OS Security
COMP9242 - Advanced OS

Toby Murray

(with thanks to Gernot Heiser, from whom some of this material is borrowed)
INTRODUCTION
What is security?

- Different things to different people:
What is security?

• Different things to different people:
What is security?

• Different things to different people:
What is security?

- Different things to different people:
What is security?

• Different things to different people:
What is security?

• Different things to different people:
What is security?

• Different things to different people:
What is security?

• Different things to different people:
What is security?

• Different things to different people:

On June 8, as the investigation into the initial intrusion proceeded, the response team shared with relevant agencies that there was a high degree of confidence that OPM systems containing information related to the background investigations of current, former, and prospective Federal government employees, and those for whom a Federal background investigation was conducted, may have been compromised.
What is security?

• Different things to different people:
What is security?

• Different things to different people:
Computer Security

• Protecting my interests that are under computer control from malign threats

• Inherently subjective
  – Different people have different interests
  – Different people face different threats

• Don’t expect one-size-fits-all solutions
  – Grandma doesn’t need an air gap
  – Windows alone is insufficient for protecting TOP SECRET classified data
    • on an Internet-connected machine
State of OS Security

• Traditionally:
  – Has not kept pace with evolving user demographics
    • Focused on e.g. Defence and Enterprise
  – Has not kept pace with evolving threats
    • Focused on protecting users from other users, not from the programs they run

• Is getting better
  – But is hindered because:
    • We don’t yet understand how to write secure code
    • OSes are getting larger and more complex
OS Security

• What is the role of the OS for security?
• Minimum:
  – provide mechanisms to allow the construction of secure systems
  – that are capable of securely implementing the intended users’/administrators’ policies
  – while ensuring these mechanisms cannot be subverted
Good security mechanisms

• Are widely applicable
• Support general security principles
• Are easy to use correctly and securely
• Do not hinder non-security priorities (e.g. productivity, generativity)
• Lend themselves to correct implementation and verification
Security Design Principles

- Saltzer+Schroeder (SOSP ’73, CACM ’74)
  - Economy of mechanism
  - Fail-safe defaults
  - Complete mediation
  - Open design
  - Separation of privilege
  - Least privilege
  - Least common mechanism
  - Psychological acceptability
Common OS Security Mechanisms

- **Access Control Systems**
  - control what each process can access

- **Authentication Systems**
  - confirm the identity on whose behalf a process is running

- **Logging**
  - for audit, detection, forensics and recovery

- **Filesystem Encryption**

- **Credential Management**

- **Automatic Updates**
Security Policies

• Define what should be protected
  – and from whom

• Often in terms of common security goals:
  – **Confidentiality**
    • X should not be learnt by Y
  – **Integrity**
    • X should not be tampered with by Y
  – **Availability**
    • X should not be made unavailable to Z by Y
Policy vs. Mechanism

• Policies accompany mechanisms:
  – access control policy
    • who can access what?
  – authentication policy
    • is password sufficient to authenticate TS access?

• Policy often restricts the applicable mechanisms

• One person’s policy is another’s mechanism
Assumptions

• All policies and mechanisms operate under certain **assumptions**
  – *e.g.* TS cleared users can be trusted not to write TS data into the UNCLASS window

• Problem: implicit or poorly understood assumptions

• Good assumptions:
  – clearly identified
  – verifiable
Risk Management

• Comes down to **risk management**
  – At the heart of all security
  – Assumptions: risks we are willing to tolerate

• Other risks:
  – we mitigate (using security mechanisms)
  – or transfer (e.g. by buying insurance)

• Security policy should distinguish which is appropriate for each risk
  – Based on a thorough **risk assessment**
Trust

• Systems always have trusted entities
  – whose misbehaviour can cause insecurity
  – hardware, OS, sysadmin ...

• Trusted Computing Base (TCB):
  – the set of all such entities

• Secure systems require trustworthy TCBs
  – achieved through assurance and verification
  – shows that the TCB is unlikely to misbehave
  – why the TCB should be as small as possible
Assurance and Formal Verification

• **Assurance:**
  – systematic evaluation and testing

• **Formal verification:**
  – mathematical proof

• Together trying to establish correctness of:
  – the **design** of the mechanisms
  – and their **implementation**

• **Certification:** establishes that the assurance or verification was done right
Covert Channels

• Information flow not controlled by security mechanism
  – confidentiality requires absence of all such

• Covert **Storage** Channel:
  – attribute of shared resource used as channel
  – controllable by access control

• Covert **Timing** Channel:
  – temporal order of shared resource accesses
  – outside of access control system
  – much more difficult to control and analyse
Covert Timing Channels

• Created by shared resource whose timing-related behaviour can be monitored
  – network bandwidth, CPU load ...

• Requires access to a time source
  – anything that allows processes to synchronise

• Critical issue is channel bandwidth
  – low bandwidth limits damage
    • why DRM ignores low bandwidth channels
  – beware of amplification
    • e.g. leaking passwords, encryption keys etc.
Summary: Introduction

- Security is very subjective
- OS security:
  - provide good security mechanisms
  - that support users’ policies
- Security depends on establishing trustworthiness of trusted entities
  - TCB: set of all such entities
  - should be as small as possible
  - Main approaches: assurance and verification
- The OS is necessarily part of the TCB
ACCESS CONTROL PRINCIPLES
Access Control

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

• AC Policy
  – Specifies allowed accesses
  – And how these can change over time

• AC Mechanism
  – Implements the policy

• Certain mechanisms lend themselves to certain kinds of policies
  – Certain policies cannot be expressed using certain mechanisms
Protection State

- **Access control matrix** defines the protection state at any instant in time

<table>
<thead>
<tr>
<th></th>
<th>Obj1</th>
<th>Obj2</th>
<th>Obj3</th>
<th>Subj2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subj1</td>
<td>R</td>
<td>RW</td>
<td></td>
<td>send</td>
</tr>
<tr>
<td>Subj2</td>
<td>RX</td>
<td></td>
<td></td>
<td>control</td>
</tr>
<tr>
<td>Subj3</td>
<td>RW</td>
<td>RWX</td>
<td>own</td>
<td>recv</td>
</tr>
</tbody>
</table>
Storing Protection State

