NICTA

seL4
Above and Beyond

Toby Murray and Thomas Sewell

Joint work with Matthew Brassil, Timothy Bourke, Peter Gammie, Xin Gao, Gerwin Klein, Corey Lewis, Daniel Matichuk and Magnus O. Myreen

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

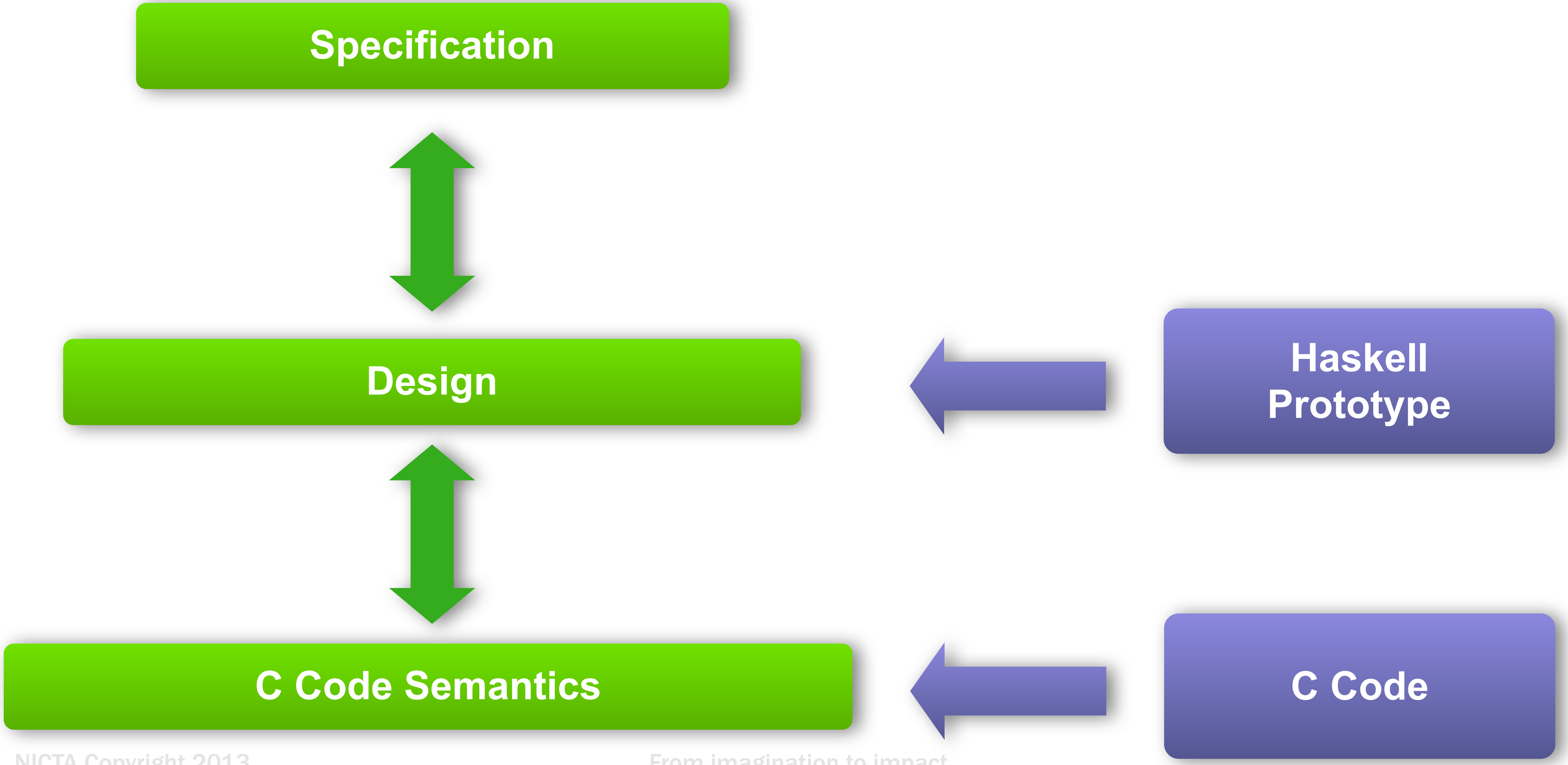
NICTA Funding and Supporting Members and Partners

Australian National University
UNSW THE UNIVERSITY OF NEW SOUTH WALES
NSW GOVERNMENT Trade & Investment
State Government Victoria
THE UNIVERSITY OF MELBOURNE
THE UNIVERSITY OF SYDNEY
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
The University Of Queensland AUSTRALIA

NICTA Copyright 2012

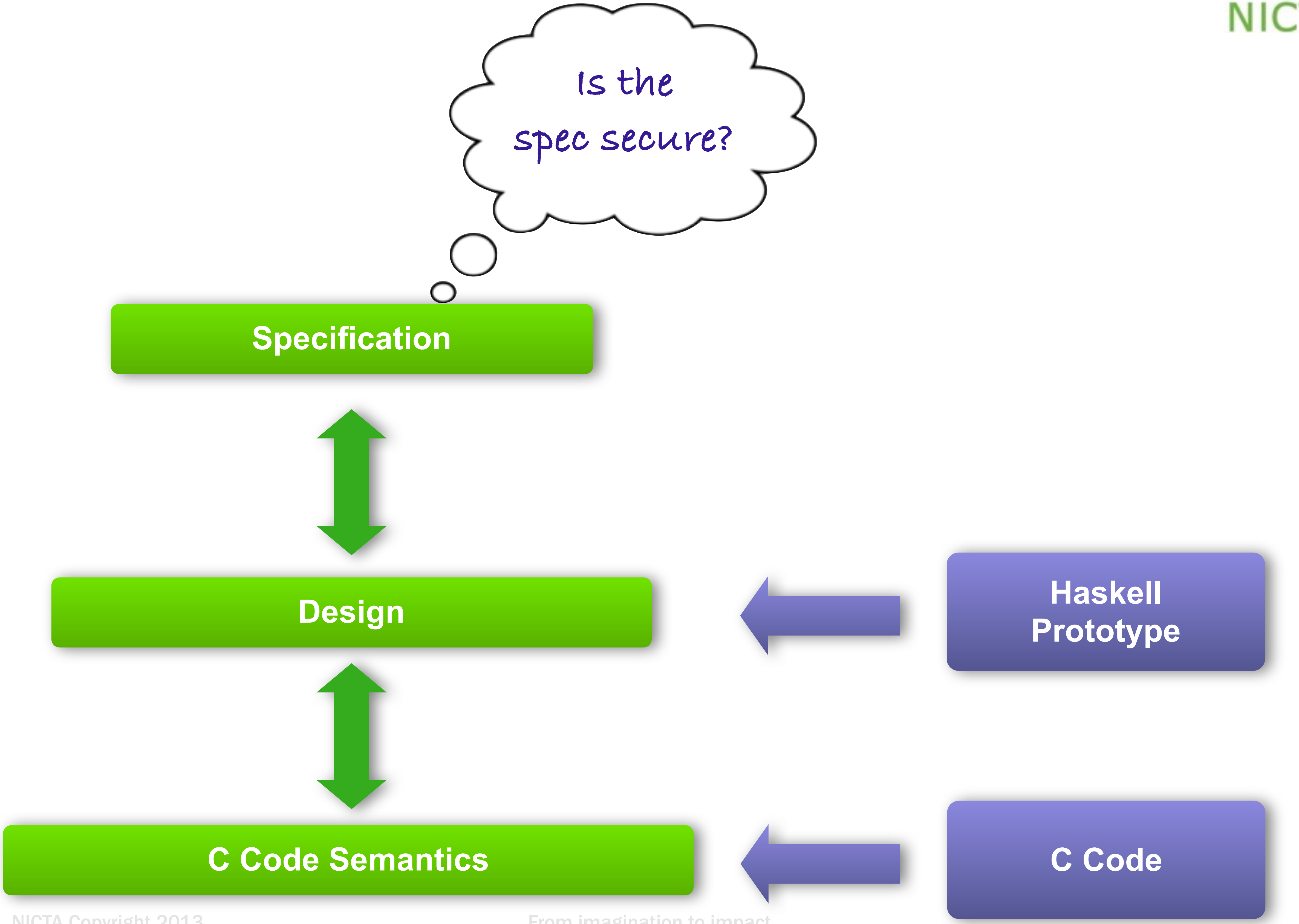
From imagination to impact

Functional Correctness Proof (2009)



NICTA Copyright 2013

Functional Correctness Proof (2009)



Is the spec secure?

Specification

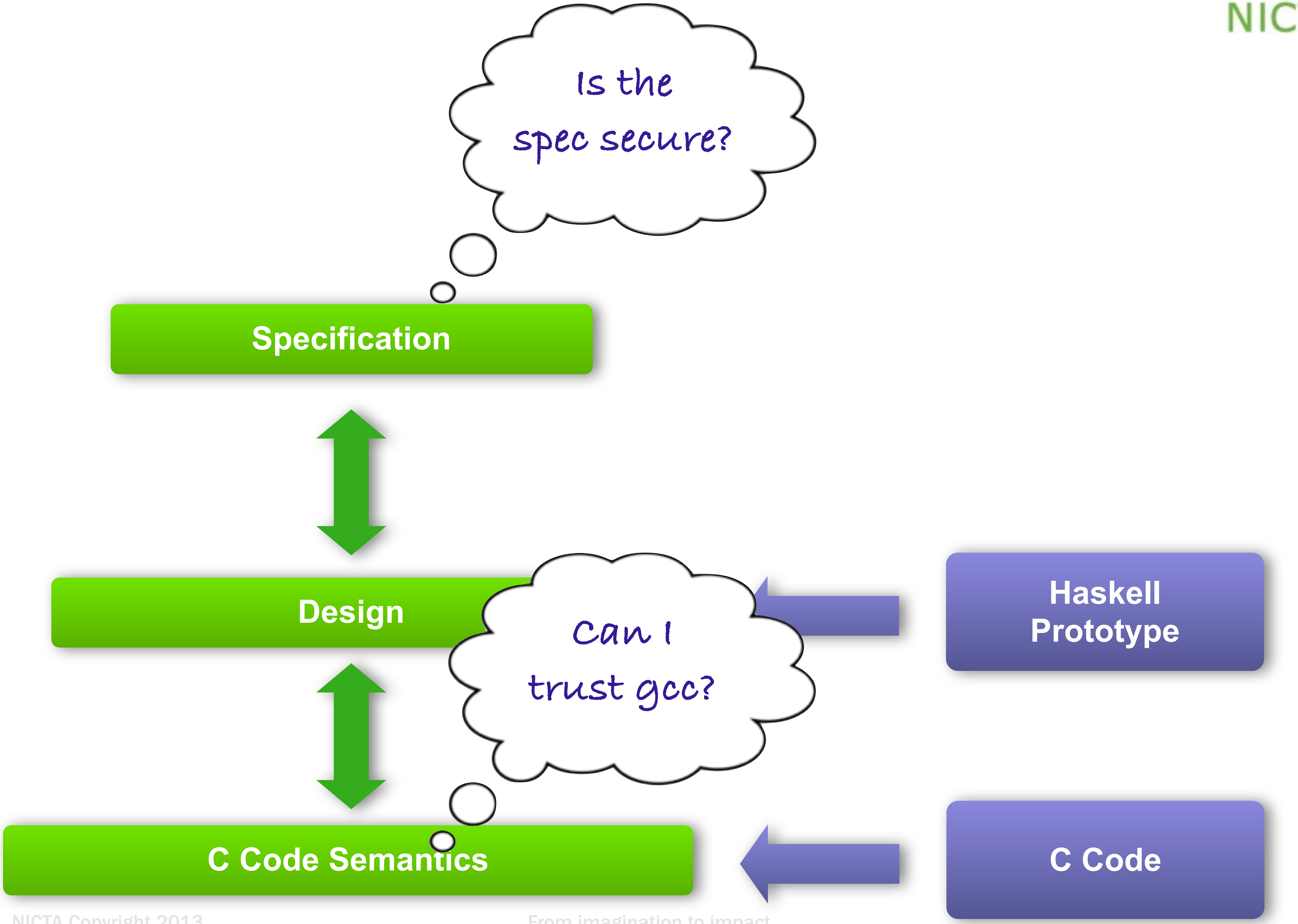
Design

C Code Semantics

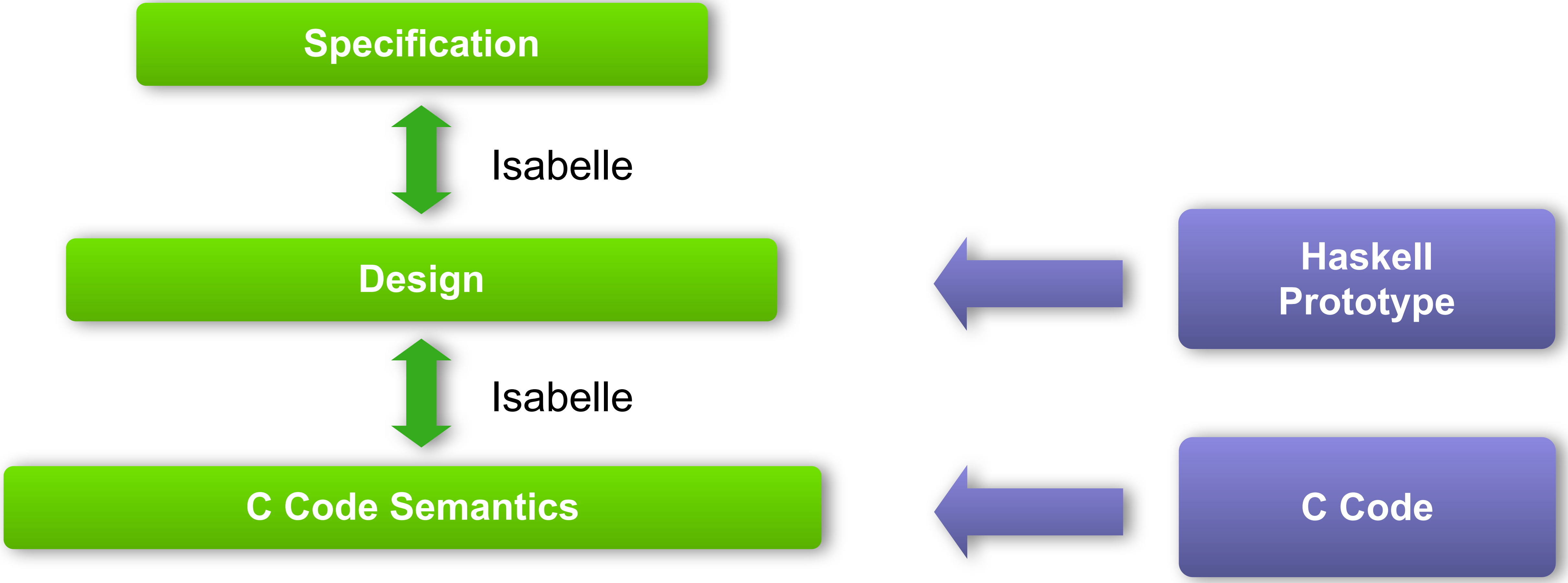
Haskell Prototype

C Code

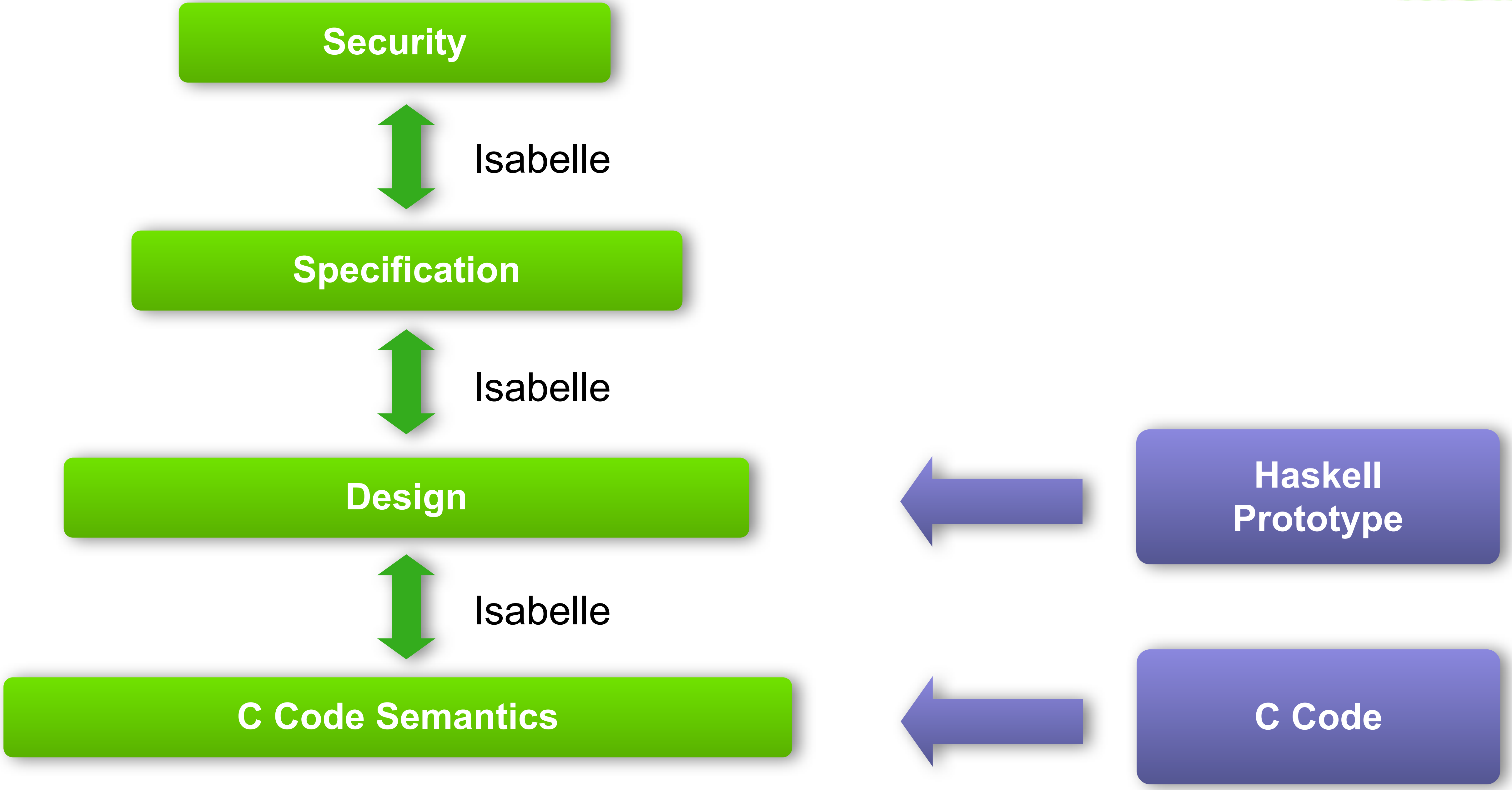
Functional Correctness Proof (2009)



