACLs

NICTA

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

- *ACL* associated with *object*
  - Propagation: meta right (e.g. owner can `chmod`)
  - Restriction: meta right
  - Revocation: meta right
  - Amplification: protected-invocation right (e.g. `setuid`)
  - Accessibility: explicit in ACL
  - Protection domain: hard (if not impossible) to determine
- 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

Access Matrix Implementation: Capabil

NICTA

Represent row-wise: *capabilities*:

- *Capability list* associated with agent
- Each capability confers a certain right to its holder
  - Propagation: copy capabilities between agents (how?)
  - Restriction: lesser rights require creation of new (derived) caps
  - Revocation: requires invalidation of caps from all agents
  - Amplification: special invocation capability
  - Accessibility: requires inspection of all capability lists (how?)
  - Protection domain: explicit in capability list
- Can have *negative rights* to:
  - reduce window of vulnerability
  - simplify management of groups of capabilities
- Only successful commercial systems: IBM System/38 / AS400 / i-Series

Capabilities

- Main advantage of capabilities is the fine-grained access control:
  - easy to provide access to specific agents
- Capability presets *prima facie* evidence of the *right to access*
  - capability ═► *object identifier* (imples 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
  - partitioned/segregated — protected by software
  - sparse — protected by sparsity (probabilistically secure, like encryption)

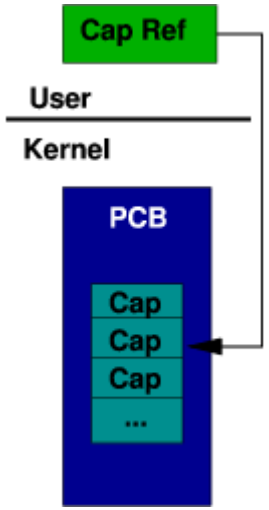
Tagged Capabilities

NICTA

- *Tag bit(s)* with every (group of) memory word(s)
  - tag identifies capabilities
  - capabilities are used and copied like “normal” pointers
  - hardware checks permissions when deferencing capability
  - modifications turn tags off (convert to plain data)
  - Only privileged instructions(kernel) can turn tags on
  - Properties:
    - ➔ propagation easy
    - ➔ restriction requires kernel to make new capability
    - ➔ revocation virtually impossible (requires memory scan)
    - ➔ amplification possible (below)
    - ➔ accessibility virtually impossible to determine
    - ➔ protection domain difficult to establish
- IBM System/38, AS/400, i-Series, many historical systems

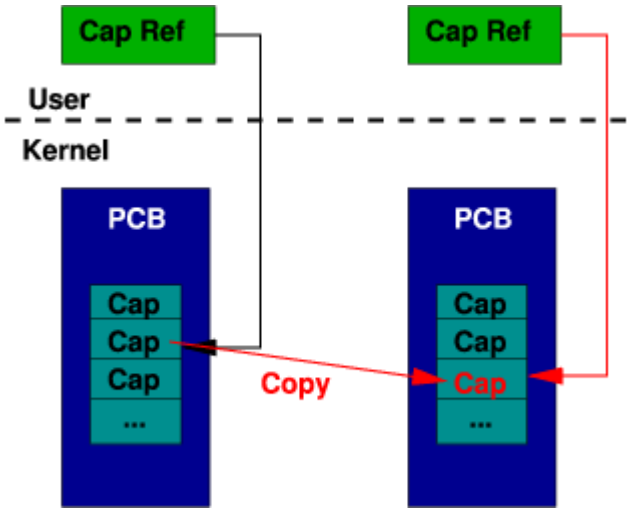
Partitioned Capabilities

- System maintains capability list (Clist) with each process
  - user code uses indirect references to caps (clist index)
    - c.f. Unix file descriptors
  - System validates access via clist when mapping any page
  - Properties:
    - validation is explicit at map time
    - propagation: system call to copy between clients
    - restriction: kernel to make new capability
    - revocation: kernel to remove cap from clist
      - one specific or all
    - accessibility: requires scanning all clists
    - protection domain: explicitly represented in clist
- Few commercial systems (KeyKOS)
- Many research systems
  - Hydra, Mach, EROS, and many others



Propagating Partitioned Capabilities

- Capabilities can be included in IPC messages
  - sender supplies cap (clist index) to system call
  2. kerne looks up cap in sender's clist
  3. kernel inserts caps into receiver's clist
  4. kernel replaces sender's clist index by receiver's and delivers message