- Not usually as access control matrix
  - too sparse, inefficient
- Two obvious choices:
  - store individual columns with each object
    - defines the subjects that can access each object
    - each such column is called the object’s **access control list**
  - store individual rows with each subject
    - defines the objects each subject can access
    - each such is called the subject’s **capability list**
Access Control Lists (ACLs)

- Subjects usually aggregated into classes
  - e.g. UNIX: owner, group, everyone
- Meta-permissions (e.g. own)
  - control class membership
  - allow modifying the ACL
- Implemented in almost all commercial OSes

<table>
<thead>
<tr>
<th>Subj1</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subj2</td>
<td></td>
</tr>
<tr>
<td>Subj3</td>
<td>RW</td>
</tr>
</tbody>
</table>
Capabilities

- **A capability** is a capability list element

<table>
<thead>
<tr>
<th>Subj1</th>
<th>Obj1</th>
<th>Obj2</th>
<th>Obj3</th>
<th>Subj2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>RW</td>
<td></td>
<td>send</td>
</tr>
</tbody>
</table>

- **Names** an object to which the capability refers
- **Confers** permissions over that object

- Less common in commercial systems
  - More common in research though
Capabilities: Implementations

• Capabilities must be unforgeable

• On conventional hardware, either:
  – Stored as ordinary user-level data, but unguessable due to sparseness
    • like a password or an encryption key
  – Stored separately (in-kernel), referred to by user programs by index/address
    • like UNIX file descriptors

• Sparse capabilities can be leaked more easily, but are easier to revoke
  – The only solution for most distributed systems
ACLs and Capabilities: Duals?

- **In theory:**
  - Dual representations of access control matrix

- **Practical differences:**
  - Naming and namespaces
    - Confused Deputies
  - Evolution of protection state
  - Forking
  - Auditing of protection state
Duals: Naming and Namespaces

• ACLs:
  – objects referenced by **name**
    • e.g. `open("/etc/passwd",O_RDONLY)`
  – require a subject (class) namespace
    • e.g. UNIX users and groups

• Capabilities:
  – objects referenced by **capability**
    • object namespace still required though
  – no subject namespace required
Duals: Confused Deputies

- ACLs: separation of object naming and permission can lead to confused deputies
  - Capabilities are both names and permissions
    - You can’t name something without having permission to it
Duals: Confused Deputies

- ACLs: separation of object naming and permission can lead to confused deputies
  - Capabilities are both names and permissions
    - You can’t name something without having permission to it

```
exec "gcc" "-o LogFile"
```
Duals: Evolution of Protection State

• ACLs:
  – Protection state changes by modifying ACLs
    • Requires certain meta-permissions on the ACL

• Capabilities:
  – Protection state changes by delegating and revoking capabilities
    • Right to delegate controlled by certain capabilities
    • e.g. A can delegate to B only if A has a capability to B that carries appropriate permissions
Duals: Forking

• What permissions should children get?

• ACLs: depends on the child’s subject
  – UNIX etc.: child inherits parent’s subject
    • Inherits all of the parent’s permissions
    • Any program you run inherits all of your authority
  – Bad for least privilege

• Capabilities: child has no caps by default
  – Parent gets a capability to the child upon fork
  – Used to delegate (only) necessary authority
  – Much better for least privilege
Duals: Auditing of Protection State

• How to work out who has permission to access a particular object (right now)?
  – ACLs: Just look at the ACL

• How to work out what objects a particular subject can access (right now)?
  – Capabilities: Just look at its capabilities

• “Who can access my stuff?” vs. “How much damage can this thing do?”
Mandatory vs. Discretionary AC

- **Discretionary Access Control:**
  - Users can make access control decisions
    - delegate their access to other users etc.

- **Mandatory Access Control (MAC):**
  - enforcement of administrator-defined policy
  - users cannot make access control decisions (except those allowed by mandatory policy)
  - can prevent untrusted applications running with user’s privileges from causing damage
MAC

• Common in areas with global security requirements
  – e.g. national security classifications

• Less useful for general-purpose settings:
  – hard to support different kinds of policies
  – all policy changes must go through sysadmin
  – hard to dynamically delegate only specific rights required at runtime
Bell-LaPadula (BLP) Model

• MAC Policy/Mechanism
  – Formalises National Security Classifications
• Every object assigned a classification
  – e.g. TS, S, C, U
• Classifications ordered in a lattice
  – e.g. TS > S > C > U
• Every subject assigned a clearance
  – Highest classification they’re allowed to learn
BLP: Rules

- **Simple Security Property** ("no read up"):  
  - s can read o iff clearance(s) \( \geq \) class(o)  
  - S-cleared subject can read U,C,S but not TS  
  - standard confidentiality

- ***-Property** ("no write down"):  
  - s can write o iff clearance(s) \( \leq \) class(o)  
  - S-cleared subject can write TS,S, but not C,U  
  - to prevent accidental or malicious leakage of data to lower levels
Biba Integrity Model

• Bell-LaPadula enforces **confidentiality**
• **Biba**: Its dual, enforces integrity
• Objects now carry **integrity** classification
• Subjects labelled by **lowest** level of data each subject is allowed to learn
• BLP order is inverted:
  – s can read o iff clearance(s) <= class(o)
  – s can write o iff clearance(s) >= class(o)
Boebert’s Attack


• Shows an attack on *sparse* capability systems that violates the *-property
  – Where caps and data are indistinguishable
  – Does not work against *partitioned* capability systems
    • Practically all capability-based kernels
Boebert’s Attack

Diagram:

- **High** → **HiSeg**
- **Low** → **LoSeg**
- **Rw_l** → **LoSeg**
- **R** → **HiSeg**
- **R** → **LoSeg**
- **Rw** → **LoSeg**
Boebert’s Attack

```
Low RW LoSeg
rw_l

High R HiSeg
rw_l

rw_l.write(rw_l)
```
Boebert’s Attack
Boebert’s Attack

```
r_l.read()
```

```
High
```

```
HiSeg
```

```
Low
```

```
LoSeg
```

```
R
```

```
r_l
```

```
rw_l
```

```
RW
```

```
R
```

```
rw_l
```

```
rw_l
```
Boebert’s Attack

High \rightarrow HiSeg

Low \rightarrow LoSeg

RW \quad r_l \quad R

rw_l \quad rw_l
Boebert’s Attack

- Low writes his cap into the low segment – from which High reads it out
Boebert’s Attack: Lessons

• Not all mechanisms suited to all policies
• Many policies treat data- and access-propagation differently
  – BLP is one example
  – Cannot be expressed using sparse capability systems
• This does not mean that capability systems and MAC are incompatible in general
Decideability

- Boebert’s attack highlights the need for **decideability** of safety in an AC system

- **Safety Problem**: given an initial protection state $s$, and a possible future protection state $s'$, can $s'$ be reached from $s$?
  - i.e. can an arbitrary (unwanted) access propagation occur?

- **HRU 1975**: undecideable in general
  - equivalent to the halting problem
Decideable AC systems

• The safety problem for an AC system is **decideable** if we can always answer this question mechanically.

• Most capability-based AC systems decideable:
  – instances of Lipton-Snyder **Take-Grant** access control model
  – Take-Grant is decideable in linear time

• Less clear for many common ACL systems
Summary: AC Principles

• ACLs and Capabilities:
  – Capabilities tend to better support least privilege
  – But ACLs can be better for auditing

• MAC good for global security requirements

• Certain kinds of policies cannot be enforced with certain kinds of mechanisms
  – e.g. *-property with sparse capabilities

• AC systems should be decideable
  – so we can reason about them
ACCESS CONTROL PRACTICE
Case Study: SELinux

- NSA-developed MAC for Linux
- Designed to protect systems from buggy applications
  - Especially daemons and servers that have traditionally run with superuser privileges
- Adds a layer of MAC atop Linux’s traditional DAC
  - Each access check must pass both the normal DAC checks and the new MAC ones
- Used widely in e.g. enterprise Linux
SELinux: Policy

• **Domain-Type Enforcement:**
  – Each process labelled with a **domain**
  – Each object labelled with a **type**
  – Central policy describes allowed accesses from domains to types

• **Example:**
  – `named` runs in `named_d` domain; `/sbin` labelled with `sbin_t` type
  – “allow named_d sbin_t:dir search”
SELinux: Domain/Type Transitions

• How domains assigned to new processes
  – upon exec() (after fork())
  – based on exec’ing domain and exec’d file type
  – “type_transition initrc_d
    squid_exec_t:process squid_d”

• how types assigned to new files/directories
  – based on domain of process creating them and type of parent directory
  – “type_transition named_t
    var_run_t:sock_file named_var_run_t”
SELinux

- Static fine-grained MAC
- Monolithic policy of high complexity
  - “The simpler targeted policy consists of more than 20,000 concatenated lines ... derived from ... thousands of lines of TE rules and file context settings, all interacting in very complex ways.”

- Limited flexibility
  - What authority should we grant a text editor?
    - Needed authority determined only by user actions
Case Study: Capsicum

• “Practical Capabilities for UNIX” (Watson et al., USENIX Security 2010)

• Designed to support least privilege in conventional systems
  – without downsides of MAC
  – through delegation

• Merged into FreeBSD 9
Capsicum: Kernel

• Capsicum adds to the FreeBSD kernel:
  – Capabilities with fine-grained access rights for standard objects (files, processes etc.)
  – Capability Mode
    • Disallows access to global namespaces (e.g. filesystem etc.)
    • All accesses must go through capabilities
    • *at() system calls can resolve only names “underneath” the passed descriptor
    • Allows access to subsets of the filesystem by directory capabilities
FreeBSD Capsicum: Capabilities

- New file descriptor type
  - Wrap traditional file descriptors
  - Carry fine-grained access rights
FreeBSD Capsicum: Capabilities

• Capability passing as for file descriptors:
  – may be inherited across fork()
  – passed via UNIX domain sockets

• Created using cap_new()
  – From a raw file descriptor and a set of rights
  – Or an existing capability
    • New cap’s rights must be a subset

• Capabilities may refer to files, directories, processes, network sockets etc.
FreeBSD Capsicum: Capability Mode

- Entered via new syscall: `cap_enter()`
  - Sets a flag that all child processes then inherit and can never be cleared once set

- Disallows access to all global namespaces:
  - Process ID (PID), file paths, protocol addresses (e.g. IP addrs), system clocks etc.
    - e.g. `open()` syscall disallowed (but `openat()` OK)
  - All accesses through delegated capabilities
    - Removes all ambient authority
FreeBSD Capsicum: *at() syscalls

- Allow lookups of paths relative to a given directory
  - specified by a directory file descriptor
  - e.g. `openat(rootdirfd,"somepath", O_RDONLY)`

- In capability mode, prevented from traversing any path above the given cap
  - e.g. `openat(dirfd,"../blah", flags)` disallowed
  - Ensures that directory caps do not confer authority to access their parents
FreeBSD Capsicum: Capability Mode

- Directory capabilities allow access to sub-parts of the filesystem namespace
FreeBSD Capsicum: Delegation

• A parent delegates to an app it invokes by:
  – fork()ing, obtaining a cap to the child
  – child drops or weakens unneeded caps, calls cap_enter(), then exec()s invoked binary

• Allows e.g. your shell to delegate sensibly to apps it invokes
  – Although apps need to be modified to do all accesses via capabilities
  – Provides an incremental path towards security
Filenames as Cap Handles

