From General Purpose to a Proof of Information Flow Enforcement

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao and Gerwin Klein
A 30-Year Dream

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that prevention rather than repair was unlikely ever to succeed [25]. A more systematic method was required, presumably one that controlled the system’s design and implementation. Thus secure operation could be demonstrated in a stronger sense than an in-service claim that the last bug had been eliminated, particularly since production systems are rarely static, and errors easily introduced.

Our research seeks to develop means by which an operating system can be shown data secure, ensuring that direct access to data must be possible only if the recording protection policy permits it. The two major components of this task are: (1) developing system architectures that minimize both the amount and complexity of software involved in both protection decisions and enforcement, by isolating them into kernel modules; and (2) applying extensive verification methods to that kernel software in order to prove that our data security criteria is met. This paper reports on the latter part, the verification experience.

Three interesting architectural issues should be noted. Related work includes the Pilots operating system project at SRI [25] which uses the hierarchical design methodology described by Robinson and Levitt in [36], and efforts to prove communications software at the University of Texas [37].

Every verification step, from the development of test-level specifications to machine-checked proof of the Pascal code, was carried out. Although these steps were not complete for all portions of the kernel, most of the work was done for much of the kernel. The remainder is clearly more of the same. We therefore consider the project essentially complete. In this paper, as each verification step is discussed, an estimate of the completed portion of that step is given, together with an indication of the amount of work required for completion. One should realize that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level CNE. Security flaws were found in our system during verification, despite the fact that the implementation was written carefully and tested extensively. An example of one detected loophole is explained in [25].

This work is aimed at several audiences: the software engineering and program verification community, since this case study comprises one of the largest realistic program proving efforts to date; the operating systems community because the effort has involved new operating system architectures; and the security community because the research is directed at the proof of secure operation. We assume the reader is familiar with common operating system concepts, with general program verification methods, and with concepts such as abstract types and structured software. Understanding of alphard

NICTA Copyright 2013
A 30-Year Dream

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>Informal</td>
<td>Informal</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
Information Flow Enforcement

Internet → Malware Filter → Work → Audit → Internet
Information Flow Enforcement

general computation within partitions

Internet → Malware Filter → Audit → Work
general computation within partitions

intransitive noninterference
Information Flow Enforcement

SINGLE-CORE SYSTEMS

general computation within partitions

Internet

Audit

Work
tl;dr
tl;dr

- Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation – doesn’t cover timing channels
tl;dr

- Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation
  - doesn’t cover timing channels

first such theorem for a general-purpose kernel
tl;dr

- Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation
  - doesn’t cover timing channels
- Proof Assumptions:
  - Formally state how to configure kernel to enforce a policy
tl;dr

- Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation
  - doesn’t cover timing channels
- Proof Assumptions:
  - Formally state how to configure kernel to enforce a policy
- Restrictions:
  - DMA disabled
  - No device IRQ delivery
  - Cannot reconfigure inter-partition comms. channels
tl;dr

• Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation
  – doesn’t cover timing channels

• Proof Assumptions:
  – Formally state how to configure kernel to enforce a policy

• Restrictions:
  – DMA disabled
  – No device IRQ delivery
  – Cannot reconfigure inter-partition comms. channels
tl;dr

- Intransitive noninterference theorem for seL4’s (8,830-line) C code implementation
  - doesn’t cover timing channels

- Proof Assumptions:
  - Formally state how to configure kernel to enforce a policy

- Restrictions:
  - DMA disabled
  - No device IRQ delivery
  - Cannot reconfigure inter-partition comms. channels

- All other syscalls available inside partitions!
  - memory allocation, revocation, IPC, cap xfer, shared memory ...
Only Kernel Change: Partition Scheduling

