



# 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

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

NICTA Funding and Supporting Members and Partners













From imagination to impact

# INTRODUCTION

# A 30-Year Dream

Operating Systems — R. Stockton Gaines, Editor

## Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los Angeles

**Data Secure Unix, a kernel structured operating system, was constructed as part of an ongoing effort at UCLA to develop procedures by which operating systems can be produced and shown secure. Program verification methods were extensively applied as a constructive means of demonstrating security enforcement.**

**Here we report the specification and verification experience in producing a secure operating system. The work represents a significant attempt to verify a large-scale, production level software system, including all aspects from initial specification to verification of implemented code.**

**Key Words and Phrases: verification, security, operating systems, protection, programming methodology, ALPHARD, formal specifications, Unix, security kernel**

**CR Categories: 4.29, 4.35, 6.35**

† Unix is a Trademark of Bell Laboratories.



This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. Authors' present addresses: B.J. Walker and G.J. Popek, Department of Computer Science, University of California, Los Angeles, CA 90024; R.A. Kemmerer, Computer Science Department, University of California, Santa Barbara, CA 93106.




### 1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous 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, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components of this task are: (1) developing system architectures that minimize 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 of data security criterion is met. This paper reports on the latter part, the verification experience. Those interested in architectural issues should see [23]. Related work includes the PSOS operating system project at SRI [25] which uses the hierarchical design methodology described by Robinson and Levitt in [26], and efforts to prove communications software at the University of Texas [31].

Every verification step, from the development of top-level specifications to machine-aided proof of the Pascal code, was carried out. Although these steps were not completed for all portions of the kernel, most of the job 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 [20]. 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 §2.5.

This work is aimed at several audiences: the software engineering and program verification communities, 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 acquainted with common operating system concepts, with general program verification methods, and with common notions of abstract types and structured software. Understanding of Alphard proof





From imagination to impact

# A 30-Year Dream

Operating Systems — R. Stockton Gaines, Editor

## Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. K…
Gerald J. Popek
University of California, Los A…

Data Secure Unix, a kernel struct… tem, was constructed as part of an op… UCLA to develop procedures by which… can be produced and shown secure. P… methods were extensively applied as … means of demonstrating security rule…

Here we report the specification a… perience in producing a secure operati… work represents a significant attempt … scale, production level software syste… ports from initial specification to veri… mented code.

Key Words and Phrases: verificati… operating systems, protection, progra… ig, ALPHARD, formal specifications, Unix, security kernel

CR Categories: 4.29, 4.35, 6.35

† Unix is a Trademark of Bell Laboratories.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. Authors' present addresses: B.J. Walker and G.J. Popek, Department of Computer Science, University of California, Los Angeles, CA 90024; R.A. Kemmerer, Computer Science Department, University of California, Santa Barbara, CA 93106.

© 1980 ACM 0001-0782/80/0200-0118 $00.75.

118

### 1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous 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, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

…ectively 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 [20]. 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 §2.5.

This work is aimed at several audiences: the software engineering and program verification communities, 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 acquainted with common operating system concepts, with general program verification methods, and with common notions of abstract types and structured software. Understanding of Alphard proof

Communications of the ACM — February 1980, Volume 23, Number 2

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

# A 30-Year Dream

Operating Systems
R. Stockton Gaines
Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. K...
Gerald J. Popek
University of California, Los A...

Data Secure Unix, a kernel struct... tem, was constructed as part of an op... UCLA to develop procedures by which... can be produced and shown secure. P... methods were extensively applied as ... means of demonstrating security rule...

Here we report the specification a... perience in producing a secure operati... work represents a significant attempt ... scale, production level software syste... ports from initial specification to veri... mented code.

Key Words and Phrases: verificati... operating systems, protection, progra... ig, ALPHARD, formal specifications, Unix, security kernel

CR Categories: 4.29, 4.35, 6.35

† Unix is a Trademark of Bell Laboratories.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. Authors' present addresses: B.J. Walker and G.J. Popek, Department of Computer Science, University of California, Los Angeles, CA 90024; R.A. Kemmerer, Computer Science Department, University of California, Santa Barbara, CA 93106.

© 1980 ACM 0001-0782/80/0200-0118 $00.75.

118

1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous 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, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

...structured software. Understanding of Alphard proof

Communications of the ACM
February 1980 Volume 23 Number 2

# Assurance

| Common Criteria | EAL4 | EAL5 | EAL6 | EAL7 | Verified |
|---|---|---|---|---|---|
| **Requirements** | Informal | Formal | Formal | Formal | Formal |
| **Functional Spec** | Informal | Semiformal | Semiformal | Formal | Formal |
| **High-Level Design** | Informal | Semiformal | Semiformal | Formal | Formal |
| **Low-Level Design** | Informal | Informal | Semiformal | Semiformal | Formal |
| **Code** | Informal | Informal | Informal | Informal | Formal |

# Assurance



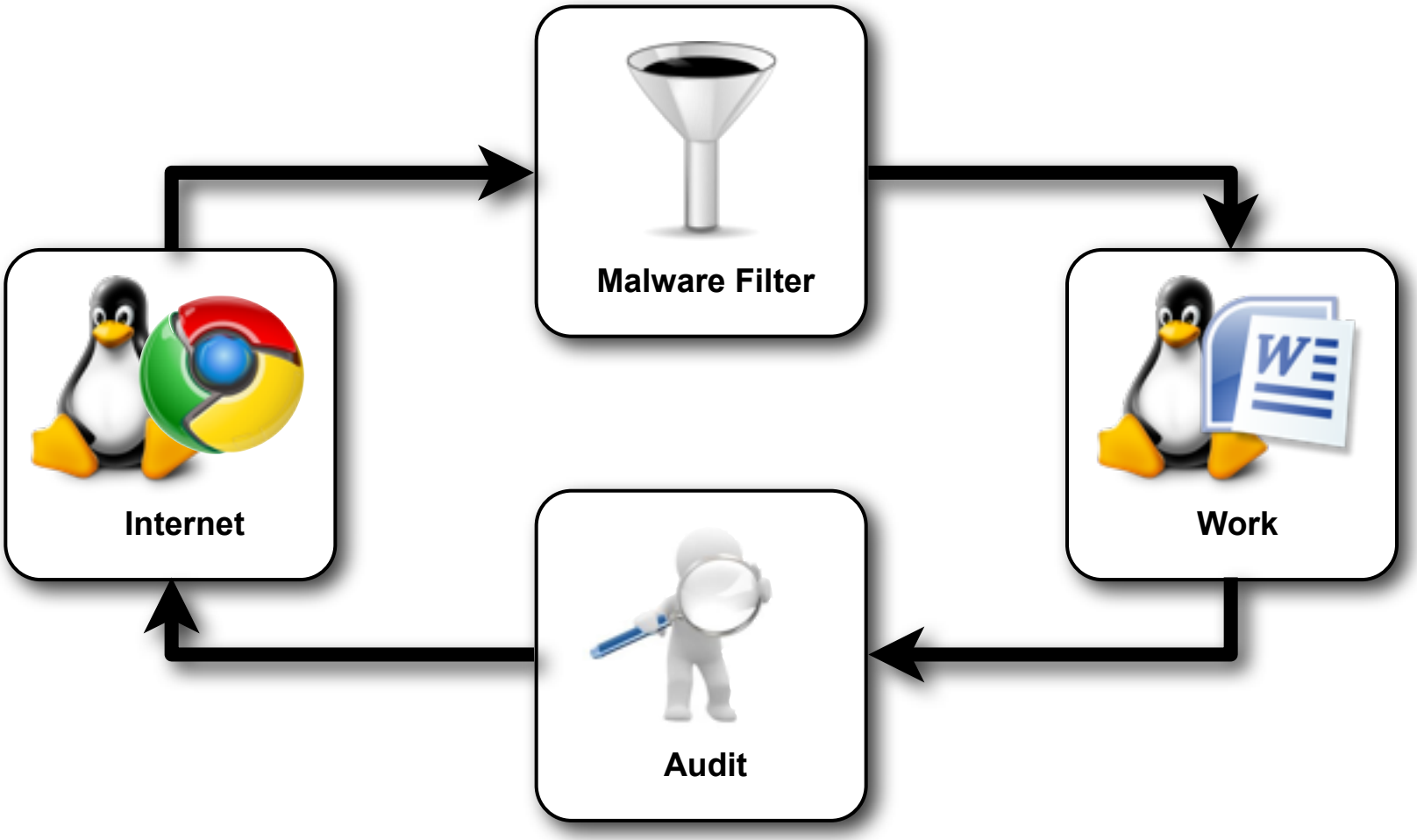
| Common Criteria | EAL4 | EAL5 | EAL6 | EAL7 | [illegible] |
|---|---|---|---|---|---|
| **Requirements** | Informal | Formal | Formal | Formal | Formal |
| **Functional Spec** | Informal | Semiformal | Semiformal | Formal | Formal |
| **High-Level Design** | Informal | Semiformal | Semiformal | Formal | Formal |
| **Low-Level Design** | Informal | Informal | Semiformal | Semiformal | Formal |
| **Code** | Informal | Informal | Informal | Informal | Formal |

# Assurance

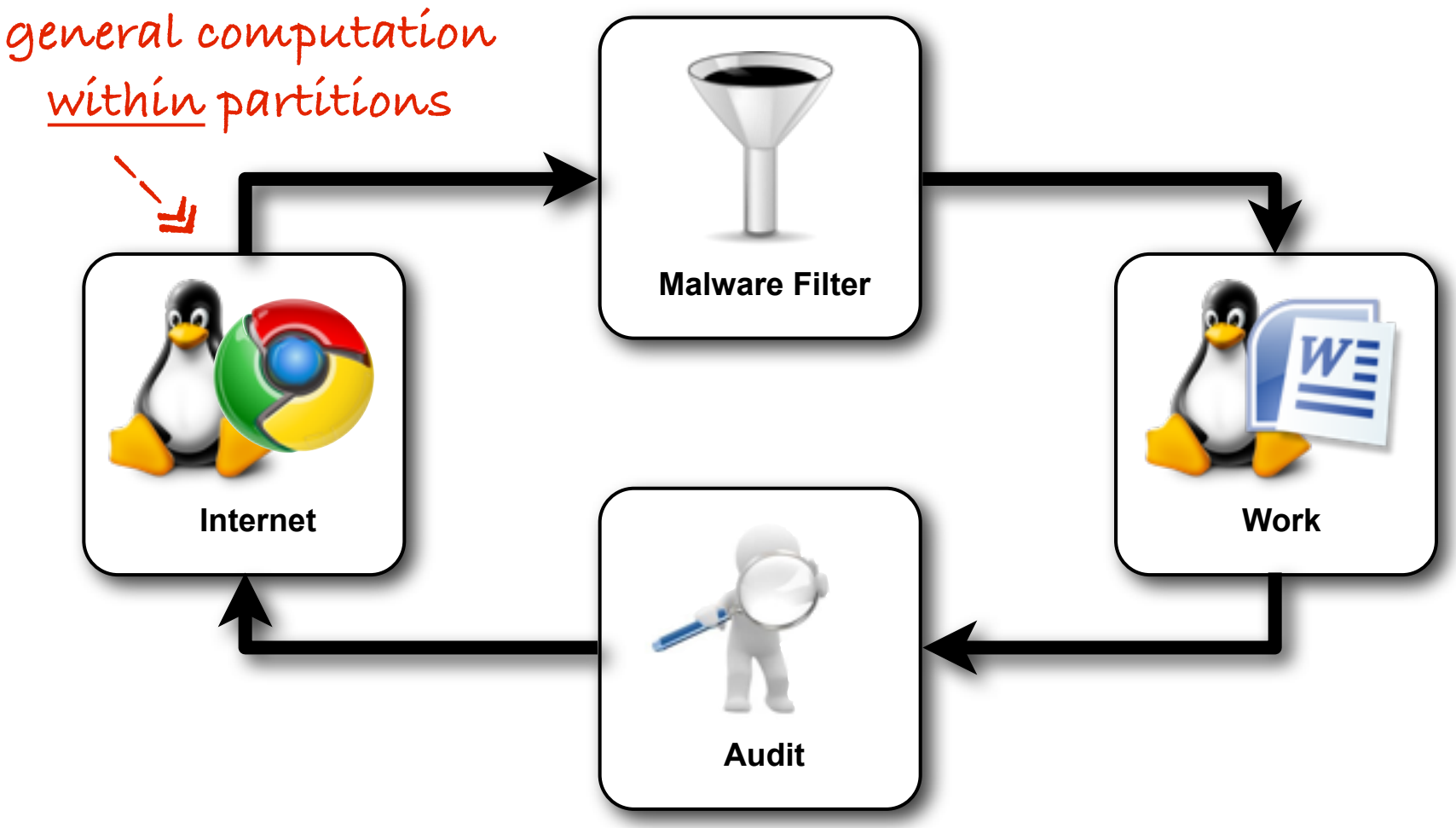
| Common Criteria | EAL4 | EAL5 | EAL6 | EAL7 | [illegible] |
|---|---|---|---|---|---|
| Requirements | Informal | Formal | Formal | [illegible] | Formal |
| Functional Spec | [illegible] | [illegible] | [illegible] | [illegible] | Formal |
| High-Le Design | [illegible] | [illegible] | [illegible] | Formal | Formal |
| Low-Leve Design | Informal | Informal | Semiformal | Semiformal | Formal |
| Code | Informal | Informal | Informal | Informal | Formal |

security proofs of the kernel's code

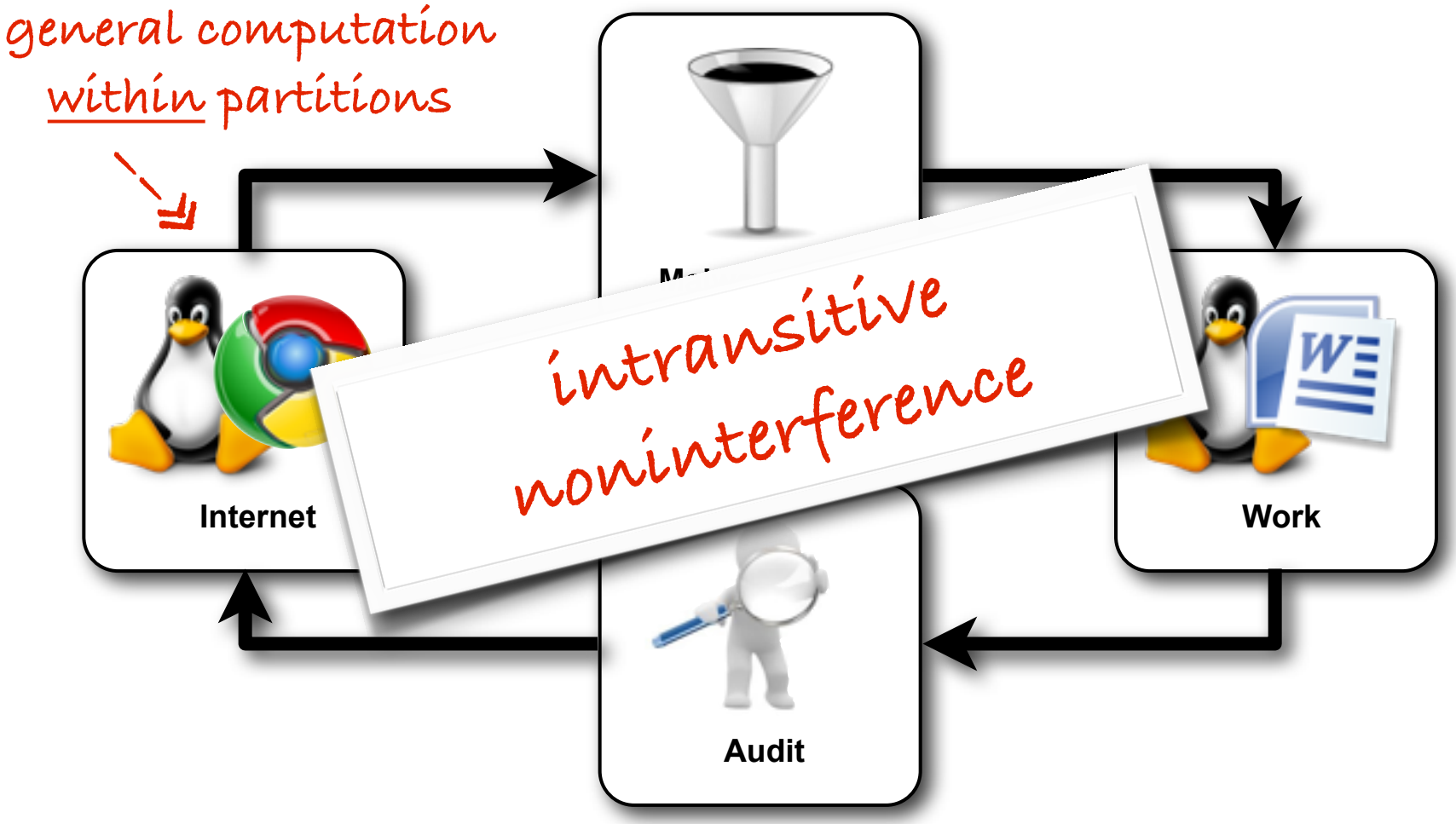
# Information Flow Enforcement



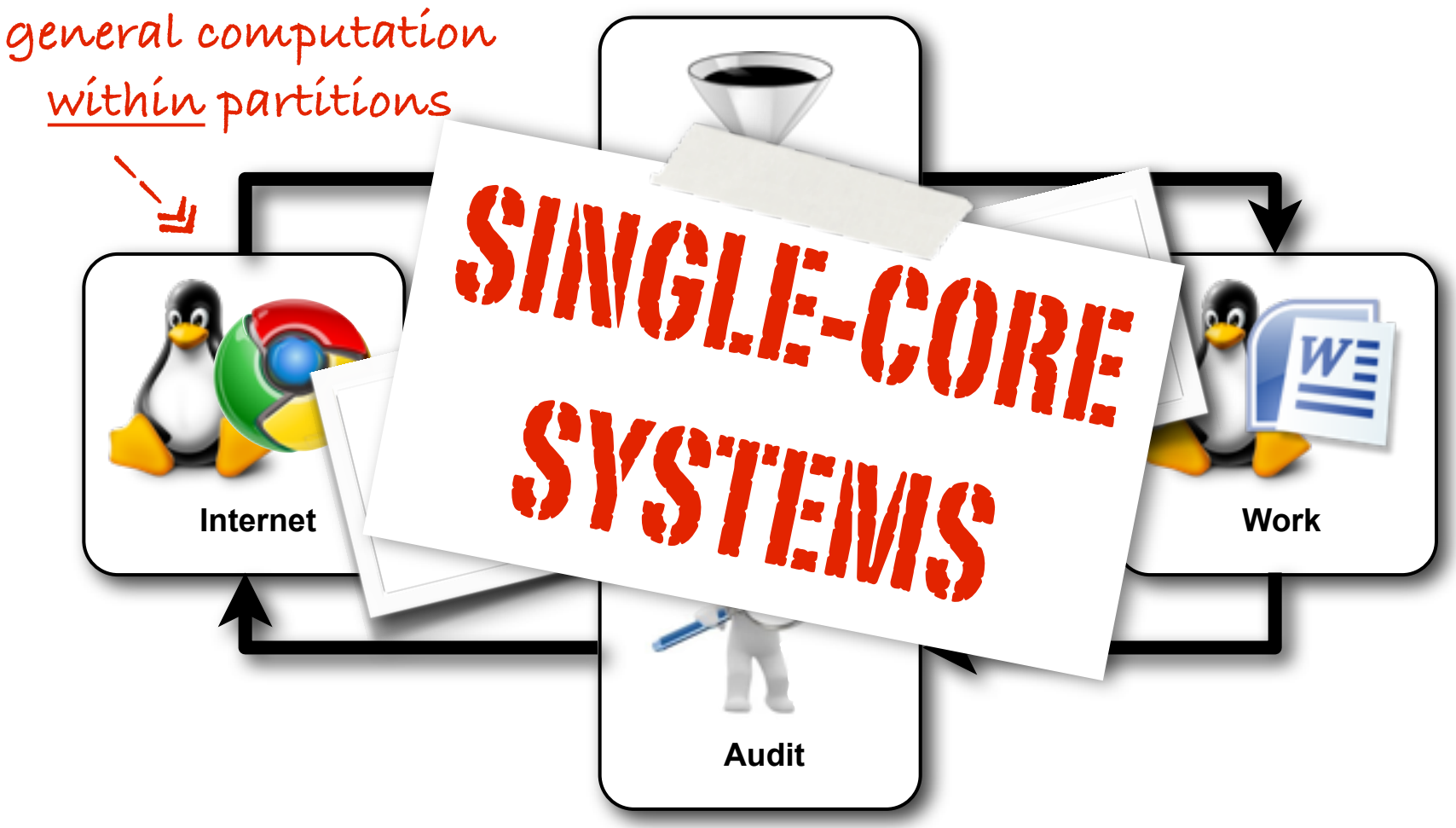
# Information Flow Enforcement



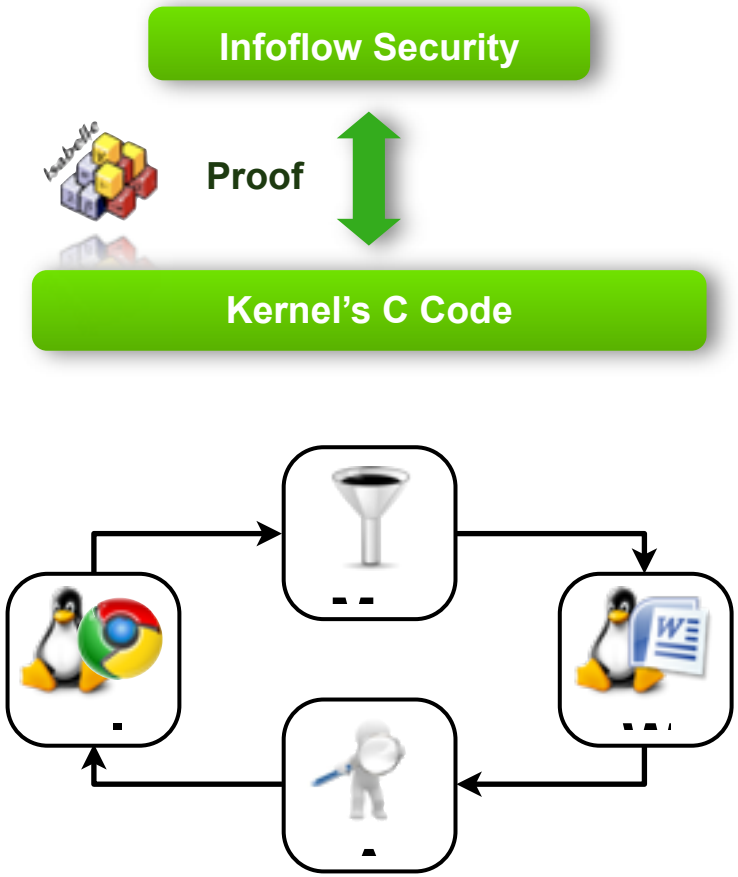
# Information Flow Enforcement



# Information Flow Enforcement

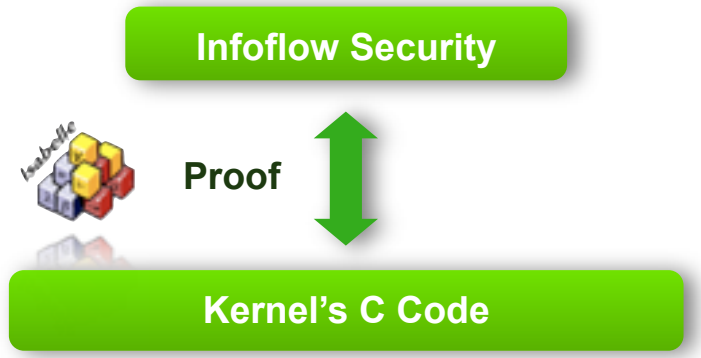


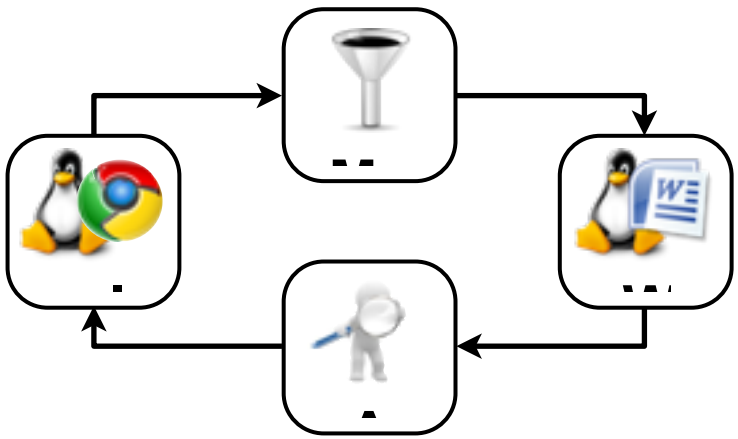
# tl;dr



# tl;dr

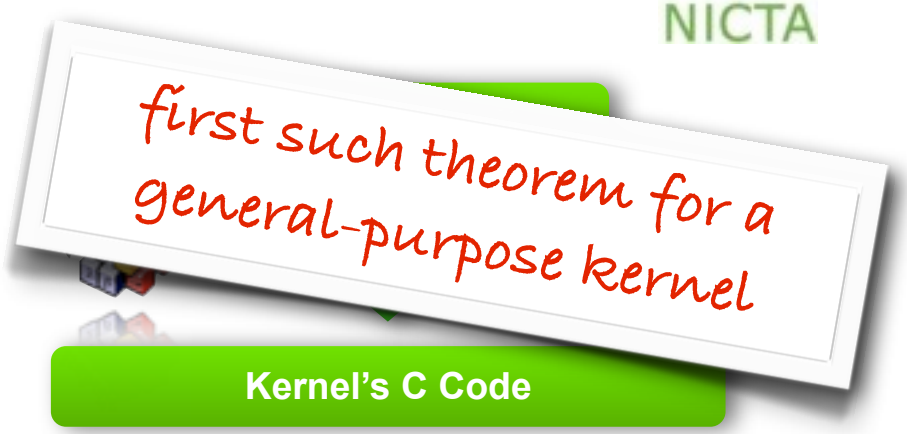
- Intransitive noninterference theorem for seL4's (8,830-line) **C code implementation**
  - doesn't cover timing channels

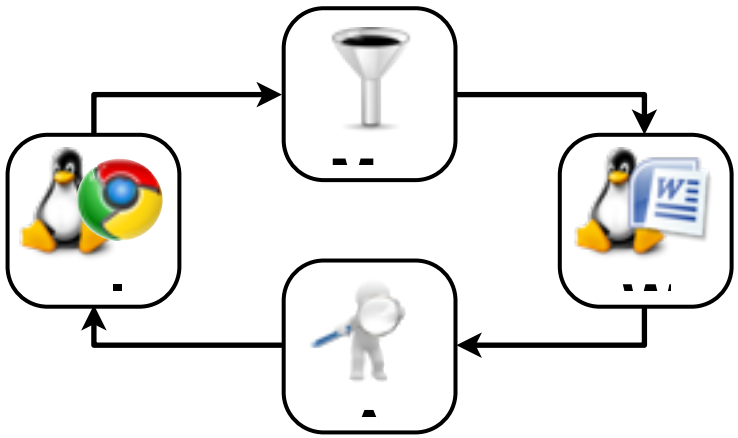


# tl;dr

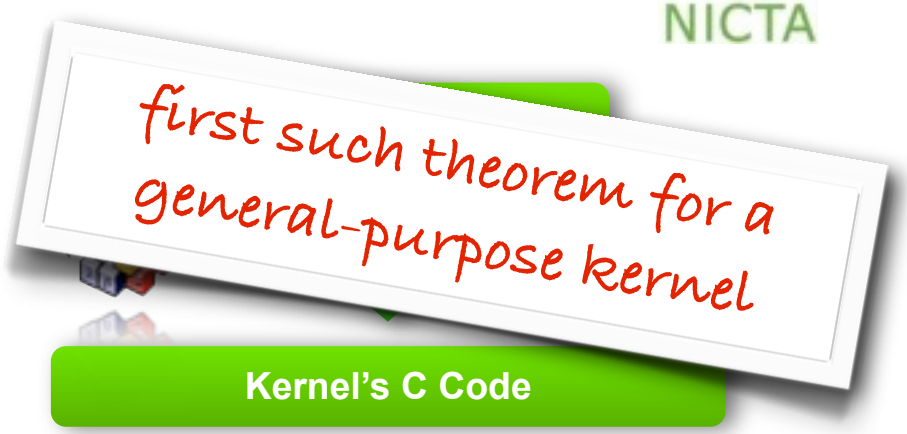
- Intransitive noninterference theorem for seL4's (8,830-line) **C code implementation**
  - doesn't cover timing channels

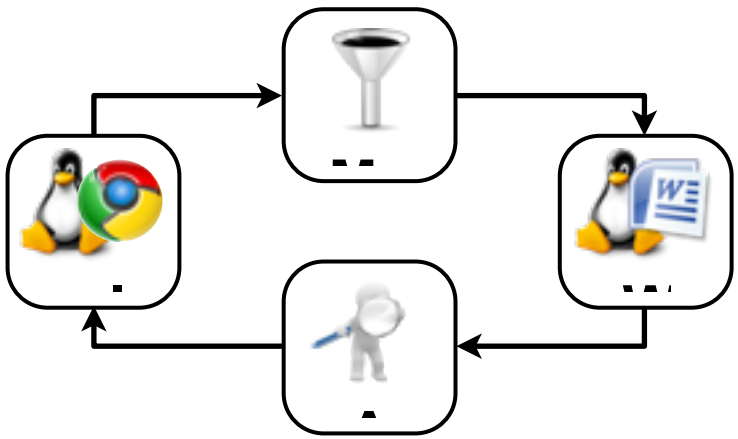


# tl;dr

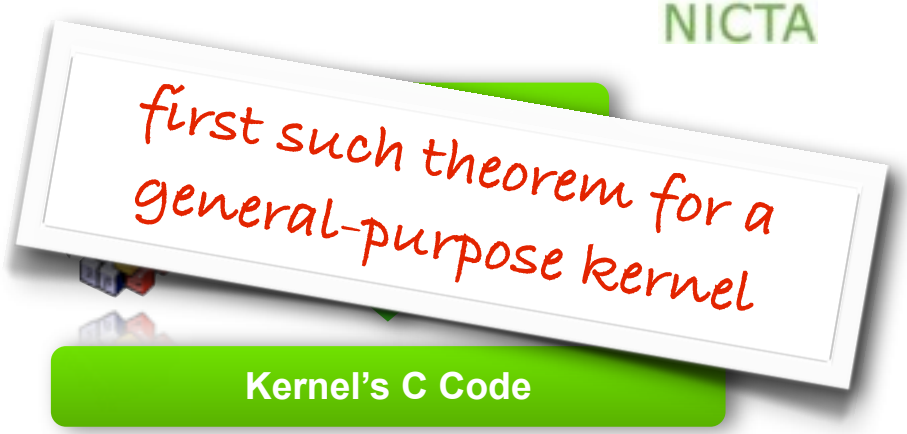
- Intransitive noninterference theorem for seL4's (8,830-line) **C code implementation**
  - doesn't cover timing channels
- Proof Assumptions:
  - Formally state how to configure kernel to enforce a policy