- **Capsicum**: `openat()` maps filenames to caps
  - relative to some root directory cap
  - filenames become capability handles

- **Unestos** (Krohn et al., HotOS 2005)
  - no global namespaces, ever
    - each process has distinct filesystem namespace, like in Plan 9
  - all resources represented in filesystem
    - e.g. `/sockets/tcp/listen/80`
  - all filenames are just string handles for caps
    - file namespace becomes simply a cap namespace
AC Mechanisms and Least Privilege

• Secure OS should support writing least-privilege applications
  – decomposing app into distinct components
  – each of which runs with least privilege

• Largely comes down to its AC system
  – some make this far more easy than others

• Example: web browser
  – handles lots of the user’s sensitive info
  – but processes lots of untrusted input
  – input processing parts need to be sandboxed
## Sandboxing Chromium (Watson et al., 2010)

<table>
<thead>
<tr>
<th>DAC</th>
<th>OS</th>
<th>Sandbox</th>
<th>LOC</th>
<th>FS</th>
<th>IPC</th>
<th>Net</th>
<th>Priv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>chroot()</td>
<td>600</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
<tr>
<td>MAC</td>
<td>OS X</td>
<td>Sandbox</td>
<td>560</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
<tr>
<td>Linux</td>
<td>SELinux</td>
<td>200</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
<tr>
<td>Caps</td>
<td>Linux</td>
<td>seccomp</td>
<td>11,300</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Capsicum</td>
<td>100</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
</tbody>
</table>
USABLE SECURITY
Users and Security

• “The single biggest cause of network security breaches is not software bugs and unknown network vulnerabilities but user stupidity, according to a survey published by computer consultancy firm @Stake.”

• “if [educating users] was going to work, it would have worked by now.”
Security Advice

• Security advice:
  – e.g. check URLs / HTTPS certs, use strong passwords, don’t write down passwords, etc.

• Is regularly rejected:
  – when it makes it impossible to get work done
    • why bosses share their passwords with their PAs
  – when there is some incentive to do so
    • why users give out their passwords for chocolate
  – when nobody ever sees any threat
    • why nobody checks HTTPS certificates
    • who here has ever faced a live MITM?
Security Advice Rejection

• Is often rational (Herley, NSPW 2009)
  – because it costs more to follow it than not to
    • advice imposes a cost on everyone
    • but only a fraction ever get attacked
    • so for most, there is not benefit

• Is because security is secondary concern
  – people get paid (only) for getting work done

• Writing good security advice is hard
  – this says more about poor system design than about the motivations of end-users
A brief digression...

- Has your bank ever reminded you not to forget your ATM card when withdrawing cash?
User Education

• Needed when the most secure way to use a system differs from the easiest
  – for rational users: “easiest” = “most profitable”
    • will be different for different people

• Is expensive
  – Cheaper to avoid need for it by careful design

• Not always possible to avoid:
  – when security and productivity goals conflict
  – e.g. need-to-know versus intelligence sharing post 9/11
Why Usable Security?

• Design Principle: Make the easiest way to use a system the most secure
  – c.f. safe defaults

• In general: exploit the user to make the system more, not less, secure
  – by aligning their incentives to produce behaviour that enhances security
  – requires good understanding of economics, human behaviour, psychology etc.
    • why these are now becoming hot topics in security research
Secure Interaction Design

• Users often behave “insecurely” because their actions cause effects different to what they expect
  – User types password into a phishing website
    • did not expect the website was fraudulent
  – User executes email attachment
    • did not expect the attachment to be dangerous

• General principle: secure systems must behave in accordance with user expectations
User Expectations

• To behave in accordance with user expectations:
  – Software must clearly convey consequences of any security choices presented to user
  – Software must clearly inform the user to keep accurate their mental model that informs their choices

• Why secure UIs require trusted paths
  – Essential security mechanism of a secure OS
Trusted Path

• Unspoofable I/O with the user
  – unspoofable output
    • so the user can believe what they see
  – unspoofable input
    • so the user knows what they say will be honoured

• Requires trustworthy I/O hardware
• For interactions via the OS, requires:
  – trustworthy drivers
  – trustworthy kernel
Secure Attention Key

• A trusted path for logging in
  – Ctrl-Alt-Del in Windows NT-based systems
  – Untrappable by applications, so unspoofable
  – Traps directly to kernel
  – Causes login prompt only to be displayed

• Requires user effort
  – So not optimal
  – But better than nothing
Hardware Trusted Paths

• For high-security situations, often cannot trust kernel or device drivers

• These use hardware-only trusted paths
  – Simple I/O hardware directly connected to security-critical device functions
    • e.g. pushbuttons (input) and LEDs (output)
  – bypasses OS
    • requires only that the hardware is trusted
Case Study: Windows UAC

Do you want to allow the following program to make changes to this computer?

Program name: Adobe Reader 8.1.2
Verified publisher: Adobe Systems, Incorporated
File origin: Hard drive on this computer
Program location: "L:\Softwares\AdbeRdr812_en_US.exe"

Do you want to allow the following program from an unknown publisher to make changes to this computer?