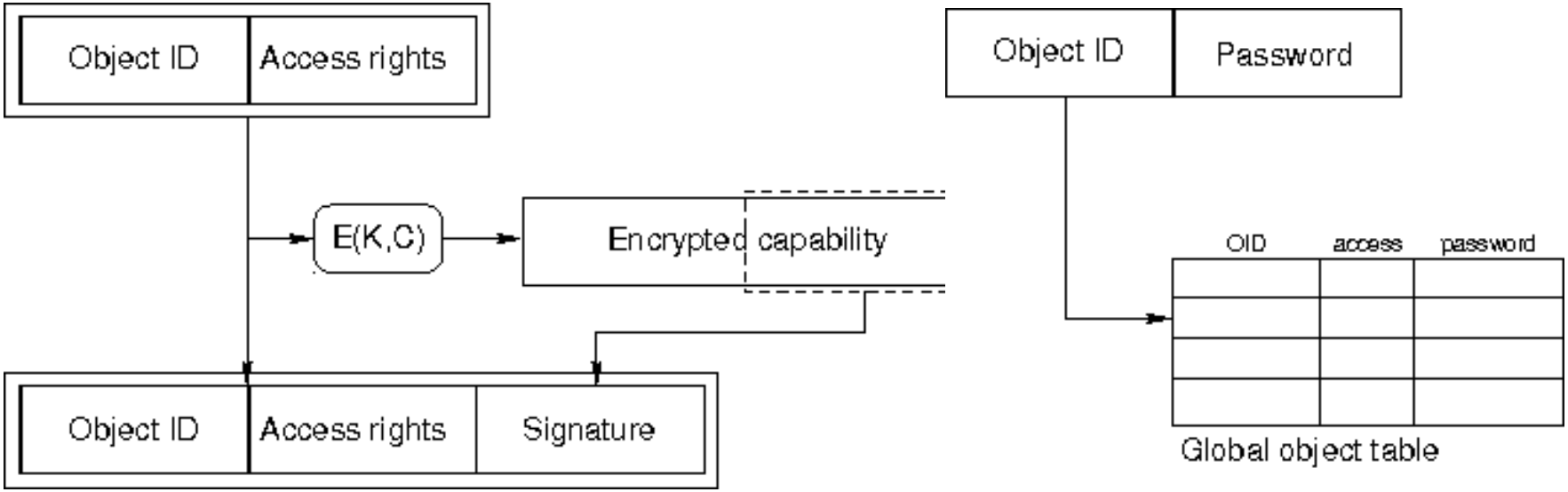


Partitioned Capabilities Summary

- Secure through protection by kernel
  - real caps live in kernel space
- Validation at mapping time ═► apps use “normal” pointers
- Fast validation (clist check is simple, validation cached by MMU)
- Propagation requires marshaling and kernel intervention
- Reference counting possible to detect unaccessible objects

Sparse Capabilities

- Basic idea similar to encryption
  - add bit string to make valid capabilities a very small subset of cap space
  - either encrypted object info or password
  - secure by infeasibility of exhaustive search of cap space



Sparse Capabilities

NICTA

- Sparse caps are user-level objects
  - can be passed like other data
    - similar to tagged caps, but without hardware support
    - validated at mapping time (explicit or implicit)
  - good match to user-level servers
    - no central authority, no kernel required on most ops
    - cannot reference-count objects
- Issues:
  - Full mediation requires extra work
    - but doable, see Mungi
    - essentially provided user-level cap segregation
  - High amplification of leaked data
    - problem with convert channels

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]
- In practice difficult to achieve due to *covert channels*

Overview

- Operating systems security overview
- Security policies
- OS security verification
- Security mechanisms
- **Design principles**
- 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

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

- 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

NICTA

- 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

- 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

- 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
- Closely related to least privilege

Least Common Mechanisms

- Avoid sharing mechanisms
  - shared mechanism ⟹ shared channel
  - potential covert channel
- Inherent conflict with other design imperatives
  - simplicity ⟹ shared mechanisms

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
- Security policies
- OS security verification
- Security mechanisms
- Design principles
- **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 explilcit
- Flexibility to support various uses
  - make authority delgatable
  - ensure mechanisms allow high-performance implementation
- Design for verifiability
  - minimize implementation complexity