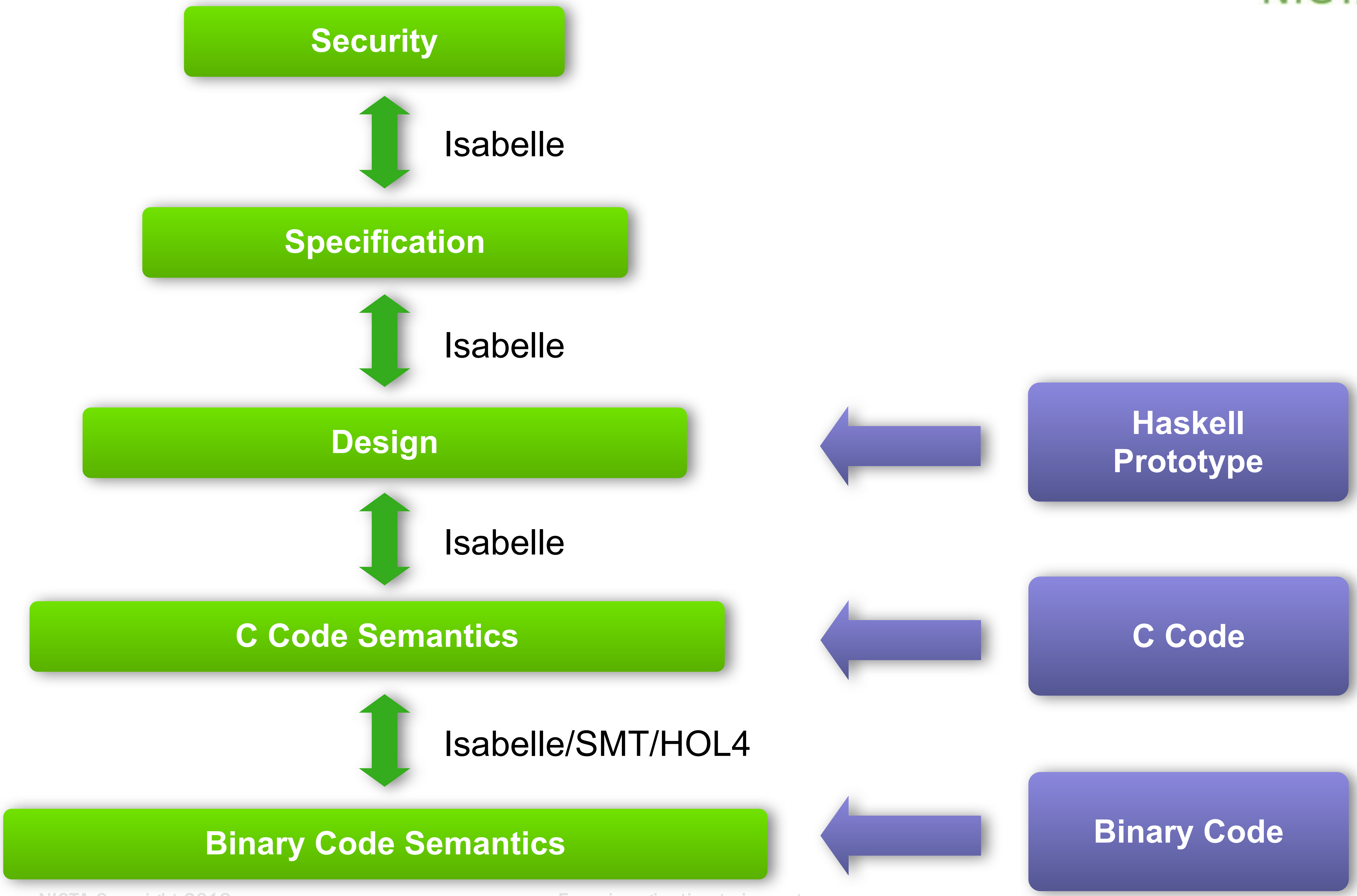
Proof Architecture (now)



Proof Architecture (now)



Proof Architecture (now)



Security

Isabelle

Specification

Isabelle

Design

Haskell Prototype

Isabelle

C Code Semantics

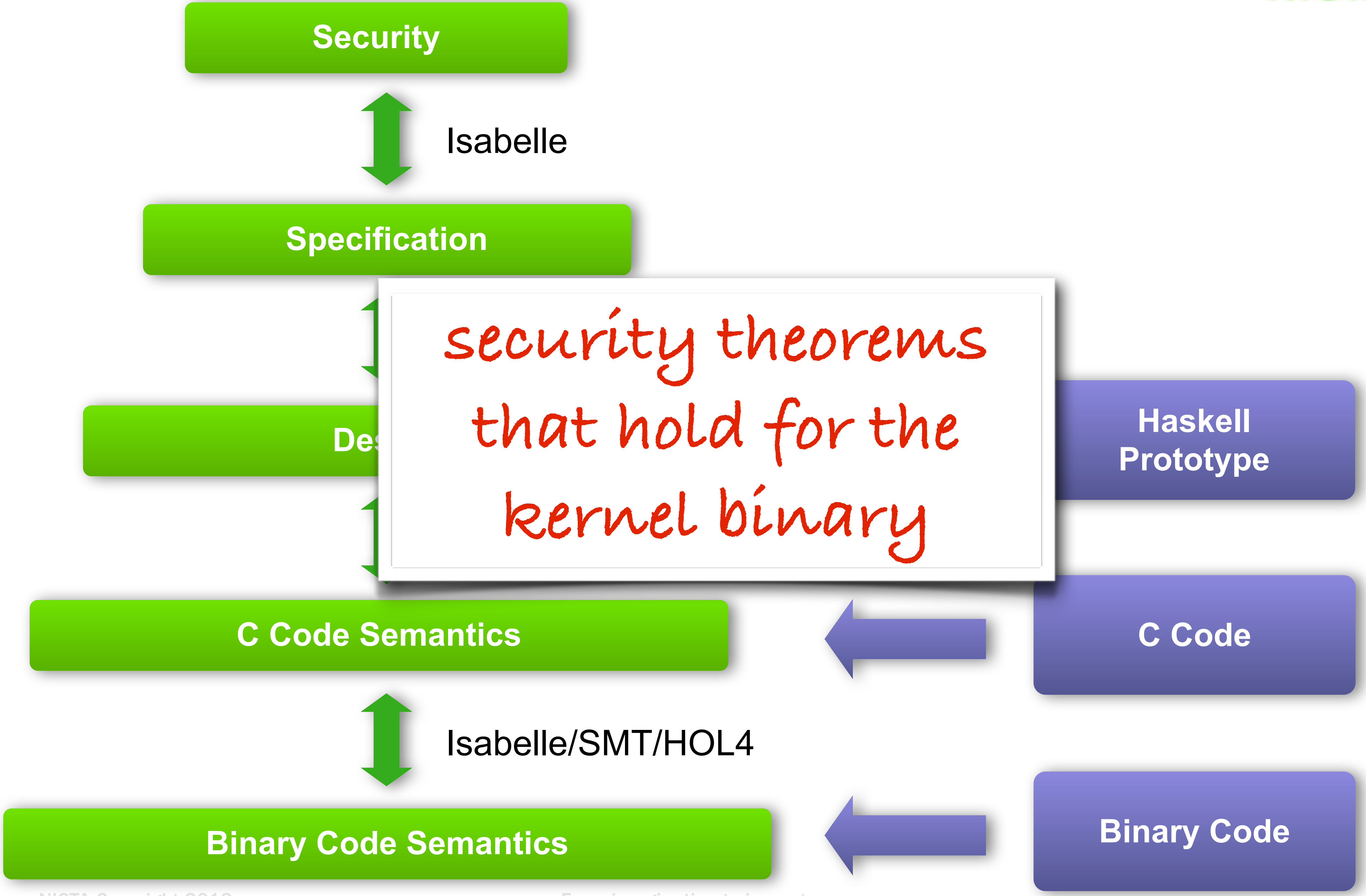
C Code

Isabelle/SMT/HOL4

Binary Code Semantics

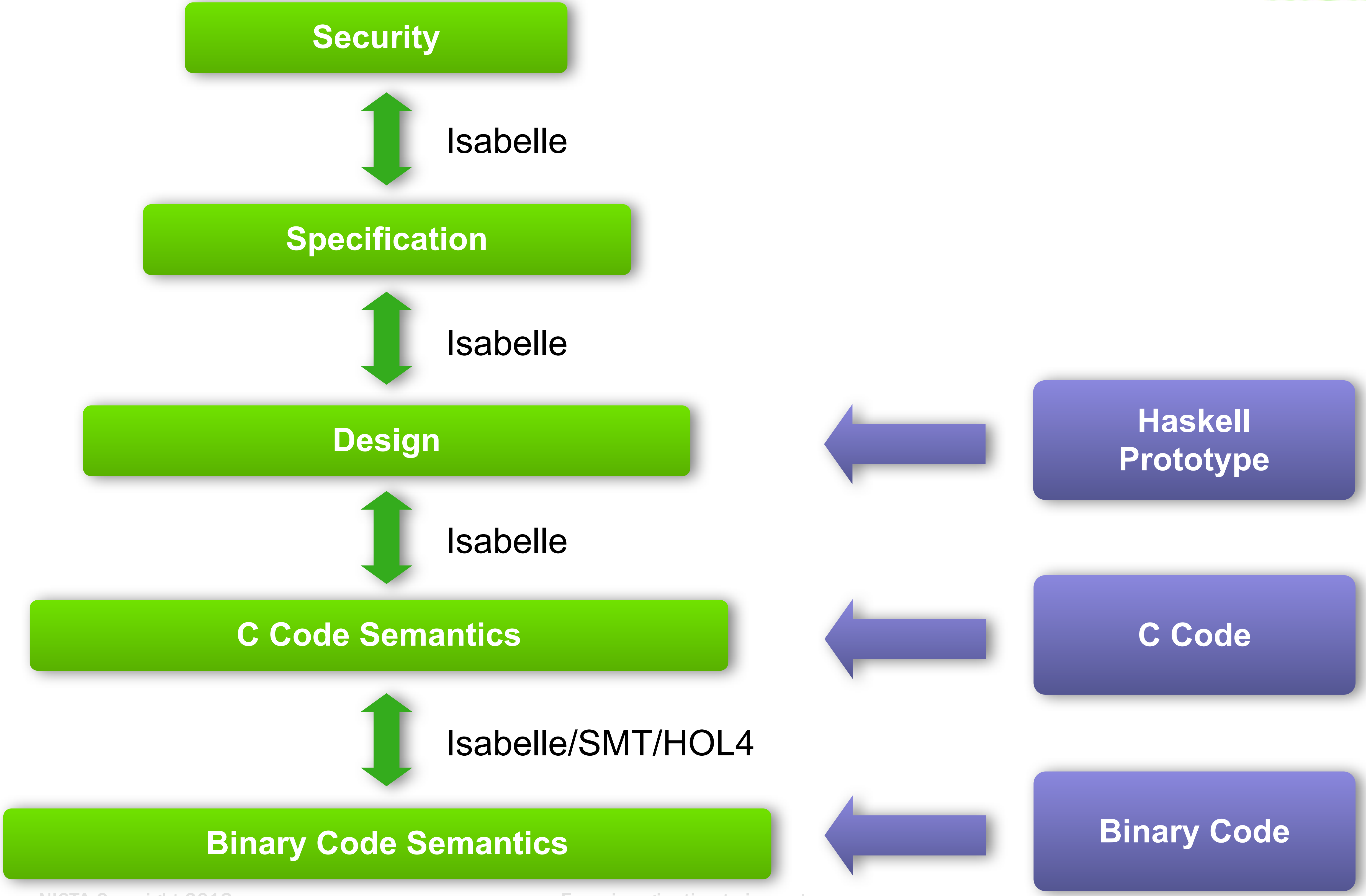
Binary Code

Proof Architecture (now)



security theorems that hold for the kernel binary

Proof Architecture (now)



Security

Isabelle

Specification

Isabelle

Design

Haskell Prototype

Isabelle

C Code Semantics

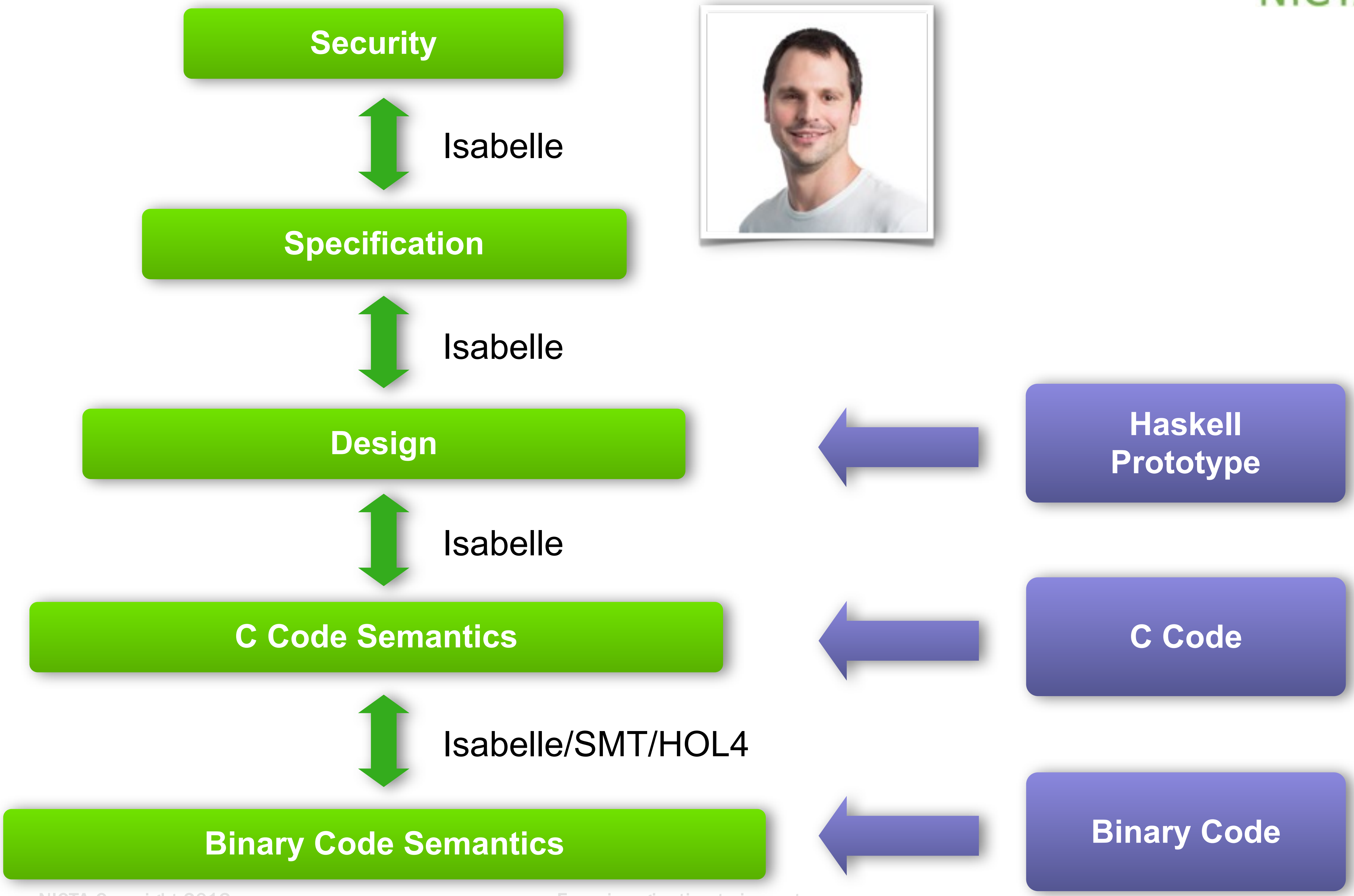
C Code

Isabelle/SMT/HOL4

Binary Code Semantics

Binary Code

Proof Architecture (now)



Security

Isabelle

Specification

Isabelle

Design

Isabelle

C Code Semantics

Isabelle/SMT/HOL4

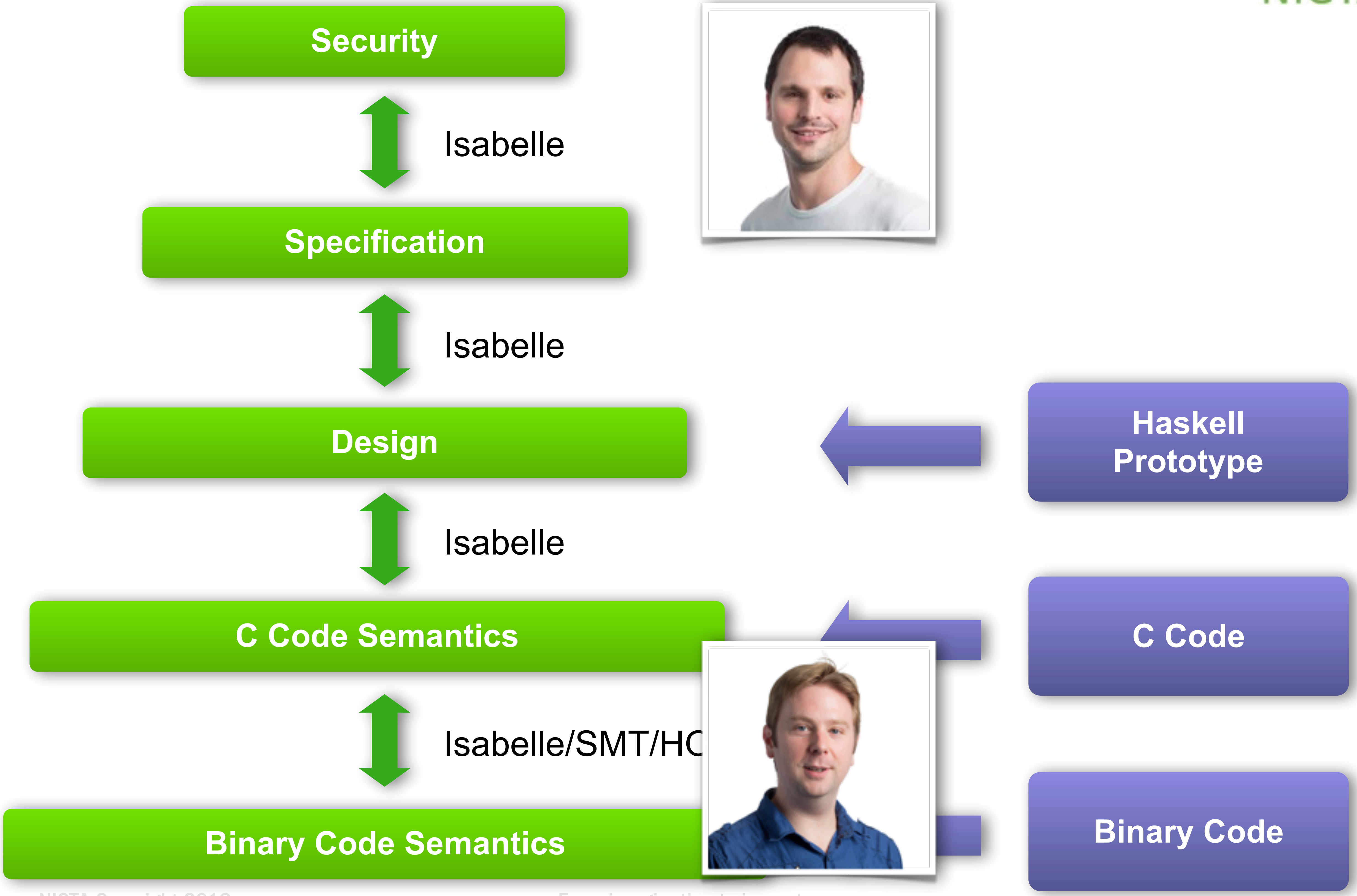
Binary Code Semantics

Haskell Prototype

C Code

Binary Code

Proof Architecture (now)



Security

Isabelle

Specification

Isabelle

Design

Isabelle

C Code Semantics

Isabelle/SMT/HC

Binary Code Semantics

Haskell Prototype

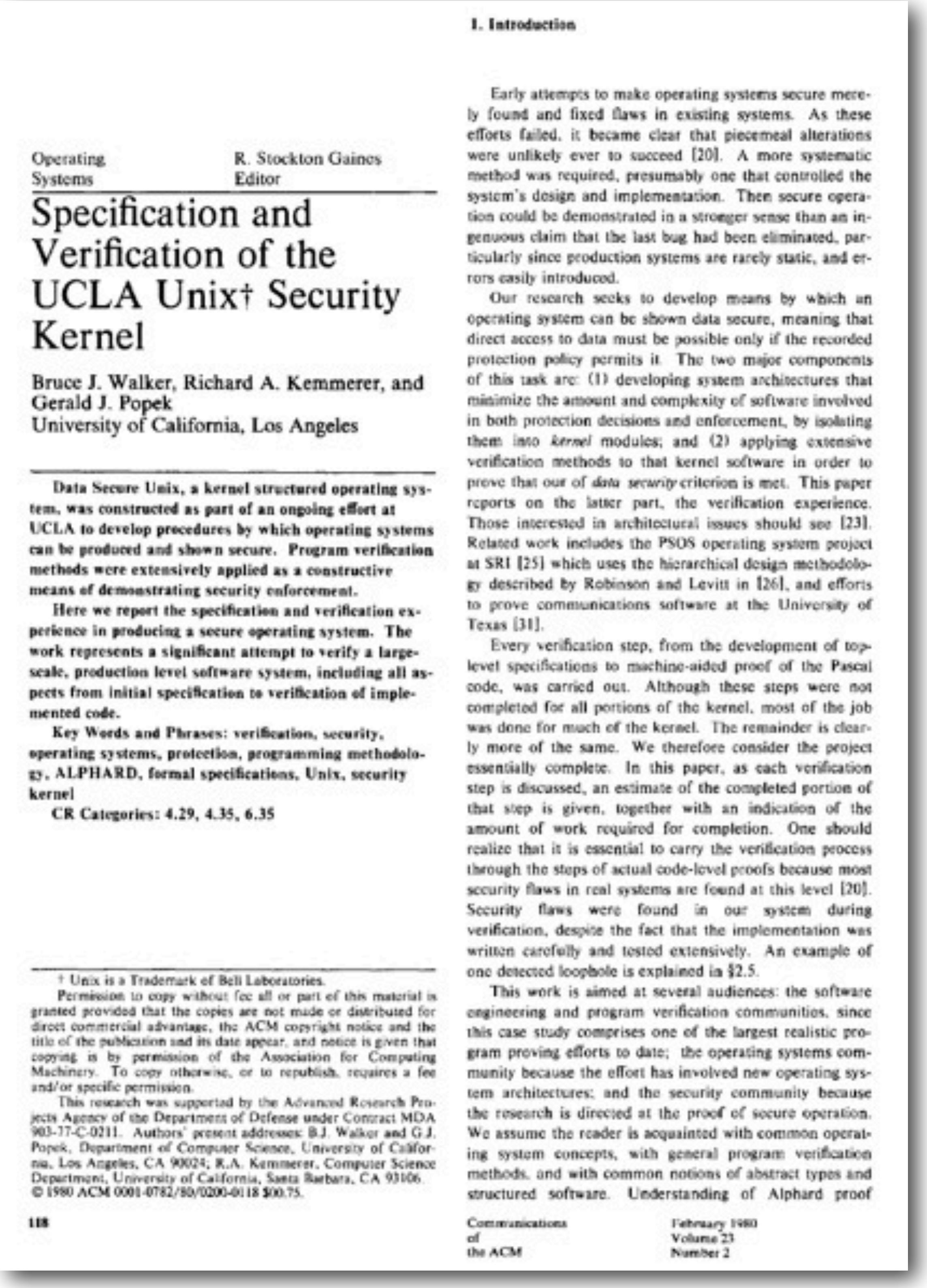
C Code

Binary Code

SECURITY

NICTA Copyright 2010

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

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.