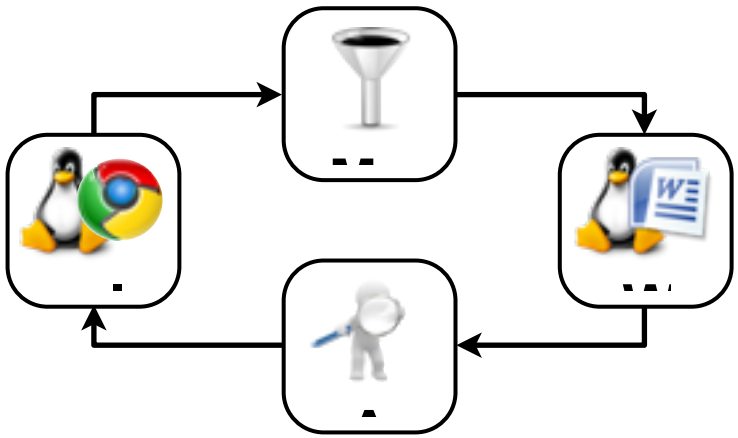


# tl;dr

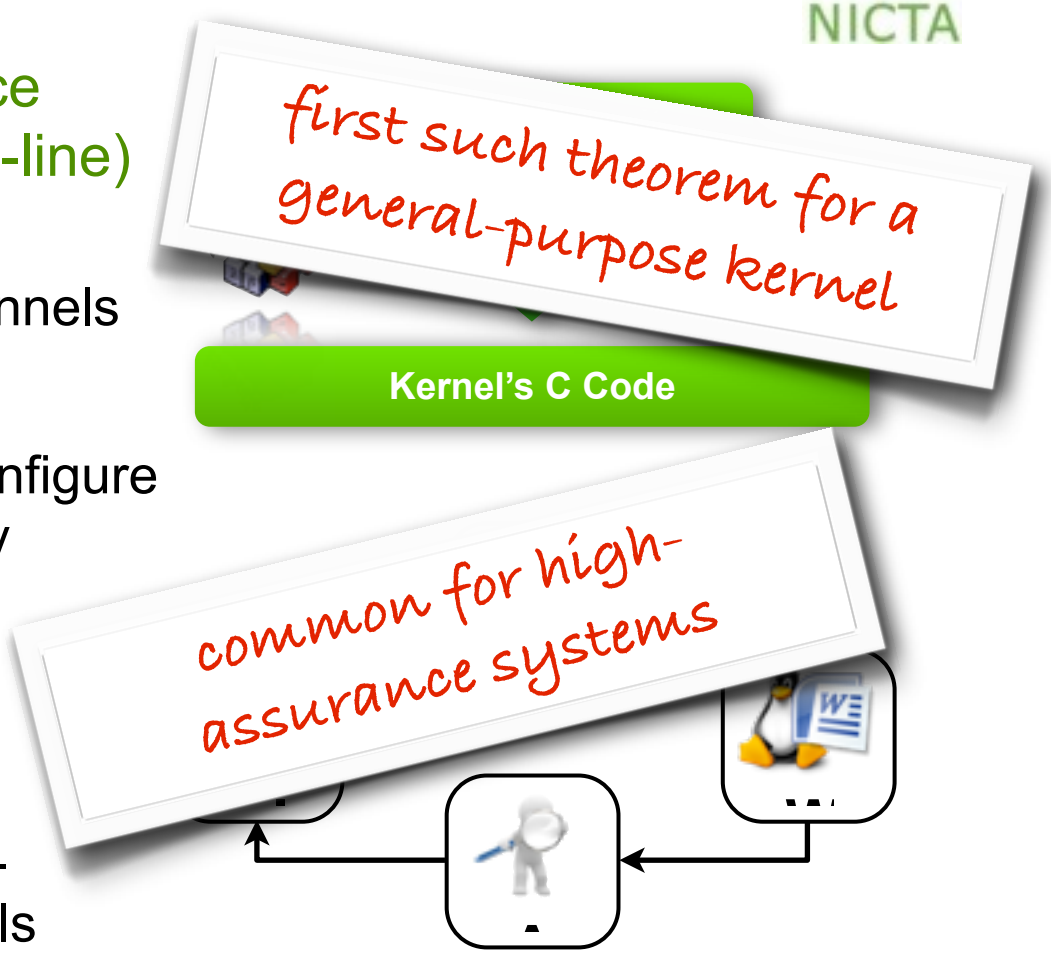
- Intransitive noninterference theorem for seL4's (8,830-line) **C code implementation**
  - doesn't cover timing channels
- Proof Assumptions:
  - Formally state how to configure kernel to enforce a policy
- Restrictions:
  - DMA disabled
  - No device IRQ delivery
  - Cannot reconfigure inter-partition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



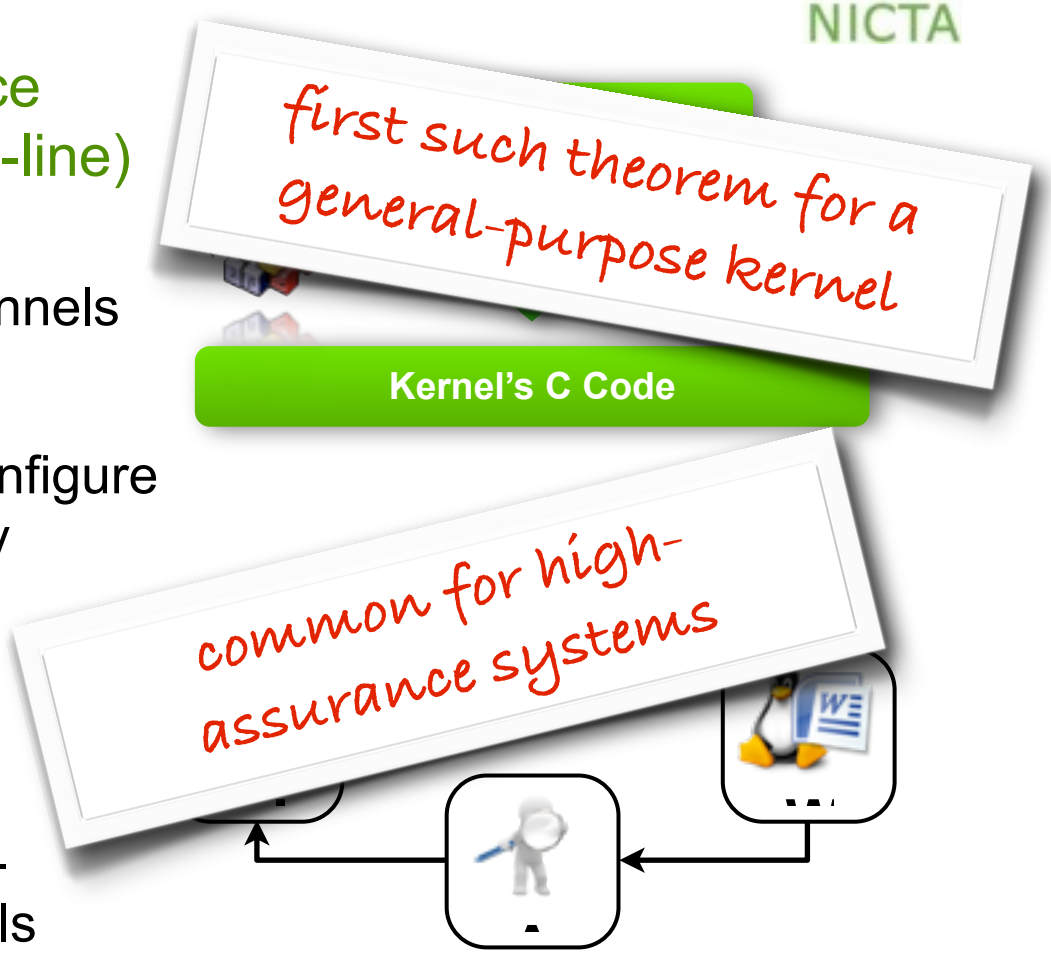
# tl;dr

- 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

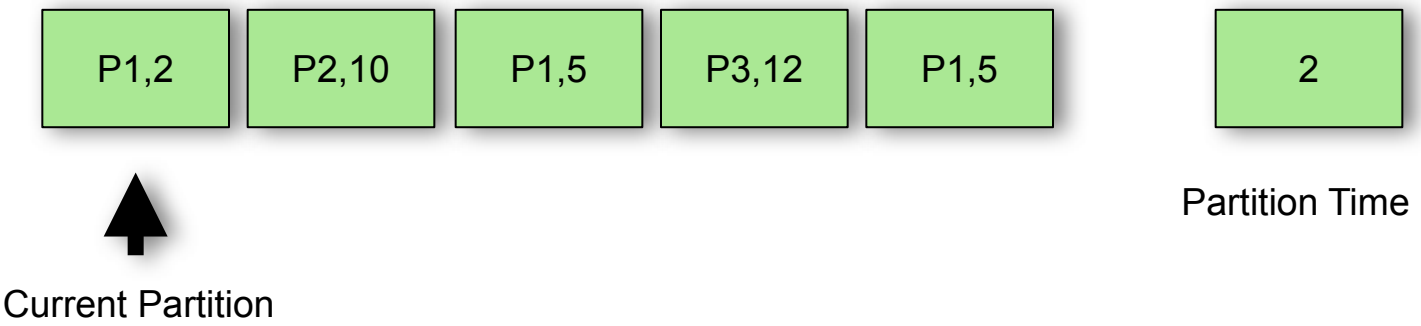


# tl;dr

- 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 ...

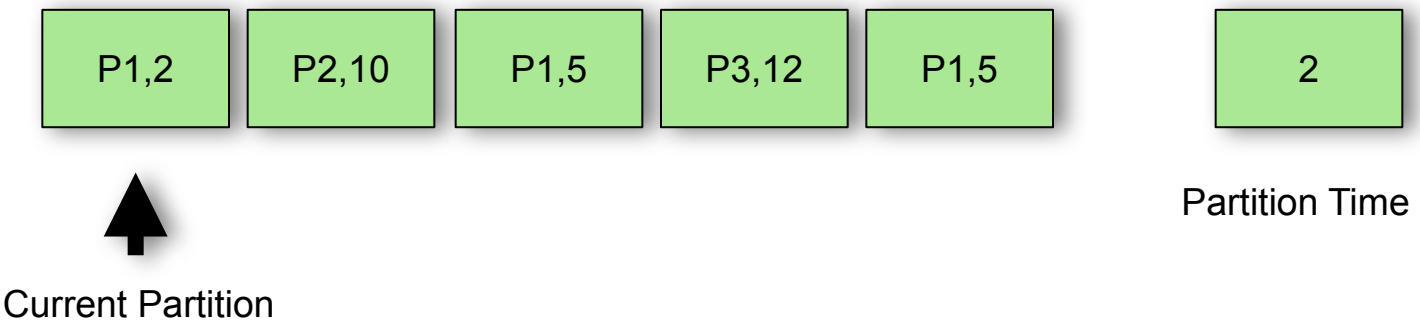


# Only Kernel Change: Partition Scheduling



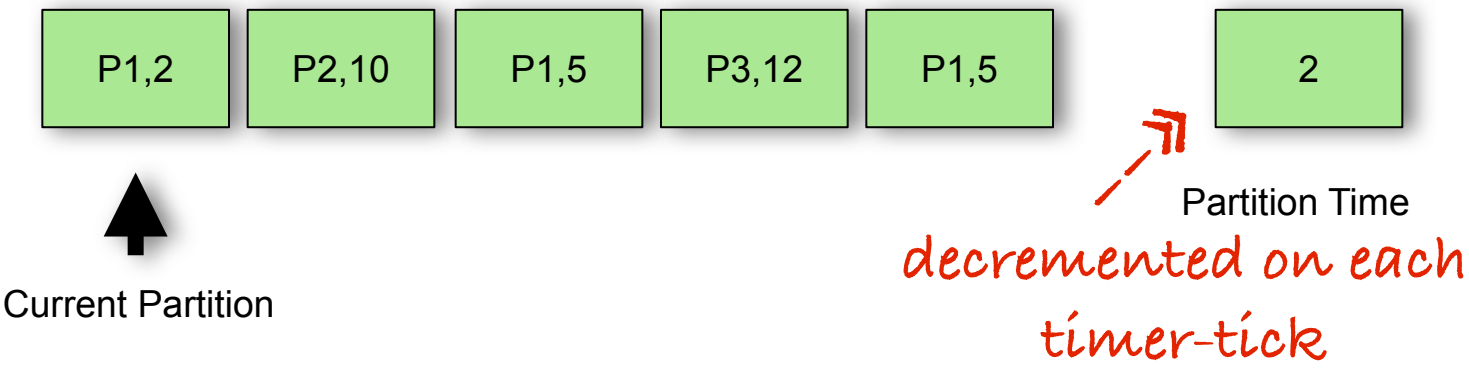
# Only Kernel Change: Partition Scheduling

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



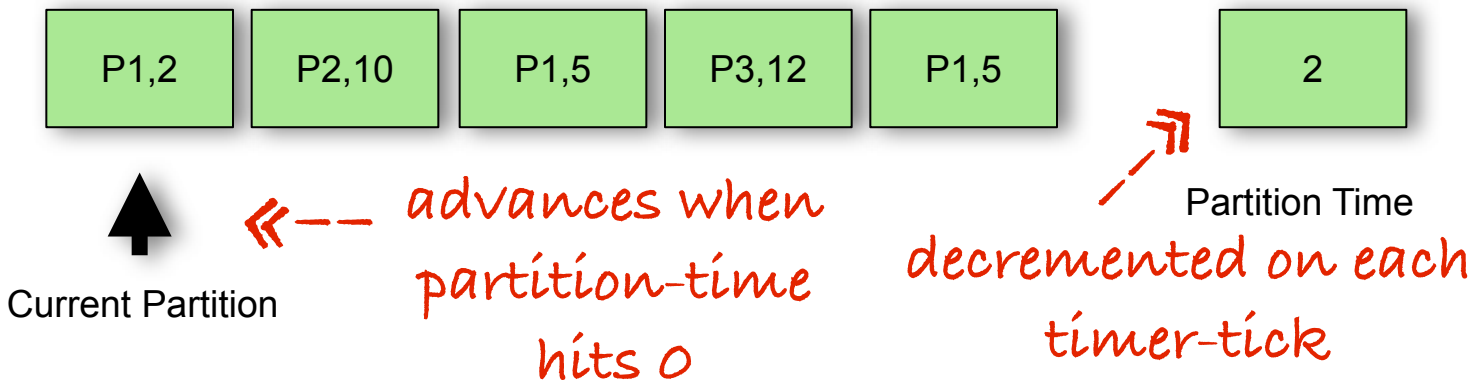
# Only Kernel Change: Partition Scheduling

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



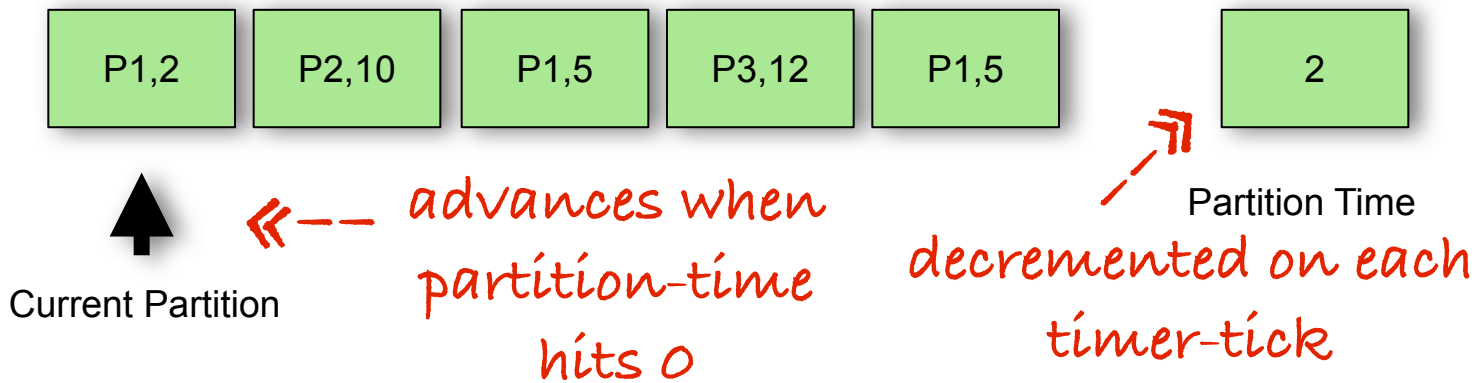
# Only Kernel Change: Partition Scheduling

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

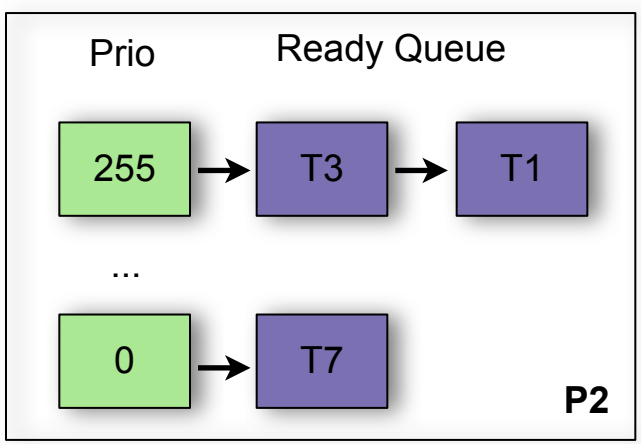


# Only Kernel Change: Partition Scheduling

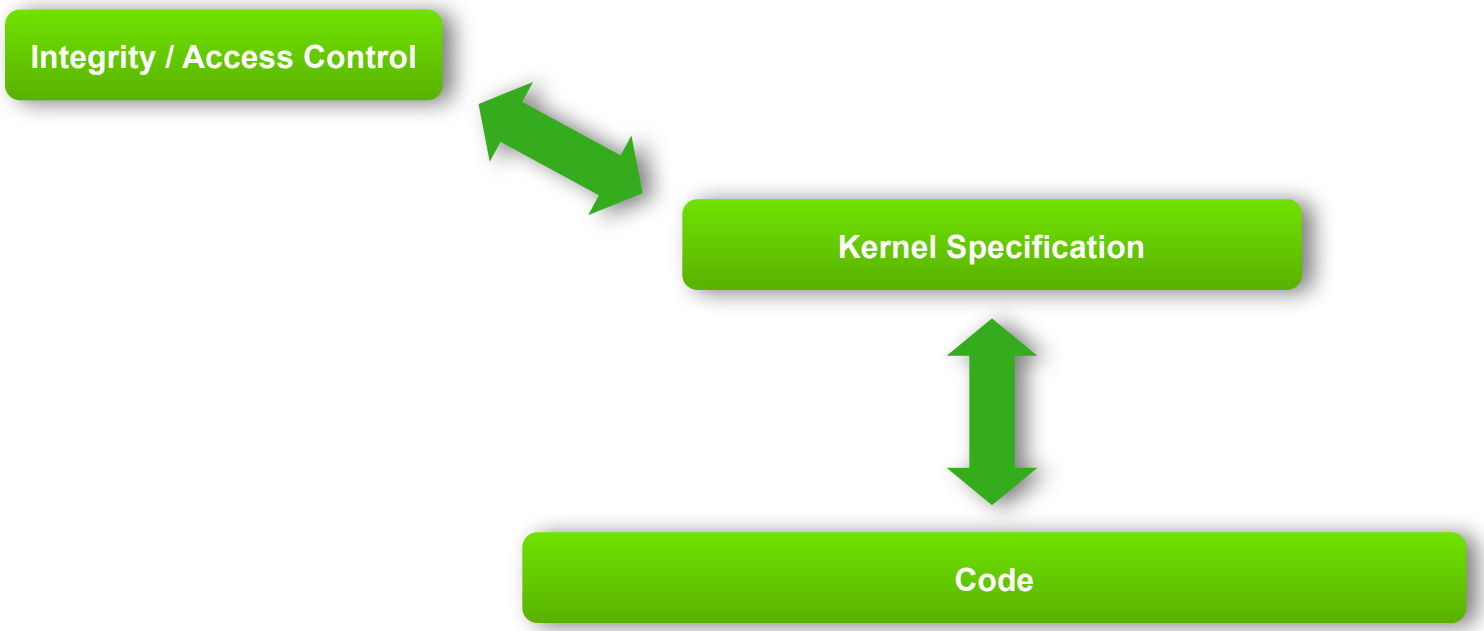
- Static round-robin schedule **between** partitions



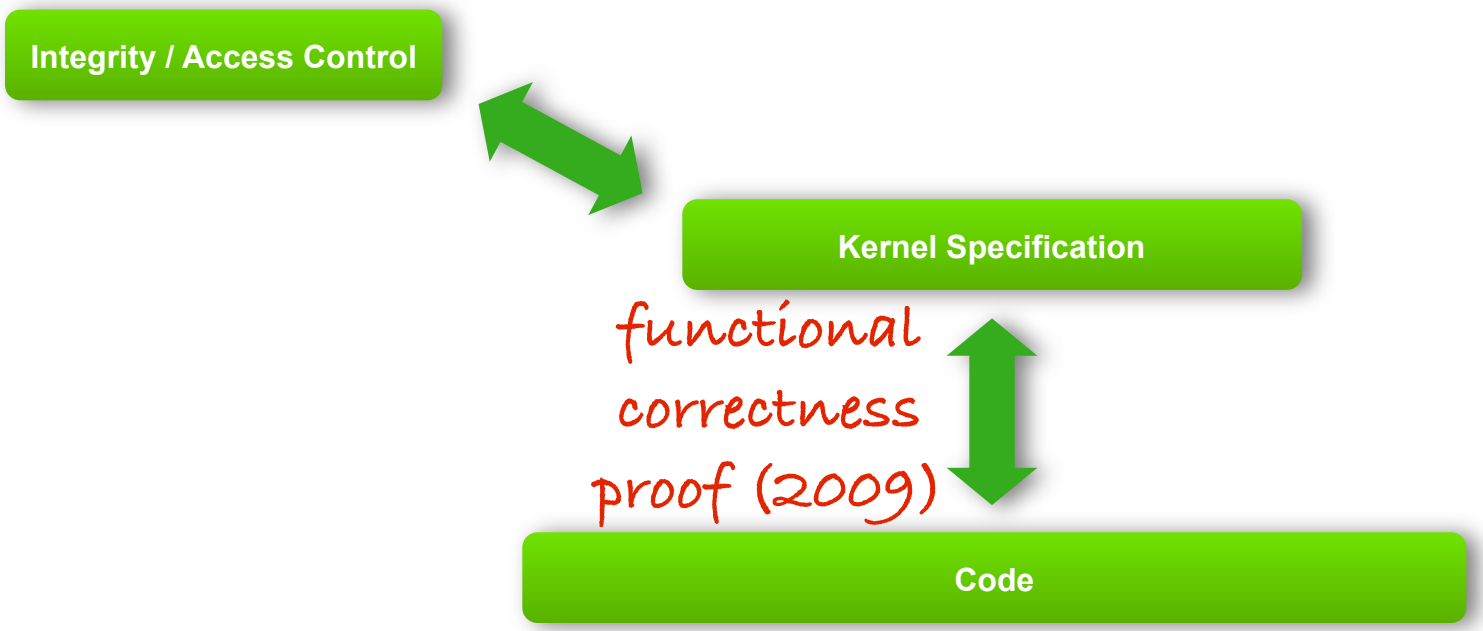

- Priority-based scheduling **within** partitions
  - Choose the highest-priority thread that is ready
  - Run idle thread if none ready
  - Any other intra-partition scheduling algorithm possible