Example: seL4

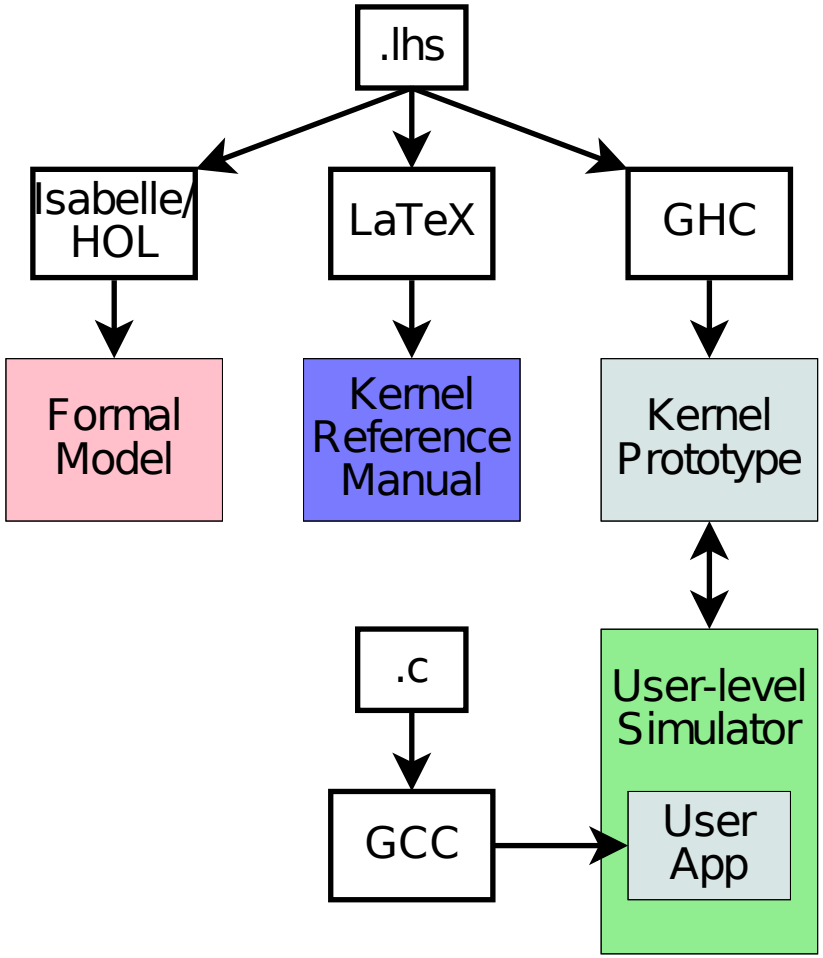
NICTA

- 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

NICTA

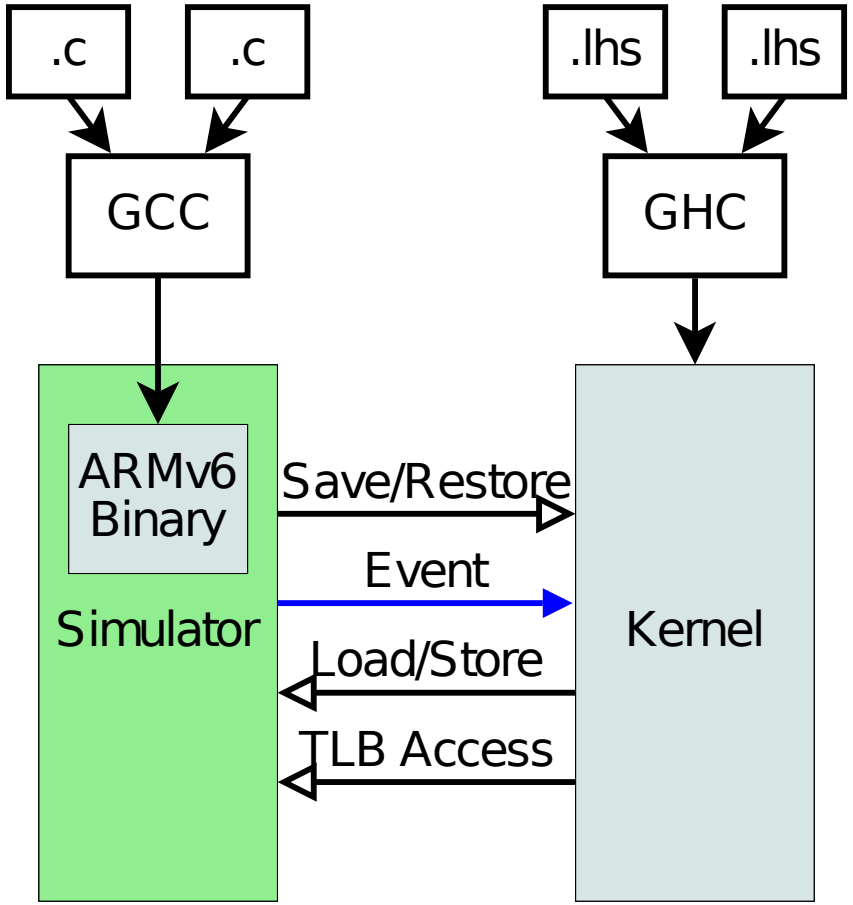
- Model the kernel in detail
- Literate Haskell to model
  - Pure functional programming language
  - Embedded documentation
  - Close to Isabelle/HOL
  - Formalized Haskell becomes intermediate representation for refinement proof
- Executable
  - Supports running user-level code