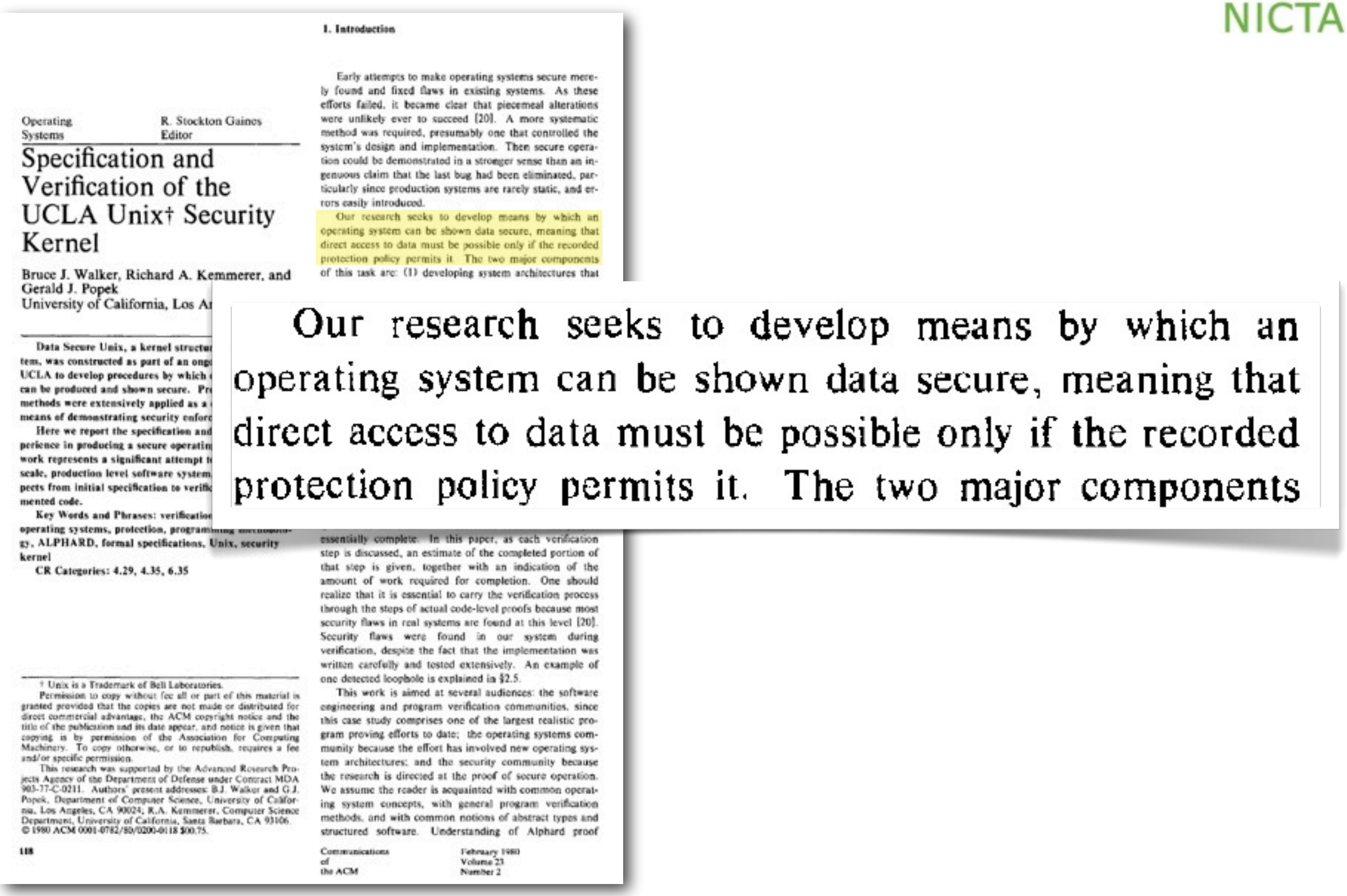
This work is aimed at several audiences: the software engineering and program verification communities, since this case study comprises one of the largest realistic program proving efforts to date; the operating systems community because the effort has involved new operating system architectures; and 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

† 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.

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 An

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

Here we report the specification and perience in producing a secure operatin work represents a significant attempt t scale, production level software system, pects from initial specification to verific mented code.

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

CR Categories: 4.29, 4.35, 6.35

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

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

† 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.

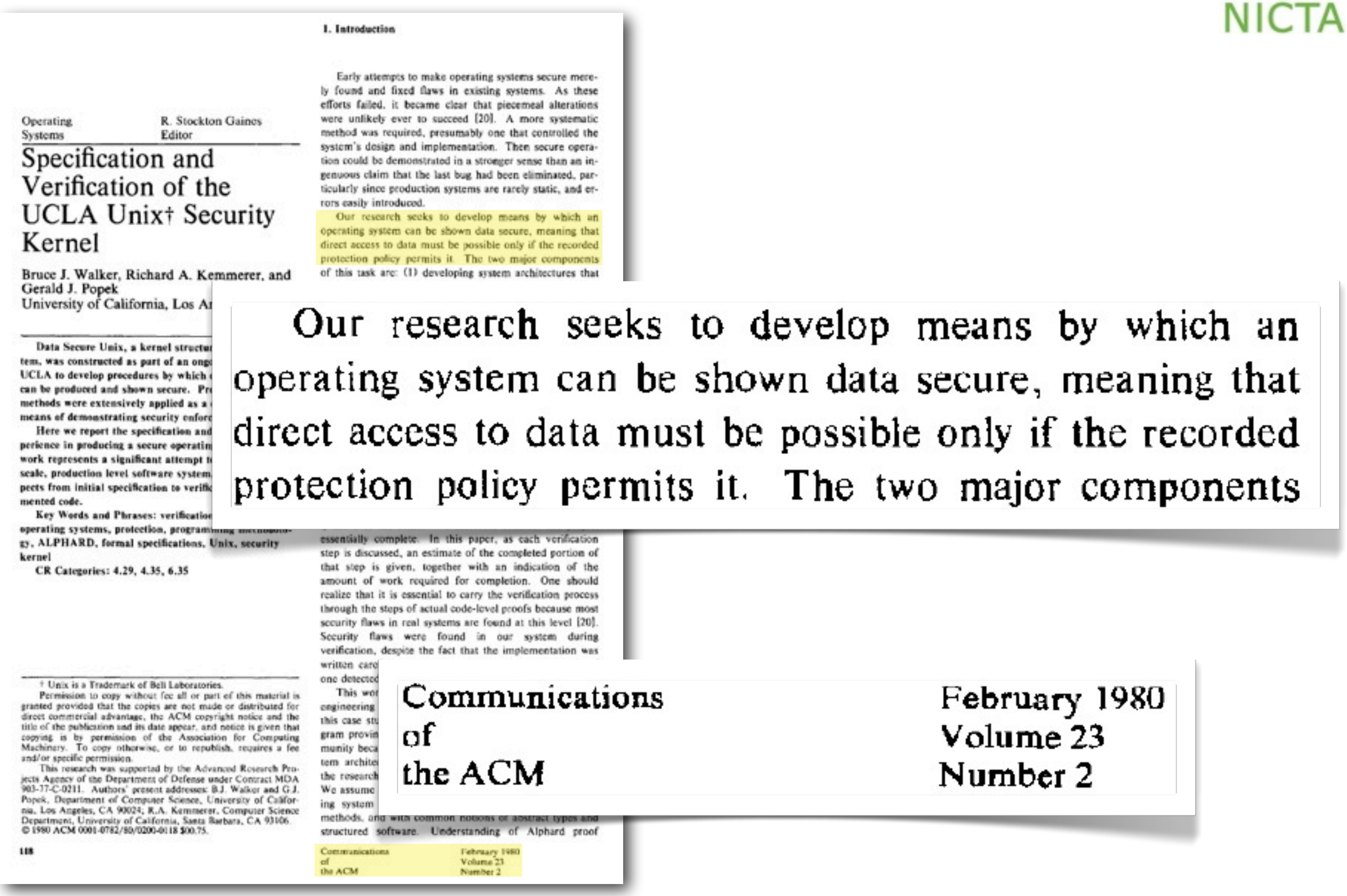
118

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. Kemmerer, and Gerald J. Popek
University of California, Los An

Data Secure Unix, a kernel structur
tem, was constructed as part of an ongo
UCLA to develop procedures by which
can be produced and shown secure. Pr
methods were extensively applied as a
means of demonstrating security enforc
Here we report the specification and
perience in producing a secure operatin
work represents a significant attempt t
scale, production level software system
pects from initial specification to verific
mented code.

Key Words and Phrases: verificatio
operating systems, protection, programming methodolo-
gy, 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 of this task are: (1) developing system architectures that

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 care
one detected

This wor
engineering
this case stu
gram provin
munity beca
tem archite
the research
We assume
ing system
methods, and with common notions of abstract types and
structured software. Understanding of Alphard proof

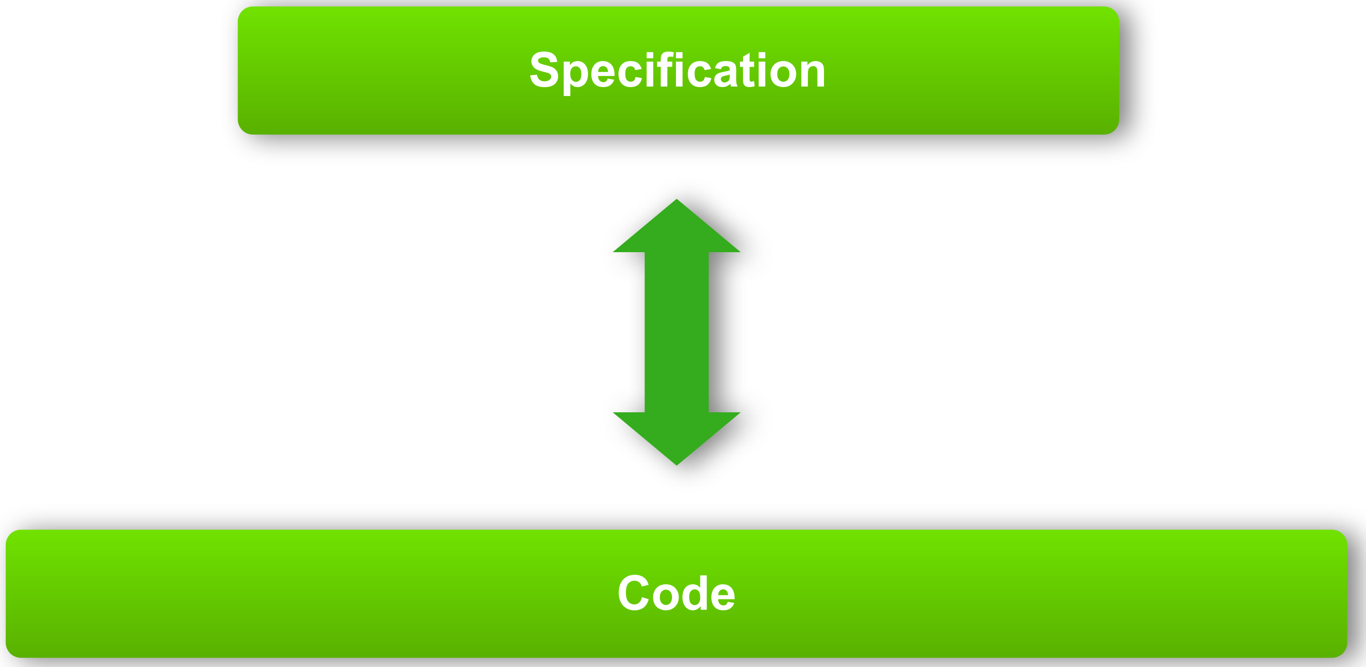
Communications of the ACM
February 1980 Volume 23 Number 2