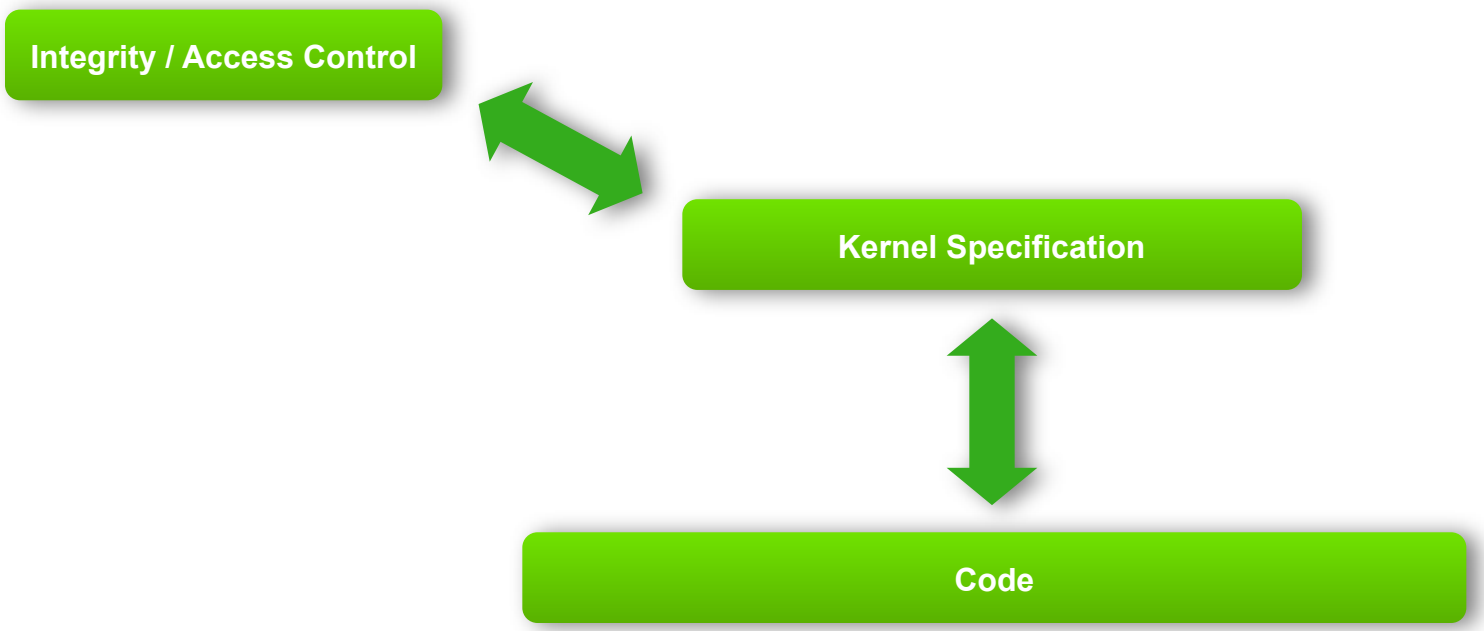
# Proof Structure



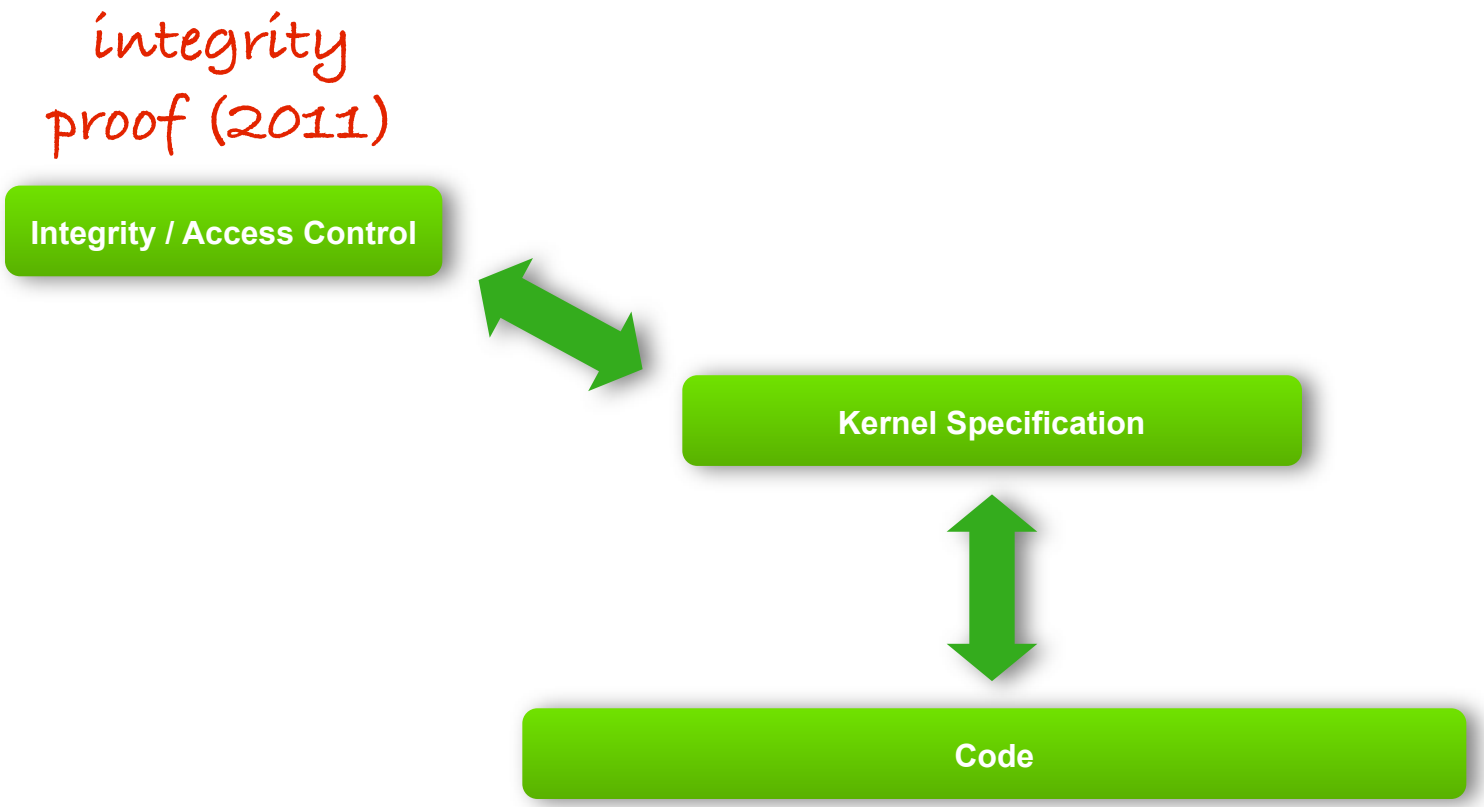
# Proof Structure



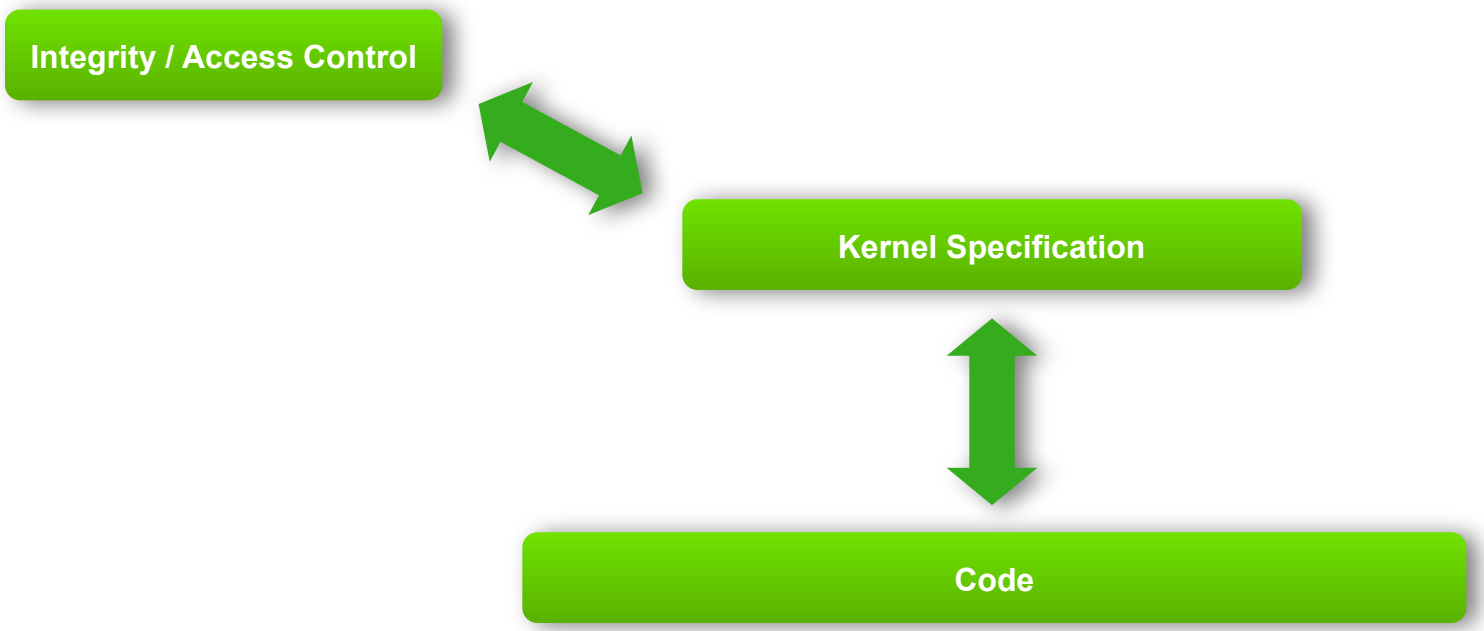
# Proof Structure



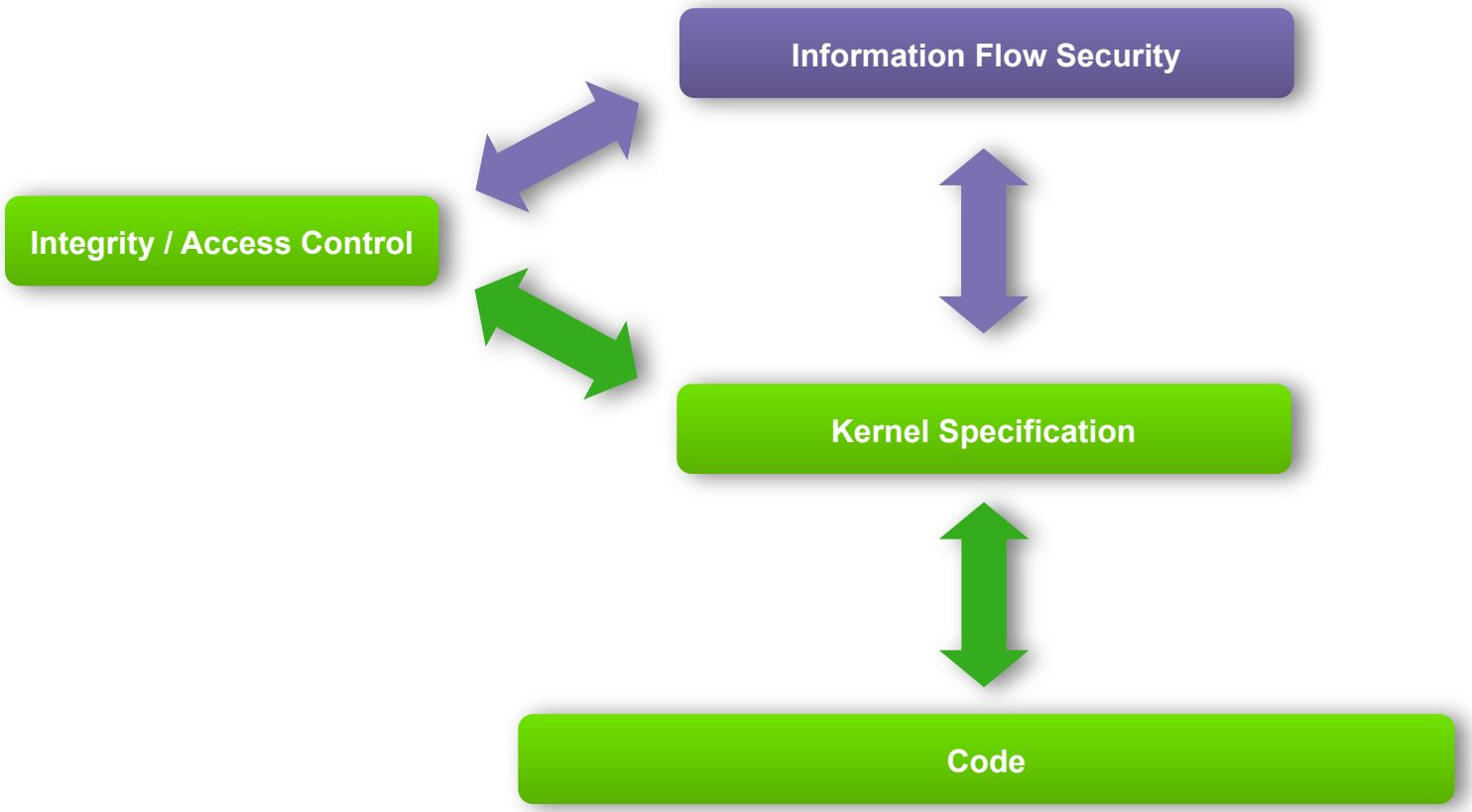
# Proof Structure



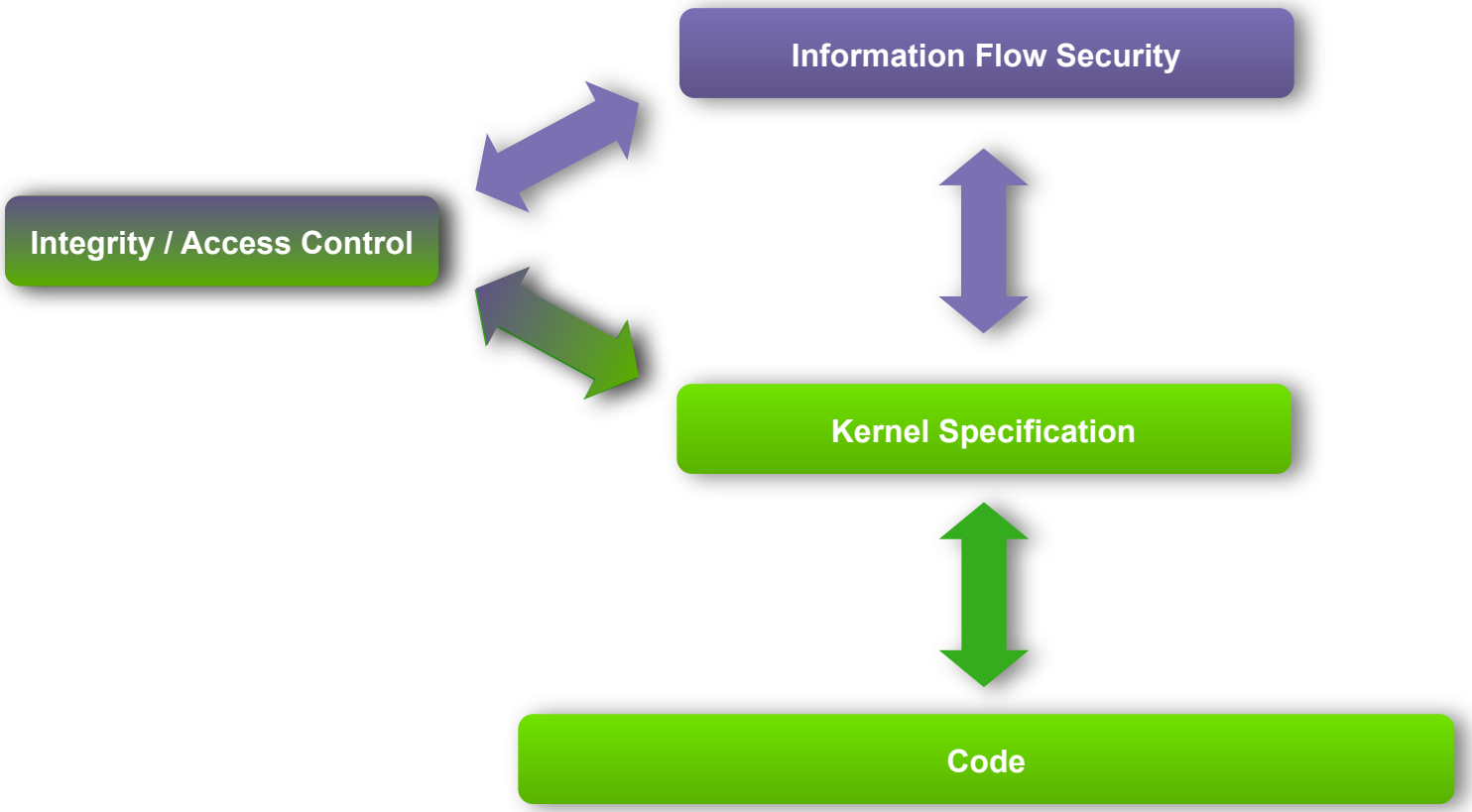
# Proof Structure



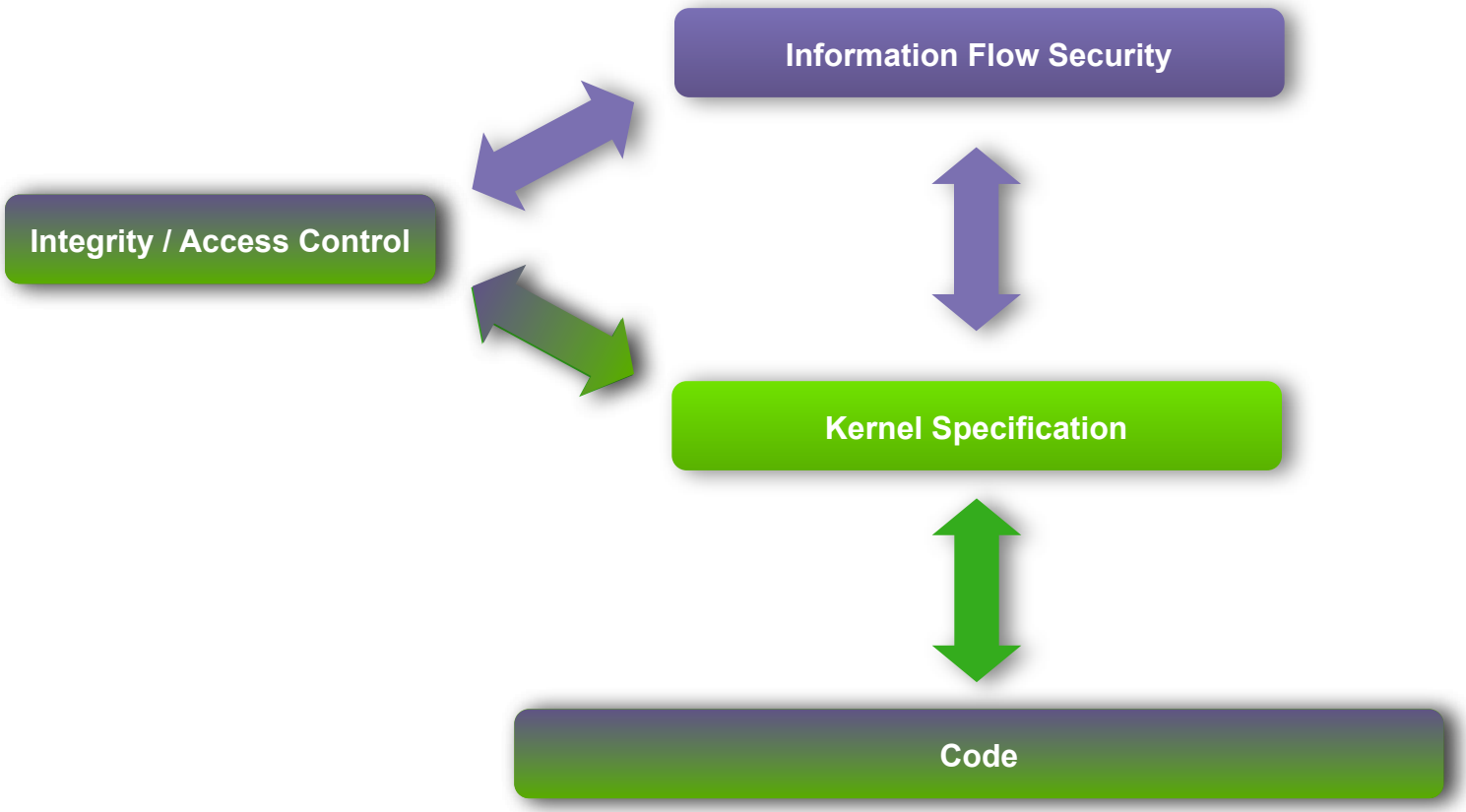
# Proof Structure



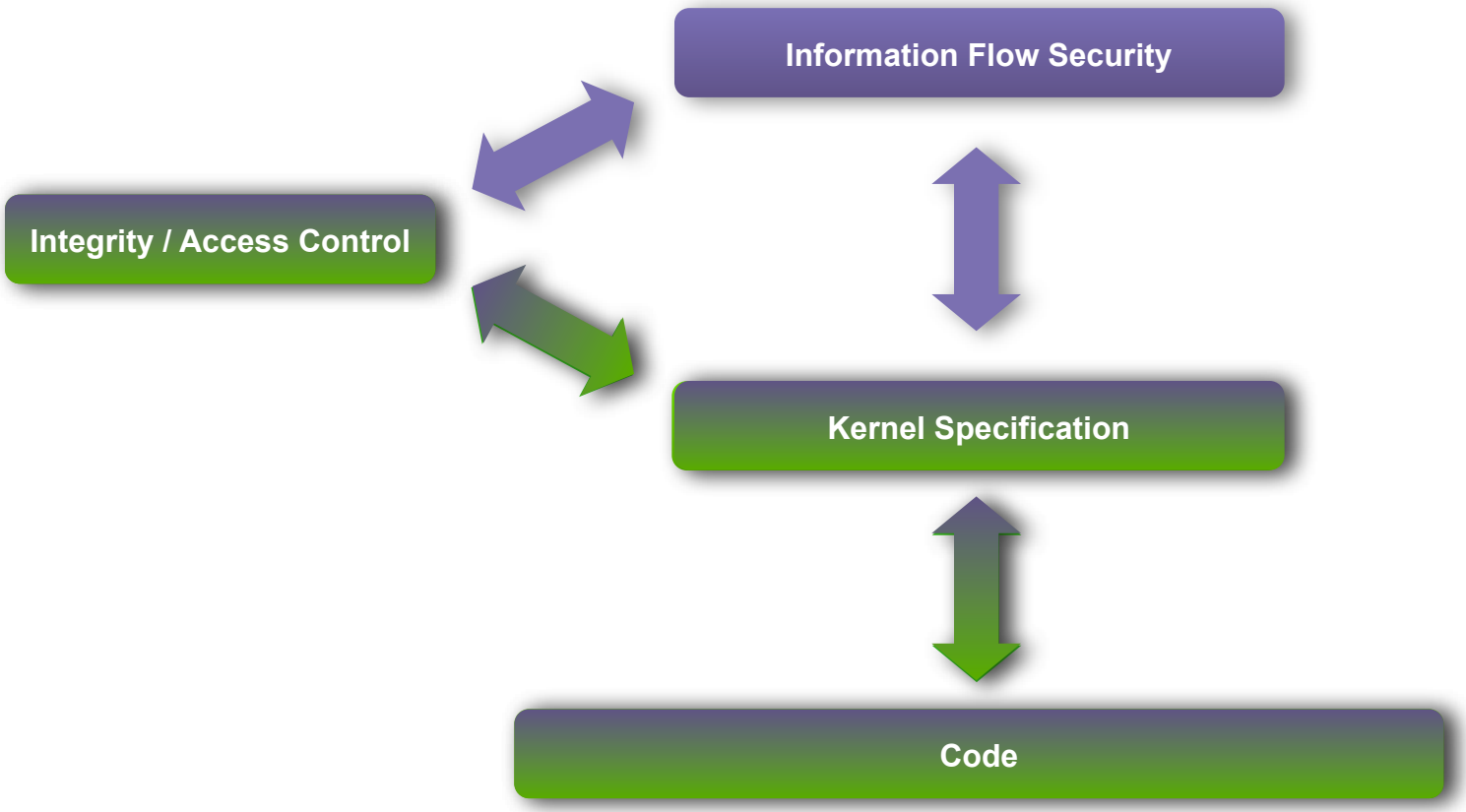
# Proof Structure



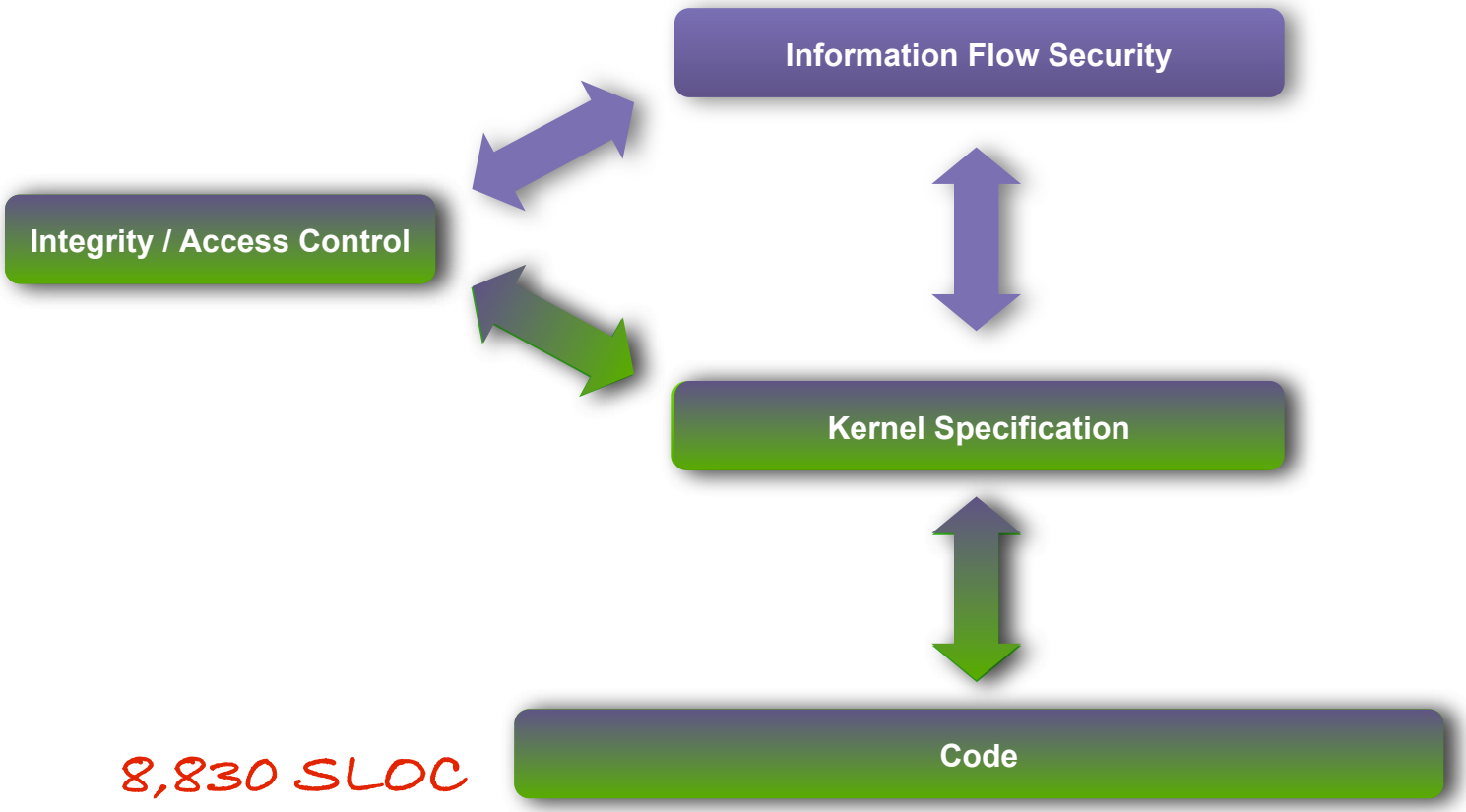
# Proof Structure



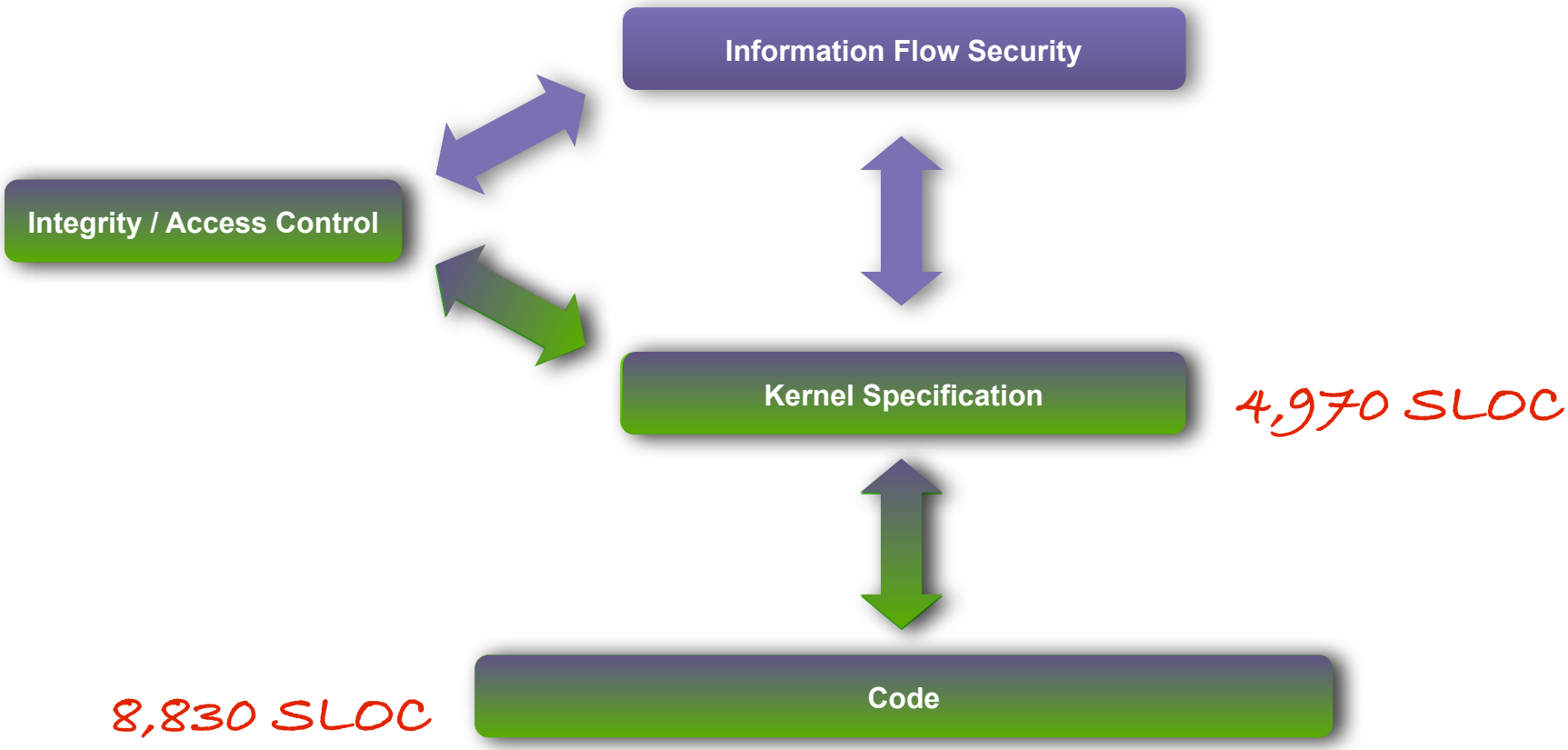
# Proof Structure



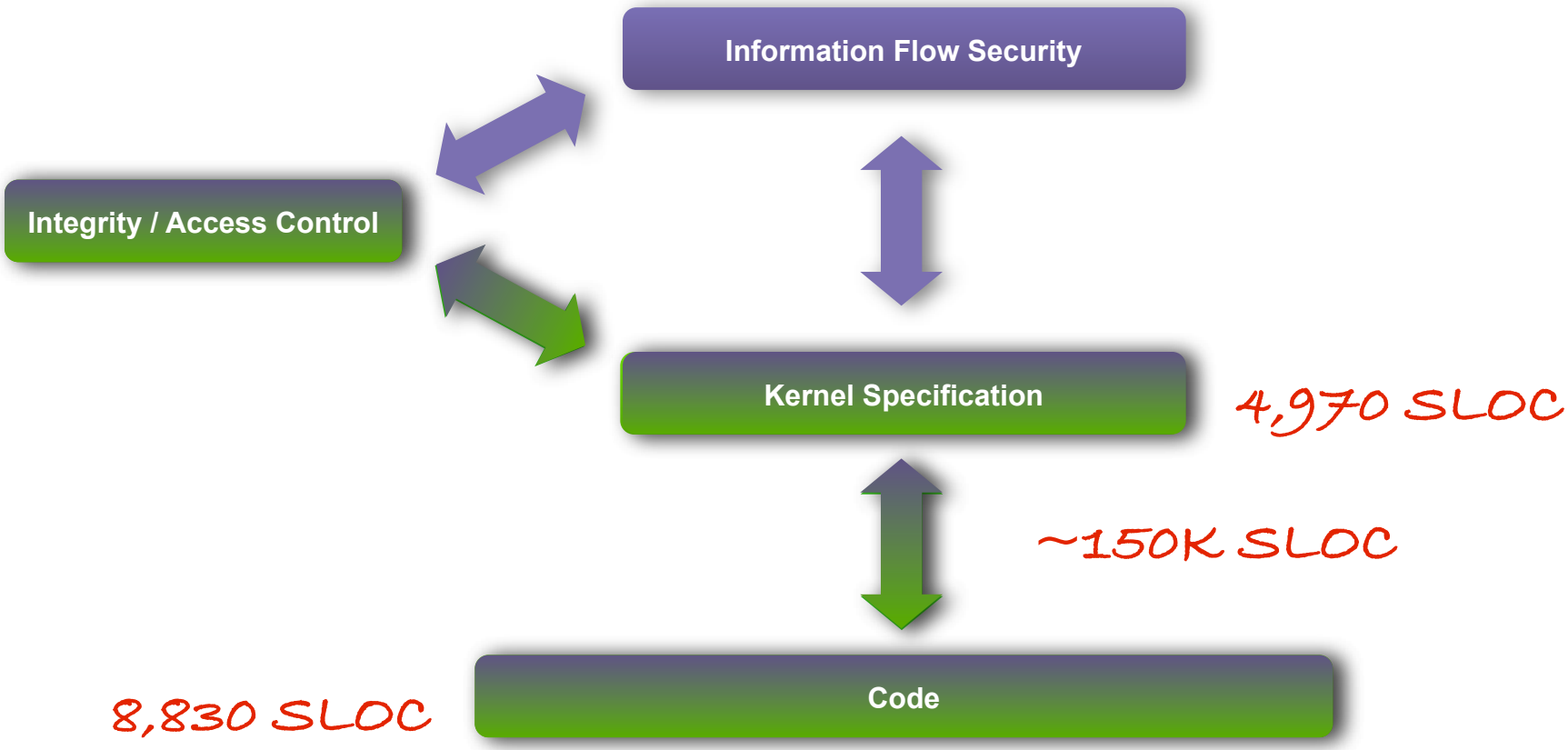
# Proof Structure



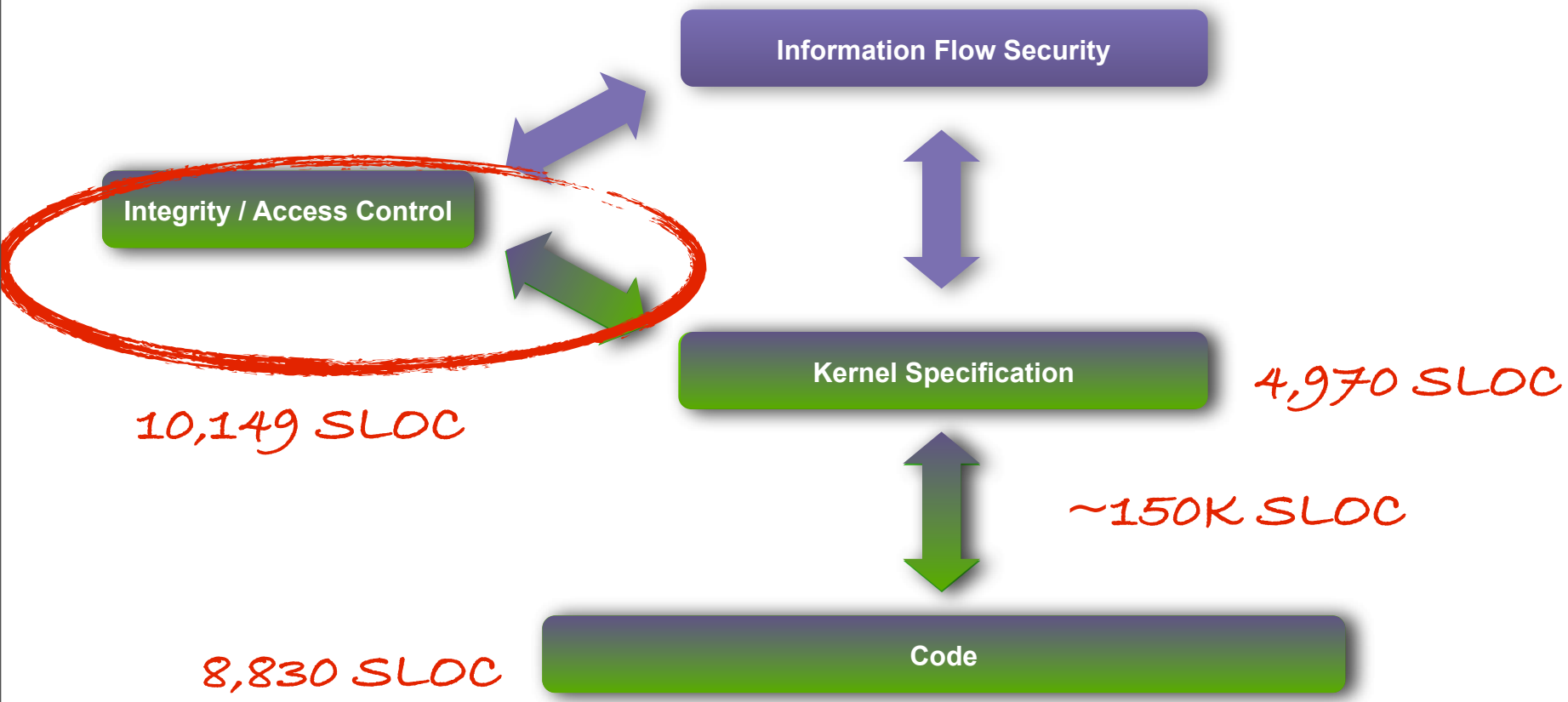
# Proof Structure



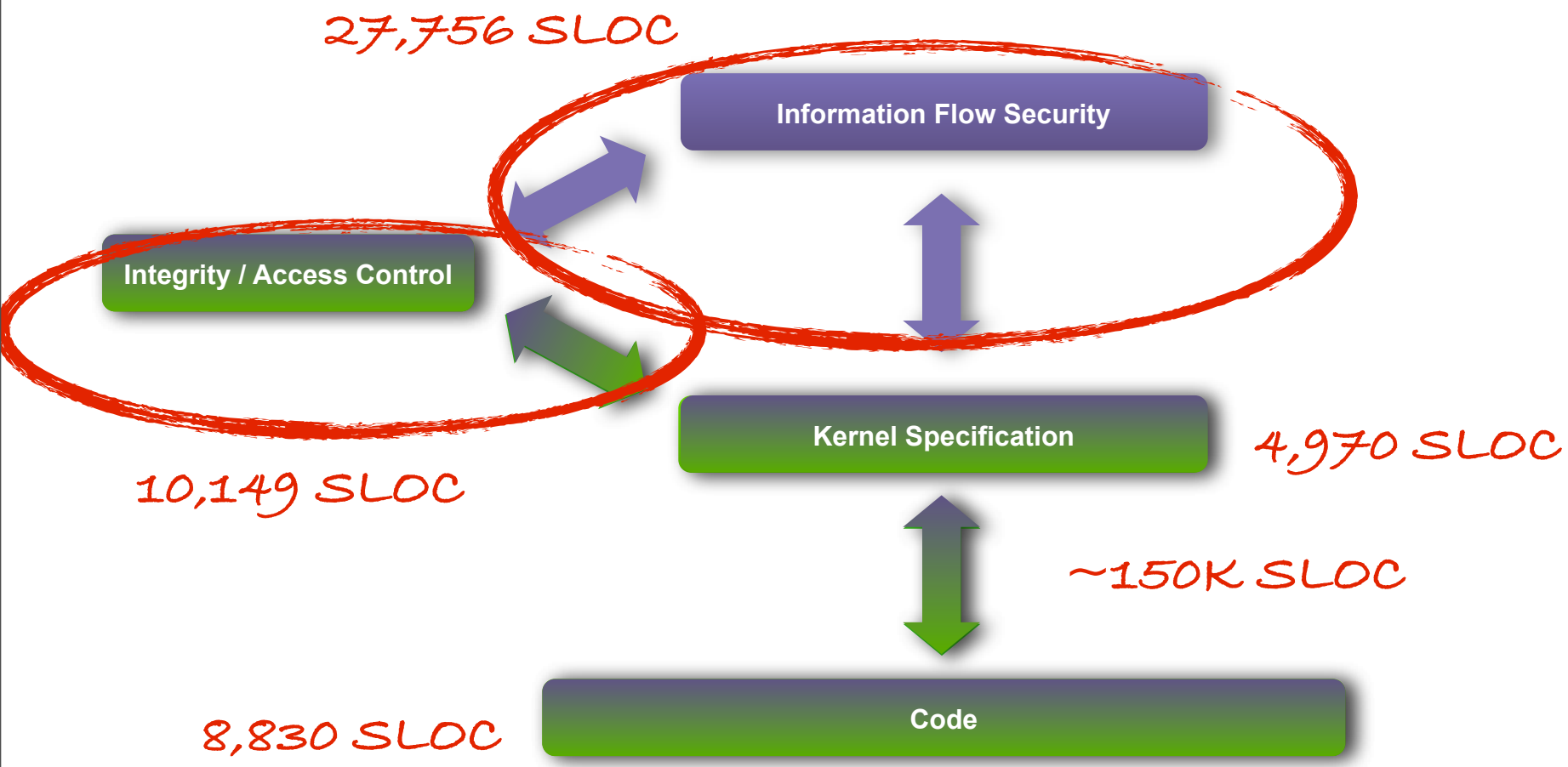
# Proof Structure



# Proof Structure



# Proof Structure



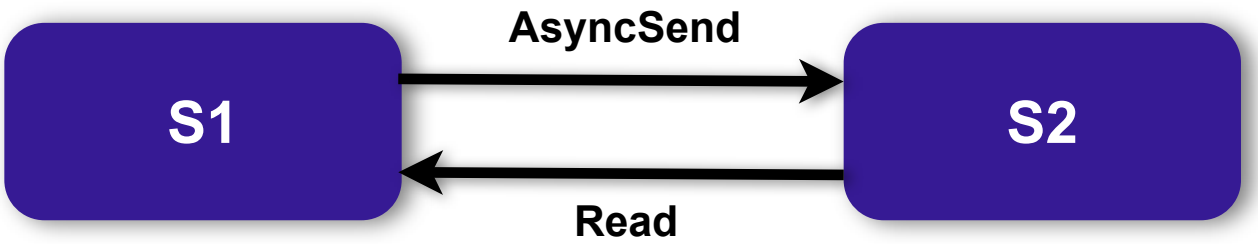
# INFORMATION FLOW

# INFORMATION FLOW

(confidentiality)

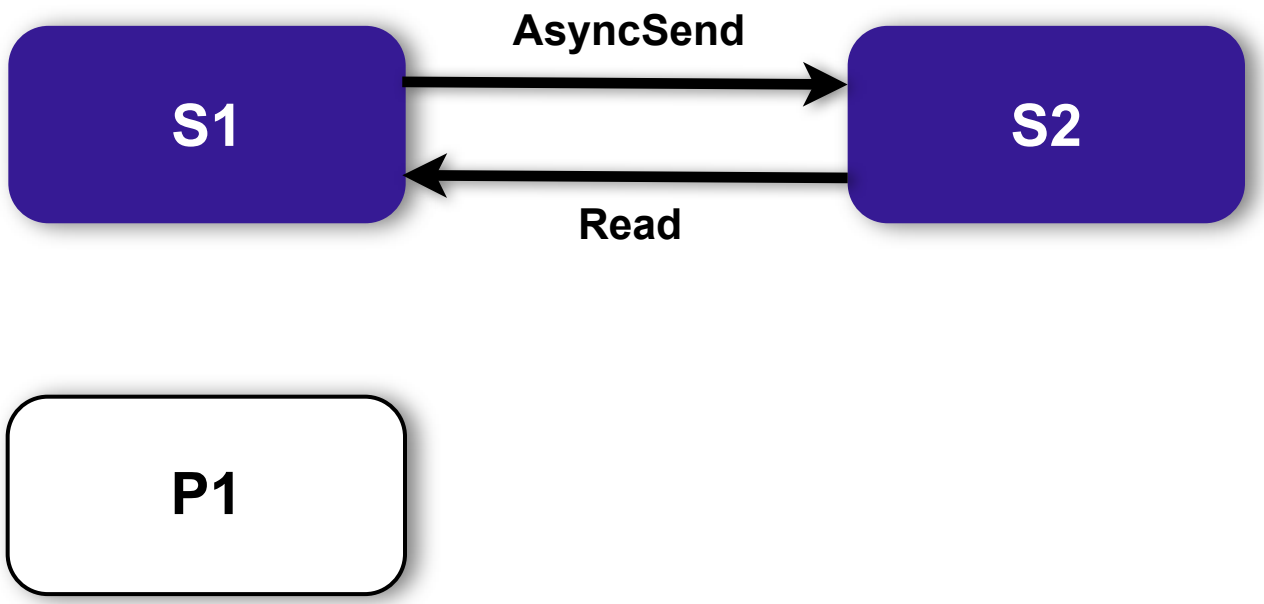
# Information Flow Policy

- Derived from access control policy