Program name: 7z457.exe
Publisher: Unknown
File origin: Downloaded from the Internet
Program location: "L:\Softwares\7z457.exe"
Windows UAC: Overview

- User prompted to confirm granting admin privileges to applications
  - distinguishes apps from “known” and unknown publishers
  - graphical trusted path used by default
    - via separate desktop session
    - prevents apps interfering with the dialog

- User offered a binary choice
  - cannot decide which privileges to grant
UAC Levels (Windows 7 and 8)

High – Always notify

– Don’t notify when “I” make changes
  • “I” is a component of Windows (e.g. launched via Control Panel)
    – potential confused deputies
  • the default

– Don’t dim desktop
  • no trusted path

Low – Never notify
UAC as Usable Security

• On an uninfected machine:
  – User should say yes always
  – This can become the most natural action

• When the user becomes infected, then:
  – Most natural action could be the least secure

• Saying yes optimises for short-term productivity
  – So users who value short-term productivity may act insecurely
Admonition vs. Designation

• UAC is example of **security by admonition** (Yee *S&P* vol 2, no 4, 2004)
  – provide a notification
  – to which user must attend to remain secure

• Alternative is **security by designation**
  – user actions simultaneously designate and authorise
    • c.f. capabilities
  – users’ security decisions inferred through their usual actions
Security by Admonition
Security by Admonition

• Example: User double-clicks an app
Security by Admonition

• Example: User double-clicks an app
Security by Admonition

• Example: User double-clicks an app

  ![User Account Control dialog box](image)

  • Answer will always be “yes”
  – unless the user clicked the wrong app
Security by Admonition

• Example: User double-clicks an app

• Answer will always be “yes”
  – unless the user clicked the wrong app

• “why did it ‘forget’ I wanted to run the app?”
Security by Designation

• Example: User double-clicks an app
  – the app just runs

• User’s act of double-clicking both:
  – designates the app to run
  – grants authority for it to run
    • c.f. capabilities

• Ordinary user actions become security designations
  – ordinary actions grant appropriate authority
  – in accordance with least privilege
Case Study: OS X Lion (etc.) Powerbox

• Automatic dynamic grants of authority to sandboxed applications
  – inferred from ordinary user actions

• OS X sandbox:
  – an app declares its needed authorities via a manifest at install time
    • create net connection, listen, capture from camera
  – sandboxed applications’ authority limited to those in its manifest
  – plus those granted to it by the user through the powerbox daemon
OS X Lion Powerbox

• Trusted daemon process: **pboxd**
• Controls open/save dialogs (and similar)
• User selects File -> Open / Save / Save As
  – pboxd launches appropriate dialog on behalf of the app
• User selects file and clicks e.g. “Open”
  – pboxd grants the app access to the specific file / directory only
• Similar mechanism used for “Recently Opened” files etc.
OS X Lion Powerbox: MS Word

• How much authority does Word need?
  – declared statically (e.g. in its manifest):
    • ability to read/execute its shared libraries
    • ability to read/write global preferences etc.
    • i.e. access to things that were created when it was installed
  – dynamically (through the powerbox):
    • the currently opened files

• That’s basically it
  – same principle can be applied to most other apps too
Least Authority Filesystem Access

• Most apps need just access to:
  – files created when the app was installed
    • /usr/lib/appname
  – system-wide space for app-specific data
    • /usr/share/appname
  – local space for user preferences
    • $HOME/.appname
  – files selected through the powerbox

• Basic idea behind OLPC’s Bitfrost least-authority security architecture
  – whose creator worked on the Lion powerbox
Inferring other needed authorities

• By application type (Yee 2004, IEEE S&P)
  – Internet
    • network access
  – Sound & Video
    • camera / mic access
  – ...

• Determined at install-time
  – user drags the app to the desired part of the applications menu
    • installs the app
    • grants it the necessary authorities
Inferring more complicated authorities

- Windows knows my default web and email clients
- Manages my passwords etc.
- Web browser has access to:
  - my bookmarks
  - web passwords, ....
- Email client has access to:
  - my mail servers
  - account names / passwords ...
- Bonus: app agnostic
Aside: App Stores and Incentives

• Apple distributes OS X Lion apps via its App Store
• Apps need to list required authorities
• Opportunity for security:
  – allows Apple to target their application auditing processes
    • because low authority apps need less auditing
  – natural incentive for developers to minimise the authorities listed by their apps
    • low authority apps can be audited faster
• Incentives are as important as technology!
Case Study: User Driven AC (S&P 2012)

• Generalises powerbox idea from files to arbitrary user-owned resources
  – camera, microphone, address book, facebook friends list

• Access decisions inferred through genuine UI interactions

• Avoids user-facing manifests and UAC/iPhone style permission popups
  – Android malware has shown that users don’t audit install-time manifests carefully
  – users tend to click-through popups
User Driven Access Control

User’s View

Photo Editor App

Camera ACG

System’s View

Camera

Resource Monitor

Policy: Which app can access camera in what fashion

Isolation container

Photo Editor App

<object src="rm://camera/takePicture"/>

ACG

1) User clicks on camera ACG

2) Take picture

3) Receive picture

Kernel
User-Driven AC

• Access Control Gadget (ACG)
  – UI element that applications can embed
  – Interacts with resource Reference Monitor
  – Interactions with ACG grant permissions to the embedding app
  – File Powerbox is but one simple ACG for files

• Protected by the OS from interference from the embedding app
  – but app can move, resize etc. embedded ACGs
ACGs and Resource Classes
ACGs and Resource Classes

- Location data
ACGs and Resource Classes

- Location data
- Microphone, camera
ACGs and Resource Classes

- Location data
- Microphone, camera
- Clipboard
ACGs and Resource Classes

• Location data

• Microphone, camera

• Clipboard

• Files
ACGs and Access Semantics

- ACGs may grant one-time, session or permanent access
  - permanent access rarely required (5% top 100 Android apps)
ACGs and Trusted Path

- ACGs require a trusted path from the OS
  - ACG input events must go directly to ACG
  - Kernel must control the cursor over ACGs

- ACGs must be isolated from app
  - although ACGs can allow customisation

- “Social engineering” attacks still possible
  - trick user into granting access to current location
  - high effort/risk for attacker
Usable Security: Summary

• Design OS security mechanisms with real users in mind
  – mechanisms that fail when users behave normally are faulty, not the other way around

• Mechanisms must convey accurate information to users
  – so they can make informed security decisions

• Mechanisms should infer security decisions from normal user actions
  – granting authority according to least privilege
ASSURANCE AND VERIFICATION
Assurance: Substantiating Trust

• Specification
  – unambiguous description of desired behaviour

• System design
  – justification that it meets specification
    • by mathematical proof or compelling argument