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

Communications
of
the ACM

February 1980
Volume 23
Number 2

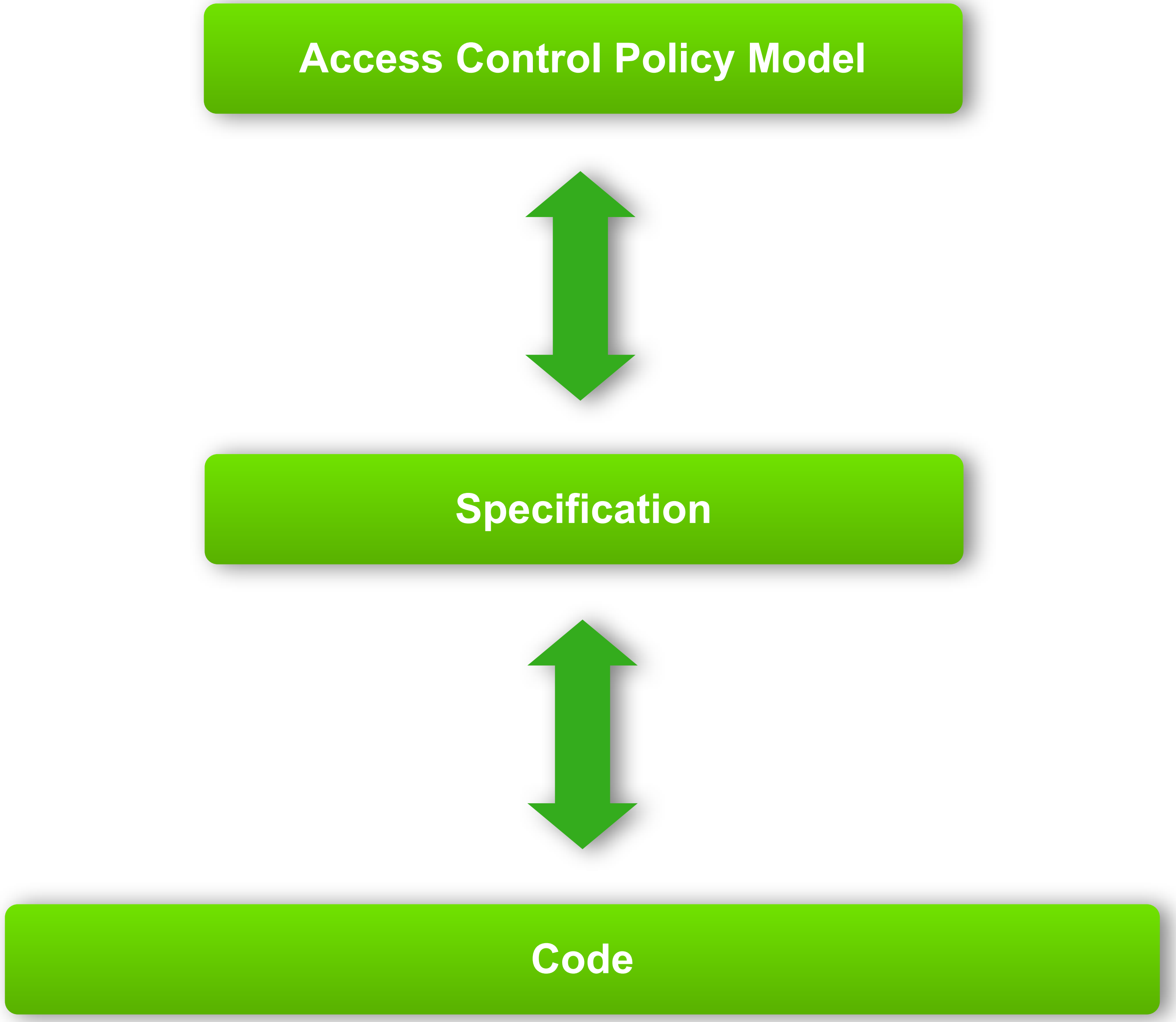
seL4 Security Proofs: Overview



Specification

Code

seL4 Security Proofs: Overview

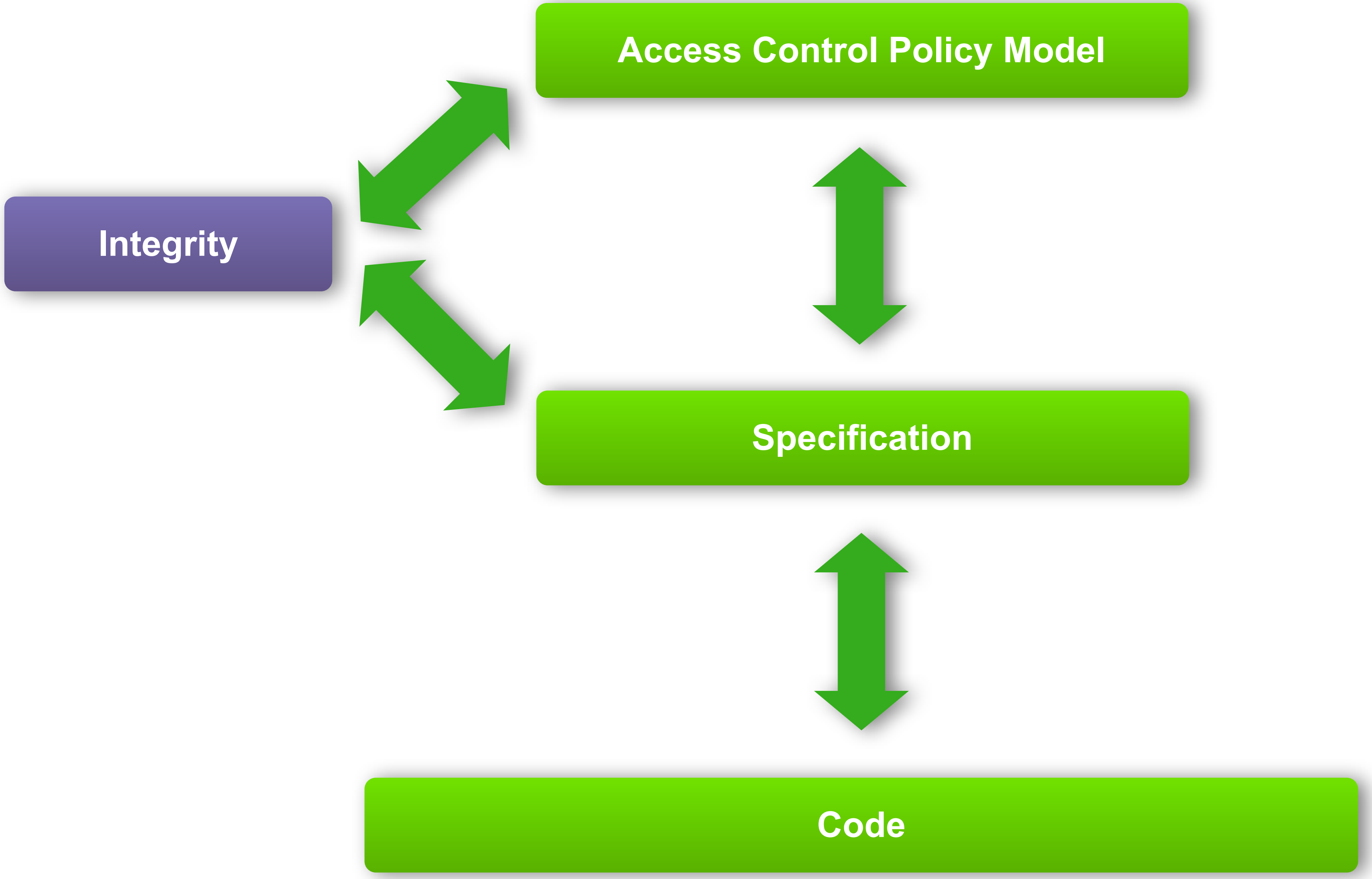


Access Control Policy Model

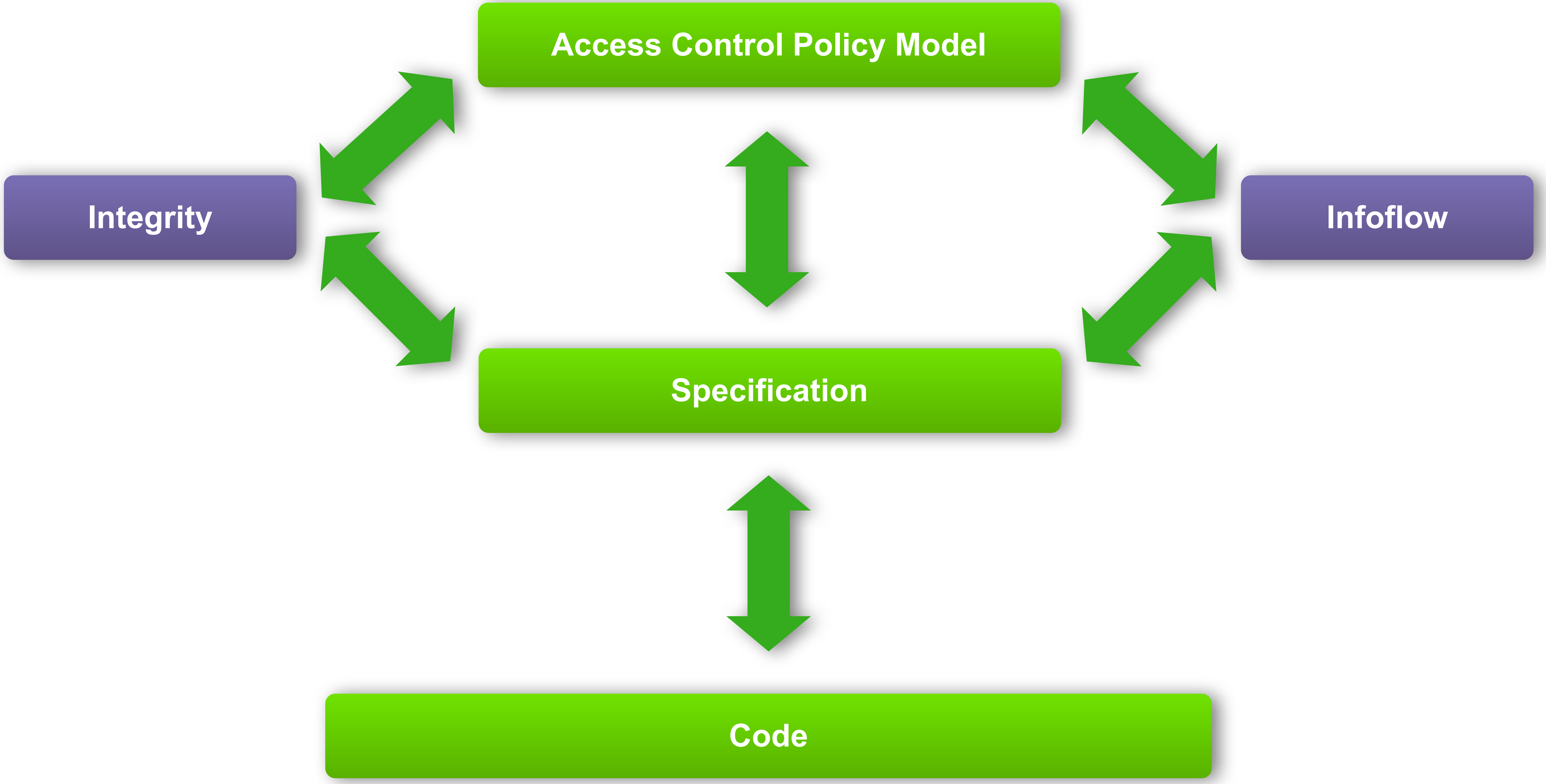
Specification

Code

seL4 Security Proofs: Overview

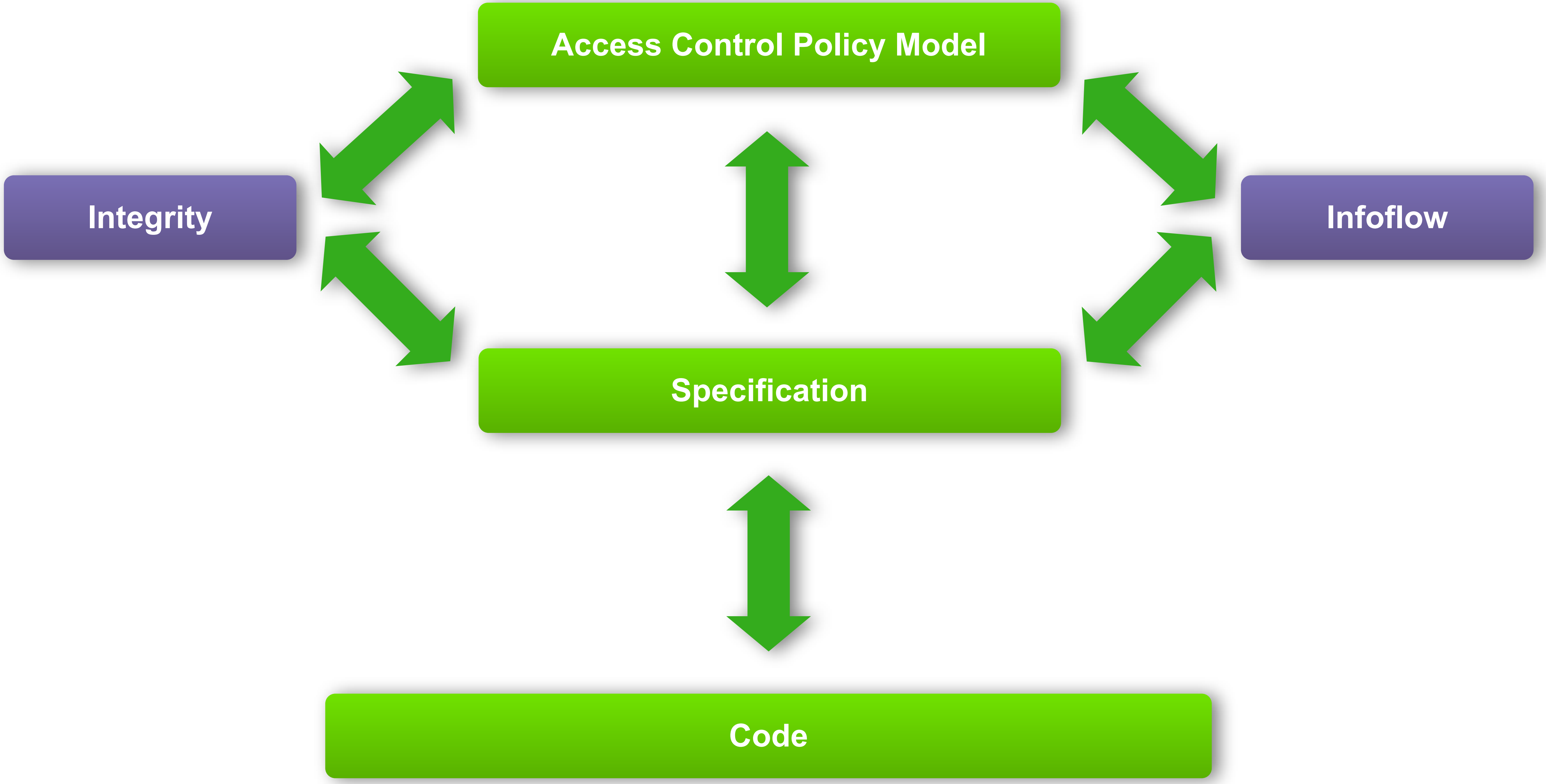


seL4 Security Proofs: Overview



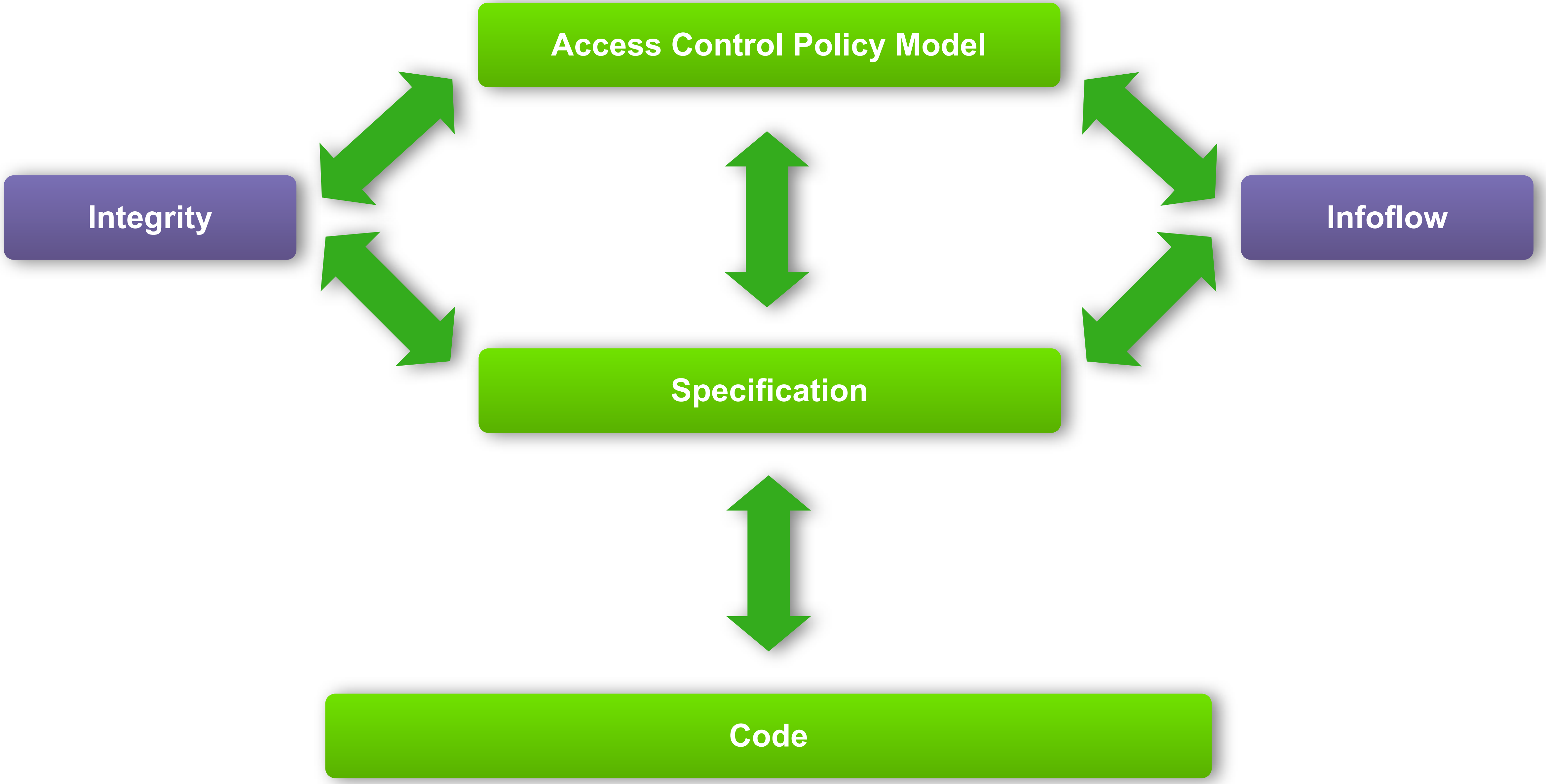
seL4 Security Proofs: Overview

Integrity + Infoflow --» Isolation



seL4 Security Proofs: Overview

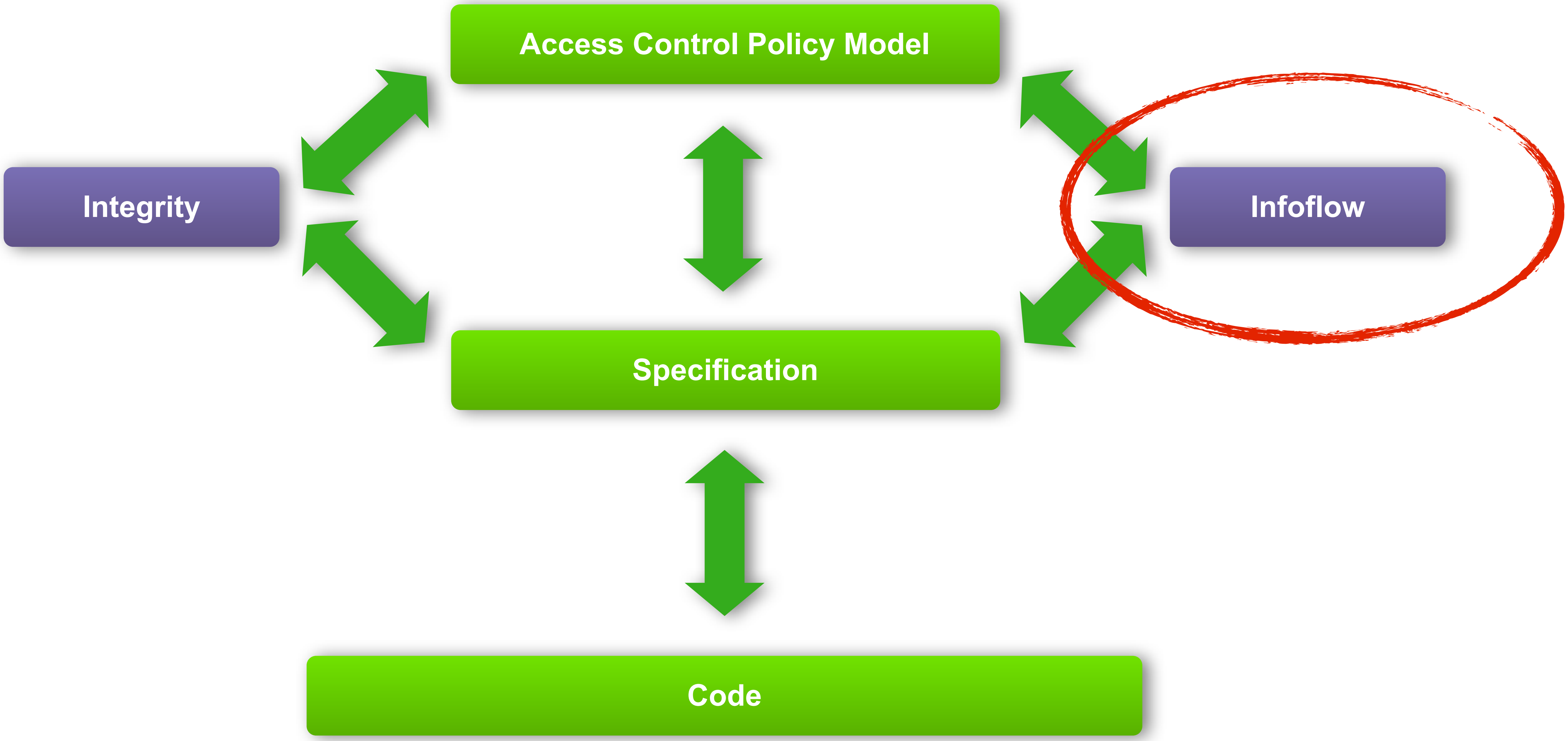
Integrity + Infoflow --» Isolation



Infoflow --» Confidentiality

seL4 Security Proofs: Overview