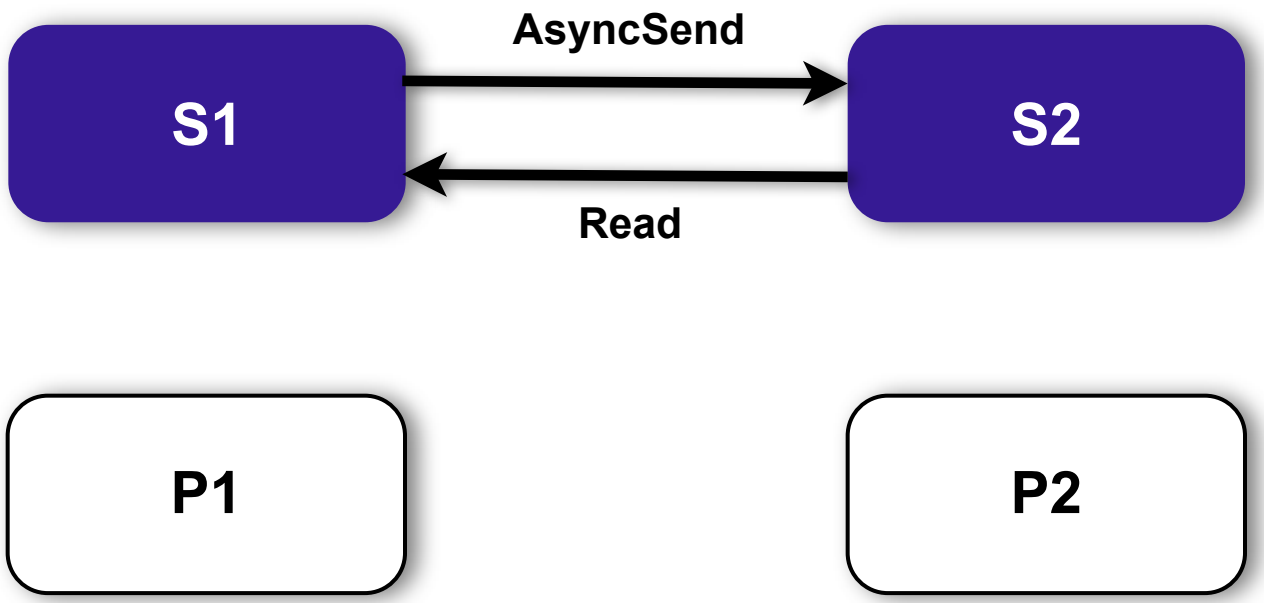
# Information Flow Policy

- Derived from access control policy



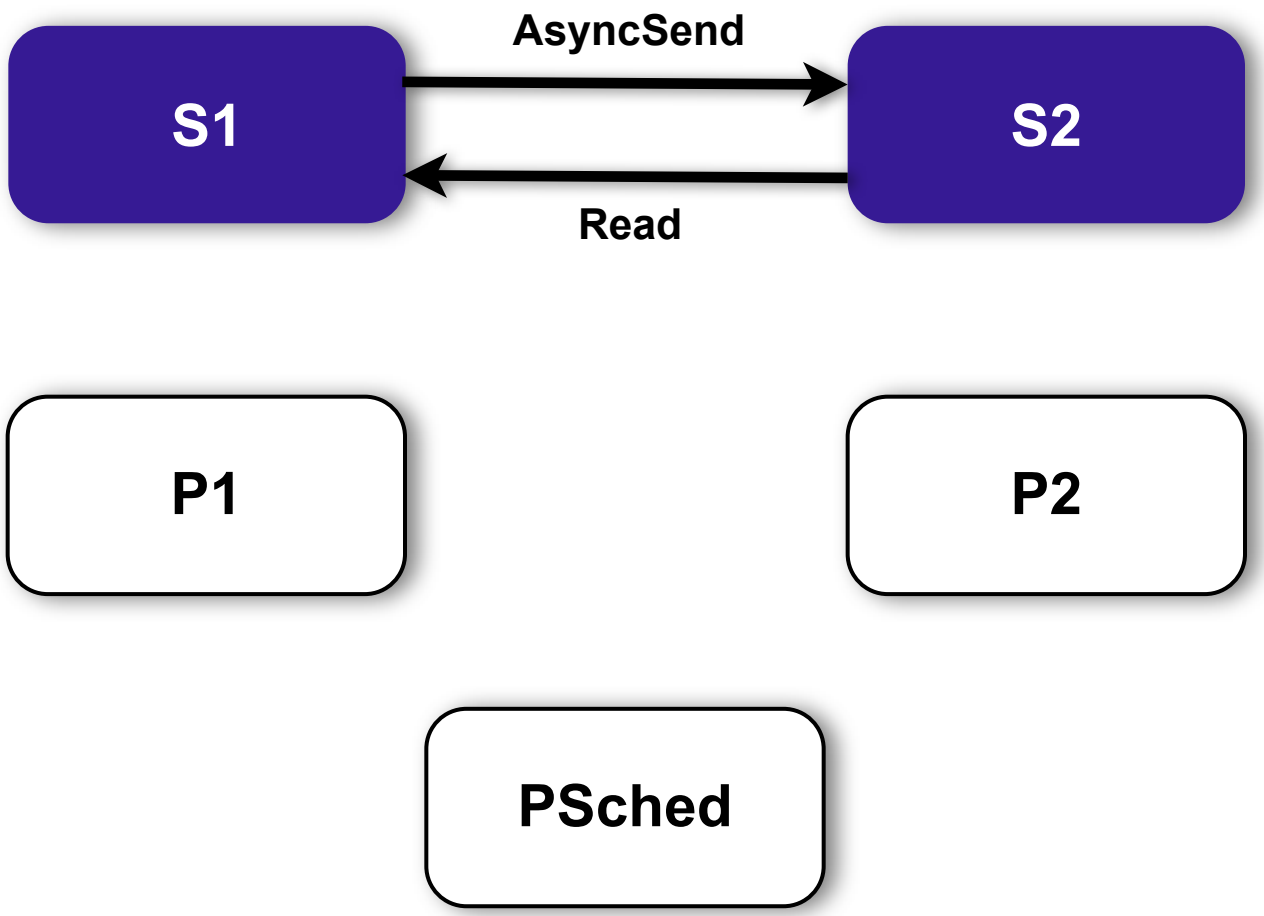
# Information Flow Policy

- Derived from access control policy



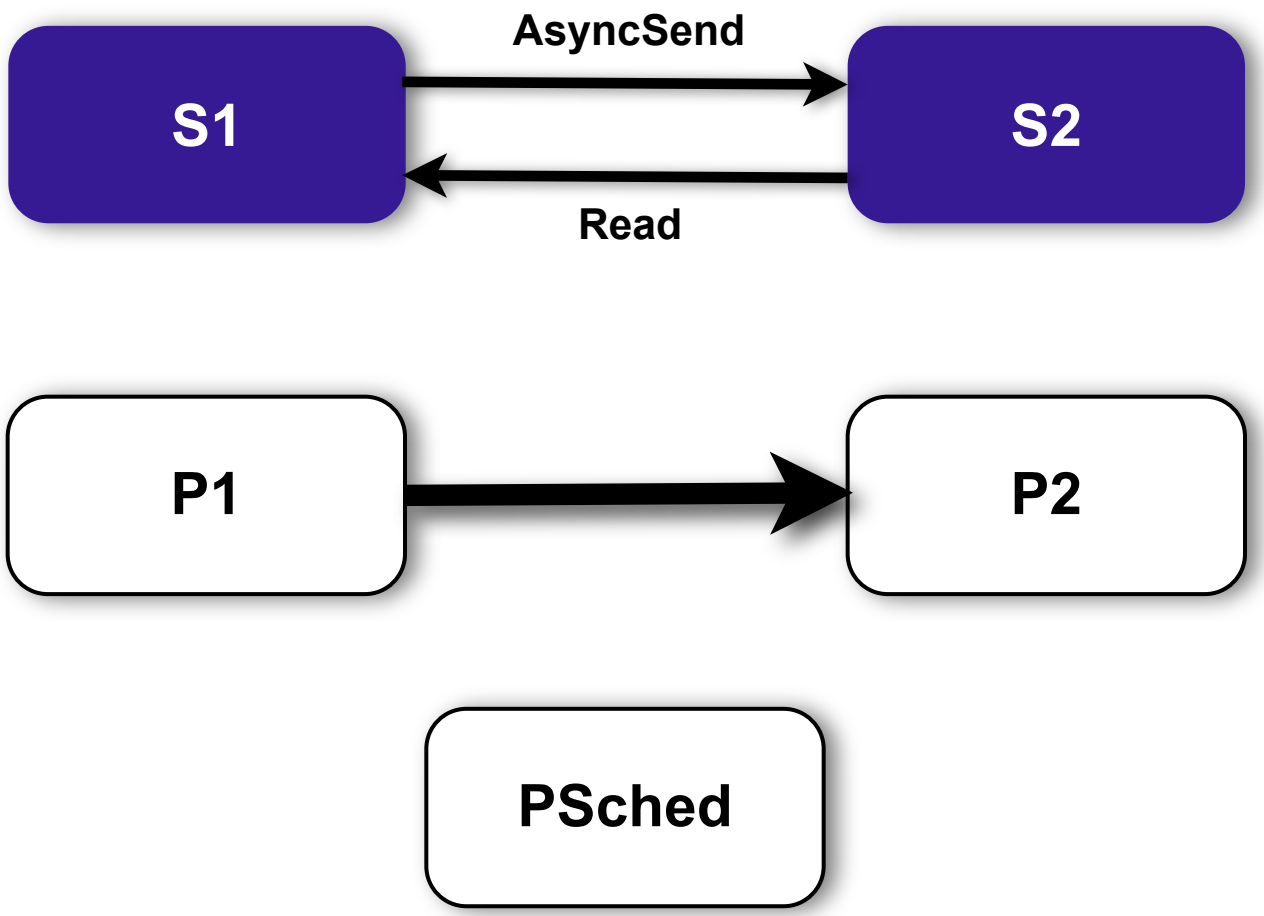
# Information Flow Policy

- Derived from access control policy



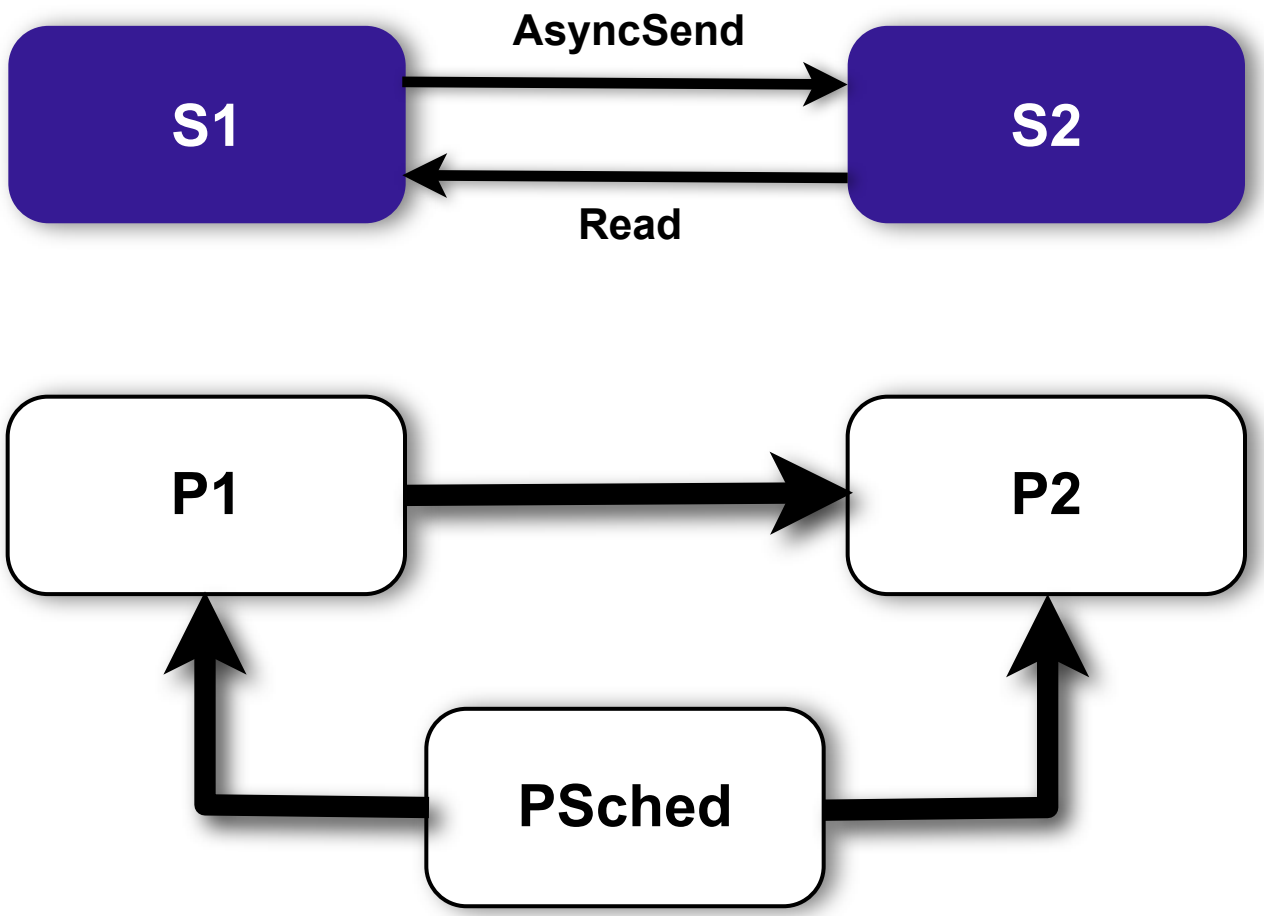
# Information Flow Policy

- Derived from access control policy



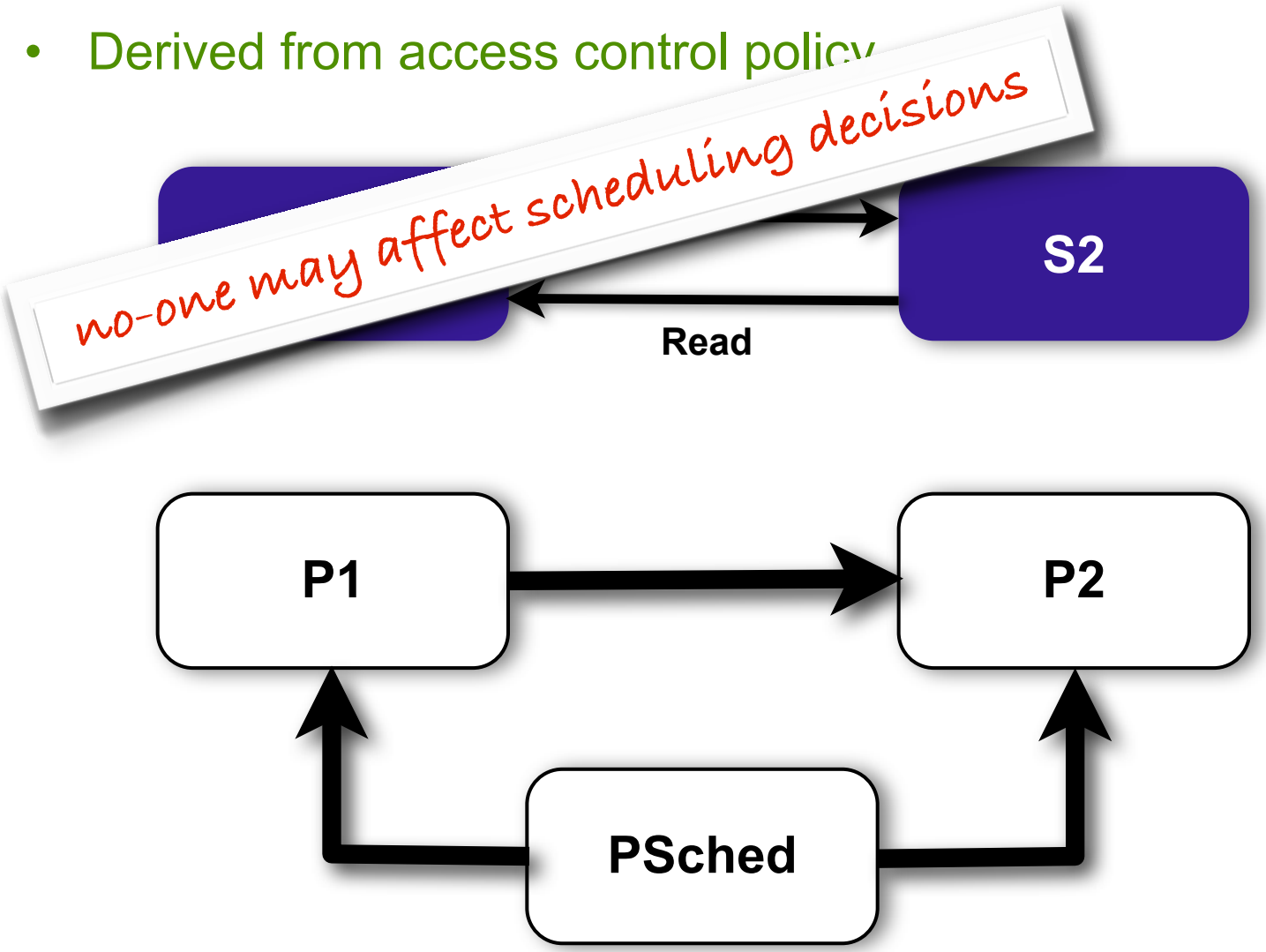
# Information Flow Policy

- Derived from access control policy



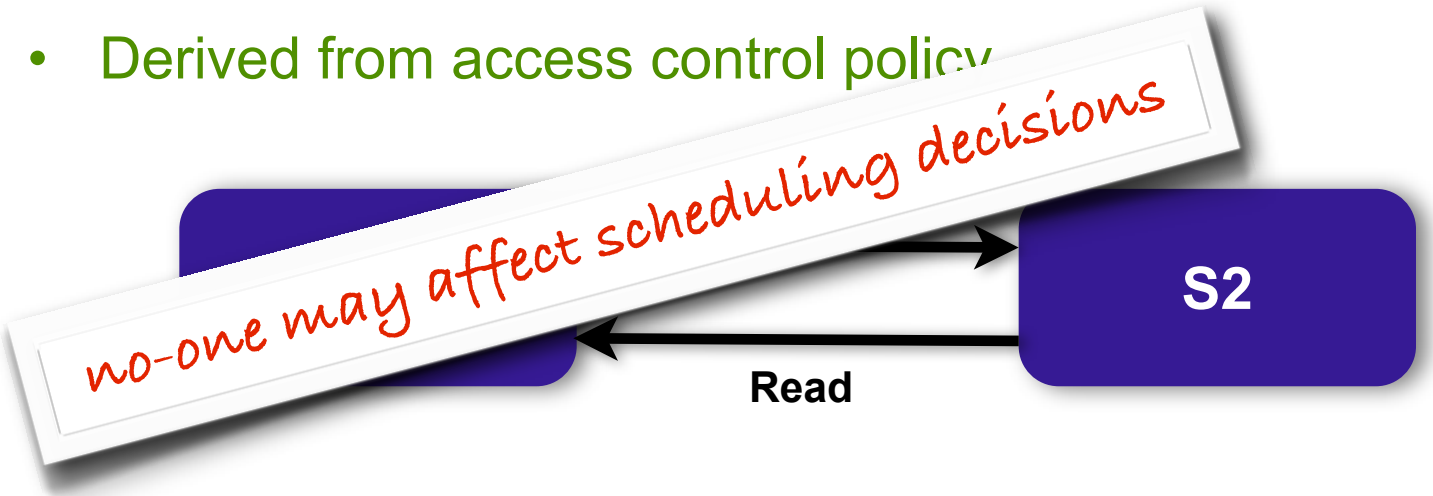
# Information Flow Policy

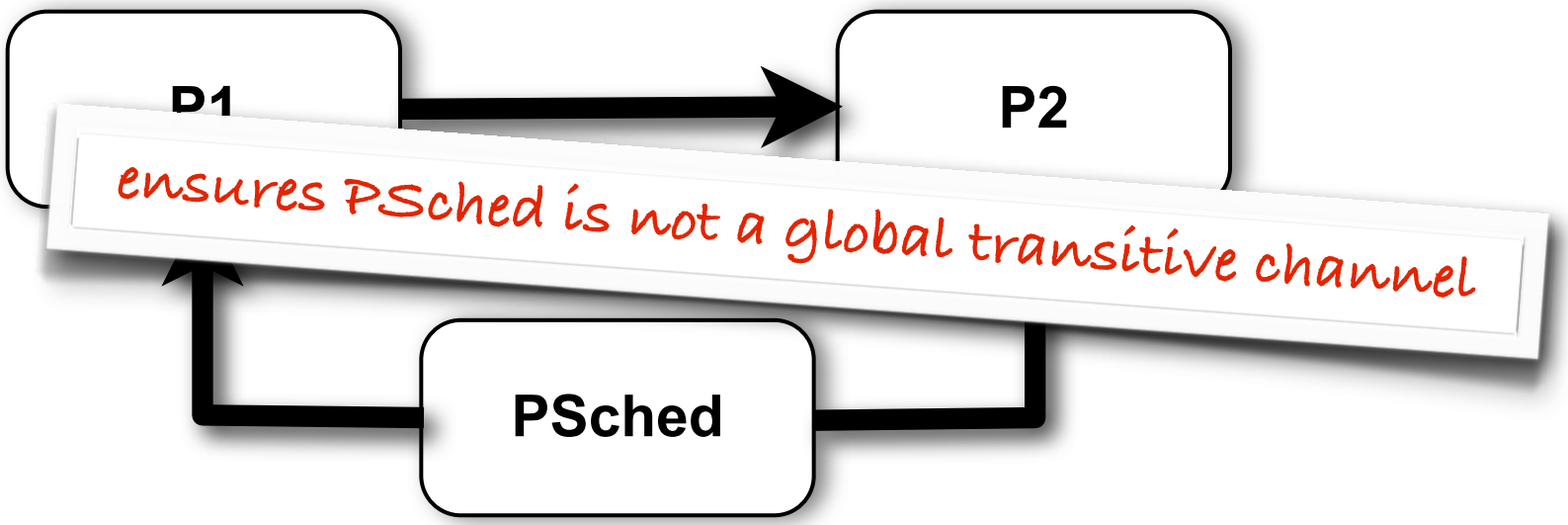
- Derived from access control policy



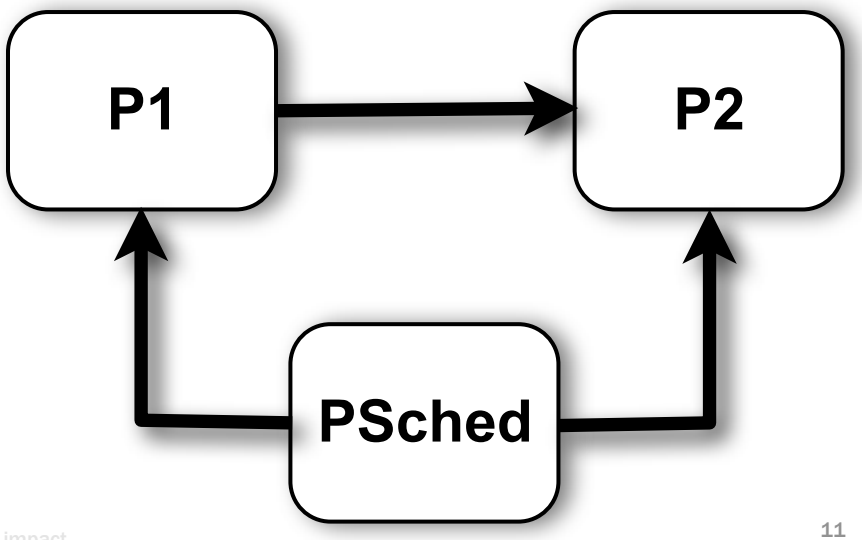
# Information Flow Policy

- Derived from access control policy



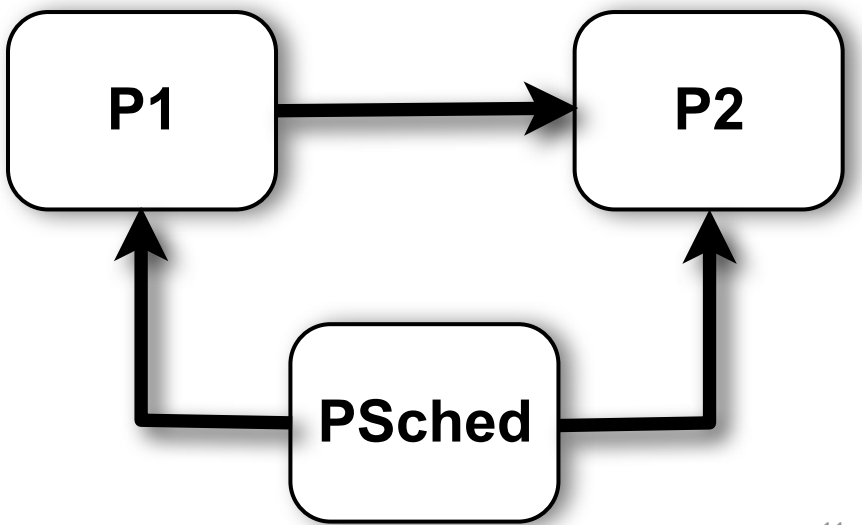



# Intransitive Nonleakage



# Intransitive Nonleakage

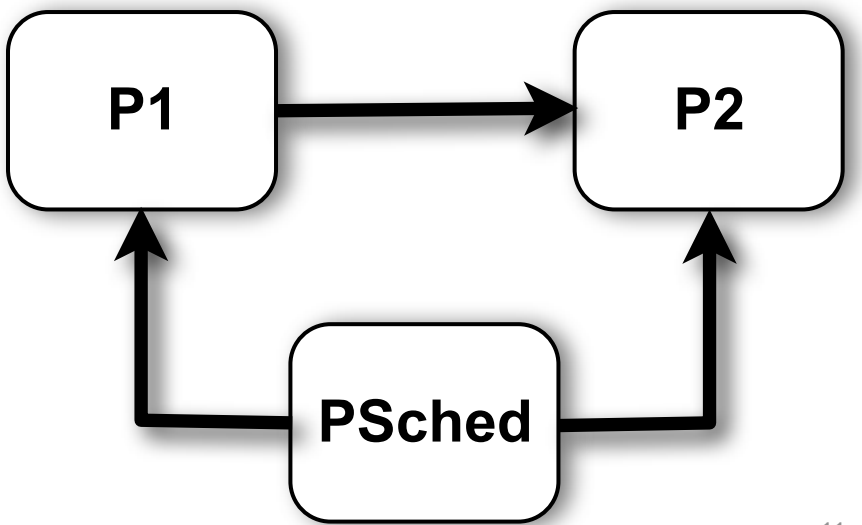
- Variant of **intransitive noninterference**
  - Asserts absence of information leaks