• Implementation
  – justification that it implements the design
    • by proof, code inspection, rigorous testing

• Maintenance
  – justifies that system use meets assumptions
Common Criteria

  - ISO standard, for general use
  - evaluates QA used to ensure systems meet their requirements

- Target of Evaluation (TOE) evaluated against Security Target (ST)
  - ST: statement of desired security properties based on Protection Profiles
Common Criteria: EALs

- 7 Evaluated Assurance Levels
  - higher levels = more thorough evaluation
  - higher cost
  - not necessarily better security

<table>
<thead>
<tr>
<th>Level</th>
<th>Requirement</th>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL1</td>
<td>not eval.</td>
<td>Informal</td>
<td>not eval.</td>
<td>not eval.</td>
</tr>
<tr>
<td>EAL2</td>
<td>not eval.</td>
<td>Informal</td>
<td>Informal</td>
<td>not eval.</td>
</tr>
<tr>
<td>EAL3</td>
<td>not eval.</td>
<td>Informal</td>
<td>Informal</td>
<td>not eval.</td>
</tr>
<tr>
<td>EAL4</td>
<td>not eval.</td>
<td>Informal</td>
<td>Informal</td>
<td>not eval.</td>
</tr>
<tr>
<td>EAL5</td>
<td>not eval.</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL6</td>
<td>Formal</td>
<td>Semi-Formal</td>
<td>Semi-Formal</td>
<td>Informal</td>
</tr>
<tr>
<td>EAL7</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Informal</td>
</tr>
</tbody>
</table>
Common Criteria Protection Profiles (PPs)

- Controlled Access PP (CAPP)
  - standard OS security, up to EAL3
- Single Level Operating System PP
  - superset of CAPP, up to EAL4+
- Labelled Security PP
  - MAC for COTS OSes
- Multi-Level Operating System PP
  - superset of CAPP, LSPP, up to EAL4+
- Separation Kernel Protection Profile
  - strict partitioning, for EAL6-7
COTS OS Certifications

• EAL3:
  – Mac OS X

• EAL4:
  – 2005: SuSE Enterprise Linux
  – 2006: Solaris 10 (EAL4+)
    • against CAPP (an EAL3 PP!)
  – 2007: Red Hat Linux (EAL4+)

• These OSes are still regularly broken!
EAL6 and above OS Certifications

• EAL6
  – Green Hills INTEGRITY-178B (EAL6+)
    • Separation Kernel Protection Profile (SKPP)
    • relatively simple hardware platform in TOE
  – Aiming for EAL7
SKPP on Commodity Hardware

• SKPP:
  – OS provides only separation

• One Box One Wire (OB1) Project
  – Use INTEGRITY-178B to isolate VMs on commodity desktop hardware
  – Leverage existing INTEGRITY certification
    • by “porting” it to commodity platform
  – Conclusion (March 2010):
    • SKPP validation for commodity hardware platforms infeasible due to their complexity
    • SKPP has limited relevance for these platforms
Common Criteria Limitations

• Very expensive
  – rule of thumb: EAL6+ costs $1K/LOC
• Too much focus on development process
  – rather than the product that was delivered
• Lower EALs of little practical use for OSes
  – c.f. COTS OS EAL4 certifications
• Commercial Licensed Evaluation Facilities licenses rarely revoked
  – Leads to potential “race to the bottom” (Anderson & Fuloria, 2009)
Formal Verification

• Based on mathematical model of system

• Proof:
  – Model satisfies security properties
    • Required by CC EAL5-7
  – The code implements the model
    • Not required by any CC EAL (informal argument only even for EAL7)

• Example: seL4 microkernel
  – 2009: proof that code implements model
  – 2011: proof that model enforces integrity
  – 2013: proof that model enforces confidentiality
Formal Verification Limitations

• Proofs are expensive
  – e.g. seL4 took ~30 py for ~10,000 LOC

• Proofs rest on assumptions
  – assume correct everything you don’t model
    • e.g. compiler, details of hardware platform, etc.
  – difficult to assume that e.g. modern x86 platform is bug free!
  – full proofs best suited for systems that run on simple hardware platform
    • e.g. embedded systems
    • otherwise they’re not yet worth the high cost
Automatic Analyses

• Algorithms that analyse code to detect certain kinds of defects
• Cannot generally “prove” code is correct
• But much cheaper than proofs
• Tradeoff between completeness and cost
• Need to choose the right tool for the job:
  – Testing
  – Automatic Analyses
  – Formal Proof
• Best strategy is to mix them appropriately
SEL4 AND SECURITY ASSURANCE
A 30-Year Dream

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

<table>
<thead>
<tr>
<th>Common Criteria</th>
<th>EAL4</th>
<th>EAL5</th>
<th>EAL6</th>
<th>EAL7</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Informal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
<tr>
<td>Functional Spec</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
<tr>
<td>High-Level Design</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
<tr>
<td>Low-Level Design</td>
<td>Informal</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
<td>Formal</td>
</tr>
<tr>
<td>Code</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Formal</td>
</tr>
</tbody>
</table>
## Assurance

<table>
<thead>
<tr>
<th>Common Criteria</th>
<th>EAL4</th>
<th>EAL5</th>
<th>EAL6</th>
<th>EAL7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Informal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
<tr>
<td>Functional Spec</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
<td>Formal</td>
</tr>
<tr>
<td>High-Level Design</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
<td>Formal</td>
</tr>
<tr>
<td>Low-Level Design</td>
<td>Informal</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
</tr>
<tr>
<td>Code</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
</tr>
</tbody>
</table>