Current Partition

Partition Time

Wednesday, 22 May 2013
Only Kernel Change: Partition Scheduling

- Static round-robin schedule *between* partitions

Current Partition

Partition Time

Wednesday, 22 May 2013
Only Kernel Change: Partition Scheduling

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

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

Current Partition

Partition Time

decremented on each timer-tick

Wednesday, 22 May 2013
Only Kernel Change: Partition Scheduling

- Static round-robin schedule between partitions

Current Partition

advances when partition-time hits 0

Partition Time
decremented on each timer-tick

Wednesday, 22 May 2013
Only Kernel Change: Partition Scheduling

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

- **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

Current Partition

---

Partition Time

advances when partition-time hits 0

decremented on each timer-tick

---

[Diagram showing the process with P1,2, P2,10, P1,5, P3,12, P1,5, and 2, with priority-based scheduling and timer-tick details.]
Proof Structure

- Integrity / Access Control
- Kernel Specification
- Code

Functional correctness proof (2009)
Proof Structure

Integrity / Access Control

Kernel Specification

Code
Proof Structure

integrity proof (2011)

Integrity / Access Control

Kernel Specification

Code
Proof Structure

Integrity / Access Control

Kernel Specification

Code
Proof Structure

- Integrity / Access Control
- Information Flow Security
- Kernel Specification
- Code
Proof Structure

- Integrity / Access Control
- Information Flow Security
- Kernel Specification
- Code

8,830 SLOC
4,970 SLOC
Proof Structure

- Integrity / Access Control
- Information Flow Security
- Kernel Specification
- Code

- 8,830 SLOC
- 4,970 SLOC
- ~150K SLOC

Wednesday, 22 May 2013
Proof Structure

Integrity / Access Control

Information Flow Security

Kernel Specification

Code

10,149 SLOC

4,970 SLOC

~150K SLOC

8,830 SLOC

Wednesday, 22 May 2013
INFORMATION FLOW

(confidentiality)
Information Flow Policy

- Derived from access control policy
Information Flow Policy

- Derived from access control policy

![Diagram](attachment:image.png)
Information Flow Policy

- Derived from access control policy

![Diagram](image_url)

- S1
- S2
- AsyncSend
- Read
- P1
- P2

Wednesday, 22 May 2013
Information Flow Policy

- Derived from access control policy

![Diagram showing information flow policy with nodes S1, S2, P1, P2, and PSched connected by arrows representing AsyncSend and Read.]
Information Flow Policy

• Derived from access control policy

![Diagram of information flow between S1, S2, P1, P2, and PSched]
Information Flow Policy

- Derived from access control policy
Information Flow Policy

- Derived from access control policy.

no-one may affect scheduling decisions
Information Flow Policy

- Derived from access control policy.

no-one may affect scheduling decisions

ensures PSched is not a global transitive channel
Intransitive Nonleakage
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks

Diagram:

- System model
- (Current partition)

P1 → P2 → PSched → P1
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks
Intransitive Nonleakage

- **Variant of intransitive noninterference**
  - Asserts absence of information leaks

```
system model (current partition) ➔ (P2) ➔ (PSched)
```

```
P1 ➔ P2 ➔ PSched
```
Intransitive Nonleakage

- Variant of intransitive noninterference
  - Asserts absence of information leaks

```
system model
(current partition)  (P2)  (PSched)  (P2)
```
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks

![Diagram](image)
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks

- Allows partitions to know of each others’ existence
  - **P1** allowed to observe that **P2** has executed
  - But not to learn anything about **P2**’s state
Intransitive Nonleakage

- **Variant of intransitive noninterference**
  - Asserts absence of information leaks

- **Allows partitions to know of each others’ existence**
  - **P1** allowed to observe that **P2** has executed
  - But not to learn anything about **P2**’s state

- **Implied assumption:**
  - Static partition-schedule is globally public knowledge
    - When **P1** executes, it thus already knows that **P2** must have exhausted its timeslice
Intransitive Nonleakage: Formally
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
Intransitive Nonleakage: Formally

- Similar to language-based noninterference

\[ s \xrightarrow{(P2)} (PSched) \xrightarrow{(P2)} (PSched) \xrightarrow{(P1)} t \]

\[ \approx \{P1, PSched\} \approx \]

\[
\begin{align*}
\text{P1} & \quad \text{P2} \\
\text{PSched} & \quad \text{PSched}
\end{align*}
\]
Intransitive Nonleakage: Formally