# Intransitive Nonleakage

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

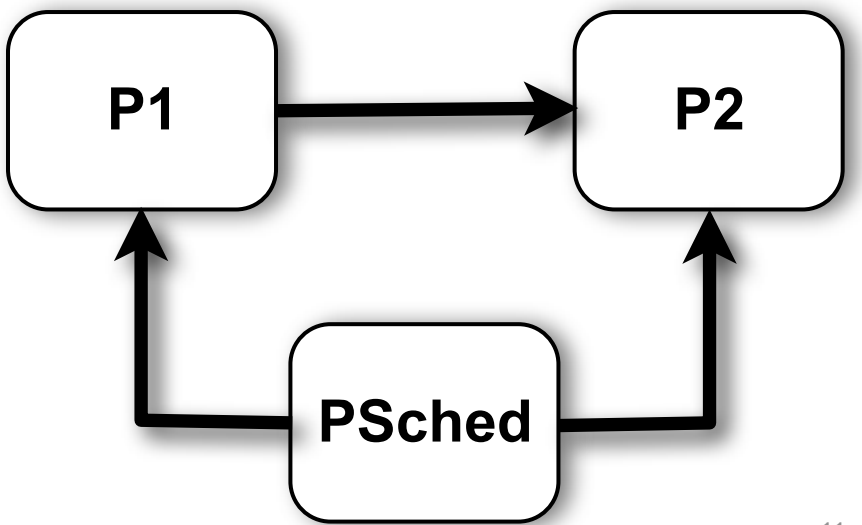
system model
(current partition)



# Intransitive Nonleakage

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

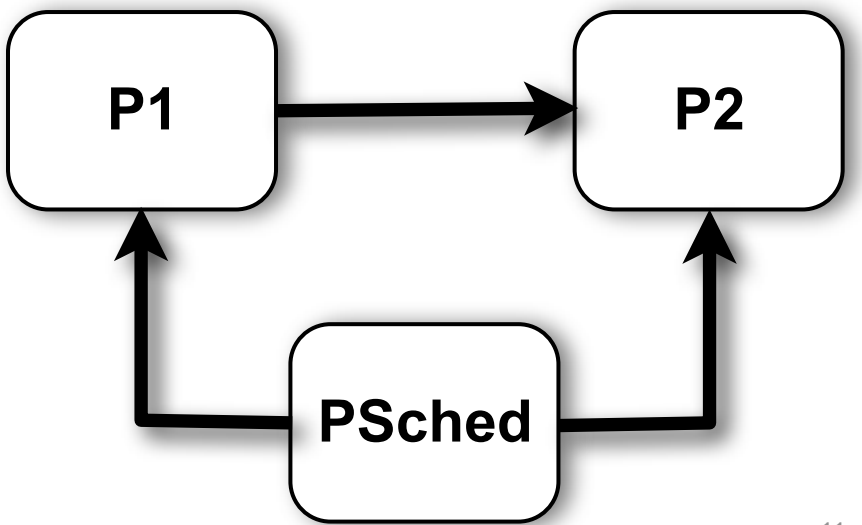
system model (current partition) → (**P2**)



# Intransitive Nonleakage

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

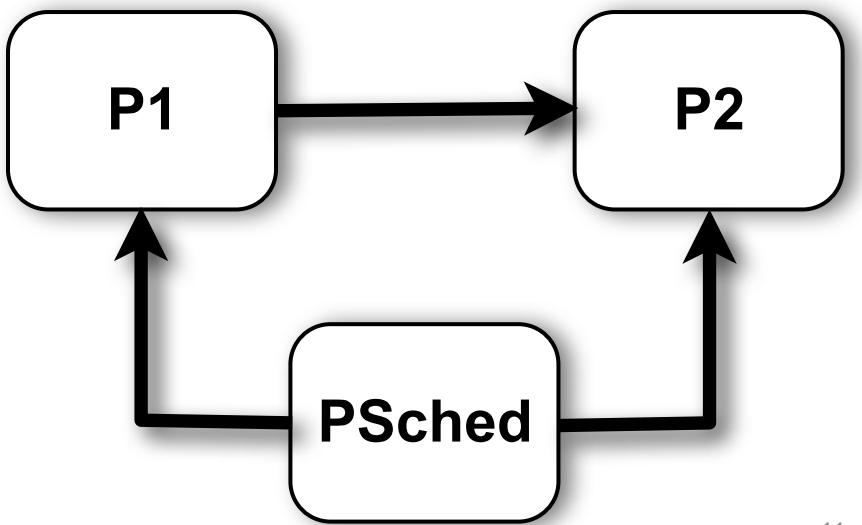
system model (current partition) → (**P2**) → (**PSched**)



# Intransitive Nonleakage

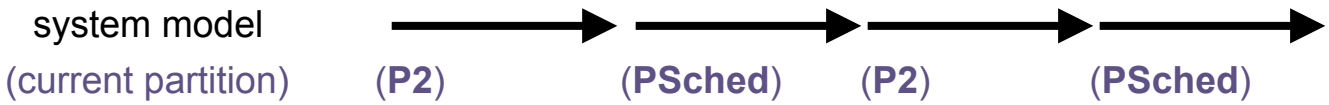
- Variant of **intransitive noninterference**
  - Asserts absence of information leaks

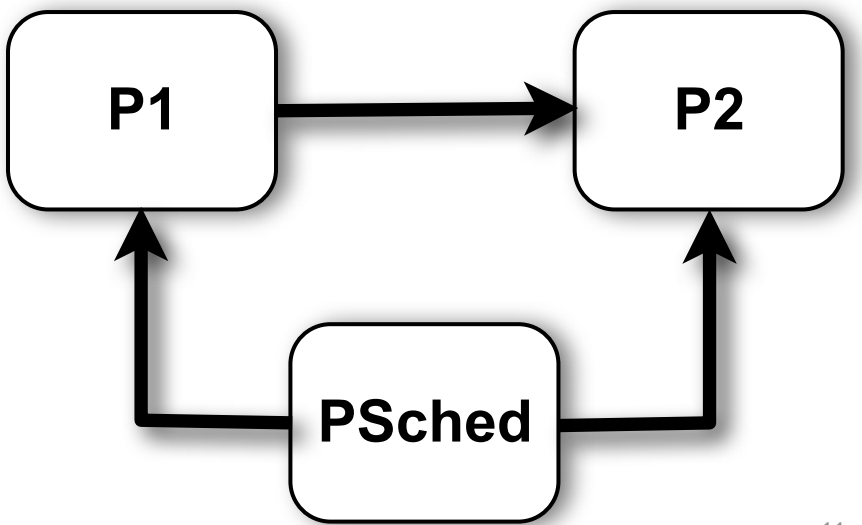
system model (current partition) → (**P2**) → (**PSched**) → (**P2**)



# Intransitive Nonleakage

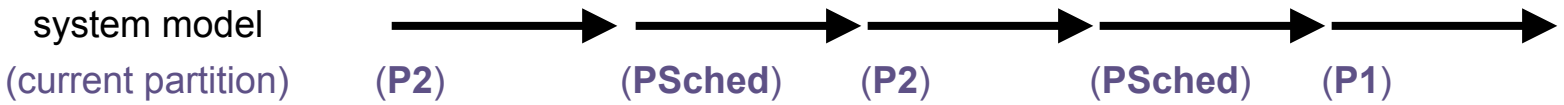
- Variant of **intransitive noninterference**
  - Asserts absence of information leaks

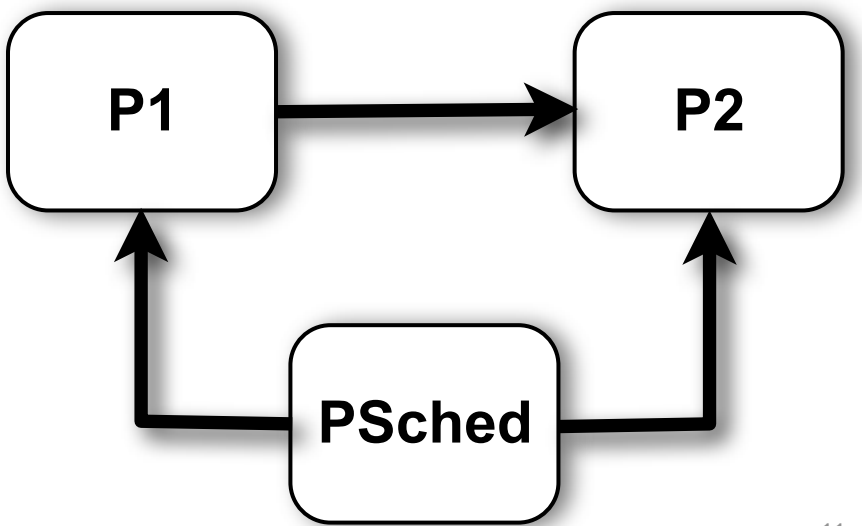


# Intransitive Nonleakage

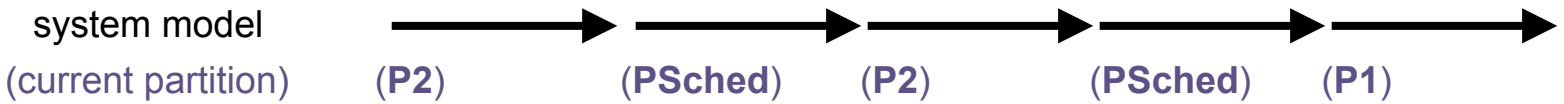
- Variant of **intransitive noninterference**
  - Asserts absence of information leaks



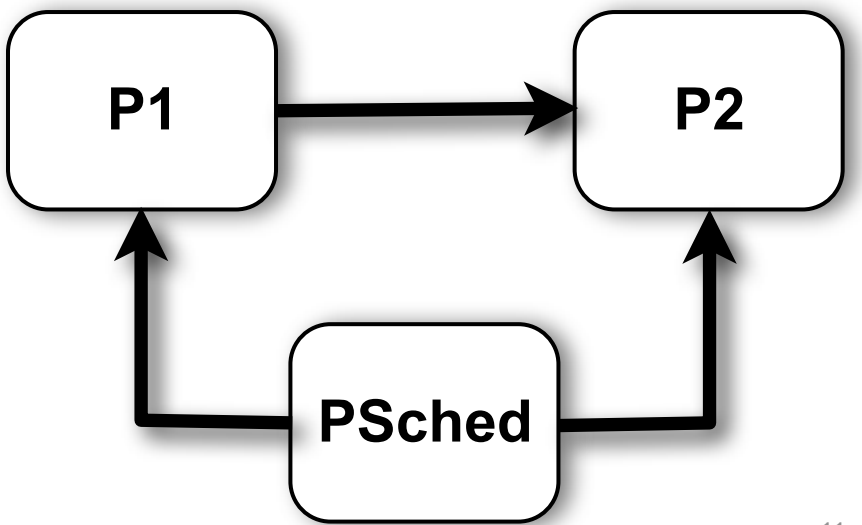



# Intransitive Nonleakage

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

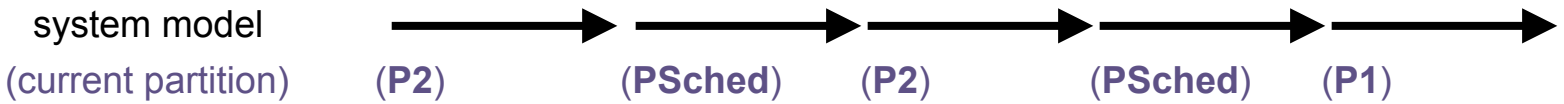



- Allows partitions to know of each others' existence
  - **P1** allowed to observe that **P2** has executed
  - But not to learn anything about **P2**'s state



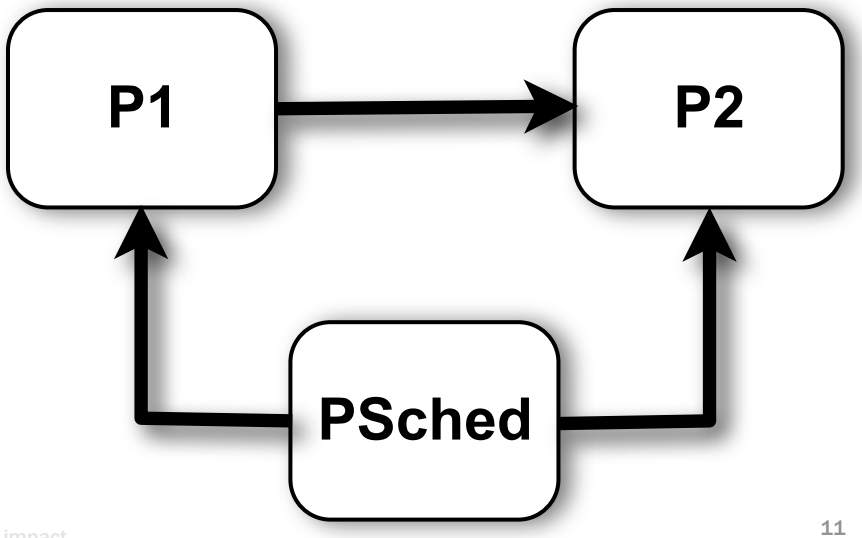
# Intransitive Nonleakage

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

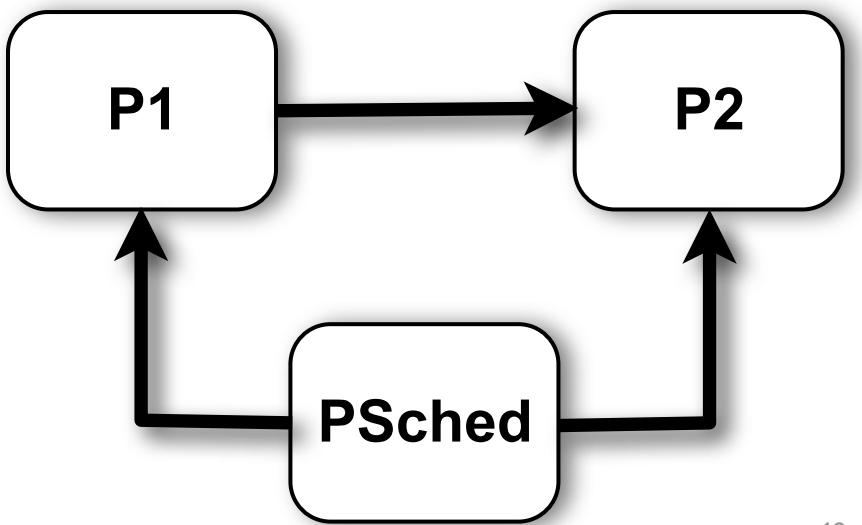



- Allows partitions to know of each others' existence
  - **P1** allowed to observe that **P2** has executed
  - But not to learn anything about **P2**'s state

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

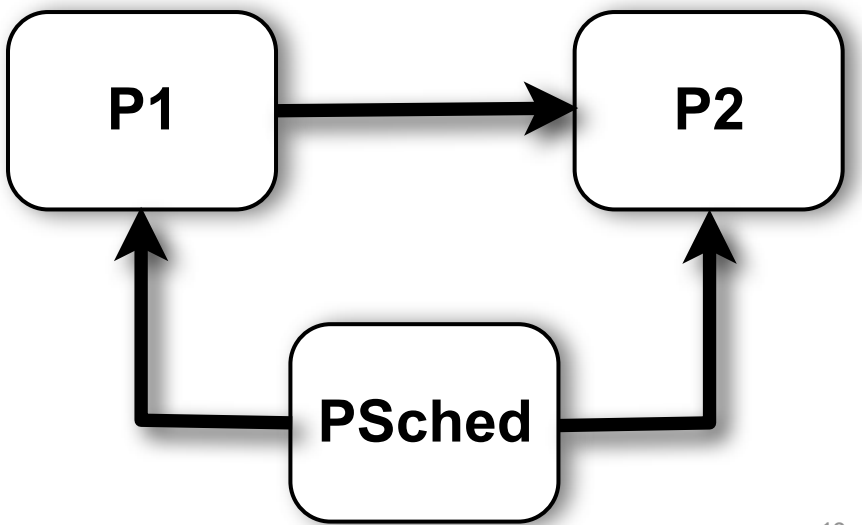


# Intransitive Nonleakage: Formally



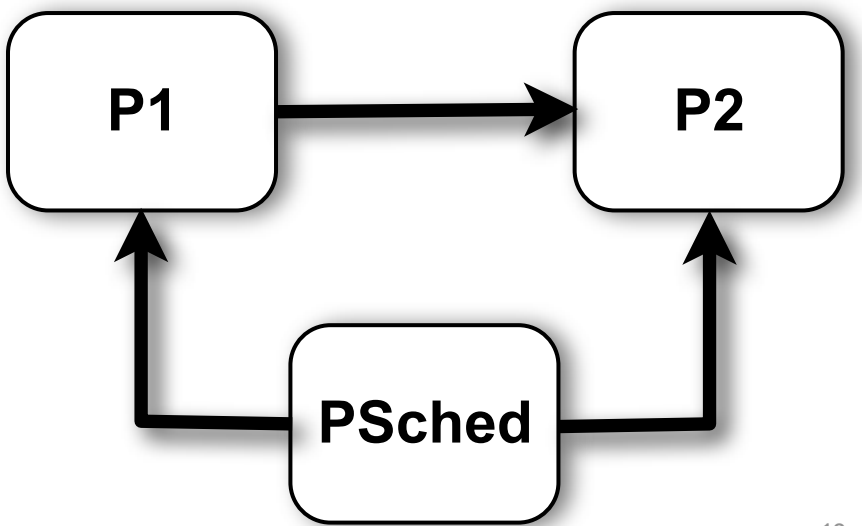
# Intransitive Nonleakage: Formally

- Similar to language-based noninterference



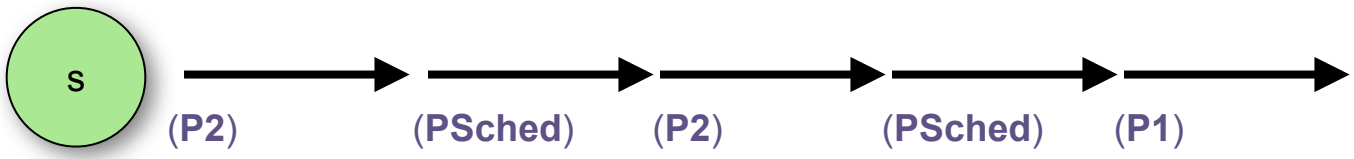
# Intransitive Nonleakage: Formally

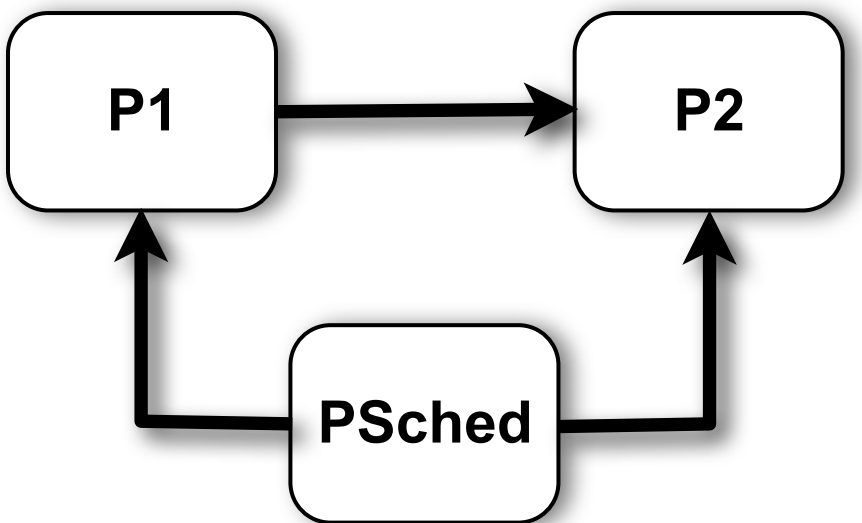
- Similar to language-based noninterference

# Intransitive Nonleakage: Formally

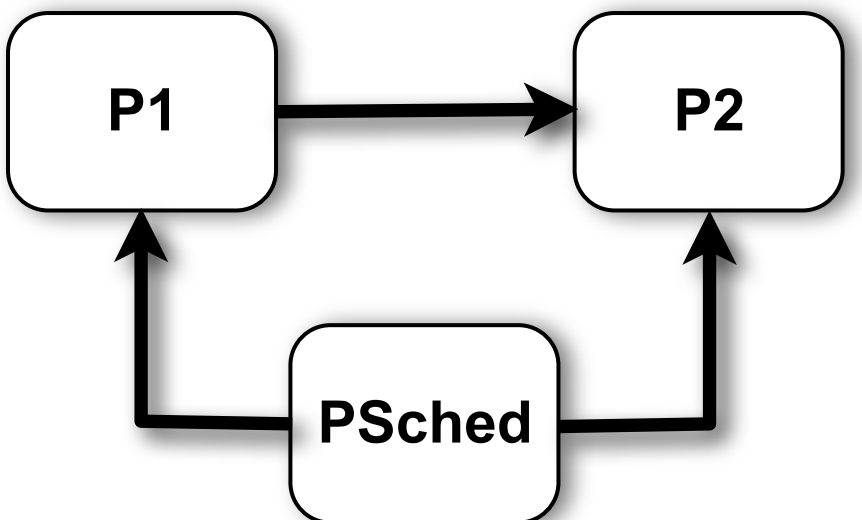
- Similar to language-based noninterference



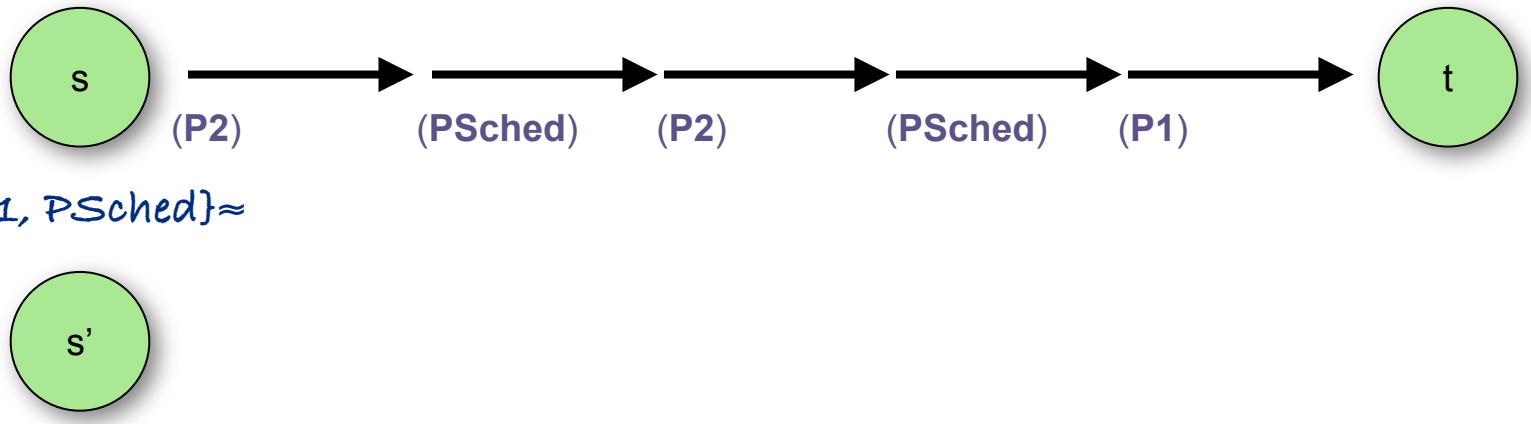
# Intransitive Nonleakage: Formally

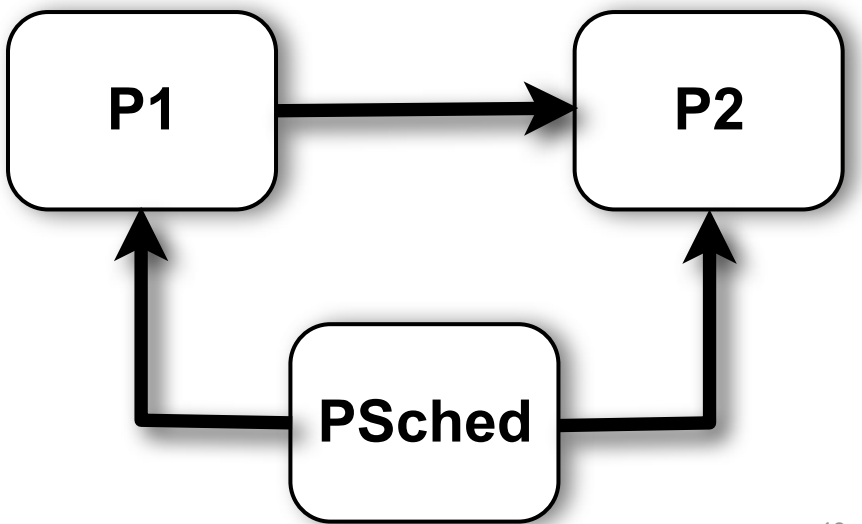
- Similar to language-based noninterference

# Intransitive Nonleakage: Formally

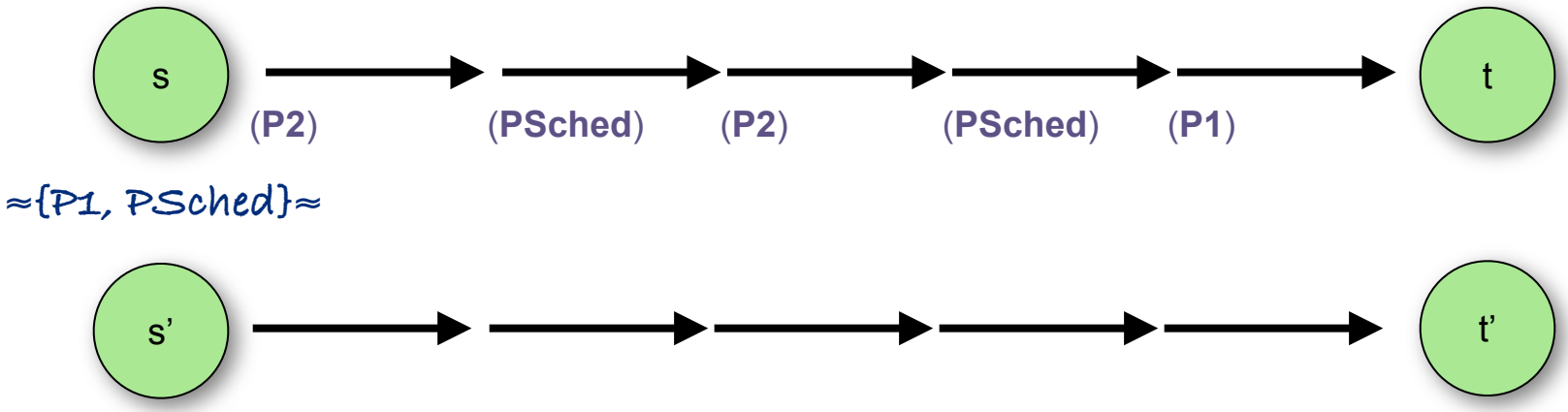
- Similar to language-based noninterference