## Assurance

<table>
<thead>
<tr>
<th>Common Criteria</th>
<th>EAL4</th>
<th>EAL5</th>
<th>EAL6</th>
<th>EAL7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Informal</td>
<td>Formal</td>
<td>Formal</td>
<td>Formal</td>
</tr>
<tr>
<td>Functional Spec</td>
<td>Informal</td>
<td></td>
<td></td>
<td>Formal</td>
</tr>
<tr>
<td>High-Level Design</td>
<td>Informal</td>
<td>Informal</td>
<td>Semiformal</td>
<td>Semiformal</td>
</tr>
<tr>
<td>Low-Level Design</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Formal</td>
</tr>
<tr>
<td>Code</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
<td>Informal</td>
</tr>
</tbody>
</table>

security proofs of the kernel’s code

sel4
seL4 Security Proofs: Overview

Specification

Code
seL4 Security Proofs: Overview

- Access Control Policy Model
- Specification
- Code
seL4 Security Proofs: Overview

- Access Control Policy Model
- Specification
- Code
- Integrity

From imagination to impact
seL4 Security Proofs: Overview

Access Control Policy Model

Integrity

Specification

Infoflow

Code
seL4 Security Proofs: Overview

Integrity + Infoflow $\rightarrow$ Isolation

- Integrity
- Infoflow
- Specification
- Access Control Policy Model
- Code
seL4 Security Proofs: Overview

Integrity + Infoflow $\rightarrow$ Isolation

Infoflow $\rightarrow$ Confidentiality
seL4 Security Proofs: Overview

Integrity + Infoflow $\rightarrow$ Isolation

Infoflow $\rightarrow$ Confidentiality
Information Flow Security (S&P 2013)

Diagram:
- Internet
- Malware Filter
- Audit
- Work
Information Flow Security (S&P 2013)

general computation within partitions
general computation within partitions

intransitive noninterference
Only Kernel Change: Partition Scheduling

Current Partition

Partitioning:
- P1,2
- P2,10
- P1,5
- P3,12
- P1,5

Partition Time: 2

NICTA Copyright 2013 From imagination to impact 110
Only Kernel Change: Partition Scheduling

- **Static round-robin schedule between** partitions

<table>
<thead>
<tr>
<th>P1,2</th>
<th>P2,10</th>
<th>P1,5</th>
<th>P3,12</th>
<th>P1,5</th>
</tr>
</thead>
</table>

Current Partition

Partition Time

2
Only Kernel Change: Partition Scheduling

- Static round-robin schedule **between** partitions

```
P1,2  P2,10  P1,5  P3,12  P1,5
```

Partition Time: 2

Current Partition

Partition Time decremented on each timer-tick
Only Kernel Change: Partition Scheduling

- Static round-robin schedule \textit{between} partitions

P1,2 \quad P2,10 \quad P1,5 \quad P3,12 \quad P1,5 \quad 2

\begin{itemize}
  \item advances when partition-time hits 0
  \item Partition Time decremented on each timer-tick
\end{itemize}
Only Kernel Change: Partition Scheduling

- Static round-robin schedule **between** partitions

  - P1,2
  - P2,10
  - P1,5
  - P3,12
  - P1,5
  - 2

- Current Partition

  - Partition Time
  - advances when partition-time hits 0

- Priority-based scheduling **within** partitions
  - Choose the highest-priority thread that is ready
  - Run idle thread if none ready
  - Any other intra-partition scheduling algorithm possible

- Prio
  - Ready Queue
  - P2
  - 255
  - T3
  - T1
  - 0
  - T7

- NICTA Copyright 2013

From imagination to impact
Problematic Kernel APIs
Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
  - the proof guarantees they will stay disabled
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts

• Inter-partition object destruction
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts

• Inter-partition object destruction
  – partition-crossing comms. channels cannot be destroyed
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts

• Inter-partition object destruction
  – partition-crossing comms. channels cannot be destroyed

not uncommon in high-assurance systems
Problematic Kernel APIs

• Leaky kernel APIs need to be disabled
  – by ensuring initially no subject has permission to use them
  – the proof guarantees they will stay disabled

• Asynchronous interrupt delivery
  – device drivers must poll for interrupts

• Inter-partition object destruction
  – partition-crossing communication channels cannot be destroyed

all kernel services available within partitions, besides async irq notification

not uncommon in high-assurance systems
Storage Channels
Storage Channels

- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...

```
  Information Flow Security
     /    /
    /      /
   Code    Kernel Specification
    \\
     \\
      \\
```
Storage Channels

- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also **all** user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible **nondeterminism**
  - **not tolerated** by nonleakage under refinement
Storage Channels

- Proof covers all storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also all user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible nondeterminism
  - not tolerated by nonleakage under refinement

```
bool l, h;
l := 0 ⨅ 1;
```
Storage Channels

- Proof covers all storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also all user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible **nondeterminism**
  - **not tolerated** by nonleakage under refinement

```c
bool l, h;
l := 0 ⊕ 1;
```

is refined by
Proof covers all storage channels present in kernel spec
- abstract kernel heap, CPU registers, physical memory, IRQ masks, ...

Also all user-visible channels read by the kernel
- those below the level of the spec appear as user-visible nondeterminism
- not tolerated by nonleakage under refinement

```
bool l, h;
l := 0 \land 1;
```

is refined by
```
bool l, h;
l := h;
```
Storage Channels

- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...

- Also **all** user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible **nondeterminism**
  - not tolerated by nonleakage under refinement

\[
\text{bool } l, h; \\
l := 0 \land 1; \\
l := h;
\]

is refined by

\[
\text{is the value of refinement-preserved noninterference}
\]
Storage Channels

- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also all user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible nondeterminism
  - not tolerated by nonleakage under refinement

**not covered:** channels absent from spec that kernel never reads

```c
bool l, h;
l := 0 \Leftrightarrow 1;
l := h;
```

is the value of refinement-preserved noninterference
Storage Channels

• Proof covers all storage channels present in kernel spec
  – abstract kernel heap, CPU registers, physical memory, IRQ masks, ...