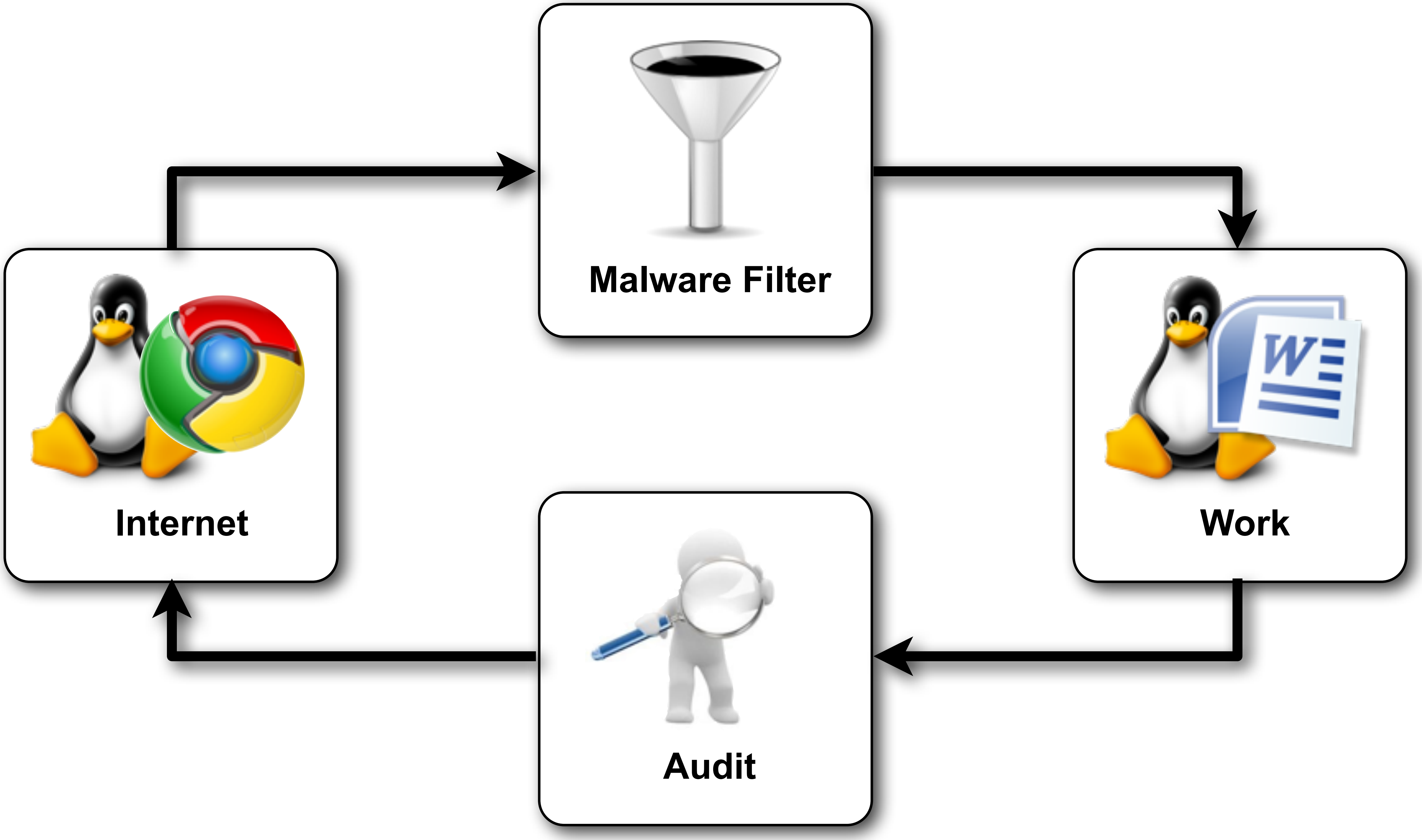
Integrity + Infoflow --» Isolation



Infoflow --» Confidentiality

NICTA Copyright 2010

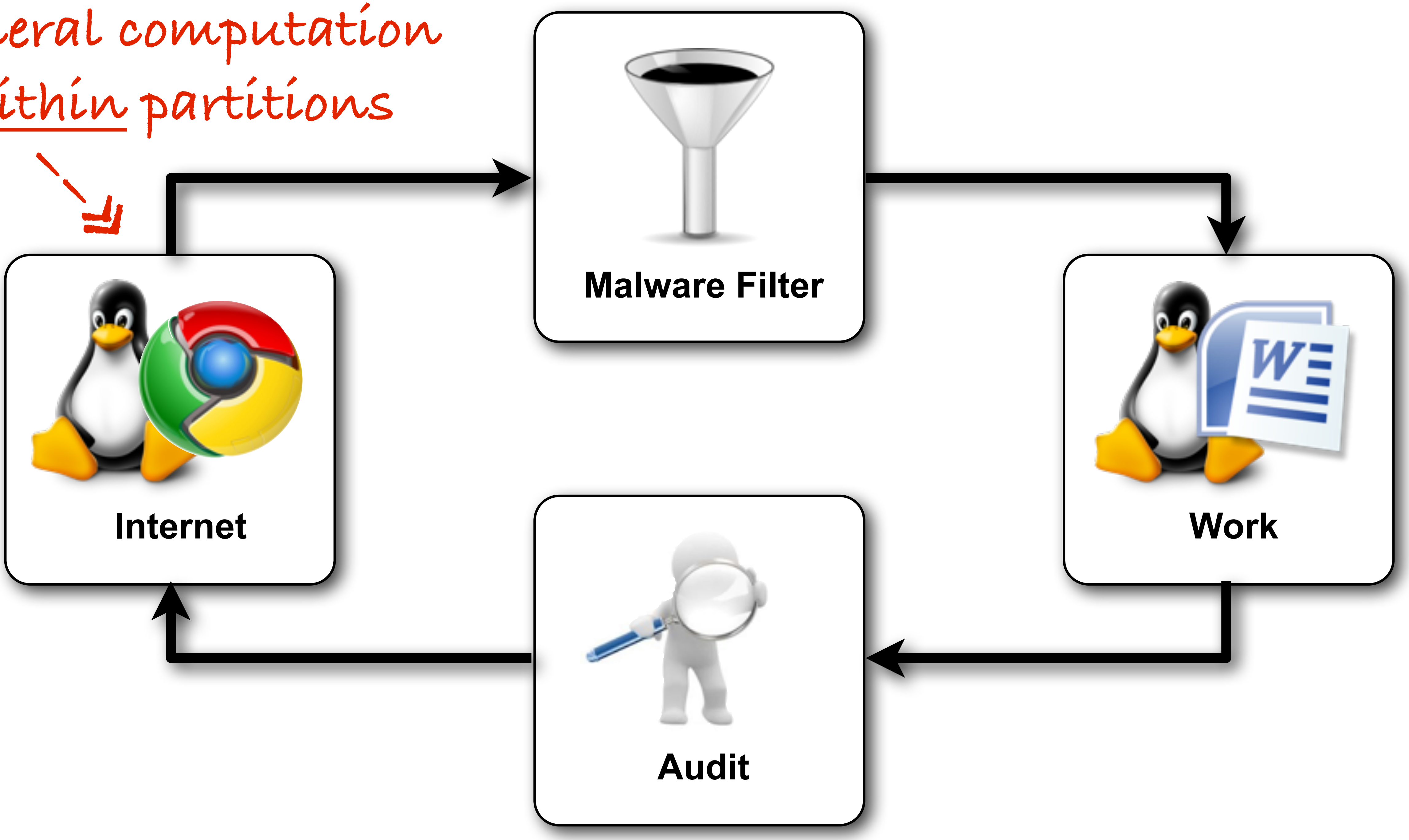
Information Flow Security



NICTA Copyright 2010

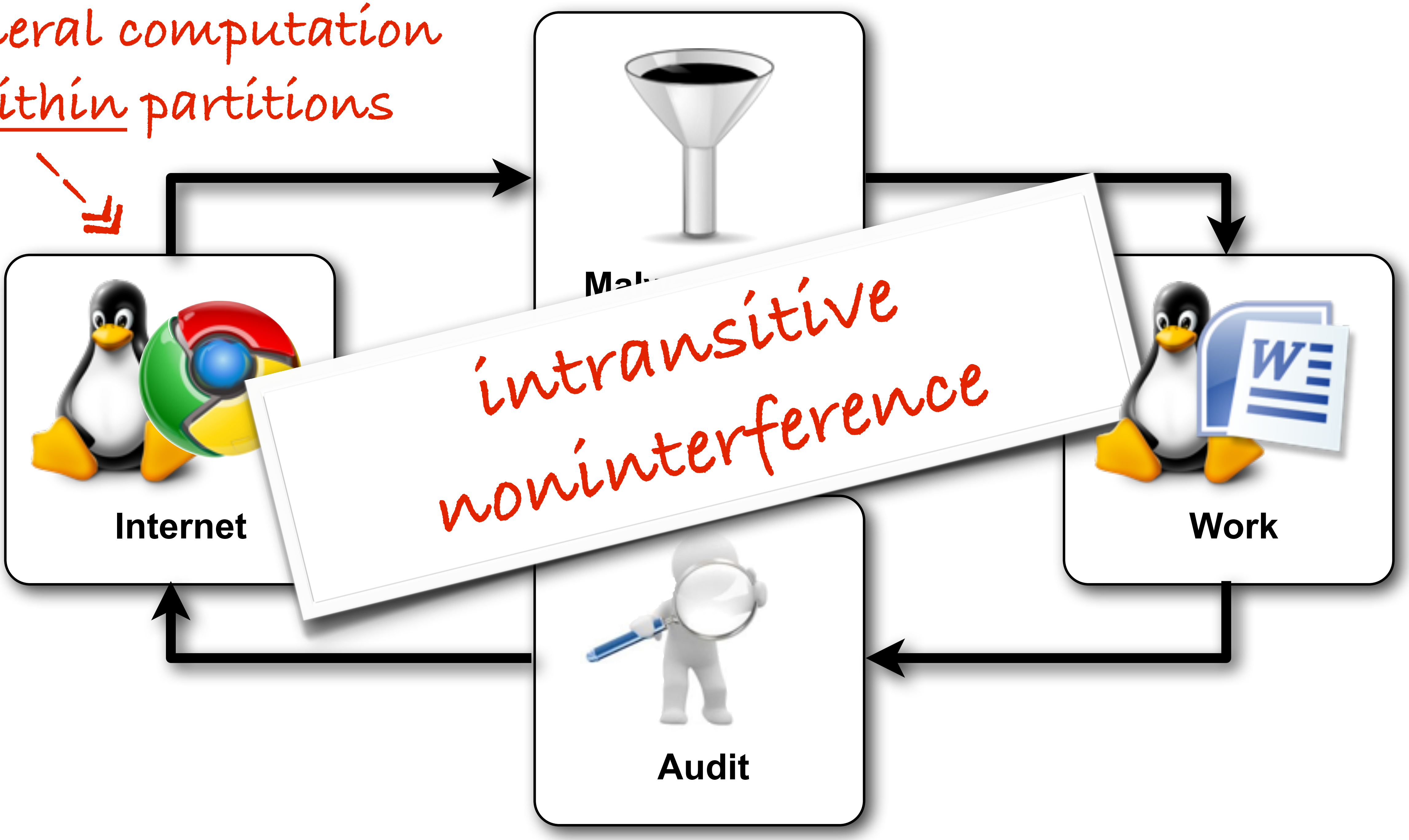
Information Flow Security

general computation
within partitions



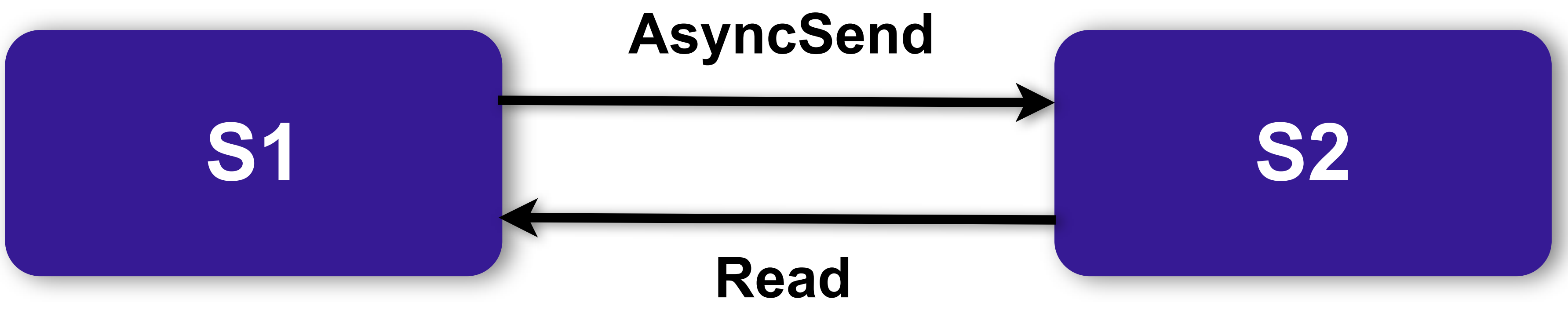
Information Flow Security

general computation
within partitions



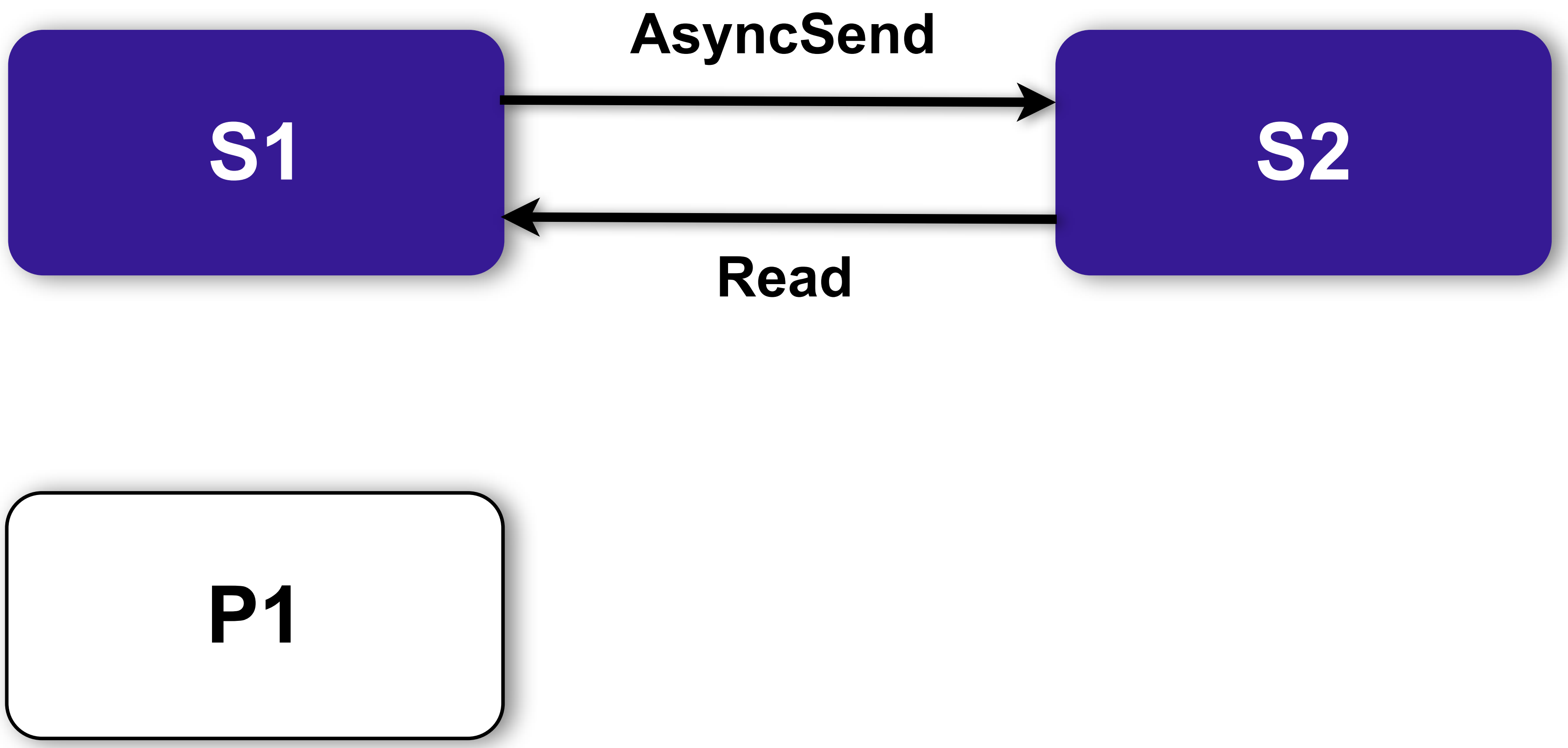
Information Flow Policy

- Derived from access control policy



Information Flow Policy

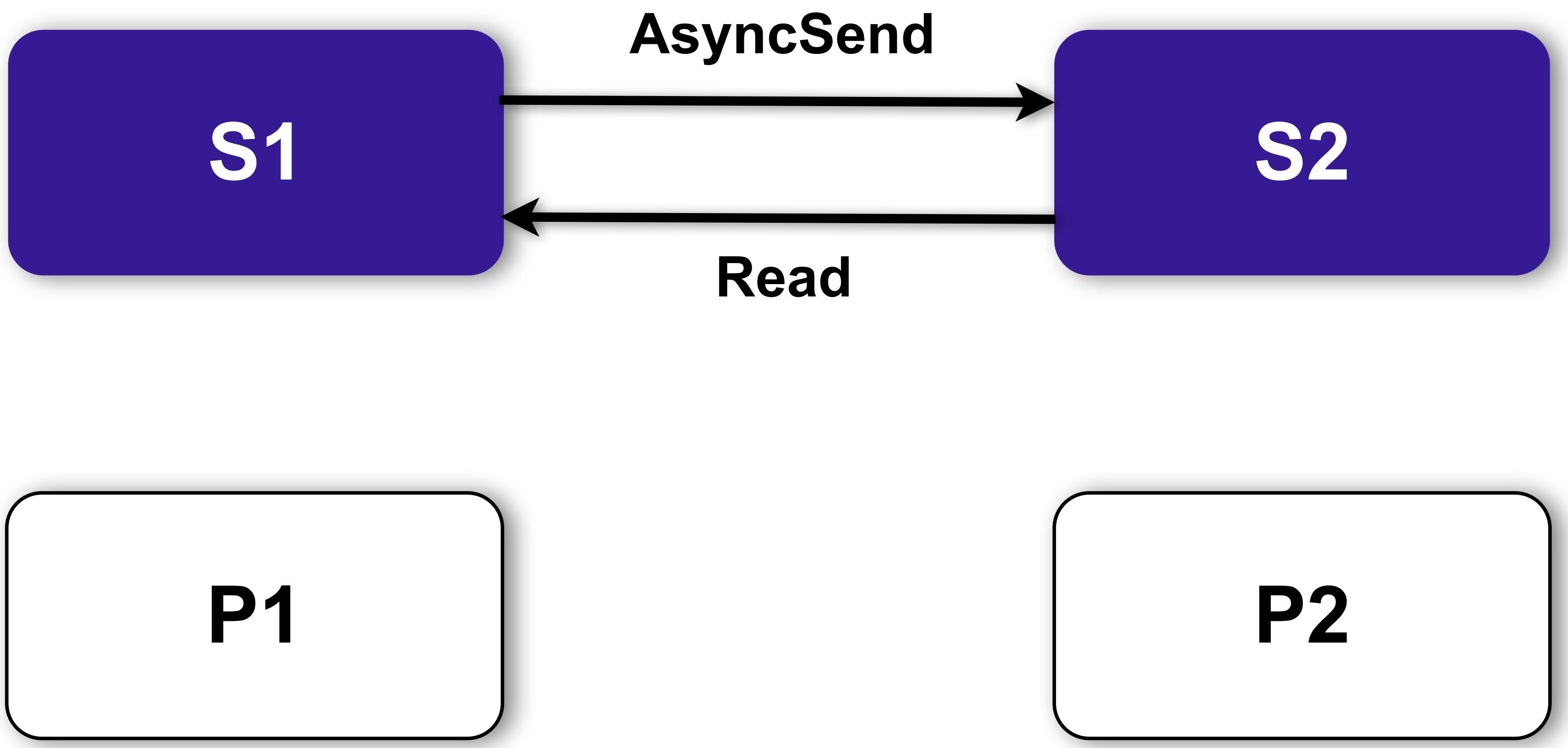
- Derived from access control policy



NICTA Copyright 2010

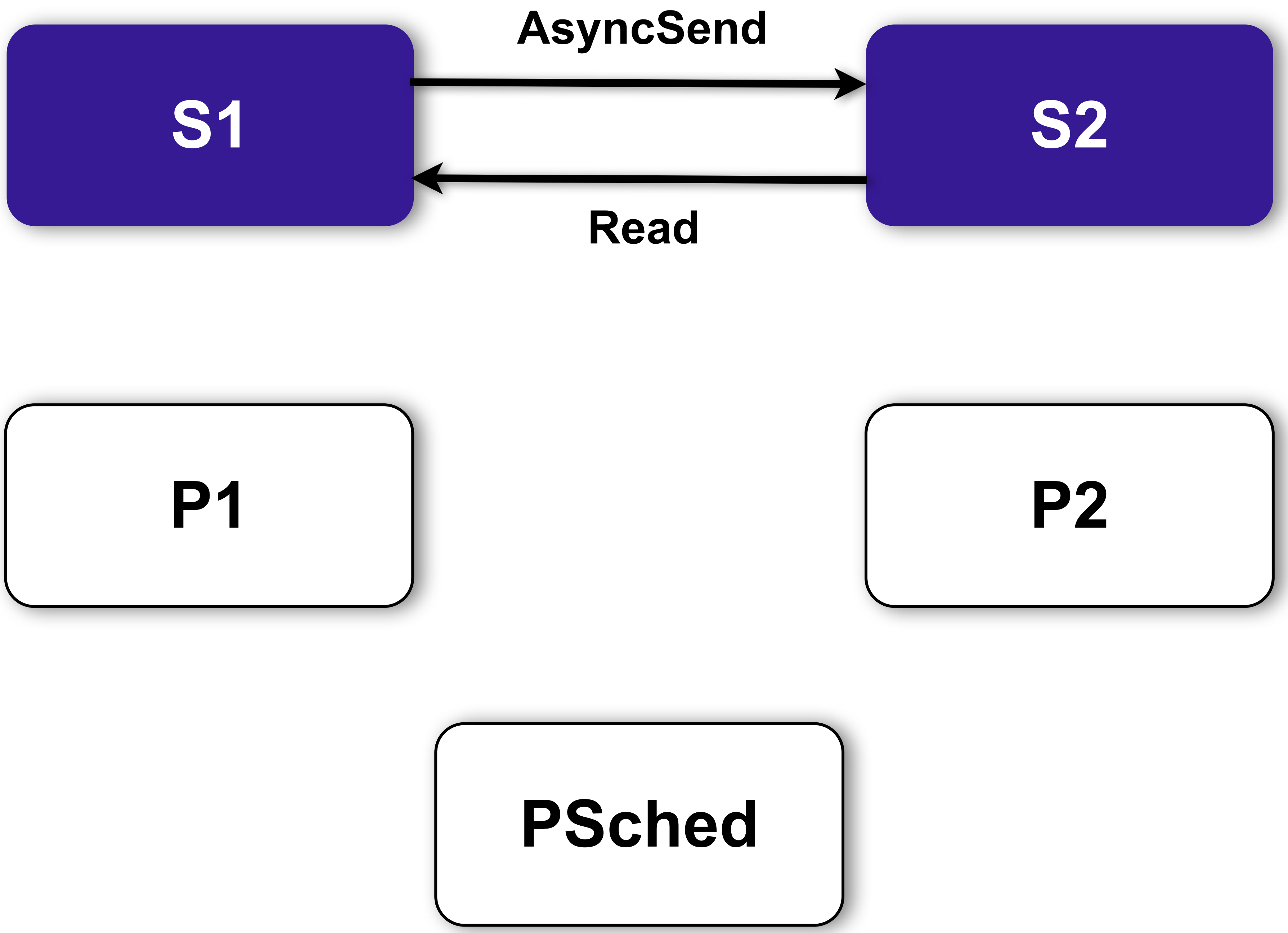
Information Flow Policy