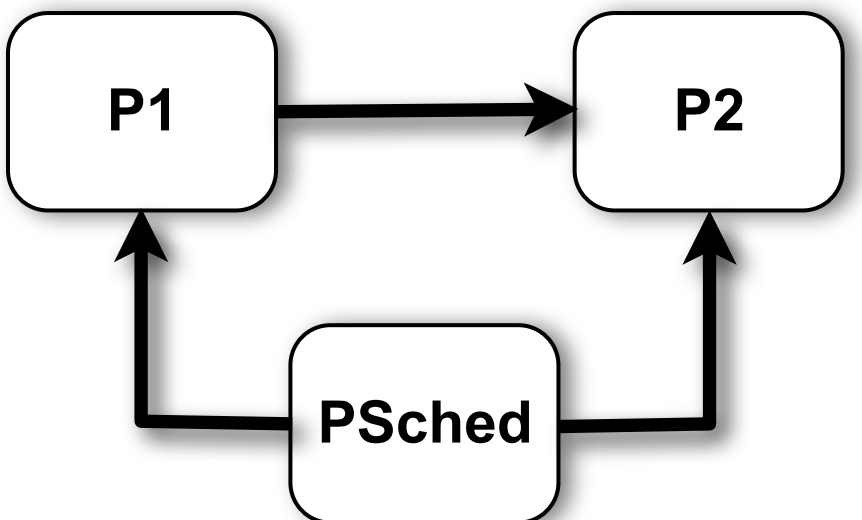


# Intransitive Nonleakage: Formally

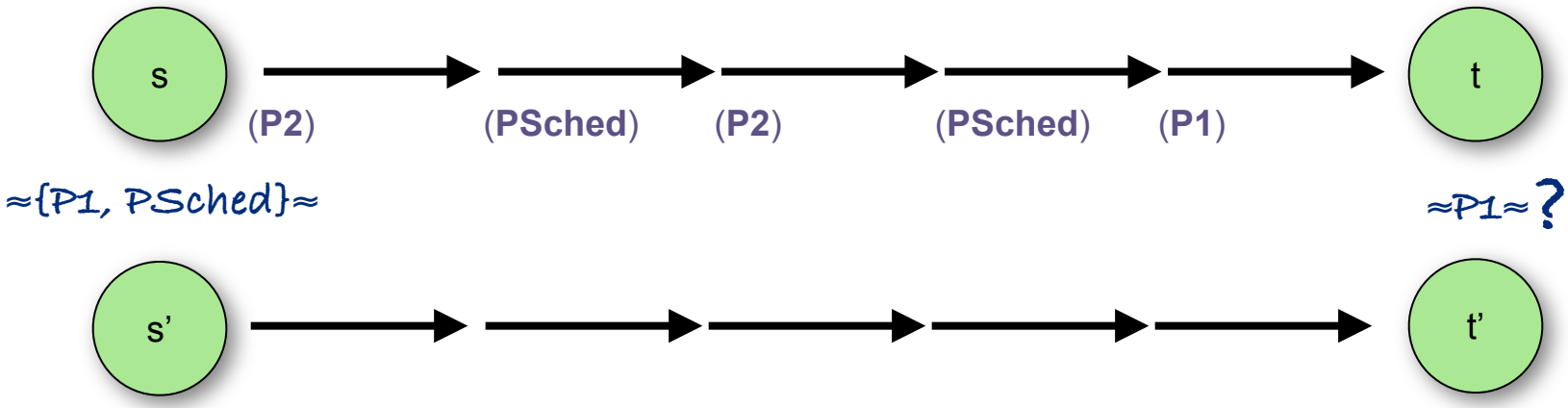
- Similar to language-based noninterference

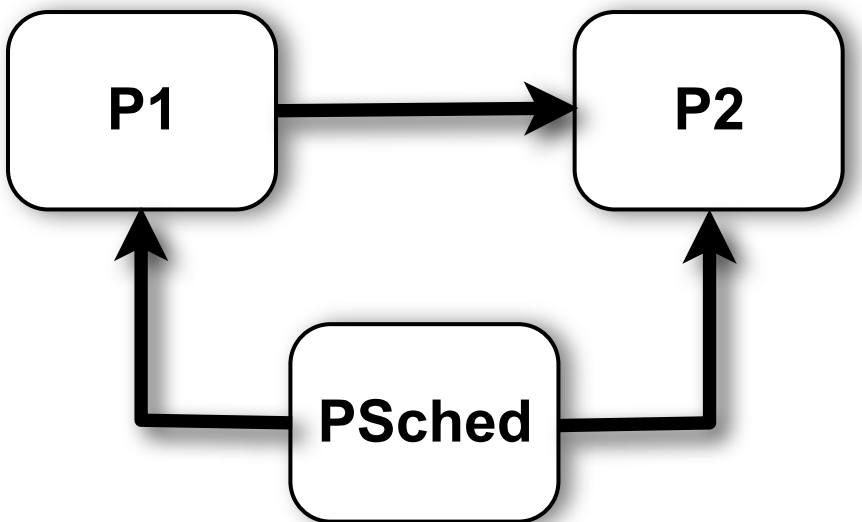


# Intransitive Nonleakage: Formally

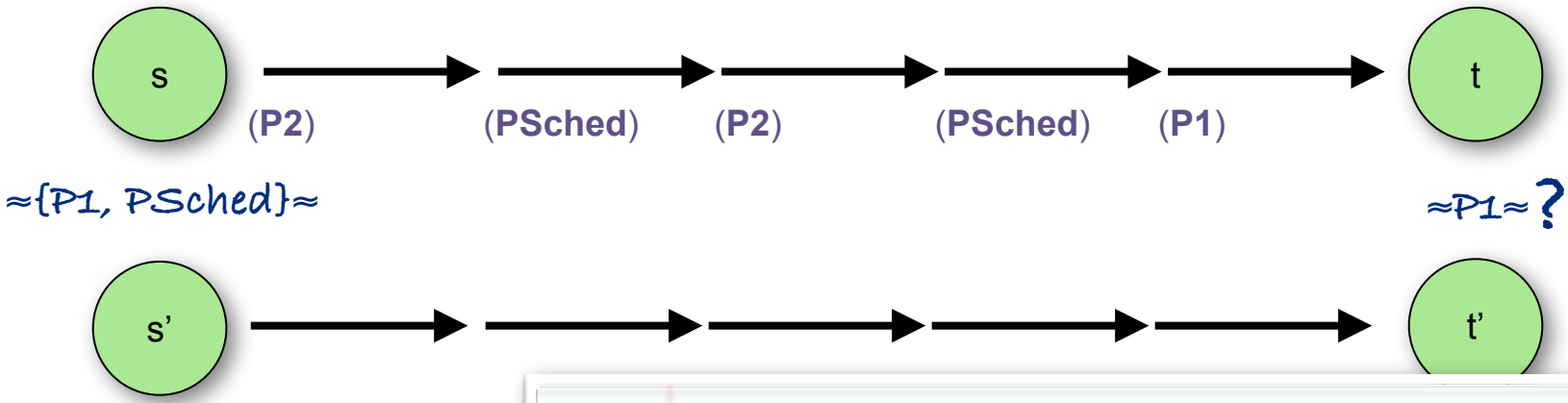
- Similar to language-based noninterference



# Intransitive Nonleakage: Formally

- Similar to language-based noninterference




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

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

# Intransitive Nonleakage: Formally

- S

informally:

≈{P1,

t

≈P1≈ ?

t'

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

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

# Intransitive Nonleakage: Formally

- S

*informally:* on a single step


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

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

# Intransitive Nonleakage: Formally

- S

*informally:* on a single step


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

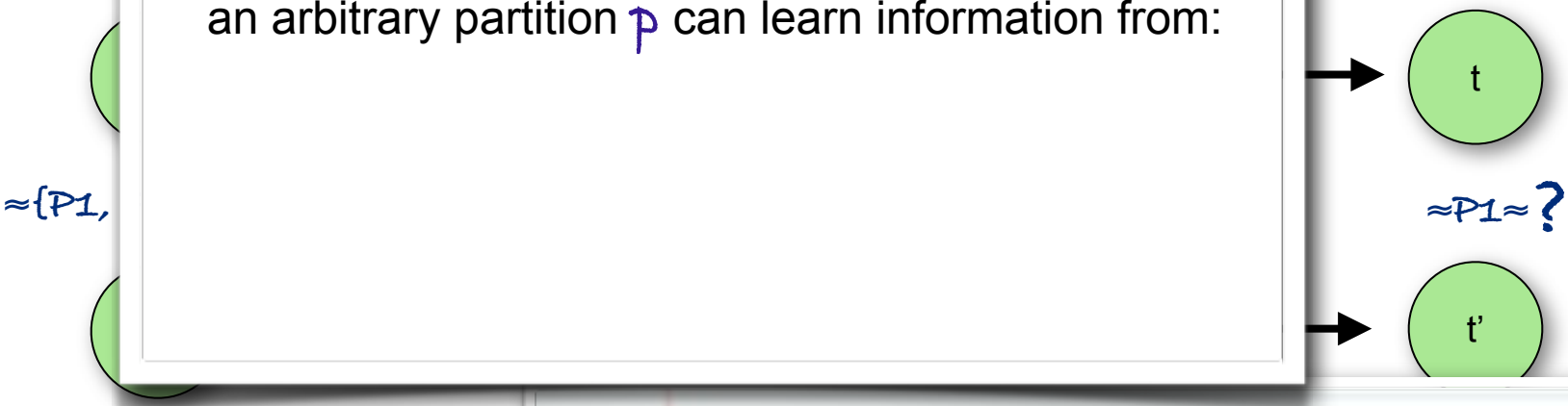
$$\forall p\ s\ s'\ t\ t'.$$
$$(s,t) \in Step \wedge (s',t') \in Step \wedge$$
$$s \approx\{p, PSched\}\approx s' \wedge$$
$$(part\ s \rightarrow p \Rightarrow s \sim part\ s \sim s') \Rightarrow$$
$$t \sim p \sim t'$$

# Intransitive Nonleakage: Formally

- S

*informally:* on a single step

an arbitrary partition p can learn information from:




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

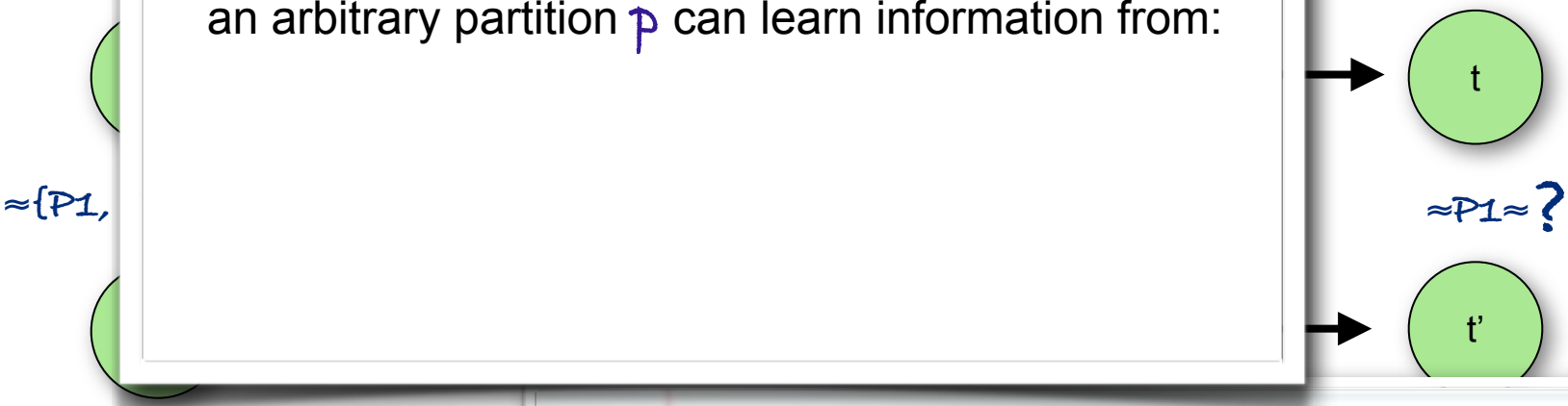
$$\forall p\ s\ s'\ t\ t'.$$
$$(s,t) \in Step \wedge (s',t') \in Step \wedge$$
$$s \approx\{p, PSched\}\approx s' \wedge$$
$$(part\ s \rightarrow p \Rightarrow s \sim part\ s\sim\ s') \Rightarrow$$
$$t \sim p\sim\ t'$$

# Intransitive Nonleakage: Formally

- S

*informally:* on a single step

an arbitrary partition p can learn information from:




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

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

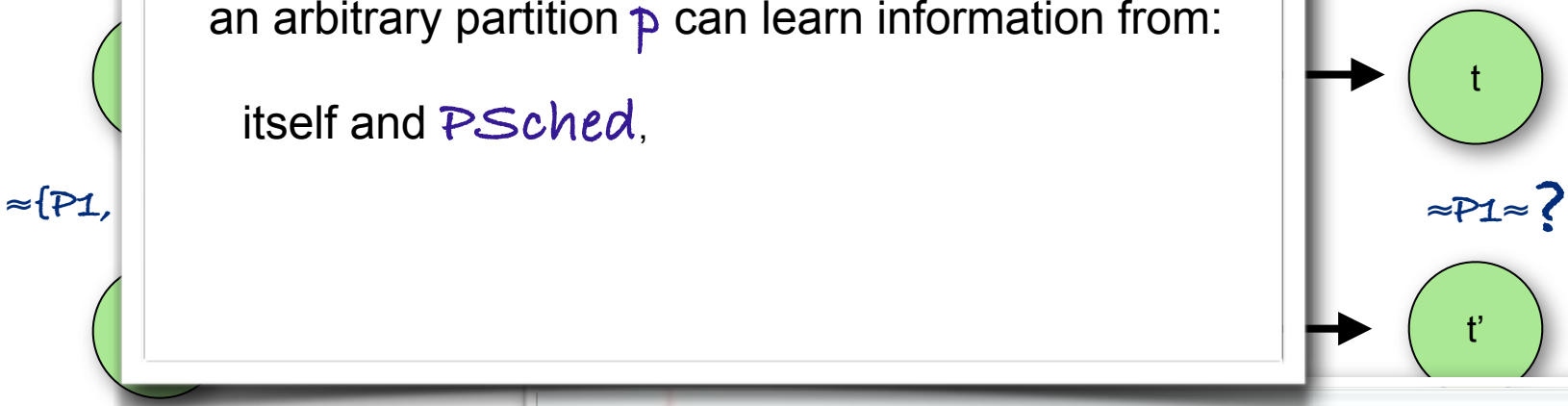
# Intransitive Nonleakage: Formally

- S

*informally:* on a single step

an arbitrary partition p can learn information from:

itself and PSched,




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

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

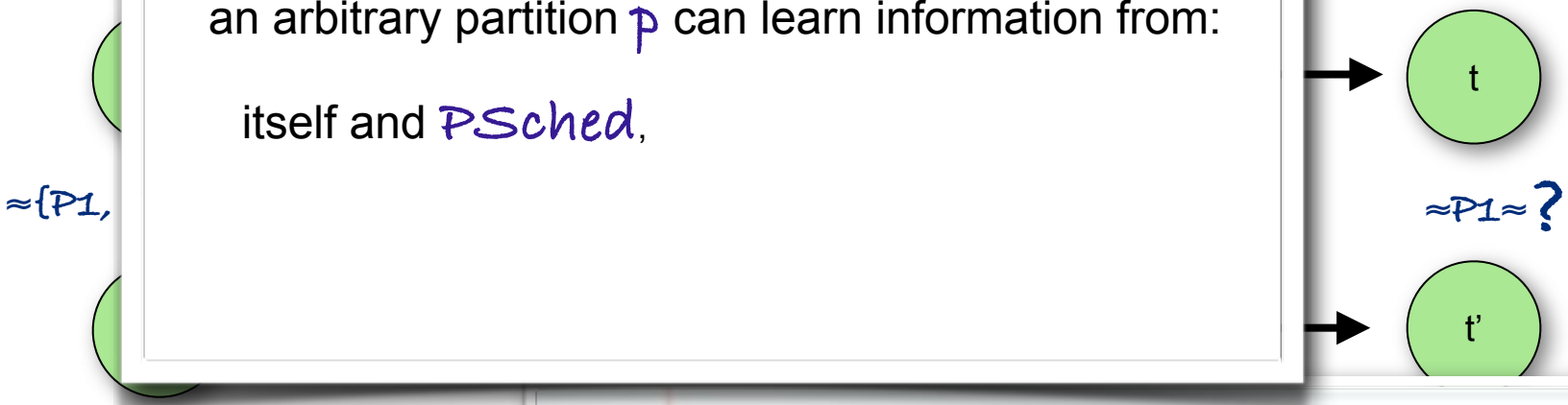
# Intransitive Nonleakage: Formally

- S

*informally:* on a single step

an arbitrary partition p can learn information from:

itself and PSched,




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

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

# Intransitive Nonleakage: Formally

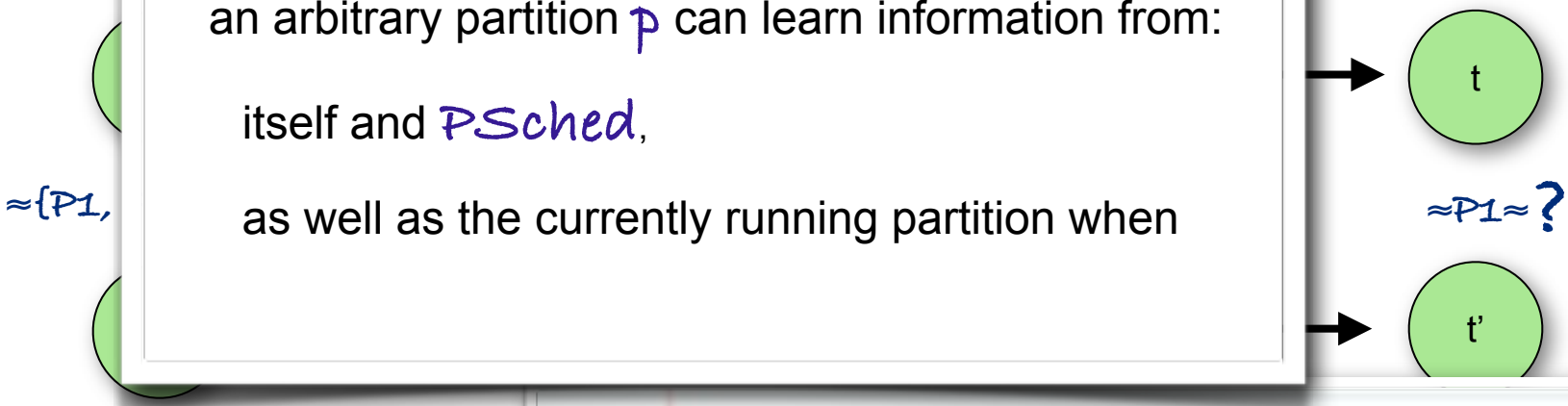
- S

*informally:* on a single step

an arbitrary partition p can learn information from:

itself and PSched,

as well as the currently running partition when




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

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

# Intransitive Nonleakage: Formally

- S

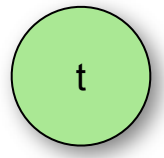
*informally:* on a single step

an arbitrary partition p can learn information from:

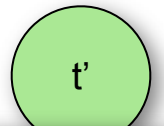
itself and PSched,

as well as the currently running partition when

≈{P1,




≈P1≈ ?




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

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

# Intransitive Nonleakage: Formally

- S

*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

≈{P1,

t

≈P1≈?

t'

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

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

# Intransitive Nonleakage: Formally

- S

*informally:* on a single step

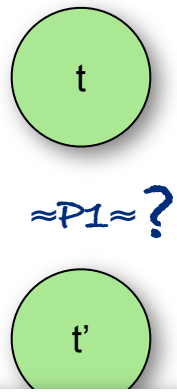
an arbitrary partition p can learn information from:

itself and PSched,

as well as the currently running partition when

the policy allows permits it to interfere with p

≈{P1,




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

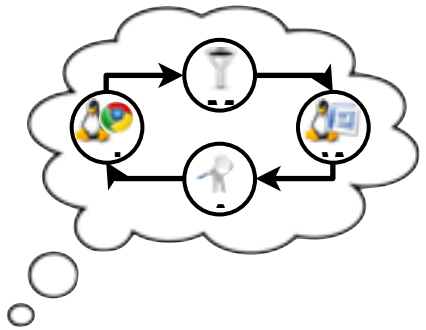
$$\forall p\ s\ s'\ t\ t'.$$
$$(s,t) \in Step \wedge (s',t') \in Step \wedge$$
$$s \approx\{p,PSched\}\approx s' \wedge$$
$$(part\ s \rightarrow p \Rightarrow s \sim part\ s \sim s') \Rightarrow$$
$$t \sim p \sim t'$$

# DISCUSSION

# DISCUSSION

(what does it mean?)

# Assurance



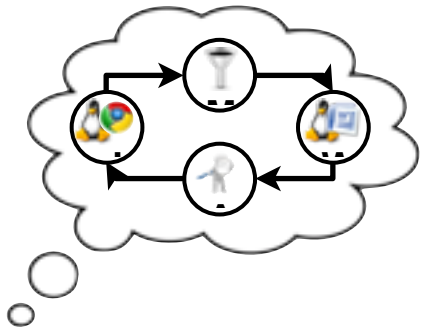

Security Property

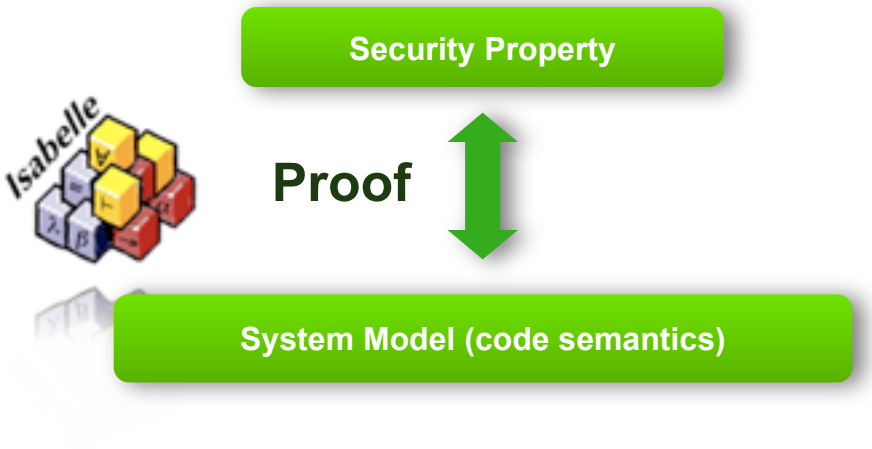
Proof

System Model (code semantics)

# Assurance

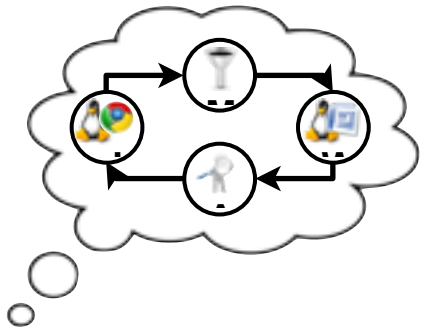
- Proofs break when:

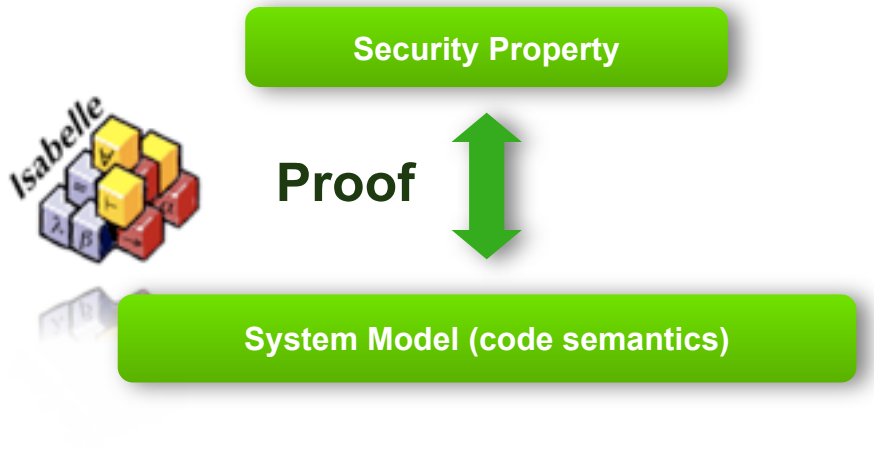





# Assurance

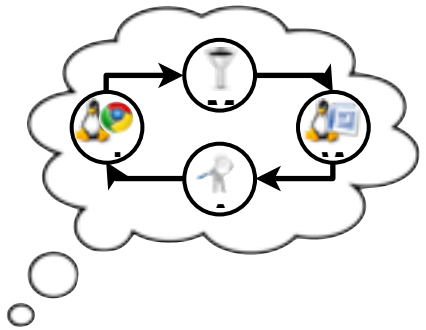
- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)

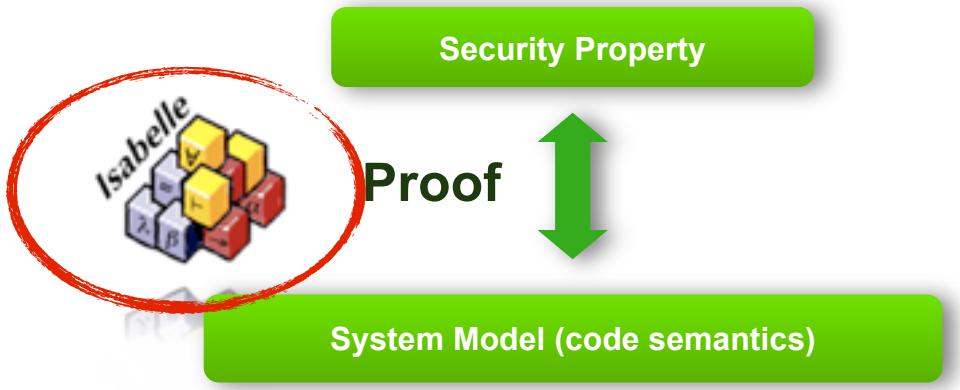





# Assurance

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

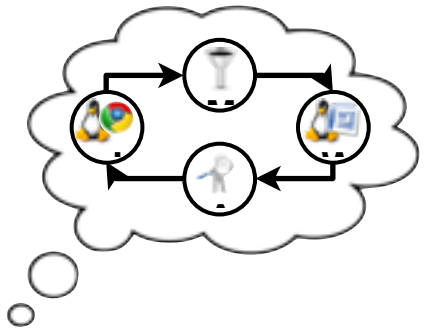


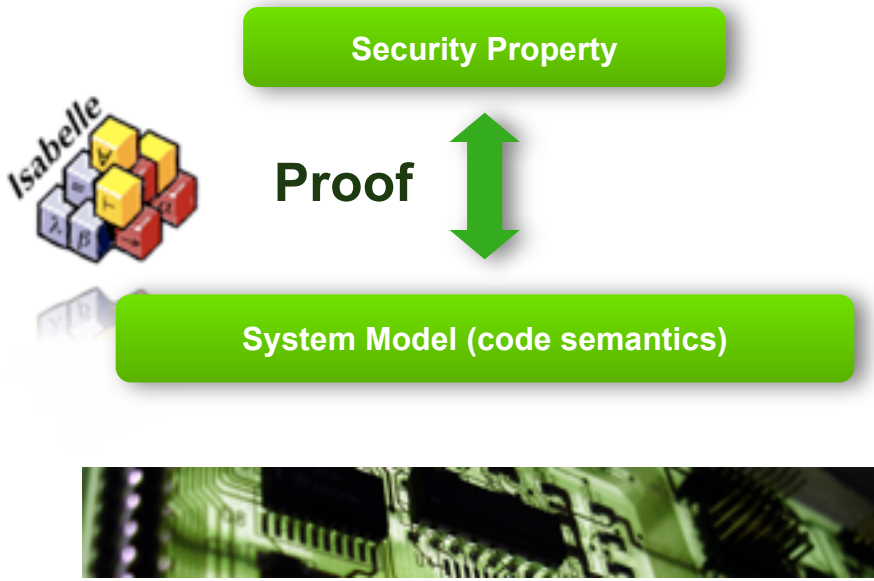



# Assurance

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

*a non-issue in practice*

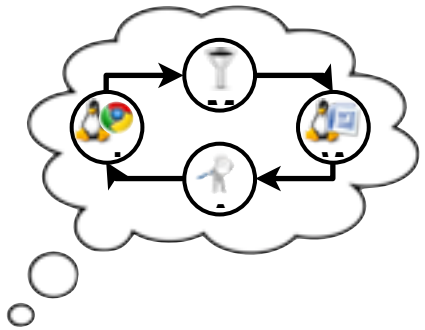




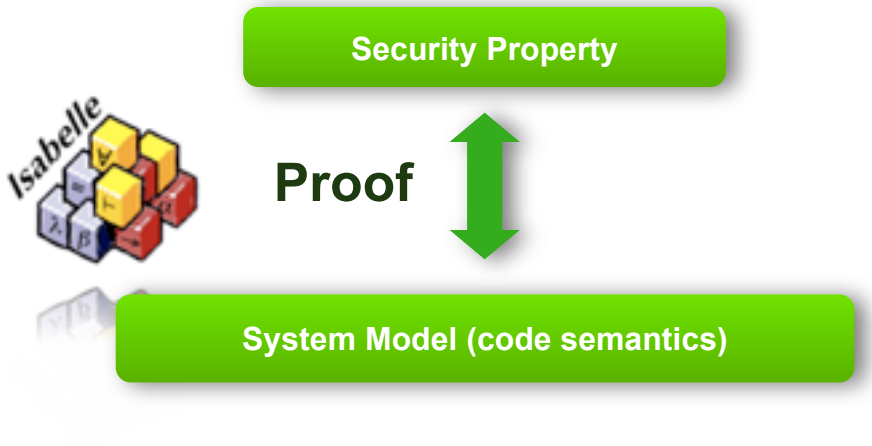
# Assurance

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

a non-issue in practice



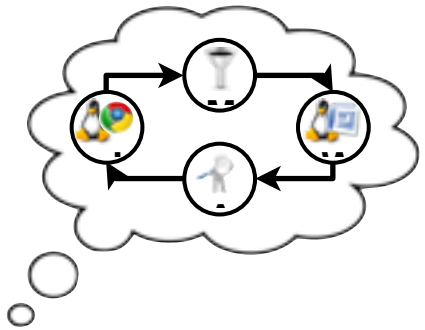
  - their assumptions are unrealistic



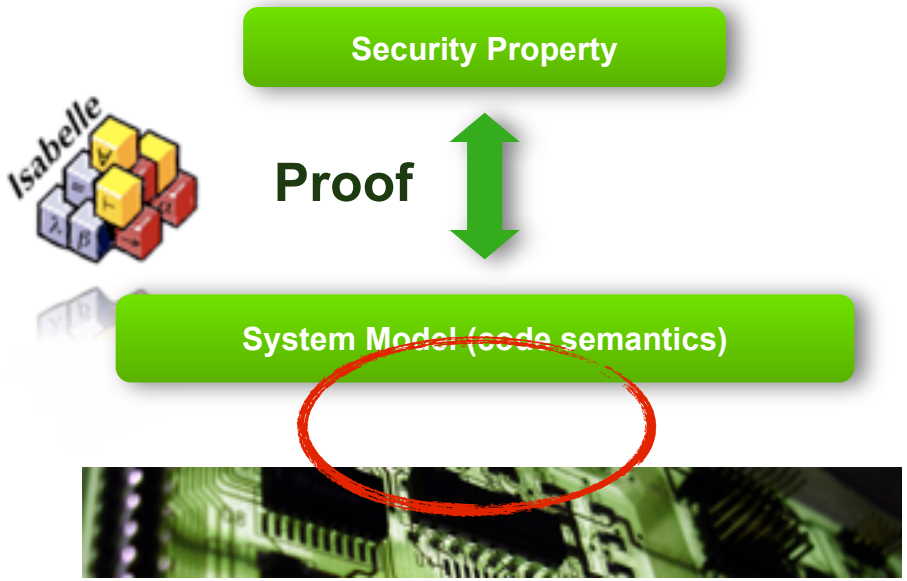

# Assurance

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

a non-issue in practice



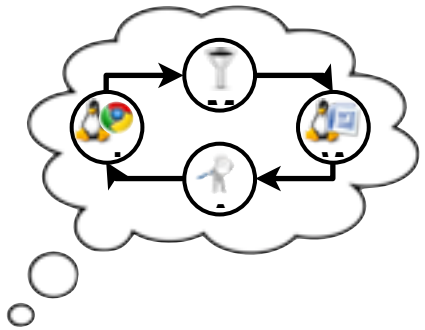
  - their assumptions are unrealistic



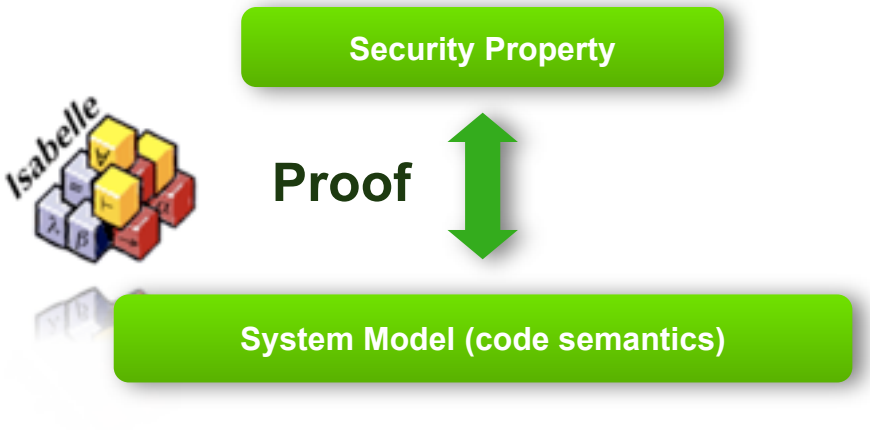
# Assurance

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

a non-issue in practice



  - their assumptions are unrealistic

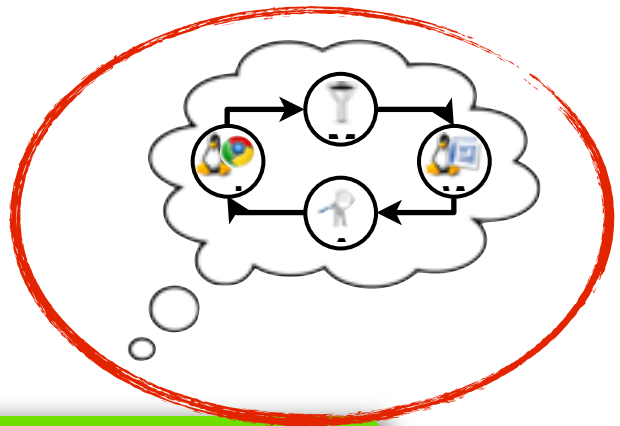



  - they don't mean what we thought they did

# Assurance

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

a non-issue in practice



  - their assumptions are unrealistic



  - they don't mean what we thought they did

# 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 fea

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 fea
  - 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 fea
  - DMA disabled