- Similar to language-based noninterference

\[ s \xrightarrow{(P2)} (PSched) \xrightarrow{(P2)} (PSched) \xrightarrow{(P1)} t \]

\[ s' \xrightarrow{} (PSched) \xrightarrow{} (P1) \xrightarrow{} t' \]

\[ \approx \{P1, PSched\} \approx \]
Intransitive Nonleakage: Formally

- Similar to language-based noninterference

\[ \approx \{ P1, PSched \} \approx \]

\[ s \xrightarrow{(P2)} (PSched) \xrightarrow{(P2)} (PSched) \xrightarrow{(P1)} t \]

\[ s' \xrightarrow{(P2)} (PSched) \xrightarrow{(P2)} (PSched) \xrightarrow{(P1)} t' \]

\[ \approx P1 \approx ? \]
Intransitive Nonleakage: Formally

- Similar to language-based noninterference

\[ \forall p \ s \ s' \ t \ t'. \\
(s, t) \in \text{Step} \land (s', t') \in \text{Step} \land \\
s \approx \{p, \text{PSched}\} \approx s' \land \\
(part s \rightarrow p \Rightarrow s \sim part s \sim s') \Rightarrow \\
t \sim p \sim t' \]

- Equivalent to single-step unwinding condition:
Informally:

\[ \forall p \ s \ s' \ t \ t'. (s, t) \in \text{Step} \land (s', t') \in \text{Step} \land s \approx \{p, \text{PSched}\} \approx s' \land (\text{part } s \leadsto p \Rightarrow s \sim \text{part } s \sim s') \Rightarrow t \sim p \sim t' \]
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
- Equivalent to single-step unwinding condition:

\[ (s,t) \in \text{Step} \land (s',t') \in \text{Step} \land \\ s \approx \{p,PSched\} \approx s' \land \\ \text{(part } s \rightarrow p \implies s \sim \text{part } s \sim s') \implies \\\ t \sim p \sim t' \]

Informally: on a single step
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
- Equivalent to single-step unwinding condition:

\[
\forall p \ s \ s' \ t \ t'. \ 
(s,t) \in \text{Step} \land (s',t') \in \text{Step} \land 

s \approx \{p, \text{PSched}\} \approx s' \land 

(part \ s \rightarrow p \Rightarrow s \sim \text{part } s \sim s') \Rightarrow 

\quad t \sim p \sim t'
\]

informally: on a single step

\[\approx \{P1, \text{PSched}\} \approx \]

\[\approx P1 \approx ?\]
Intransitive Nonleakage: Formally

- **Informally:** on a single step
  an arbitrary partition $p$ can learn information from:

- Equivalent to single-step unwinding condition:

$$\forall p \ s \ s' \ t \ t'. 
   (s,t) \in \text{Step} \land (s',t') \in \text{Step} \land 
   s \approx \{p, P\text{Sched}\} \approx s' \land 
   (\text{part } s \hookrightarrow p \Rightarrow s \sim \text{part } s \sim s') \Rightarrow 
   t \sim p \sim t'$$
Intransitive Nonleakage: Formally

- **Informally:** on a single step

  an arbitrary partition $p$ can learn information from:

  $\forall p \; s \; s' \; t \; t'$. 
  $(s, t) \in \text{Step} \land (s', t') \in \text{Step} \land 
  s \approx \{p, \text{PSched}\} \approx s' \land 
  (\text{part } s \Rightarrow p \Rightarrow s \sim \text{part } s \sim s') \Rightarrow 
  t \sim p \sim t' $

- Equivalent to single-step unwinding condition:
Intransitive Nonleakage: Formally

- **Informally:** on a single step
  
an arbitrary partition \( p \) can learn information from:
  
  itself and \( PSched \),

- **Equivalent to single-step unwinding condition:**

\[
\forall p \ s \ s' \ t \ t'. \\
(s, t) \in \text{Step} \land (s', t') \in \text{Step} \land \\
s \approx \{p, PSched\} \approx s' \land \\
\text{(part } s \sim p \Rightarrow s \sim \text{part } s \sim s') \Rightarrow \\
t \sim p \sim t'
\]
Intransitive Nonleakage: Formally