- Derived from access control policy



Information Flow Policy

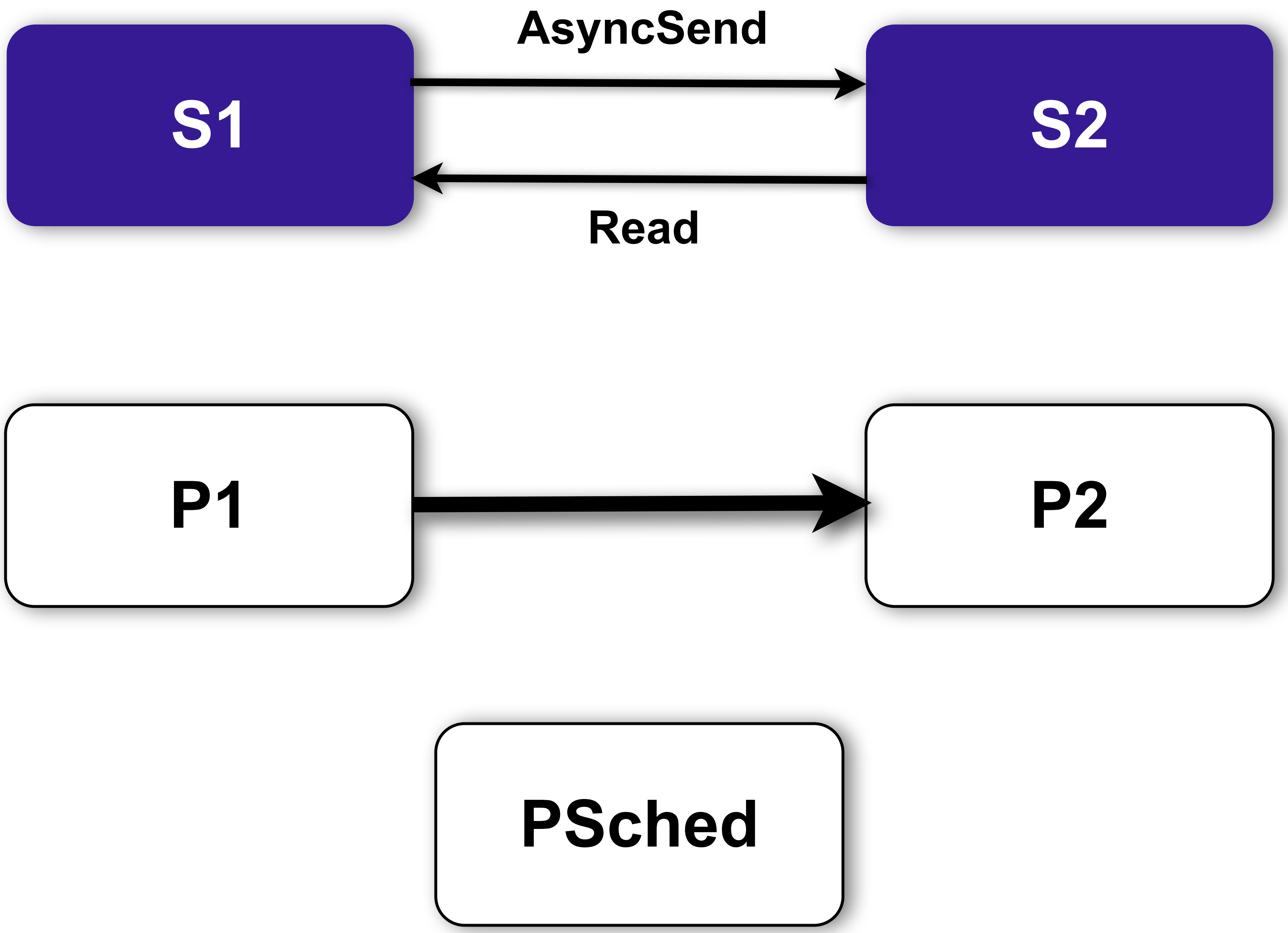
- Derived from access control policy



NICTA Copyright 2010

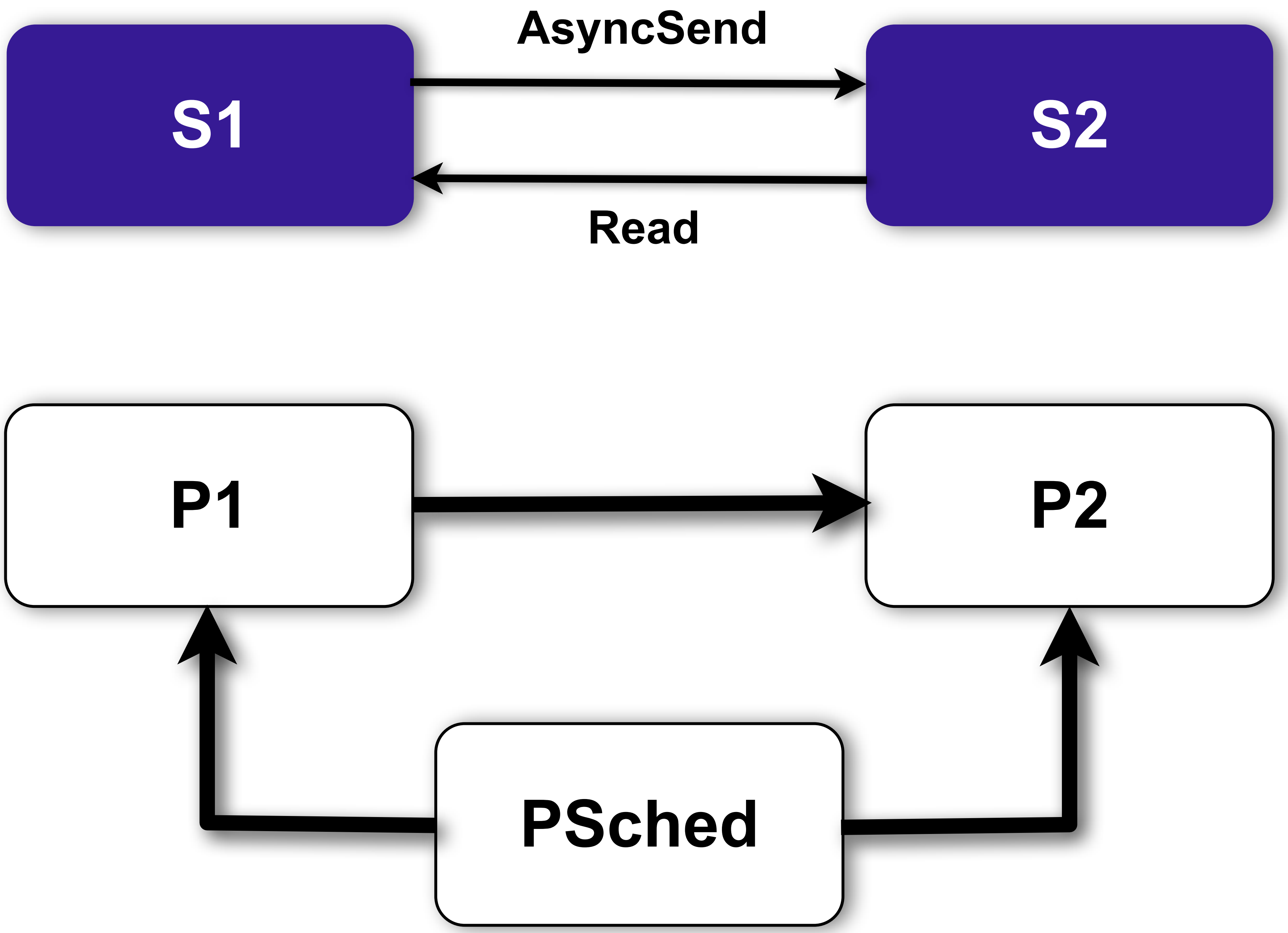
Information Flow Policy

- Derived from access control policy



Information Flow Policy

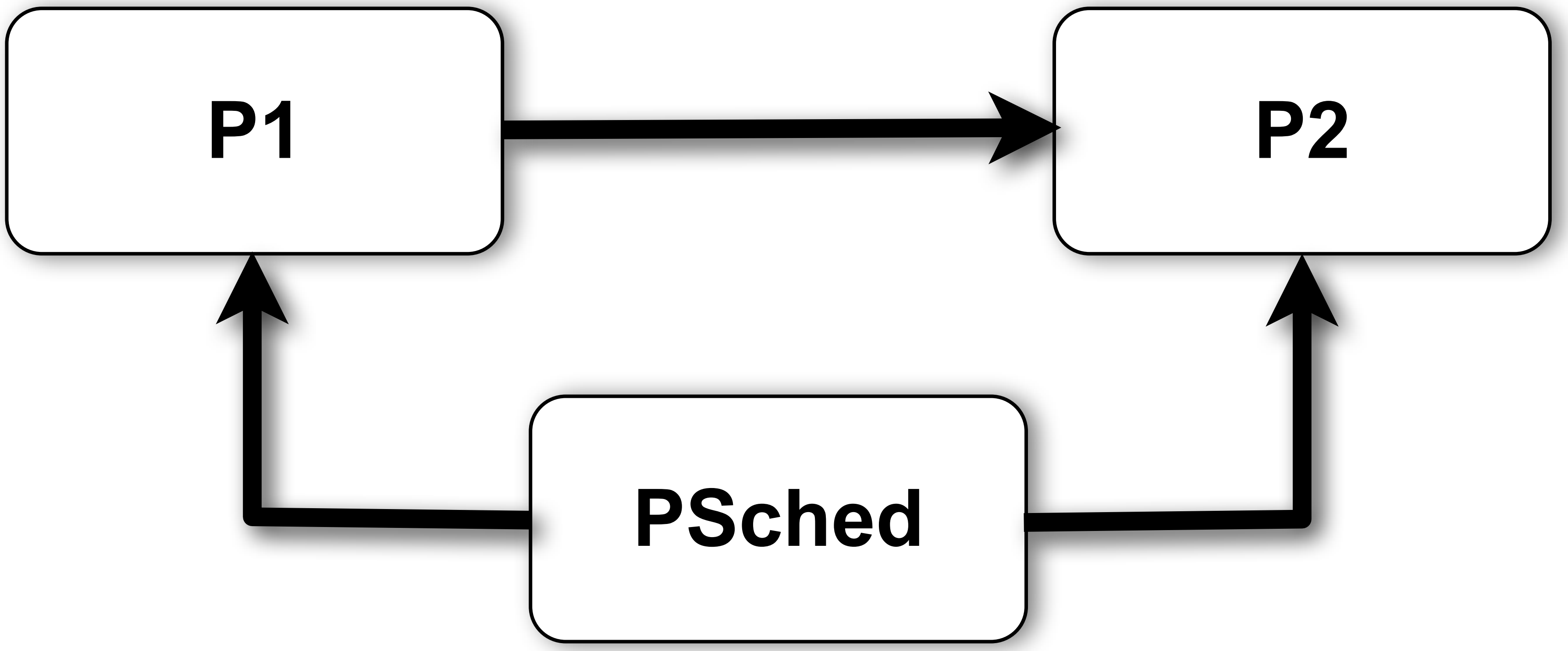
- Derived from access control policy



Information Flow Policy

- Derived from access control policy

no-one may affect scheduling decisions



P1

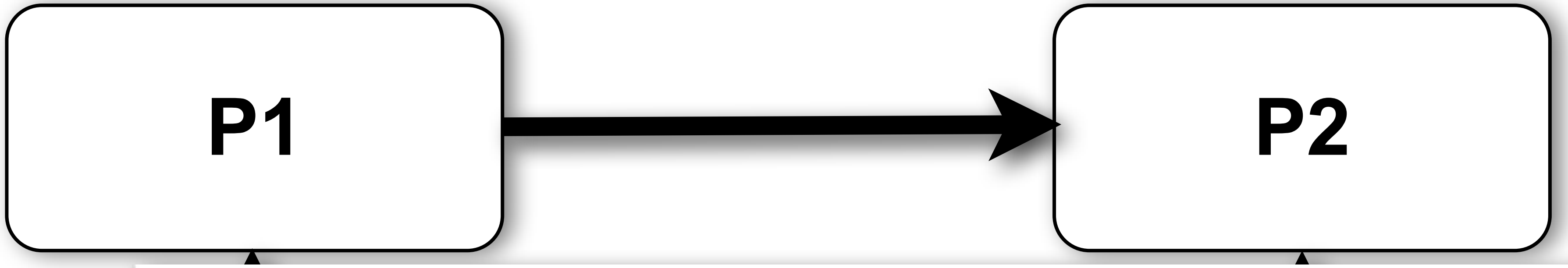
P2

PSched

Information Flow Policy

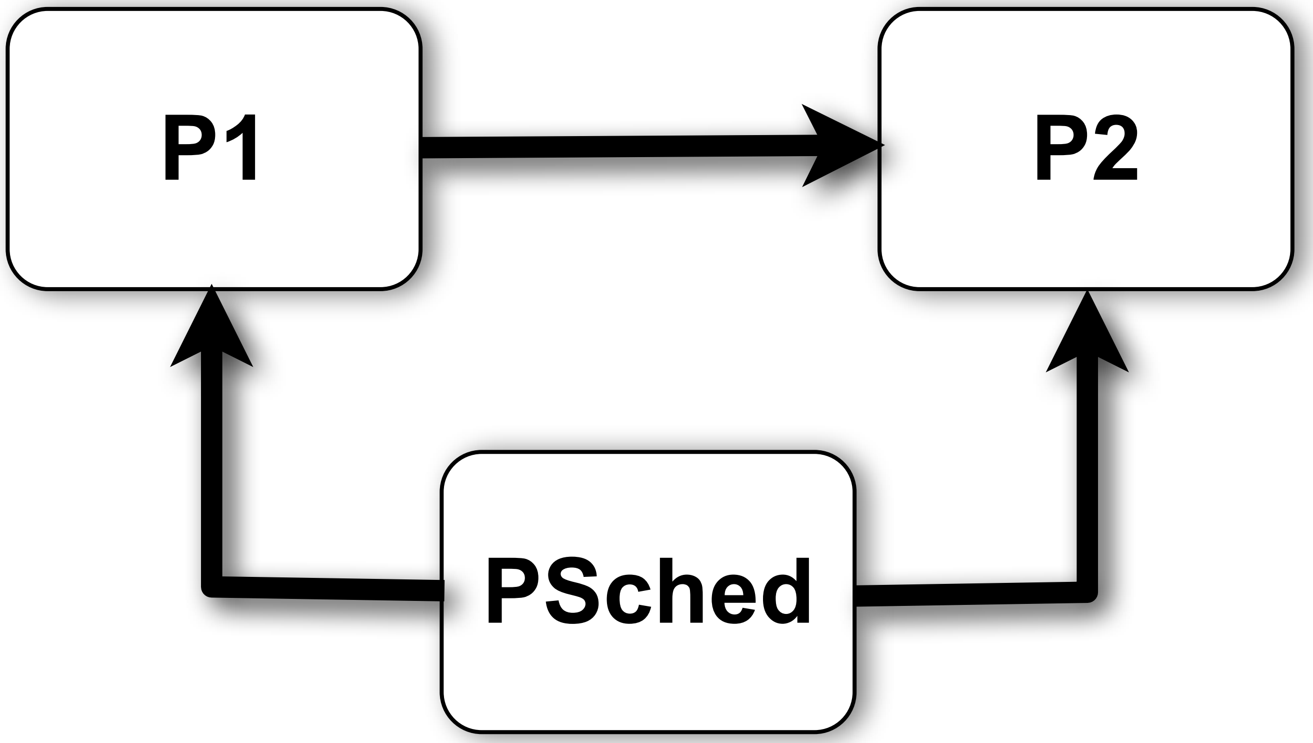
- Derived from access control policy

no-one may affect scheduling decisions



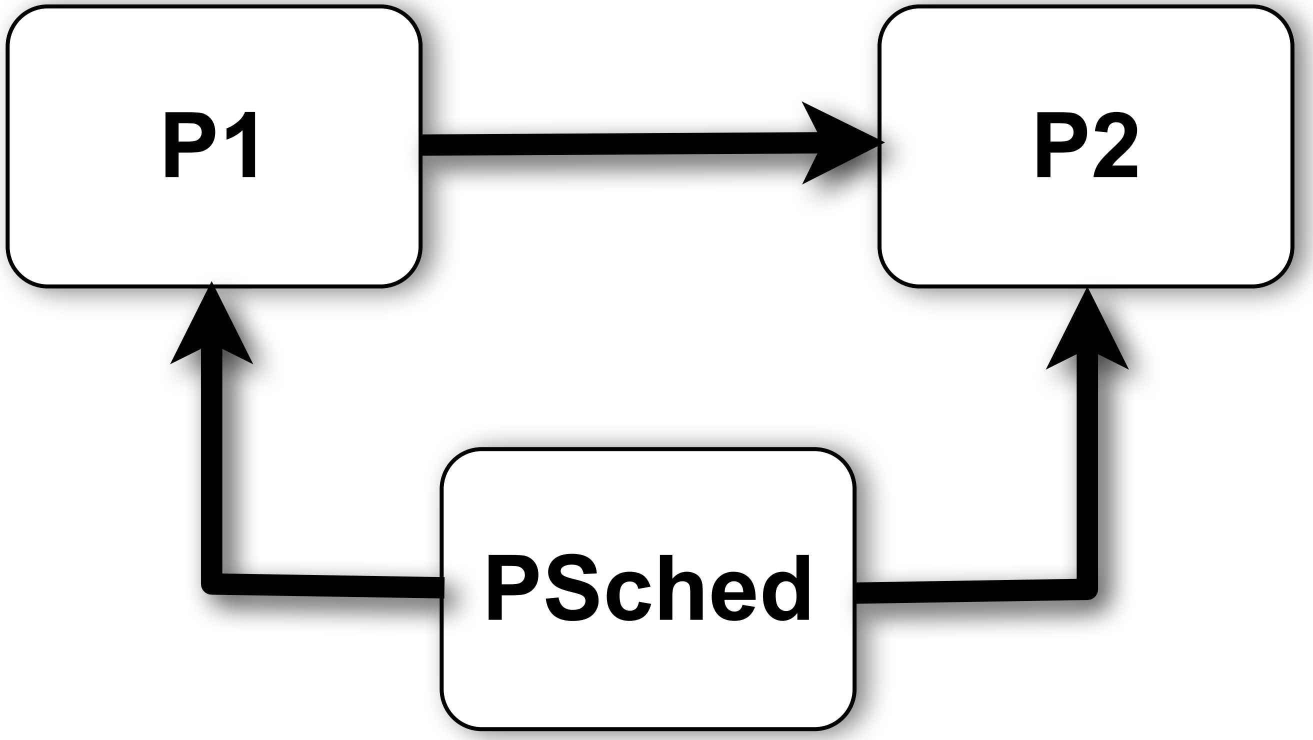
ensures PSched is not a global transitive channel