system init correctness proof: in progress

- 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 fea
  - DMA disabled

system init correctness proof: in progress

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

# Assumptions

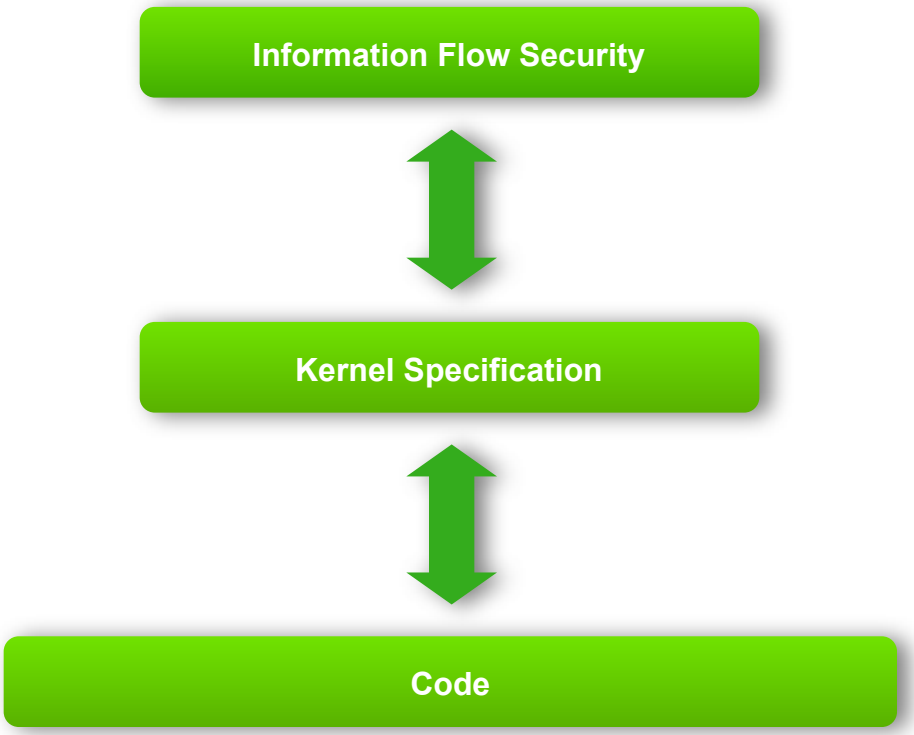
- All those of functional correctness proofs
  - compiler correctness, cache and TLB management, 450 lines of hand-written assembly code
- Correct initialisation
  - state of system after configuration enforces access policy, and
  - meets wellformedness assumptions
    - leaky API fea
  - DMA disabled

system init correctness proof: in progress

- User-space has no info sources that are not modelled
  - contents of emory

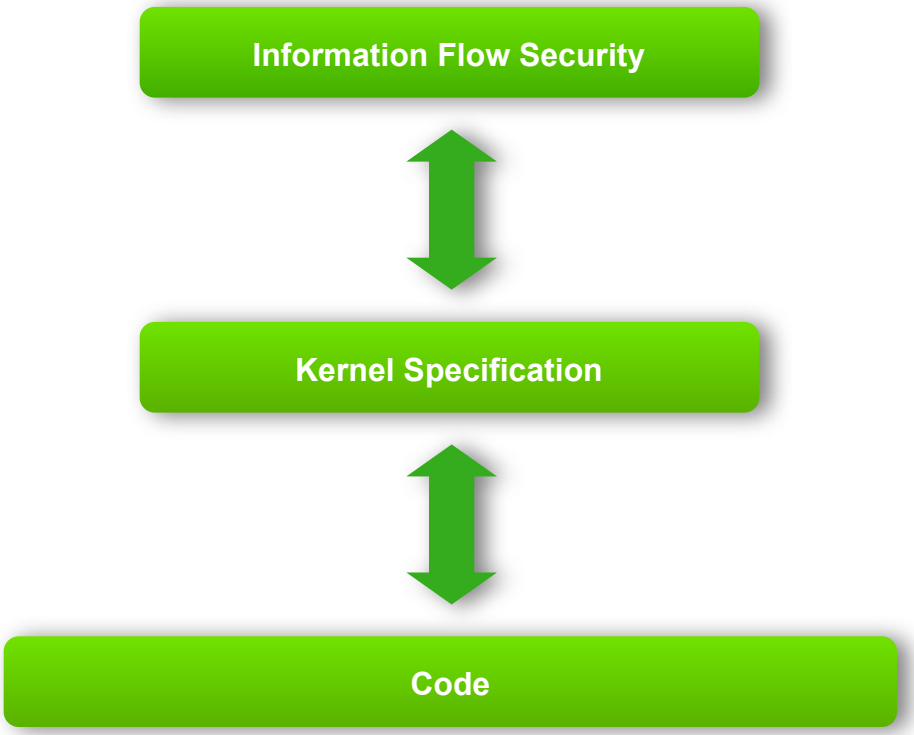
what about covert channels?

# Storage Channels



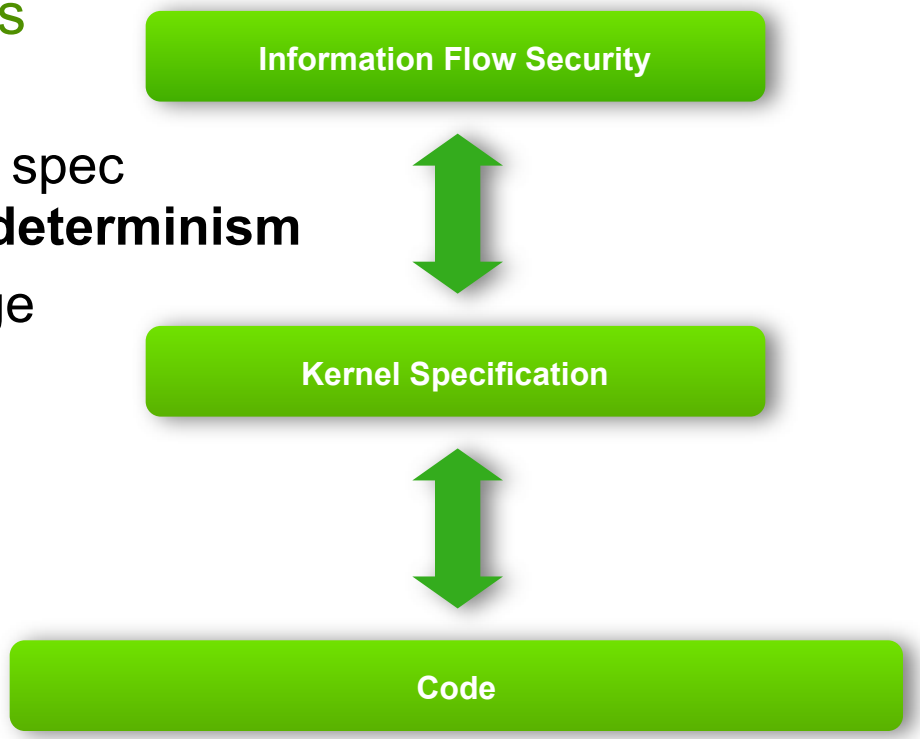
# Storage Channels

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



# Storage Channels

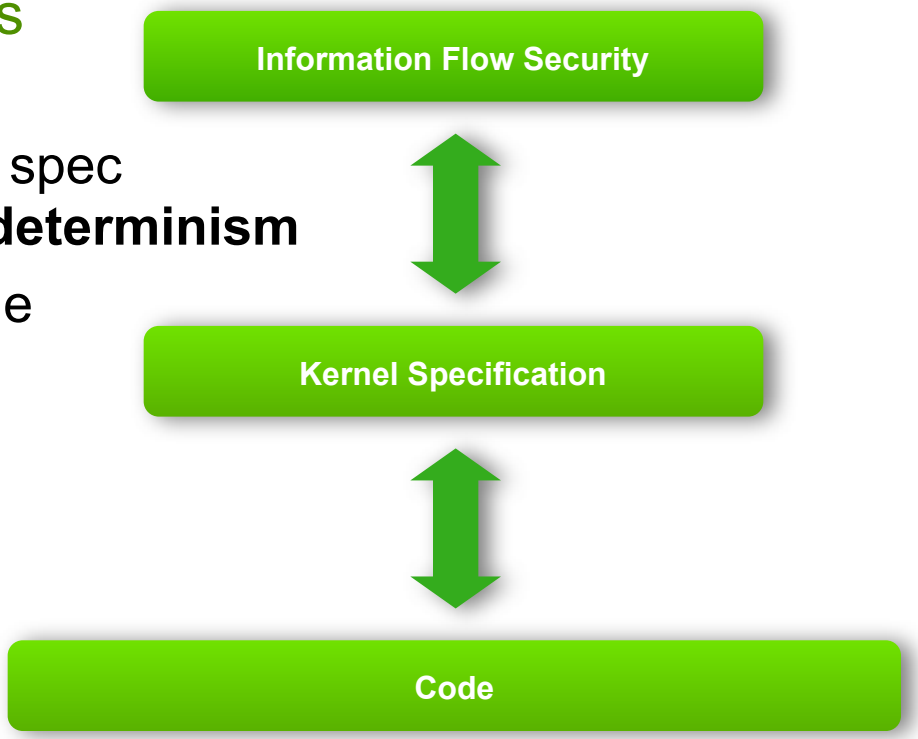
- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also **all** user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible **nondeterminism**
  - **not tolerated** by nonleakage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



# Storage Channels

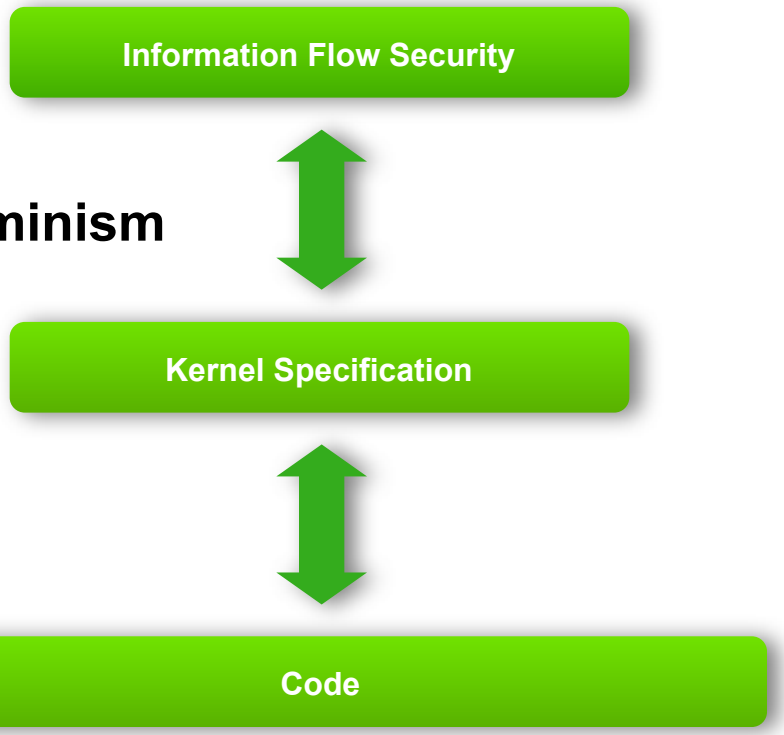
- Proof covers **all** storage channels present in kernel spec
  - abstract kernel heap, CPU registers, physical memory, IRQ masks, ...
- Also **all** user-visible channels read by the kernel
  - those below the level of the spec appear as user-visible **nondeterminism**
  - **not tolerated** by nonleakage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

```
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

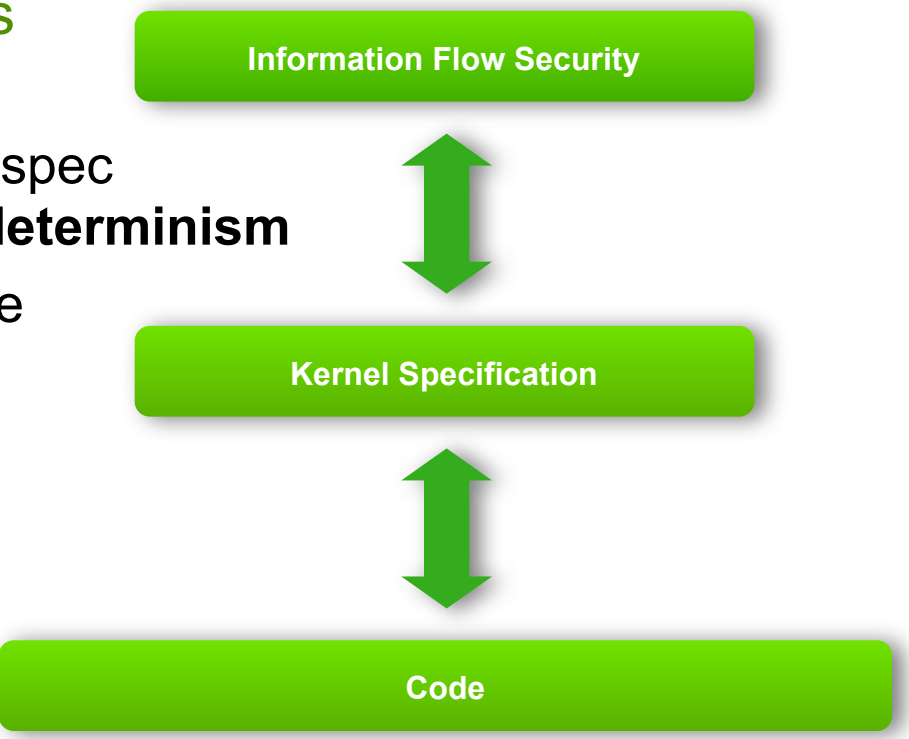



```
bool l, h;
l := 0 ⊓ 1;
```

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



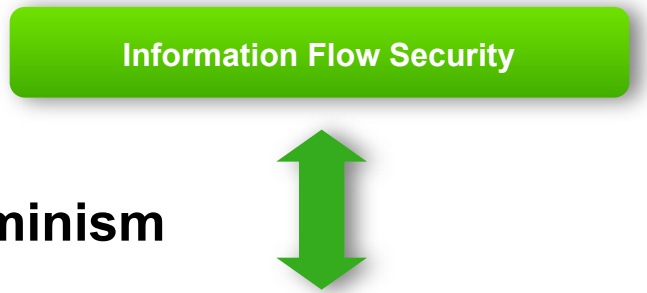

```
bool l, h;
l := 0 ⊓ 1;
```

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 tole**rated by nonleakage under r




# Storage Channels

- Proof covers **all** storage channels pre nt in kernel spec
  - abstract kernel heap, CPU ical memory, IRQ masks.
- Also
  - re
  - u

not covered: channels absent from spec that kernel never reads

ion Flow Security

inism

eakage

```
bool l, h
l := 0 ⊓ h
```

is the value of refinement-preserved noninterference

is refined by