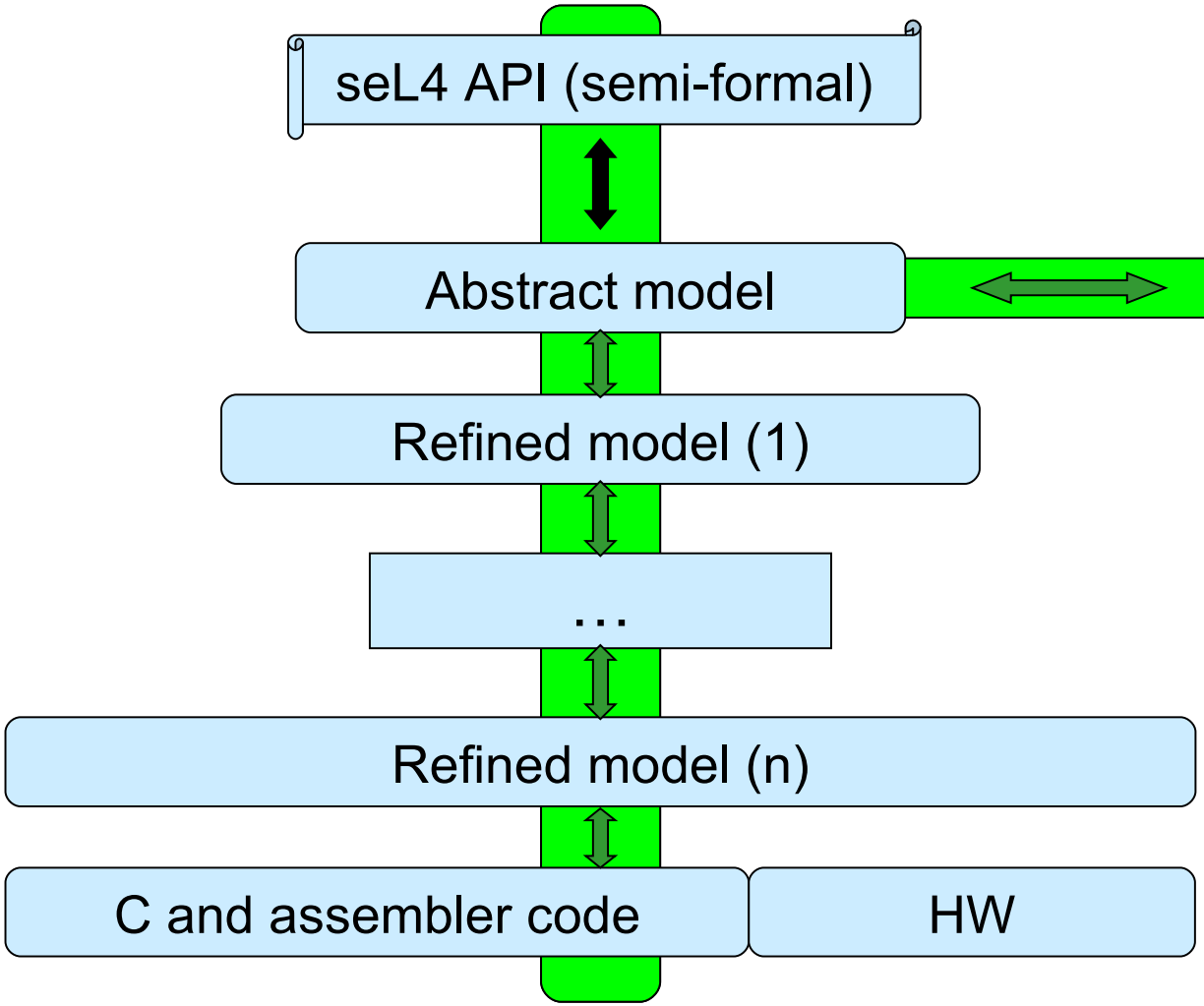
Simulation of User-Level Execution

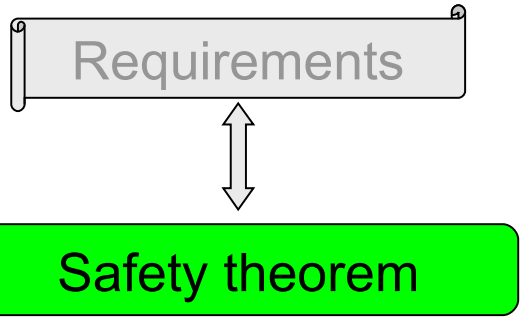
NICTA

- Simulator for user-level ISA
  - M5 Alpha simulator
  - Locally-developed ARMv6 simulator
  - QEMU
- Executes compiled user-level binaries
- Sends events to the Haskell kernel
- Combination allows running complete boot image
- Port system components before kernel implementation is complete



L4.verified: Formal Correctness Proof





- Prove correspon-dence of code and specification
- Refinement proofs in Isabelle/HOL

Thank you!