Intransitive Nonleakage



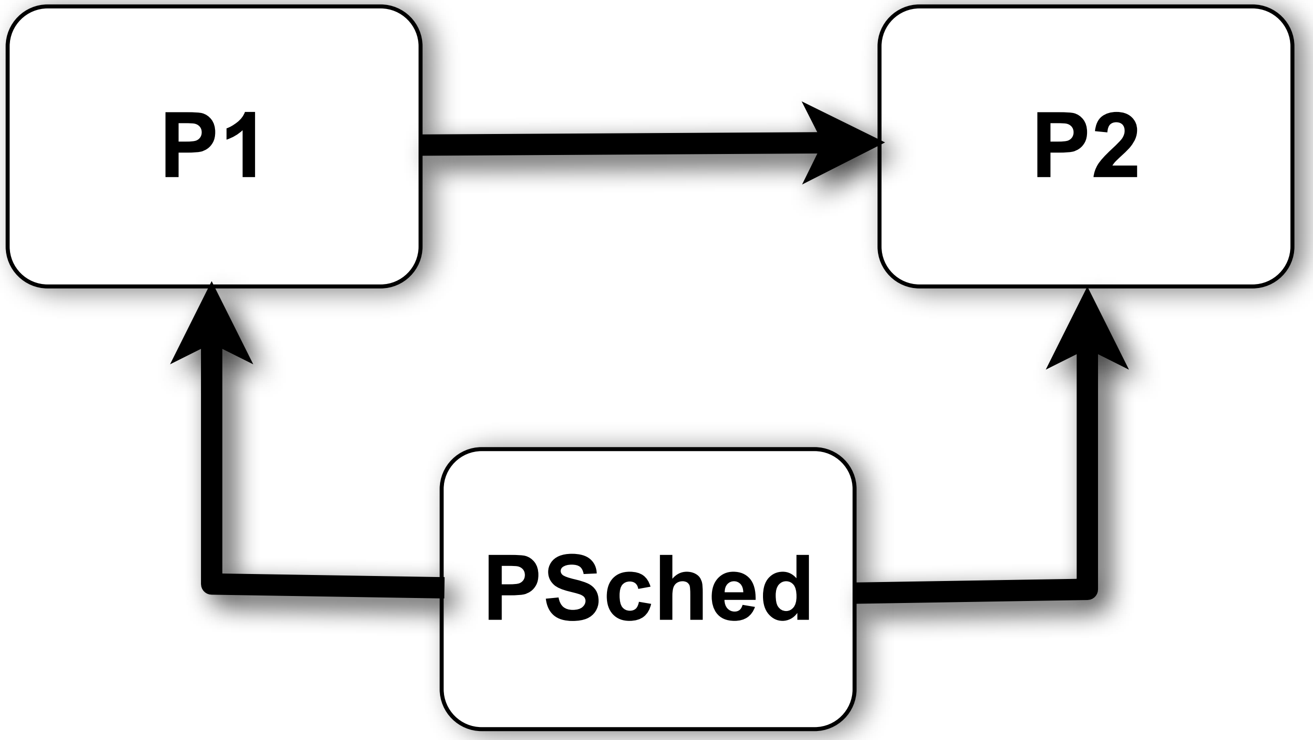
Intransitive Nonleakage

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



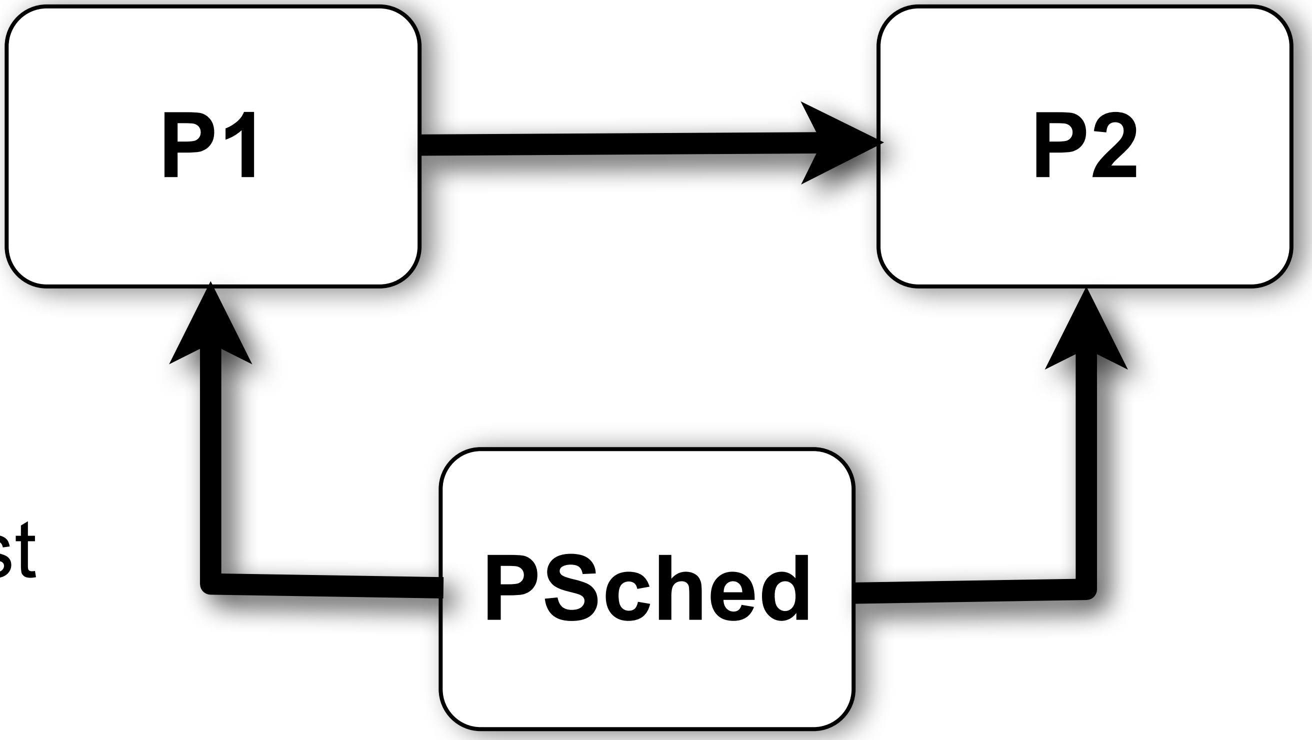
Intransitive Nonleakage

- Variant of **intransitive noninterference**
  - Asserts absence of information leaks
- 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
- Sufficient because scheduler follows a fixed round-robin partition-schedule
  - **Implied assumption:** everyone is allowed to know the static partition-schedule
  - When P2 executes, it thus already knows that P1 must have finished executing



Problematic Kernel APIs

Problematic Kernel APIs

- Leaky kernel APIs need to be disabled

Problematic Kernel APIs

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

Problematic Kernel APIs

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

Problematic Kernel APIs

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

Problematic Kernel APIs

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

- Asynchronous interrupt delivery

Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
  - the proof guarantees they will stay disabled
- Asynchronous interrupt delivery
  - device drivers must poll for interrupts

Problematic Kernel APIs

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

- Asynchronous interrupt delivery
  - device drivers must poll for interrupts

Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
  - the proof guarantees they will stay disabled
- Asynchronous interrupt delivery
  - device drivers must poll for interrupts
- Inter-partition object destruction

Problematic Kernel APIs

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

- Asynchronous interrupt delivery
  - device drivers must poll for interrupts

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

Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
  - the proof guarantees they will stay disabled
- Asynchronous interrupt delivery
  - device drivers must poll for interrupts
- Inter-partition object destruction
  - partition-crossing comms. channels cannot be destroyed

not uncommon in high-assurance systems

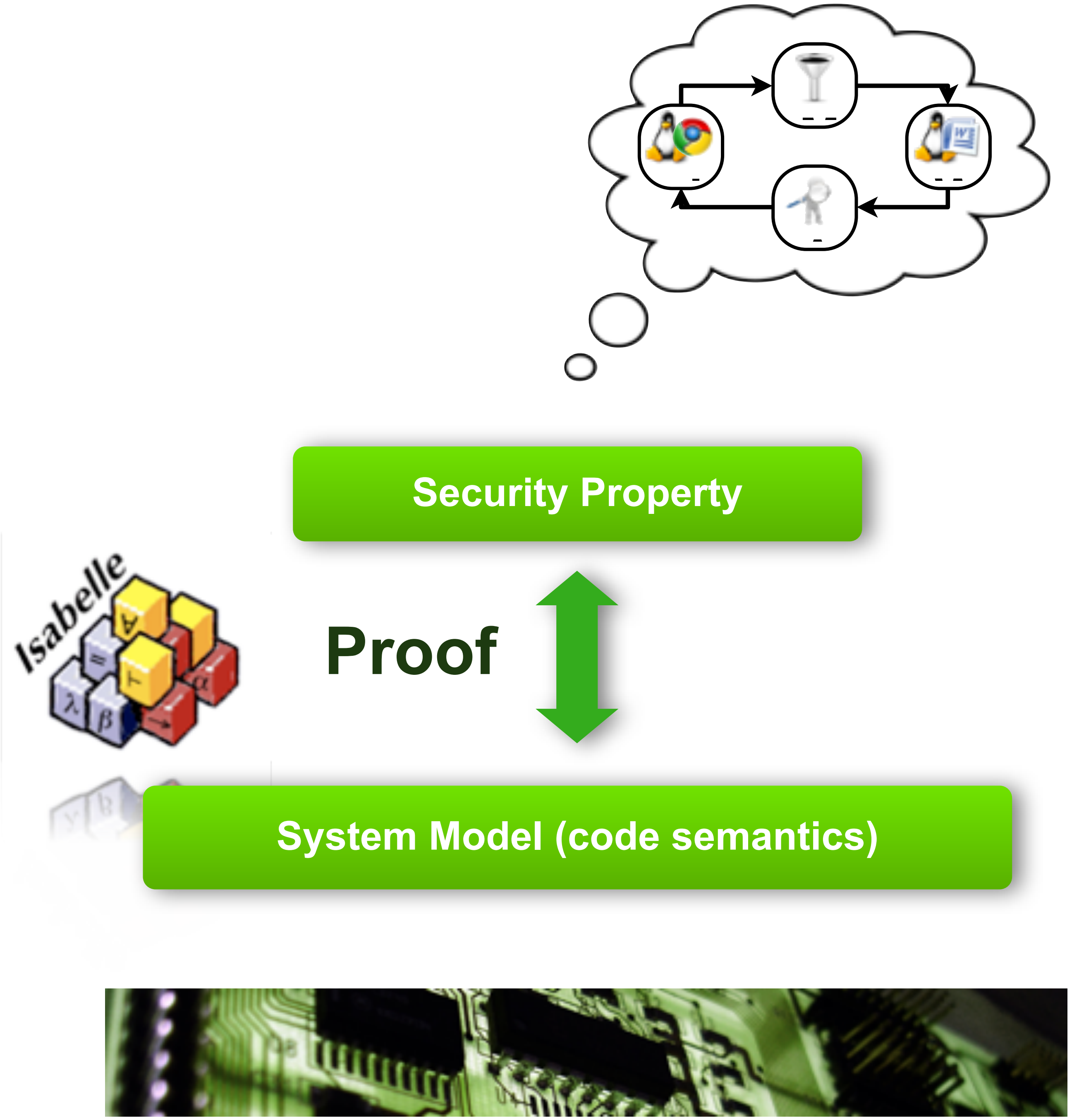
Problematic Kernel APIs

- Leaky kernel APIs need to be disabled
  - by ensuring initially no subject has permission to use them
  - the proof guarantees they will stay disabl
- Asynchronou
  - device d
- Inter-partiti
  - partition-cr be destroyed

all kernel services available within partitions, besides async irq notification

not uncommon in high-assurance systems

Assurance



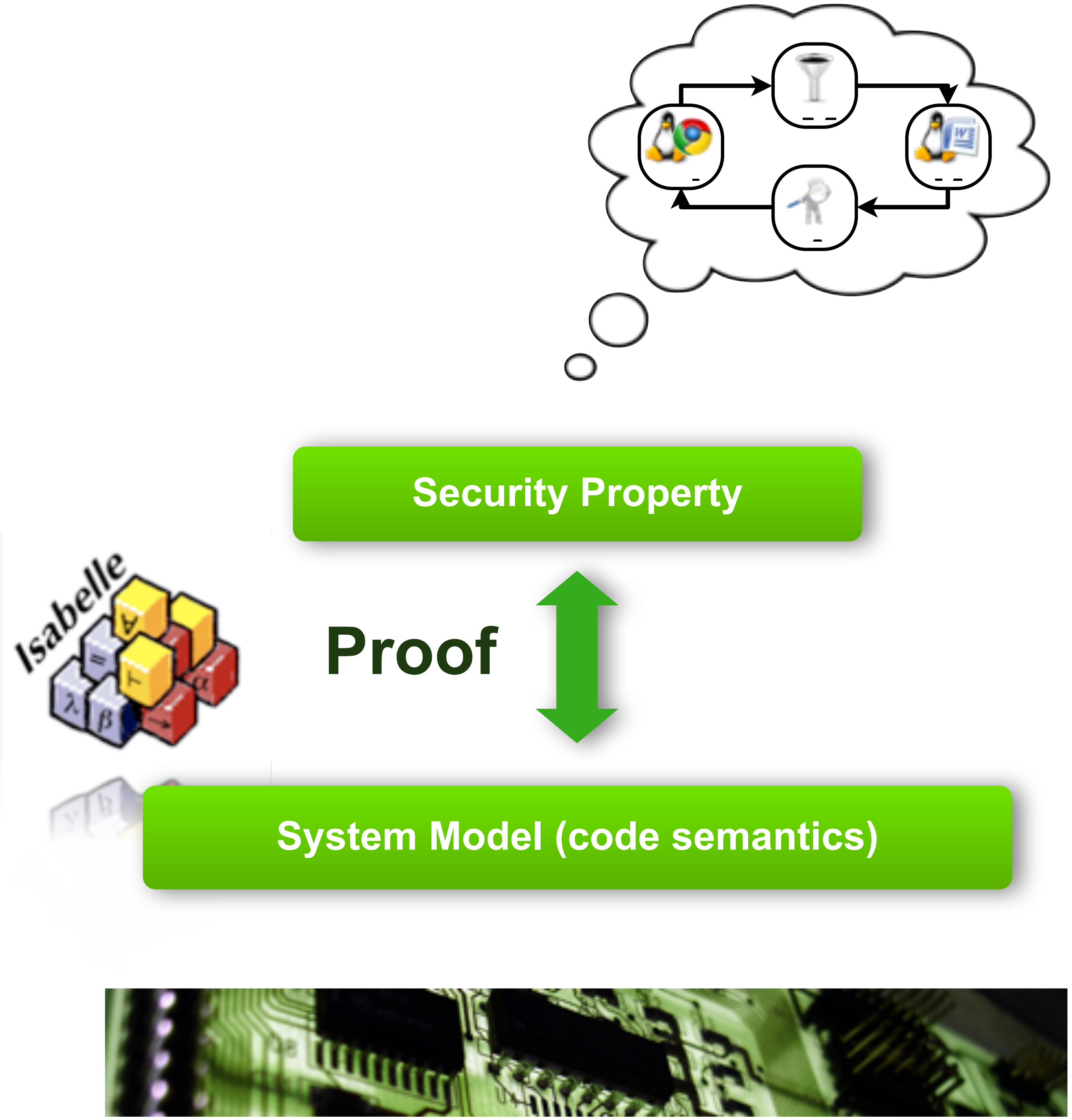
Security Property

Proof

System Model (code semantics)

Assurance

- Proofs break when:



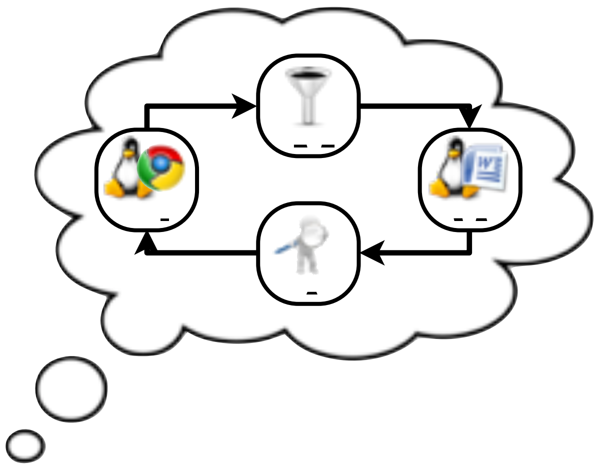
Security Property

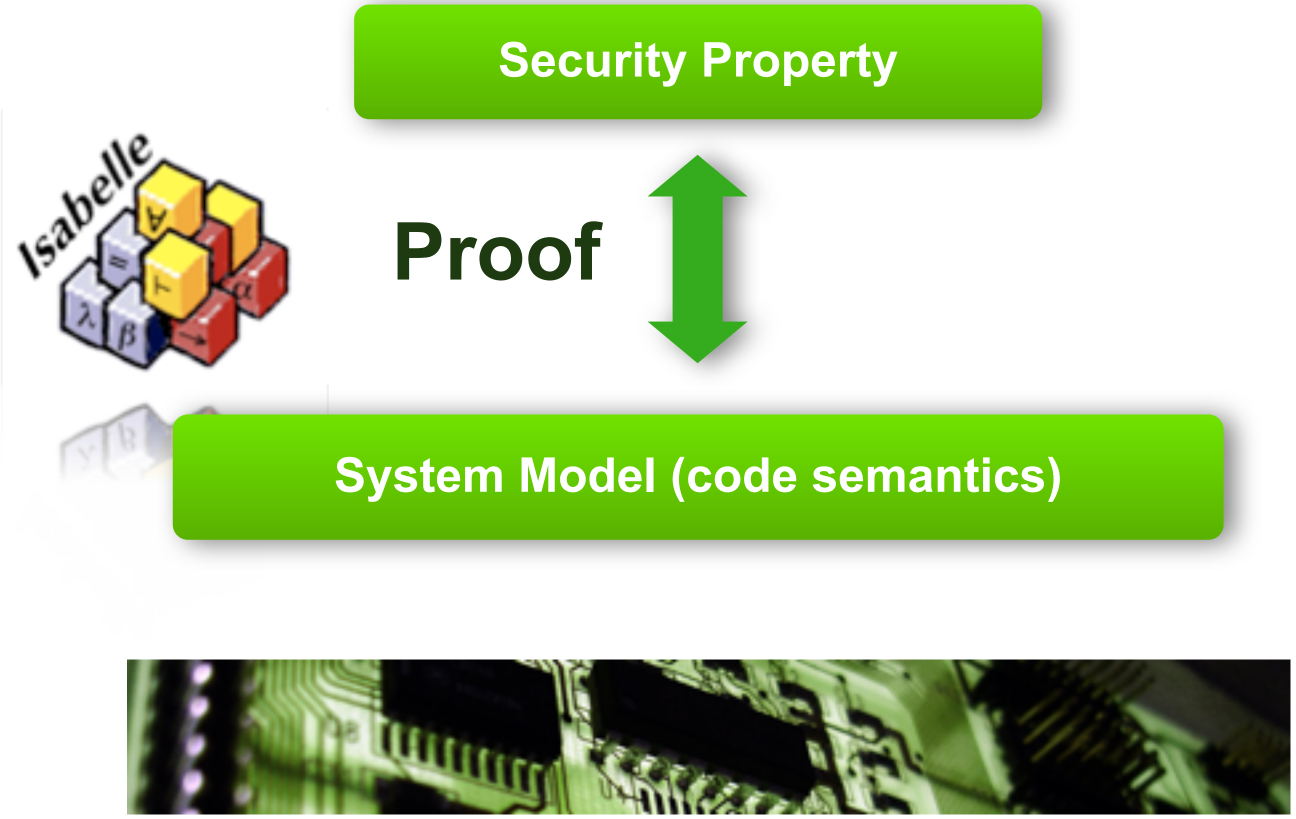
Proof

System Model (code semantics)

Assurance

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





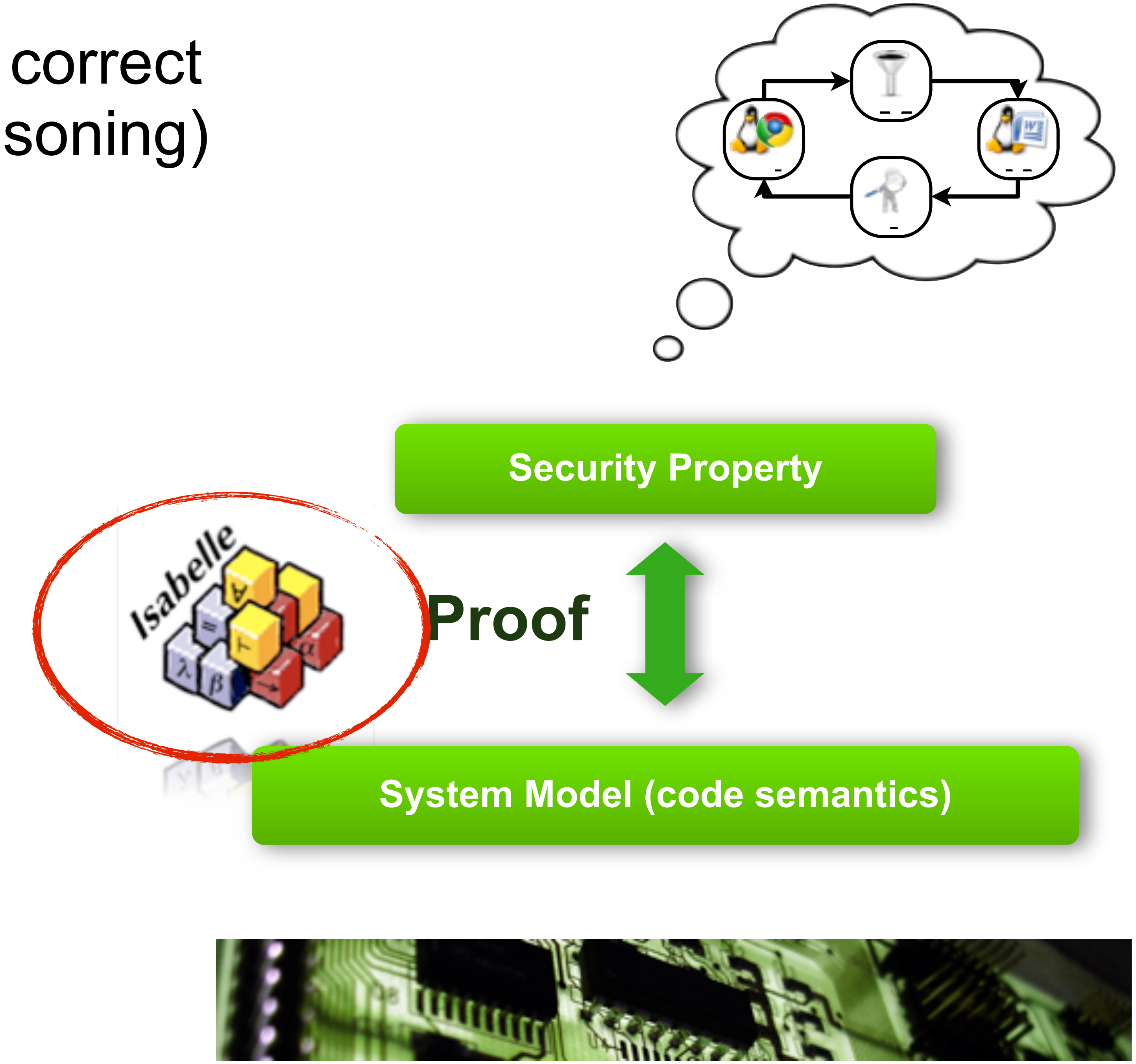
Security Property

Proof

System Model (code semantics)