```
l := h;
```

# Storage Channels

- Proof covers **all** storage channels pre nt in kernel spec
  - abstract kernel heap, CPU ical memory, IRQ masks.
- Also
  - re
  - u


is refined by

# 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

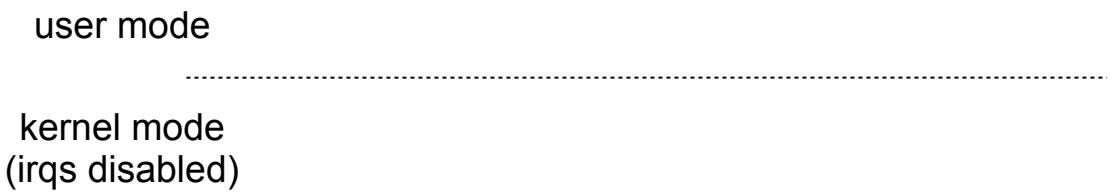
- The proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()

# 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

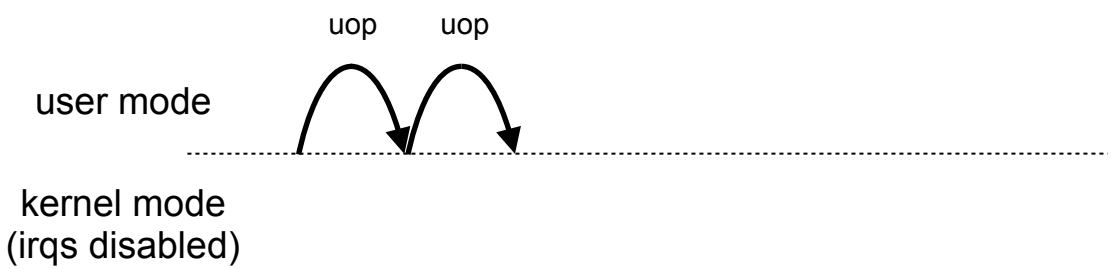


# Timing Channels

- 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

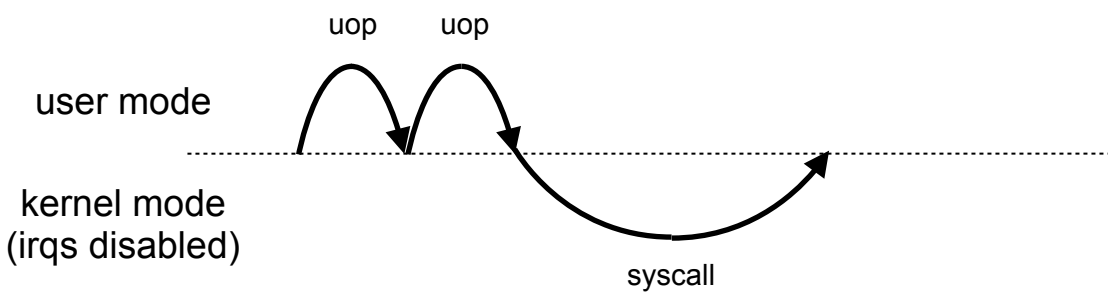
# Timing Channels

- 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



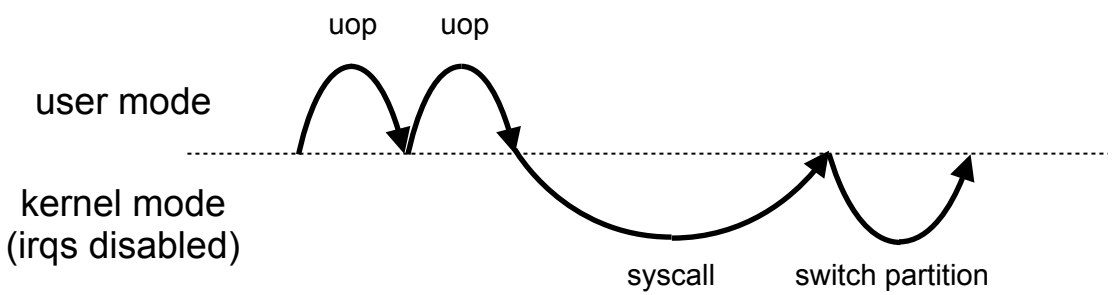
# Timing Channels

- 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



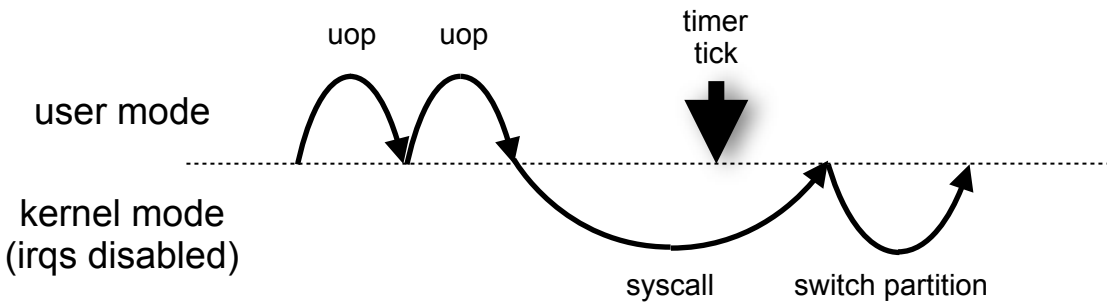
# Timing Channels

- 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



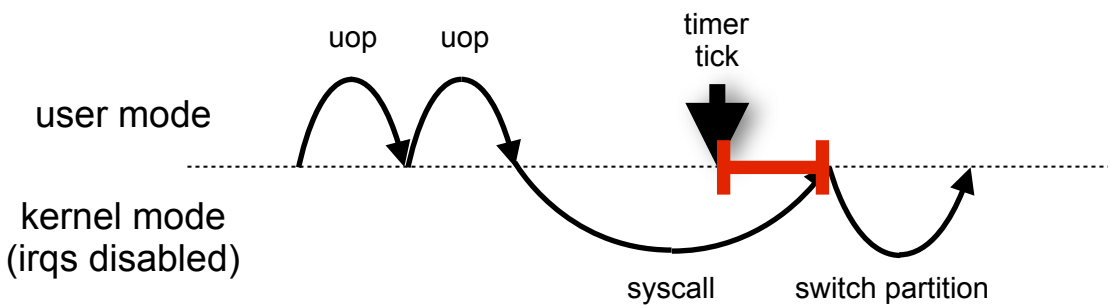
# Timing Channels

- 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



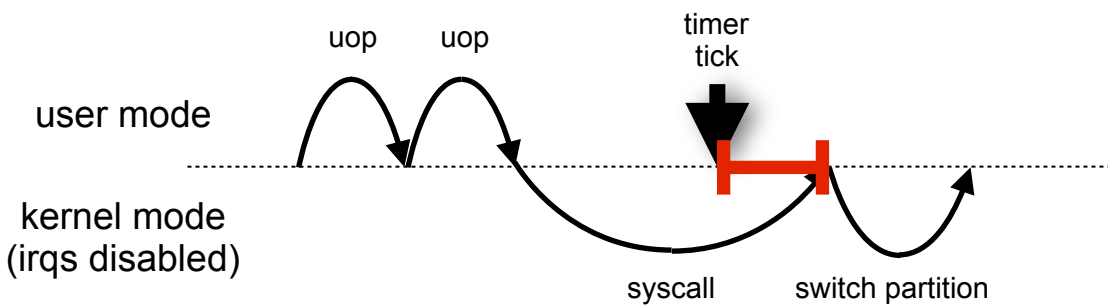
# Timing Channels

- 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

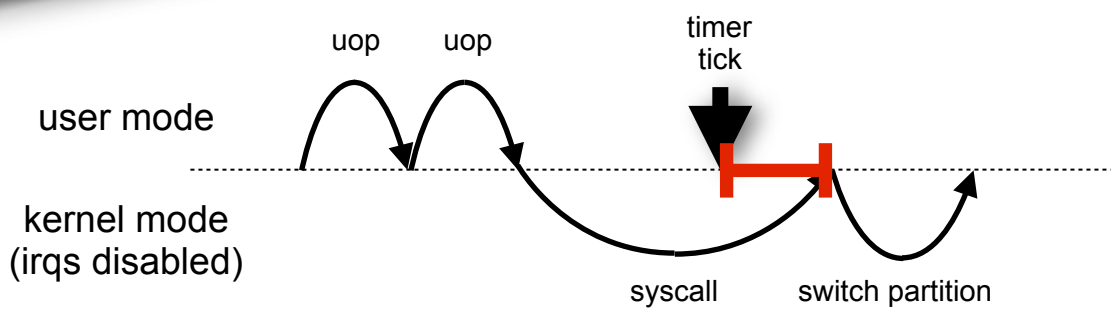



- Others: caches, CPU temp. etc.

# Timing Channels

- The proof a… …nnels
- e.g…
  - …ble
  - …nning calls e.g. Revoke()
  - …y syscall

must be mitigated by complementary techniques




- Others: caches, CPU temp. etc.

# Timing Channels

- The proof a… …nnels
- e.g…
  - …ble
  - …nning calls e.g. Revoke()
  - …

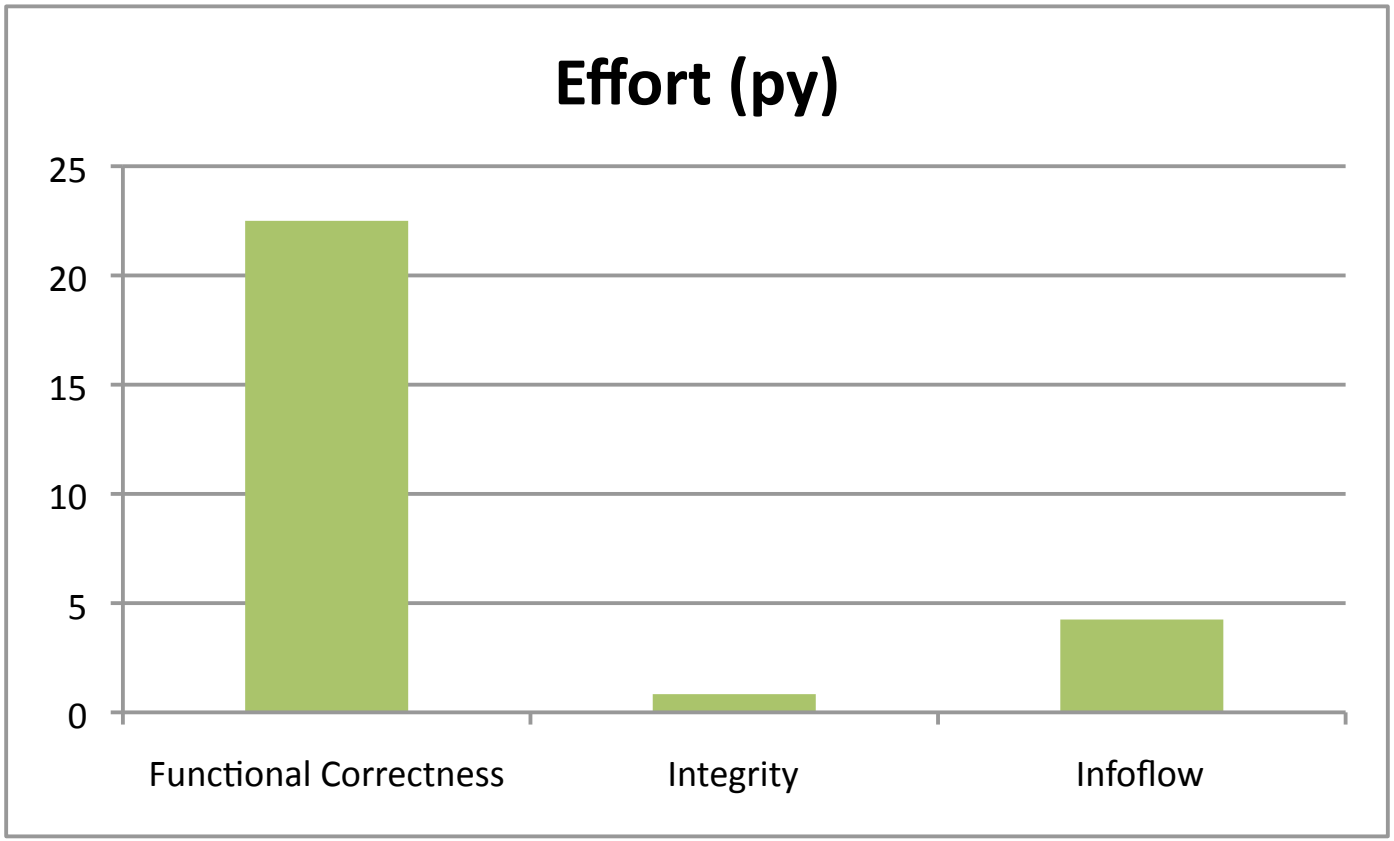
must be mitigated by complementary techniques

mitigation strategy depends on risk profile of deployment

- Others: caches, CPU temp. etc.

# Lesson

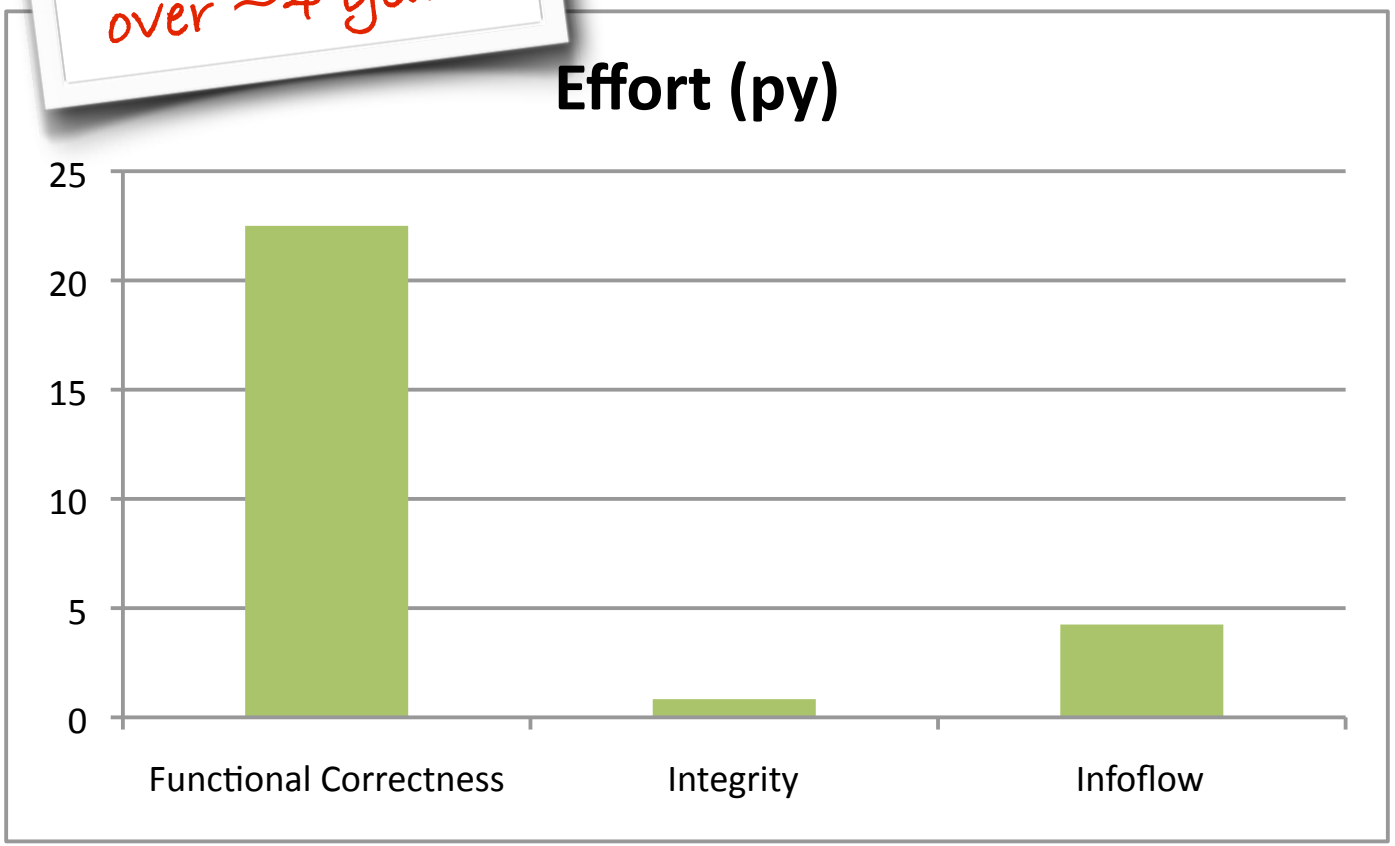
- Functional correctness enables cheap security proofs



# Lesson

- Functional c… enables cheap security proofs

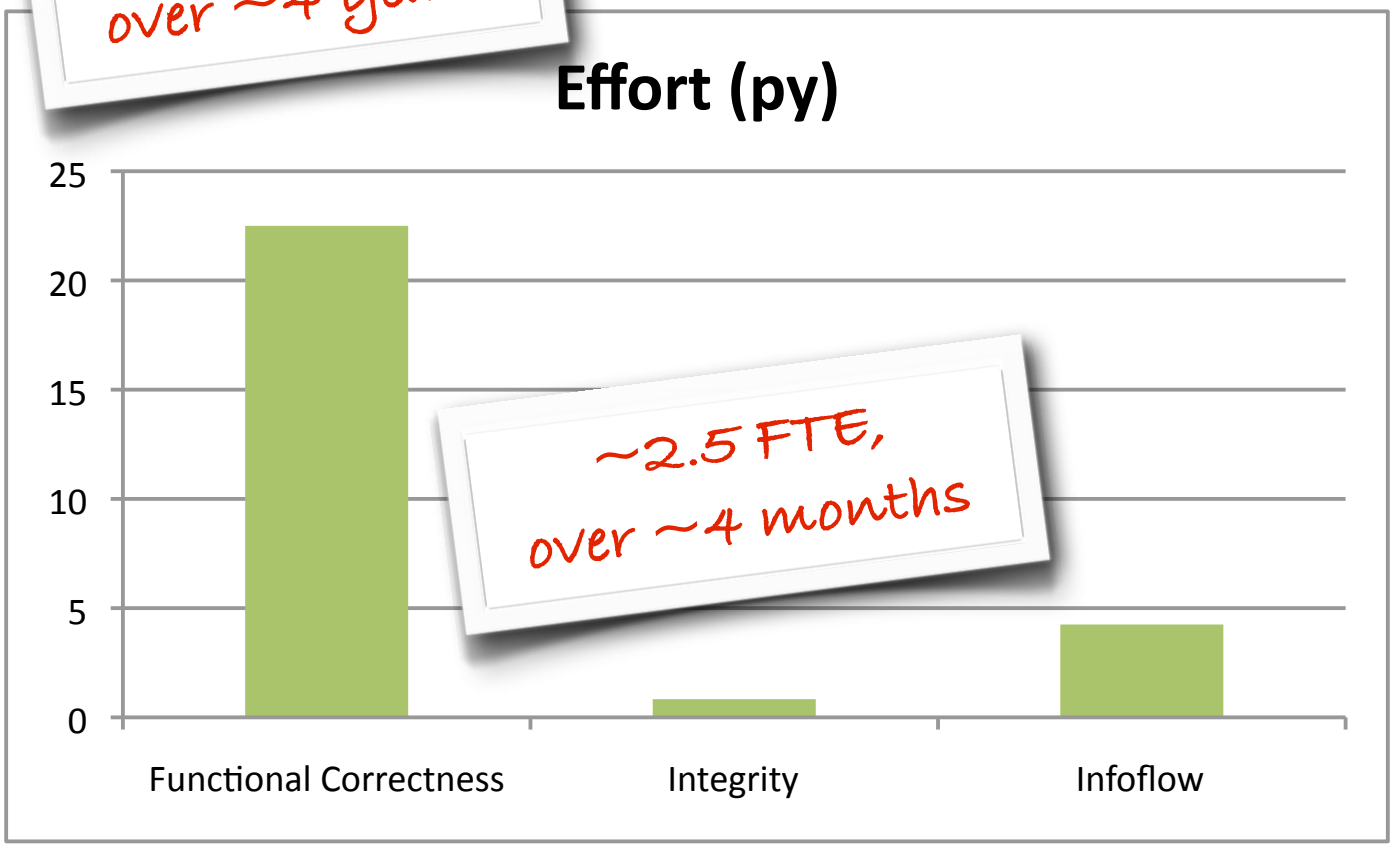
~6 people, over ~4 years



# Lesson

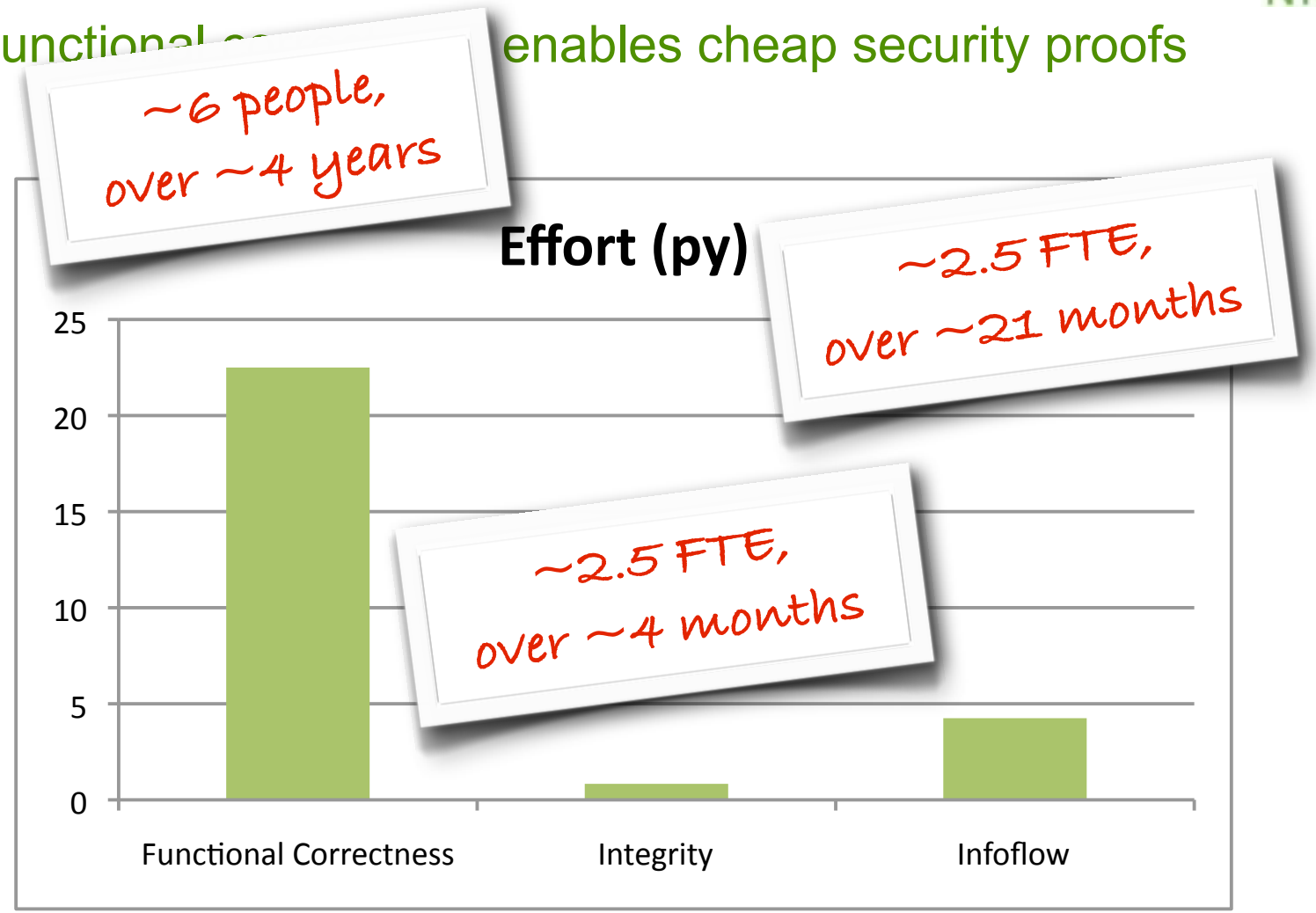
- Functional [illegible] enables cheap security proofs

~6 people, over ~4 years



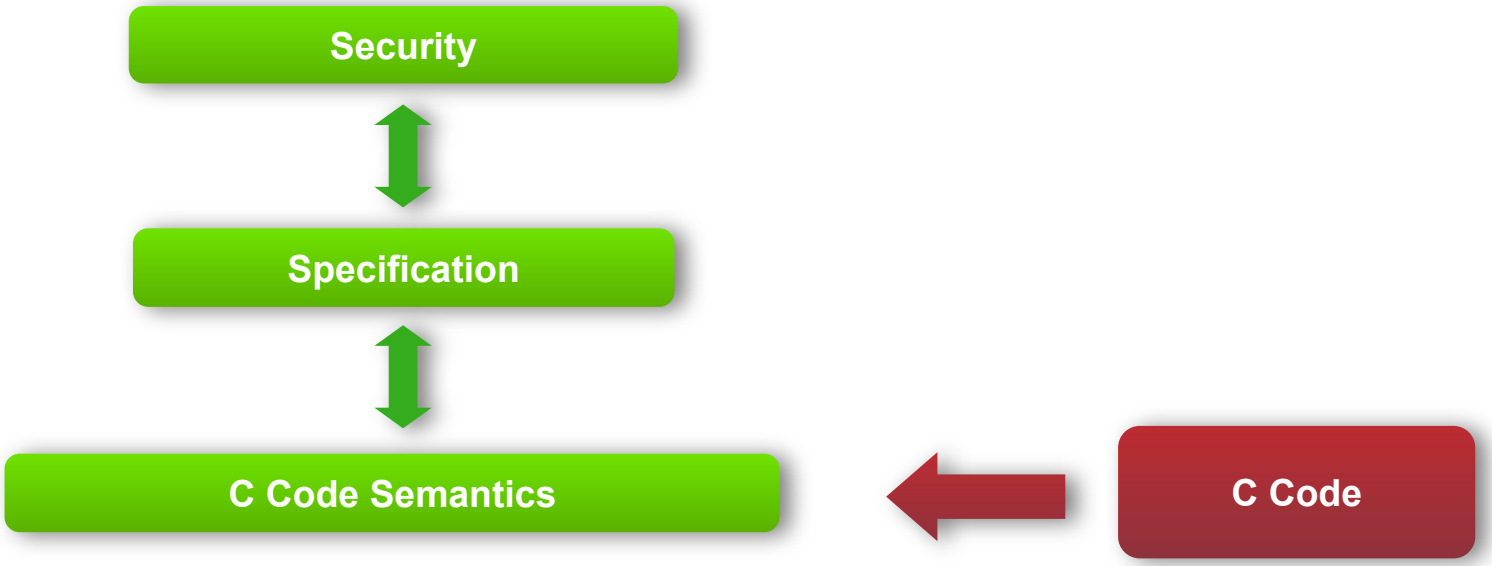
# Lesson

- Functional [illegible] enables cheap security proofs

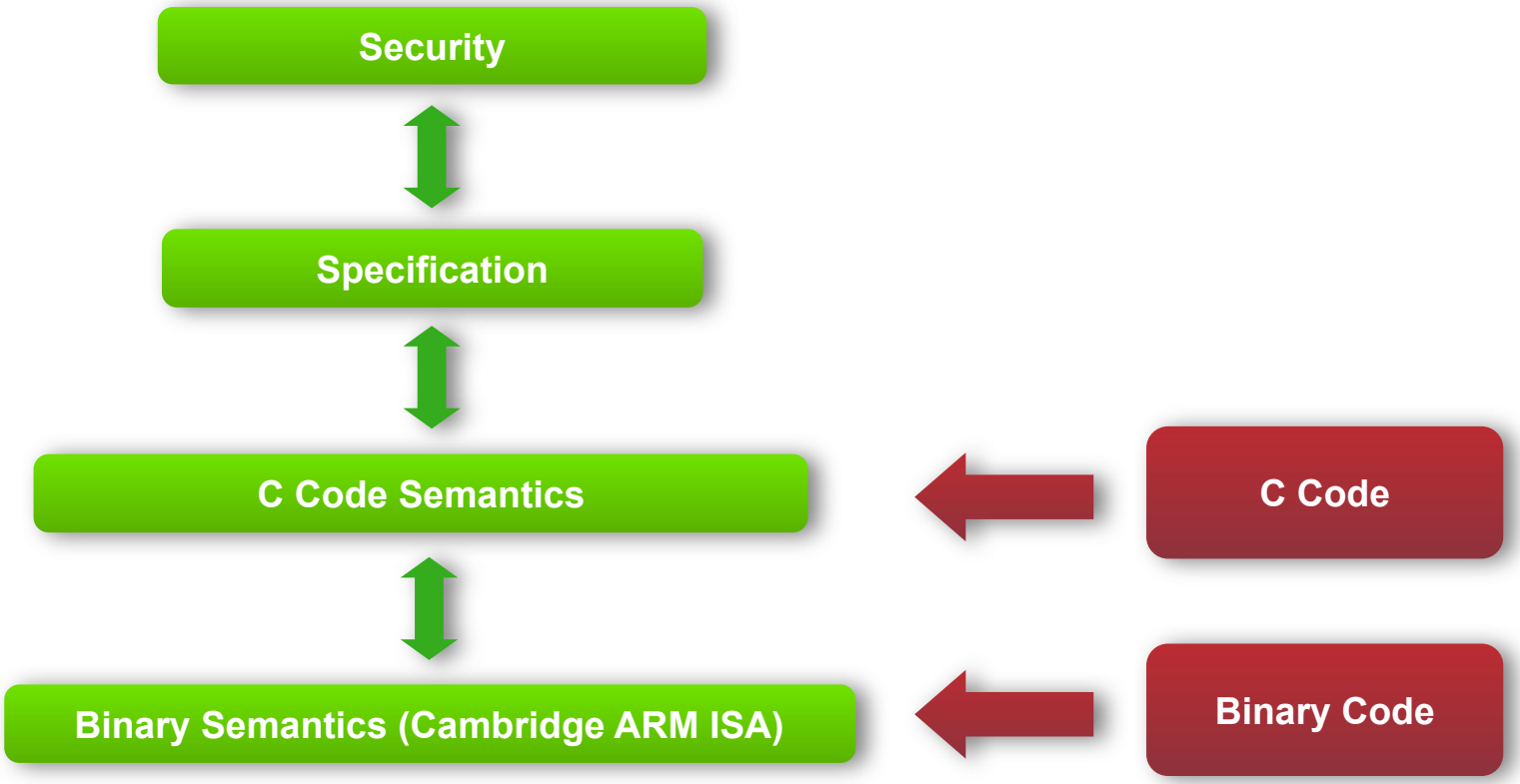


# BREAKING NEWS

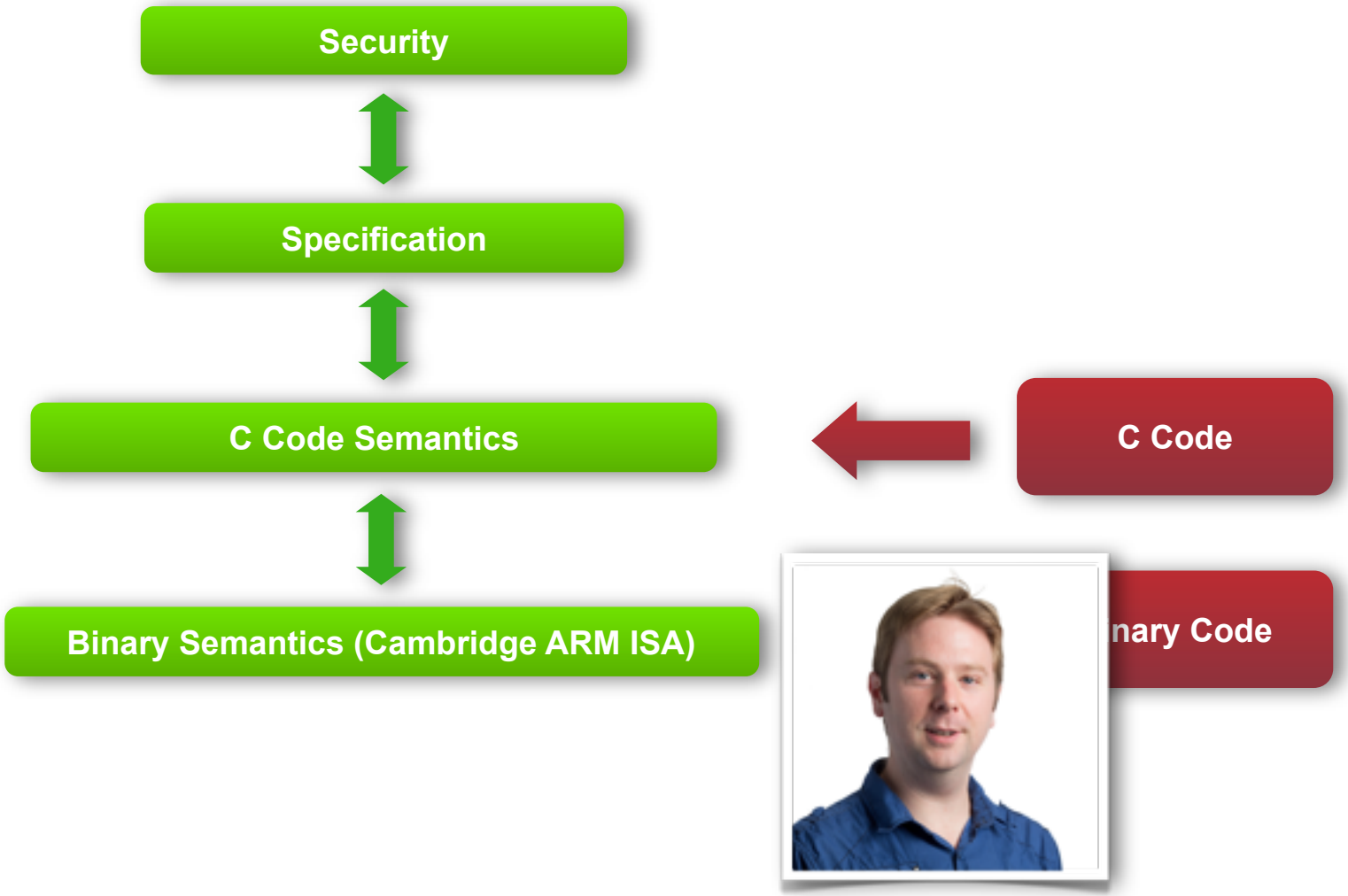
# Security Theorems for the Kernel Binary



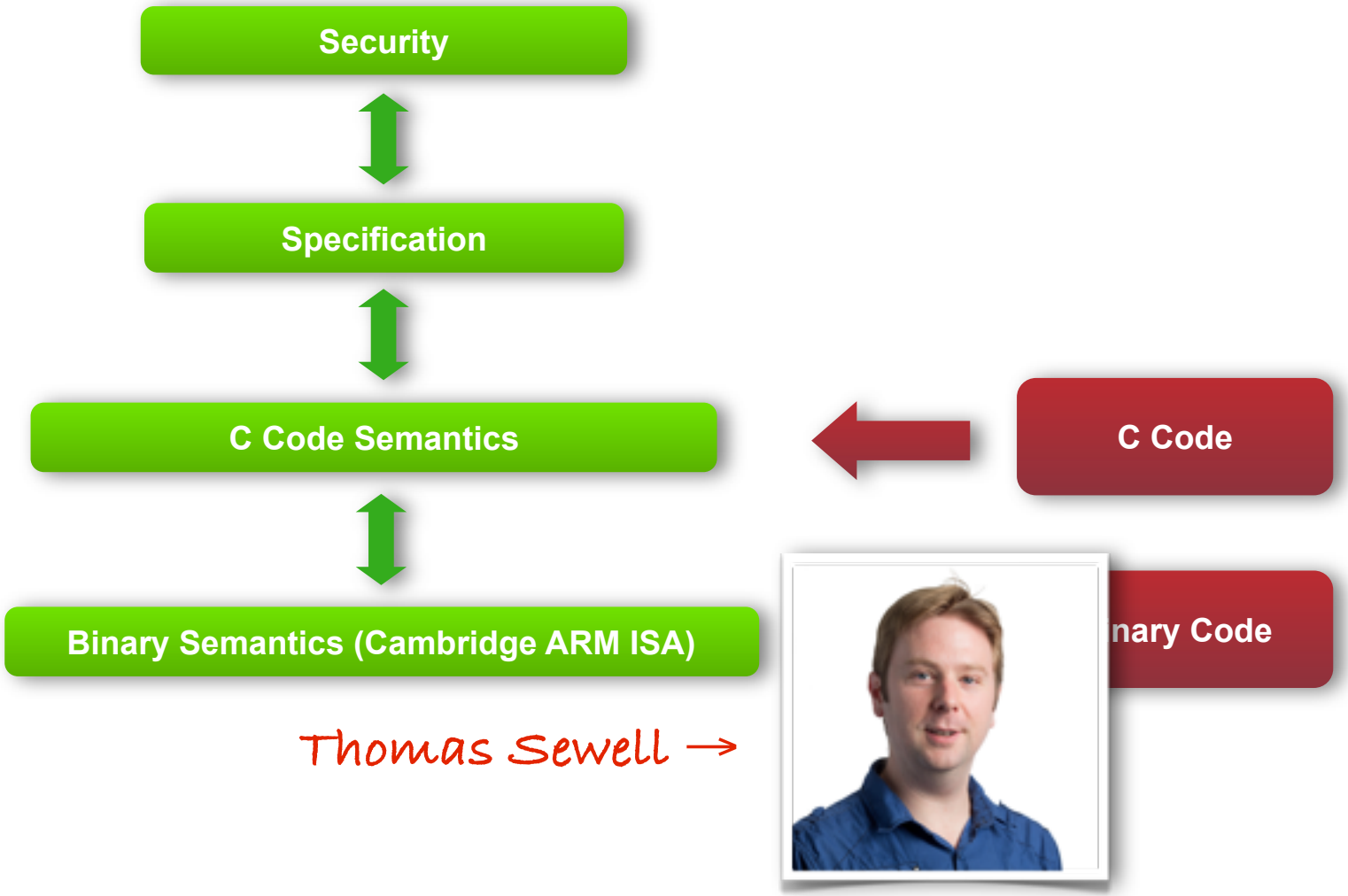
# Security Theorems for the Kernel Binary



# Security Theorems for the Kernel Binary



# Security Theorems for the Kernel Binary

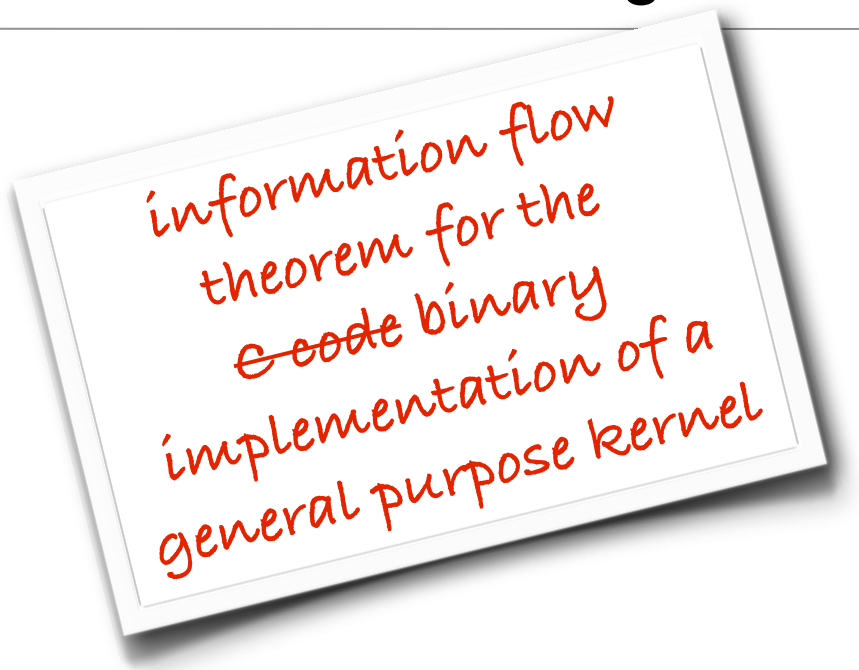


# Security Theorems for the Kernel Binary

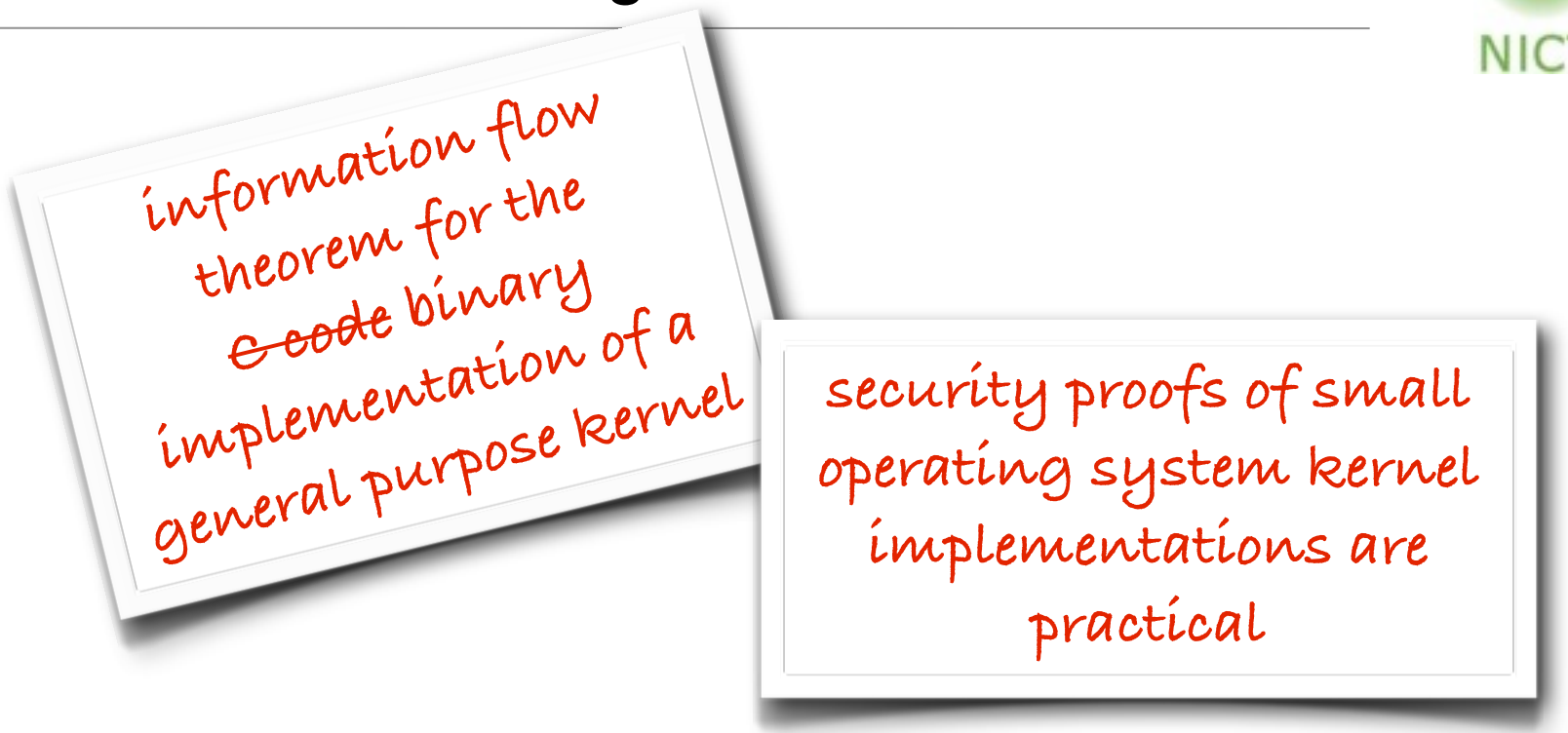
# CONCLUSION

# Take-Home Message

# Take-Home Message



# Take-Home Message



# 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.


NICTA
Thank You

NICTA

# Thank You