- Similar to language-based noninterference
- Equivalent to single-step unwinding condition:

\[
\forall p \ s \ s' \ t \ t'. \\
(s, t) \in \text{Step} \land (s', t') \in \text{Step} \land \\
\approx\{p, \text{PSched}\} \approx s' \land \\
\text{part } s \leadsto p \implies s \sim \text{part } s \sim s' \implies \\
t \sim p \sim t'
\]

informally: on a single step

an arbitrary partition \( p \) can learn information from:

- itself and \( \text{PSched} \),
Intransitive Nonleakage: Formally

- **Informally:** on a single step
  
an arbitrary partition \( p \) can learn information from:
  
itself and \( PSched \),
  
as well as the currently running partition when

\[ \forall p \ s \ s' \ t \ t'. \]
\[ (s,t) \in \text{Step} \land (s',t') \in \text{Step} \land \]
\[ s \approx \{p,PSched\} \approx s' \land \]
\[ (\text{part} \ s \rightarrow p \Rightarrow s \sim \text{part} s \sim s') \Rightarrow \]
\[ t \sim p \sim t' \]
Intransitive Nonleakage: Formally

**Informally:** on a single step

an arbitrary partition $p$ can learn information from:

- itself and $PSched$,

as well as the currently running partition when

\[
\forall p \ s \ s' \ t \ t'. \\
(s,t) \in \text{Step} \land (s',t') \in \text{Step} \land \\
s \approx \{p,PSched\} \approx s' \land \\
(part \ s \not\rightsquigarrow p \Rightarrow s \not\rightsquigarrow \text{part } s \not\rightsquigarrow s') \Rightarrow \\
t \not\approx p \not\approx t'
\]

**Equivalently to single-step unwinding condition:**
Intransitive Nonleakage: Formally

- **Informally:** on a single step
  an arbitrary partition $p$ can learn information from:
  - itself and $PSched$,
  - as well as the currently running partition when
    the policy allows permits it to interfere with $p$

- **Equivalent to single-step unwinding condition:**

$$
\forall p \ s \ s' \ t \ t'. \\
(s, t) \in \text{Step} \land (s', t') \in \text{Step} \land \\
s \approx\{p, PSched\} \approx s' \land \\
(part \ s \rightarrow p \Rightarrow s \sim part \ s \sim s') \Rightarrow \\
t \sim p \sim t'
$$

Wednesday, 22 May 2013
Intransitive Nonleakage: Formally

- \begin{align*}
\forall p \ s \ s' \ t \ t'. \\
(s,t) \in \text{Step} \wedge (s',t') \in \text{Step} \wedge \\
s \approx \{p, \text{PSched}\} \approx s' \wedge \\
(part \ s \not\hookrightarrow p \Rightarrow s \not\hookrightarrow \text{part} \ s \not\hookrightarrow s') \Rightarrow \\
t \not\hookrightarrow p \not\hookrightarrow t'
\end{align*}

informally: on a single step

an arbitrary partition $p$ can learn information from:

- itself and $\text{PSched}$,
- as well as the currently running partition when

the policy allows permits it to interfere with $p$
DISCUSSION
DISCUSSION

(what does it mean?)
Assurance

- Proofs break when:
Assurance

- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)
Assurance

- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)
Assurance

• Proofs break when:
  – they are not logically correct (involve incorrect reasoning)

  a non-issue in practice
Assurance

• Proofs break when:
  – they are not logically correct (involve incorrect reasoning)
  – their assumptions are unrealistic

Security Property

Proof

System Model (code semantics)
Assurance

- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)
  - their assumptions are unrealistic

A non-issue in practice
Assurance

- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)
  - their assumptions are unrealistic
  - they don’t mean what we thought they did

\[ \text{a non-issue in practice} \]
Assurance

• Proofs break when:
  – they are not logically correct (involve incorrect reasoning)
  – their assumptions are unrealistic
  – they don’t mean what we thought they did