• Also all user-visible channels read by the kernel
  – those below the level of the spec appear as user-visible nondeterminism
  – not tolerated by nonleakage under refinement

not covered: channels absent from spec that kernel never reads
  e.g. undocumented hardware APIs

bool l, h;
l := 0
\[ l := 0 \]\n
is refined by

\[ l := h; \]

refinement-preserved noninterference
Timing Channels
Timing Channels

- The proof says nothing about timing channels
Timing Channels

• The proof says nothing about timing channels
• e.g. jitter in scheduler
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
Timing Channels

• The proof says nothing about timing channels
• e.g. jitter in scheduler
  – seL4 syscalls are generally non-preemptible
    • except at well-defined points during long-running calls e.g. Revoke()
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be delayed by syscall
Timing Channels

• The proof says nothing about timing channels
• e.g. jitter in scheduler
  – seL4 syscalls are generally non-preemptible
    • except at well-defined points during long-running calls e.g. Revoke()
  – partition switch can be delayed by syscall

user mode

-------------------------------------------------------------------------------

kernel mode
(irqs disabled)
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be delayed by syscall
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be **delayed** by syscall

```
    user mode
    -------
  uop     uop
    -------

 kernel mode
 (irqs disabled)
```
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be \texttt{delayed} by syscall
Timing Channels

• The proof says nothing about timing channels
• e.g. jitter in scheduler
  – seL4 syscalls are generally non-preemptible
    • except at well-defined points during long-running calls e.g. Revoke()
  – partition switch can be delayed by syscall
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
     - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be delayed by syscall
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be delayed by syscall
Timing Channels

- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be delayed by syscall

- Others: caches, CPU temp. etc.
Timing Channels

• The proof says nothing about timing channels
  • e.g. jitter in scheduler
    – seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
    - partition switch can be delayed by syscall
  • Others: caches, CPU temp. etc.

must be mitigated by complementary techniques
Timing Channels

- The proof says nothing about timing channels
  - e.g. jitter in scheduler
    - seL4 syscalls are generally non-preemptible
      - except at well-defined points during long-running calls e.g. Revoke()
    - partition switch can be delayed by syscall
  - Others: caches, CPU temp. etc.

must be mitigated by complementary techniques

mitigation strategy depends on risk profile of deployment
Lesson

- Functional correctness enables cheap security proofs
Lesson

- Functional correctness enables cheap security proofs

Effort (py)

- ~6 people, over ~4 years

- Functional Correctness
- Integrity
- Infoflow
Lesson

- Functional correctness enables cheap security proofs

![Graph showing effort](graph.png)

~6 people, over ~4 years

~2.5 FTE, over ~4 months
Lessons

- Functional correctness enables cheap security proofs

Effort (py)

- Functional Correctness: ~6 people, over ~4 years
- Integrity: ~2.5 FTE, over ~4 months
- Infoflow: ~2.5 FTE, over ~21 months
Analysing Timing Channels (CCS 2014)

- Formal verification framework has no explicit concept of time
- Modern architectures too complex to accurately model timing
- Use experimental approach:
  - Implement timing channels exploits and measures
  - Sound information theoretic approach to analysis

- Objective:
  - Understand and quantify kernel-relevant timing channels
  - Develop minimal mechanisms to mitigate channels
    - Must not compromise seL4’s generality and verifiability
    - Must not degrade performance when timing channels don’t matter
Example: cache contention channel

- Partitions compete for space in the cache
- Low partition can observe High partition’s cache footprint
Unmitigated Cache Channel

- Cache channel measures on the Exynos4412 (ARMv7, Cortex-A9)
  - Estimated bandwidth: 2,350 b/s
  - Each column is 1,000 samples
  - This graph captures 4.77 TiB of uncompressed data
Cache Colouring

Partition cache between different security domains to prevent cache interference

Allocation of colours to domains implemented outside kernel
• Existing kernel memory allocation mechanism already sufficient

Kernel modified to duplicate its own code on-demand
• Each partition has its own copy, preventing i-cache interference
Effectiveness of Cache Colouring

Bandwidth: 2350 b/s → 27 b/s
Timing Channel Observations

• Empirical measurement is essential
  – coupled with sound information theory for analysis

• Has revealed unexpected channels
  – cycle counter influenced by cache misses, branch mispredicts

• Well-designed kernel mechanisms can be effective
  – ... but manufacturers are working against us
OS DESIGN FOR SECURITY
OS Design for Security

• Minimise kernel code
  – can bypass all security, inherent part of TCB

• How?:
  – generic mechanisms
  – no policies, only mechanisms
  – mechanisms as simple as possible
  – exclude all code that doesn’t need to be privileged to support secure systems
  – minimise covert channels
    • no global namespaces, or absolute time
Security and Concurrency

• Avoid concurrent access to security state
  – leads easily to security vulnerabilities
• Time of Check-to-Time-of-Use (TOCTTOU)
  – common in privileged reference monitors

```c
if (access("file", W_OK) != 0) {
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));

symlink("/etc/passwd", "file");
```

– Make rights checks atomic with accesses
– Why most system-call wrappers don’t work
Unexpected Concurrency

• Example: FreeBSD Capsicum vulnerability
  – `openat()` with paths involving multiple “..”s
  – activity can occur between each “..” lookup
  – second process races with first to ensure each “..” lookup succeeds, using `renameat()`
  – allows escaping of sandboxes

• Solutions:
  – complicate the lookup code
  – disallow multiple “..”s in pathnames (simpler)

• Second solution was chosen
Designing Secure Mechanisms

• Simplify security mechanisms
  – Because they are hard enough to get right in the first place

• Ensure mechanisms are well-defined
  – make policy and granting authority explicit

• Flexibility to support various uses
  – support explicit delegation of authority

• Design for verifiability
  – minimise implementation complexity
Example: seL4

- Simple AC mechanism: capabilities
  - supports least privilege, decideable
- No in-kernel concurrency
  - single kernel stack, poll for IRQs
- Formal proof of implementation correctness
- Formal proof that design (and so code) enforces relevant security properties:
  - integrity (ITP, 2011)
  - confidentiality (S&P, 2013)
QUESTIONS?