Security Property

Proof

System Model (code semantics)

a non-issue in practice
Assumptions
Assumptions

- All those of functional correctness proofs
Assumptions

- All those of functional correctness proofs
  - compiler correctness, cache and TLB management, 450 lines of hand-written assembly code
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and
Assumptions

- All those of functional correctness proofs
  - compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

- Correct initialisation
  - state of system after configuration enforces access policy, and
  - meets wellformedness assumptions
Assumptions

- All those of functional correctness proofs
  - compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

- Correct initialisation
  - state of system after configuration enforces access policy, and
  - meets wellformedness assumptions
    - leaky API features disabled
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and

  – meets wellformedness assumptions
    • leaky API features

  system init correctness proof: in progress
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and
  – meets wellformedness assumptions
    • leaky API features
  – DMA disabled

system init correctness proof: in progress
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and
  – meets wellformedness assumptions
    • leaky API features disabled
  – DMA disabled

• User-space has no info sources that are not modelled
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and
  – meets wellformedness assumptions
    • leaky API features
  – DMA disabled

• User-space has no info sources that are not modelled
  – contents of registers and accessible physical memory

system init correctness proof: in progress
Assumptions

• All those of functional correctness proofs
  – compiler correctness, cache and TLB management, 450 lines of hand-written assembly code

• Correct initialisation
  – state of system after configuration enforces access policy, and
  – meets wellformedness assumptions
    • leaky API features disabled
  – DMA disabled

• User-space has no info sources that are not modelled
  – contents of registers and accessible physical memory

system init correctness proof: in progress

what about covert channels?
Storage Channels

- Information Flow Security
- Kernel Specification
- Code

Wednesday, 22 May 2013
Storage Channels

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

```c
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
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 \lor 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

```c
bool l, h;
l := 0 ⨅ 1;
```

is refined by

```c
bool l, h;
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

**is the value of** refinement-preserved noninterference

```plaintext
bool l, h;
l := 0 ⨅ 1;
```
• 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

refinement-preserved noninterference
  – is refined by

bool l, h;
l := 0 ⨅ 1;
l := h;

NICTA Copyright 2013
From imagination to impact

Wednesday, 22 May 2013
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
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
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

![Graph showing effort in py for different categories: Functional Correctness, Integrity, Infoflow.]

- ~6 people, over ~4 years
Lesson

• Functional correctness enables cheap security proofs

~6 people, over ~4 years

~2.5 FTE, over ~4 months
Lesson

- Functional correctness enables cheap security proofs

![](chart.png)

~6 people, over ~4 years

~2.5 FTE, over ~21 months

~2.5 FTE, over ~4 months

Wednesday, 22 May 2013
Security Theorems for the Kernel Binary

Security

Specification

C Code Semantics

C Code
Security Theorems for the Kernel Binary

1. Security
2. Specification
3. C Code Semantics
4. Binary Semantics (Cambridge ARM ISA)
5. C Code
6. Binary Code
Security Theorems for the Kernel Binary

Security

Specification

C Code Semantics

Binary Semantics (Cambridge ARM ISA)

C Code

Binary Code
Security Theorems for the Kernel Binary

Security

Specification

C Code Semantics

Binary Semantics (Cambridge ARM ISA)

Thomas Sewell

C Code

Binary Code
Security Theorems for the Kernel Binary

C Code Semantics

Binary Semantics (Cambridge ARM ISA)

Specification

Security

will appear at PLDI 2013

Thomas Sewell →

Wednesday, 22 May 2013
Take-Home Message
Take-Home Message

information flow theorem for the C code binary implementation of a general purpose kernel
Take-Home Message

information flow theorem for the C code binary implementation of a general purpose kernel

security proofs of small operating system kernel implementations are practical
Take-Home Message

information flow theorem for the C code binary implementation of a general purpose kernel

security proofs of small operating system kernel implementations are practical

demand nothing less.
Thank You
Thank You