Assurance

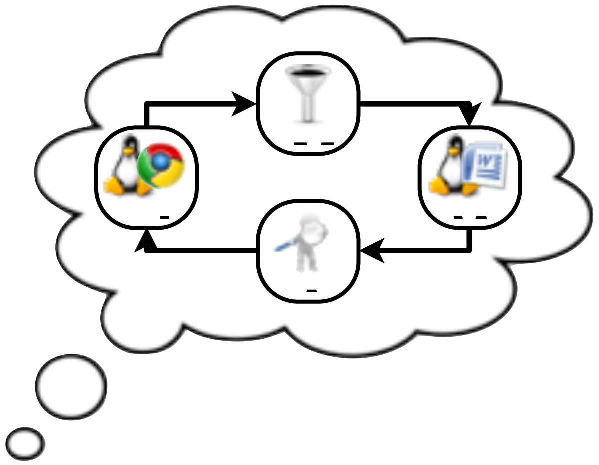
- Proofs break when:
  - they are not logically correct (involve incorrect reasoning)

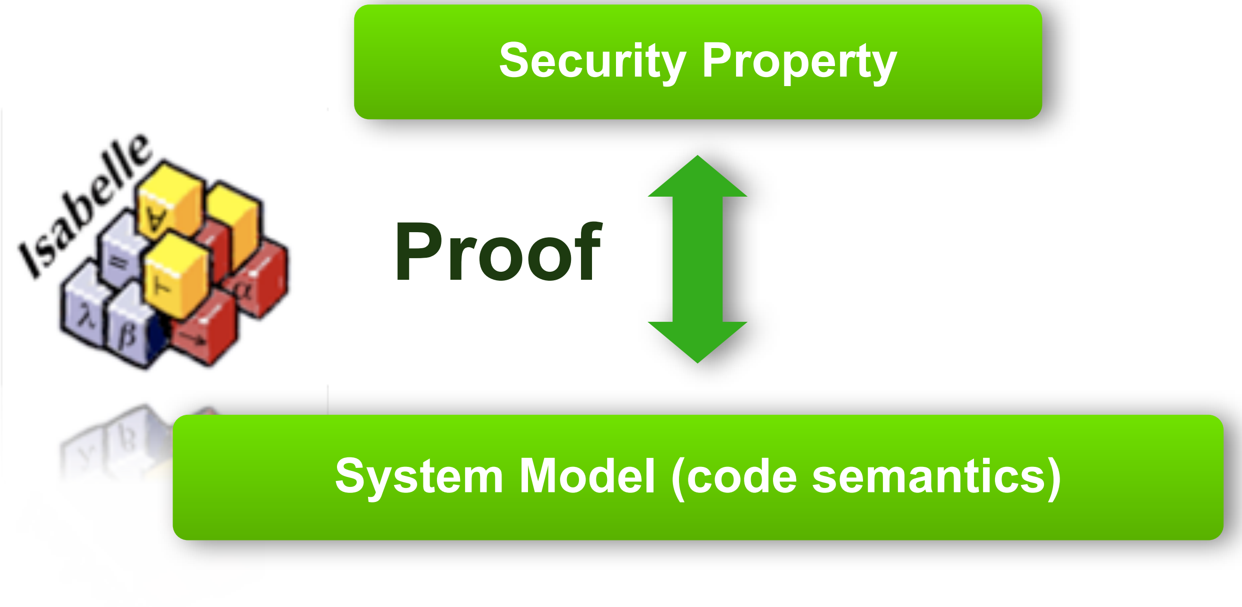


Assurance

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

a non-issue in practice



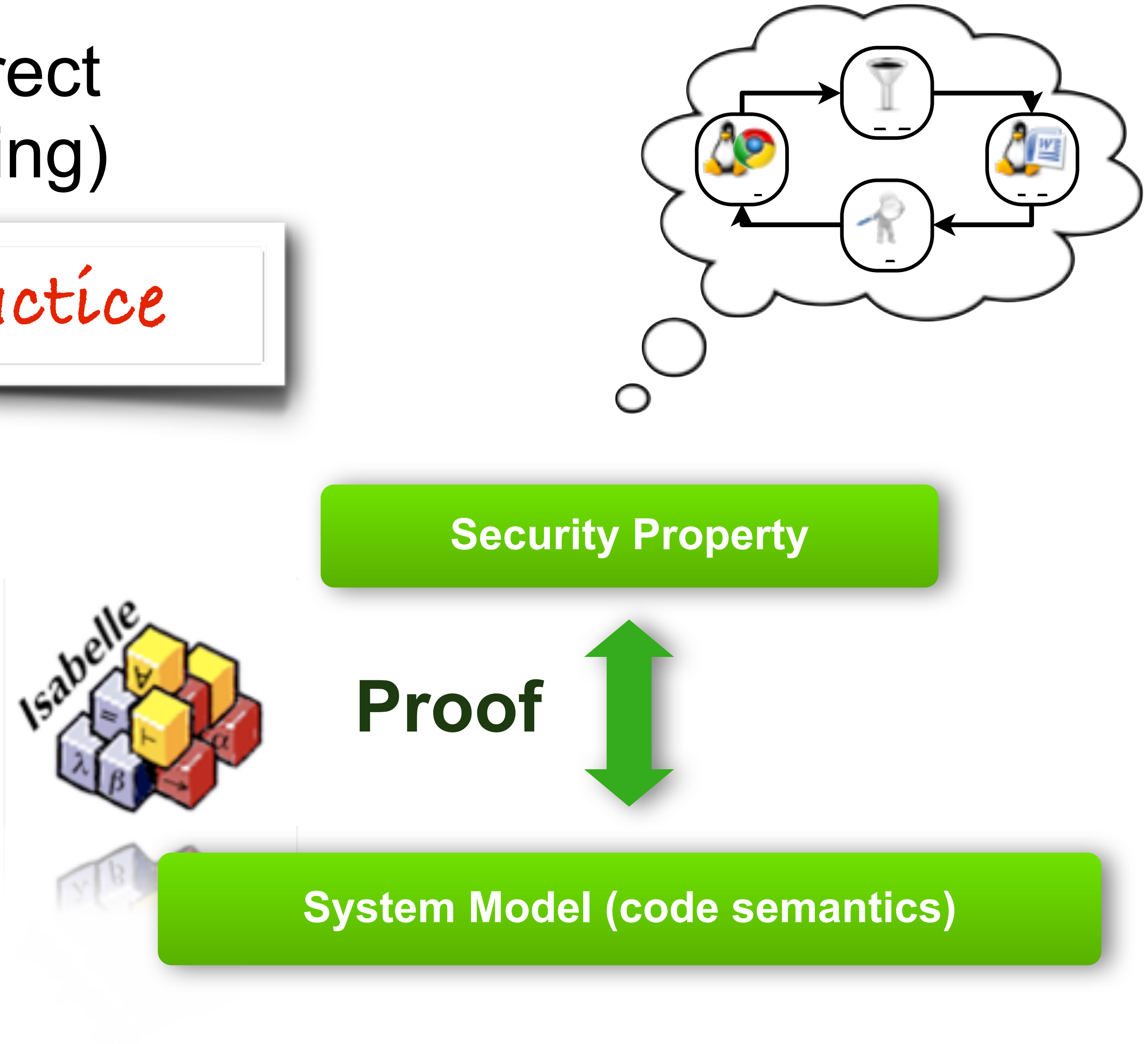


Assurance

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

a non-issue in practice

  - their assumptions are unrealistic

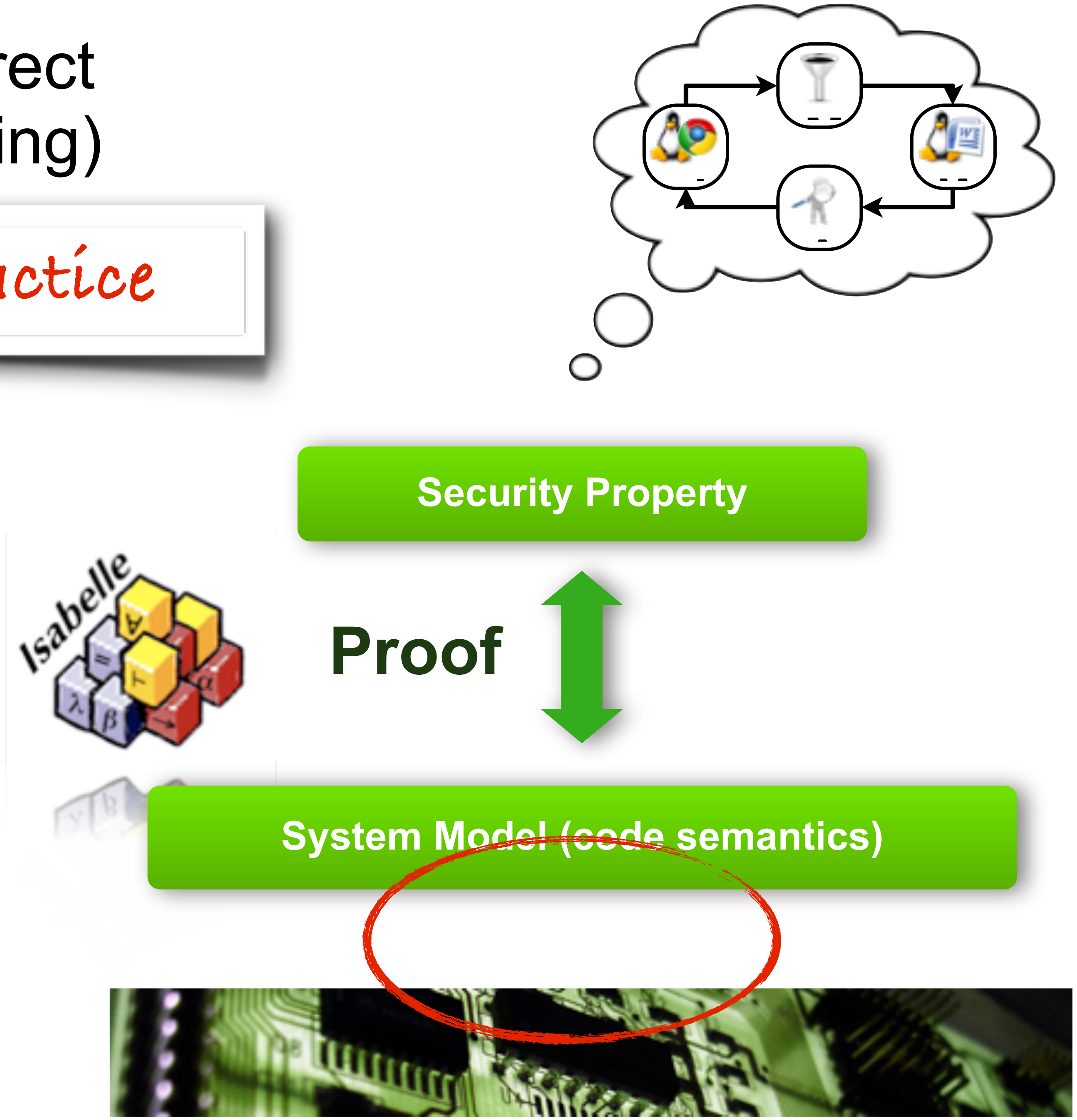


Assurance

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

a non-issue in practice

  - their assumptions are unrealistic



Security Property

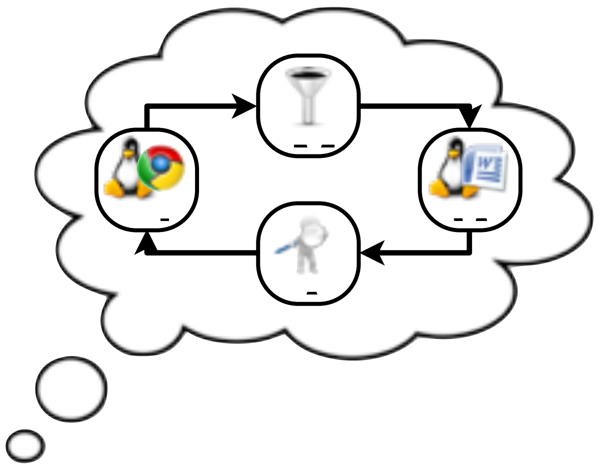
Proof

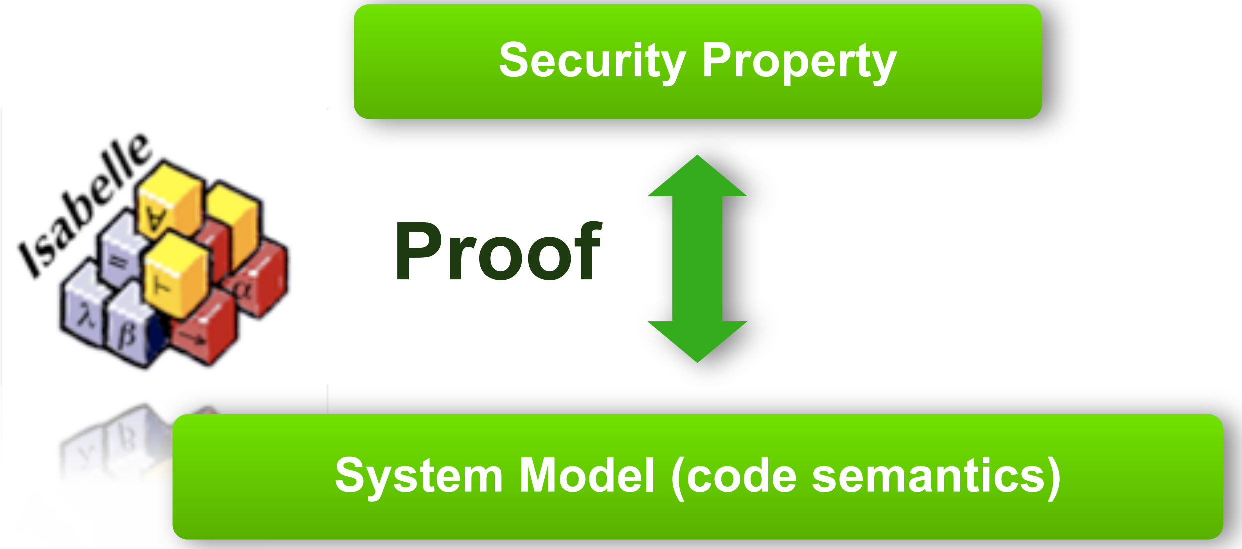
System Model (code semantics)

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



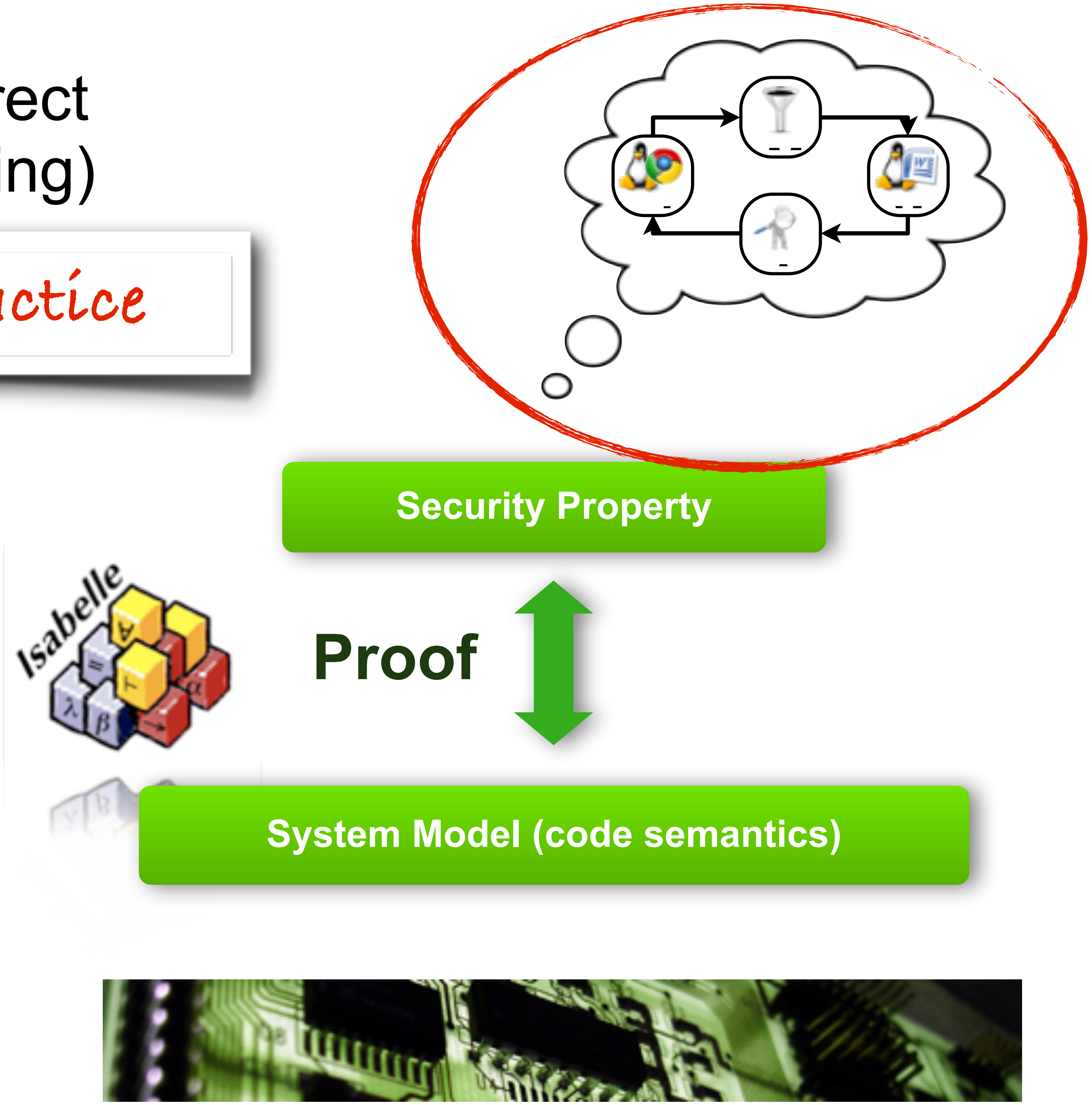


Assurance

- 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



Security Property

Proof

System Model (code semantics)

Assumptions

Assumptions

- All those of functional correctness proofs

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results

- Correct initialisation

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results
- Correct initialisation
  - system state after configuration implements access policy, and

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results

- Correct initialisation
  - system state after configuration implements access policy, and
  - meets wellformedness assumptions

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results

- Correct initialisation
  - system state after configuration implements access policy, and
  - meets wellformedness assumptions

leaky API features disabled

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results

- Correct initialisation
  - system state after configuration implements access policy, and
  - meets wellformedness assumptions

    leaky API features disabled

  - DMA disabled

Assumptions

- All those of functional correctness proofs
  - because we build on top of those results
- Correct initialisation
  - system state after configuration implements 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
  - because we build on top of those results

- Correct initialisation
  - system state after configuration implements access policy, and
  - meets wellformedness assumptions

leaky API features disabled

  - DMA disabled

- User-space h t modelled

what about covert channels?

Covert Channels

Covert Channels

- Infoflow proof says nothing about timing channels

Covert Channels

- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler

Covert Channels

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

Covert Channels

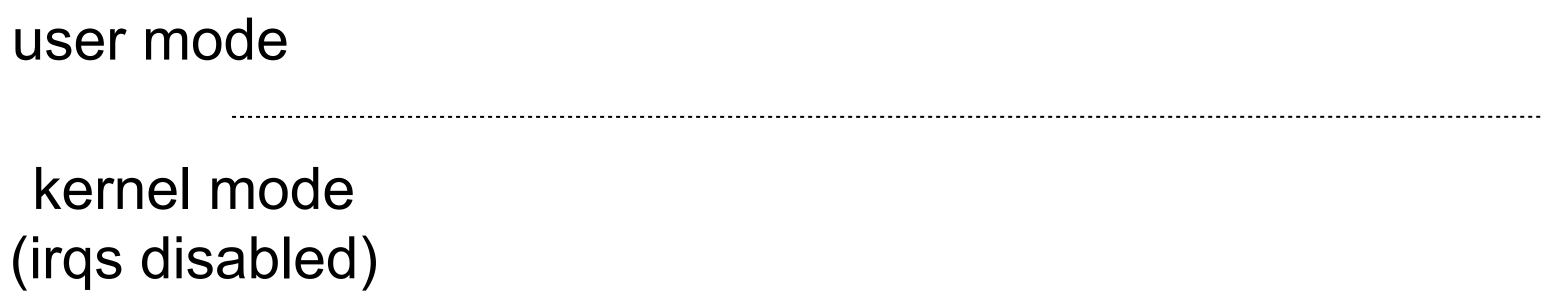
- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()

Covert Channels

- Infoflow 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

Covert Channels

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

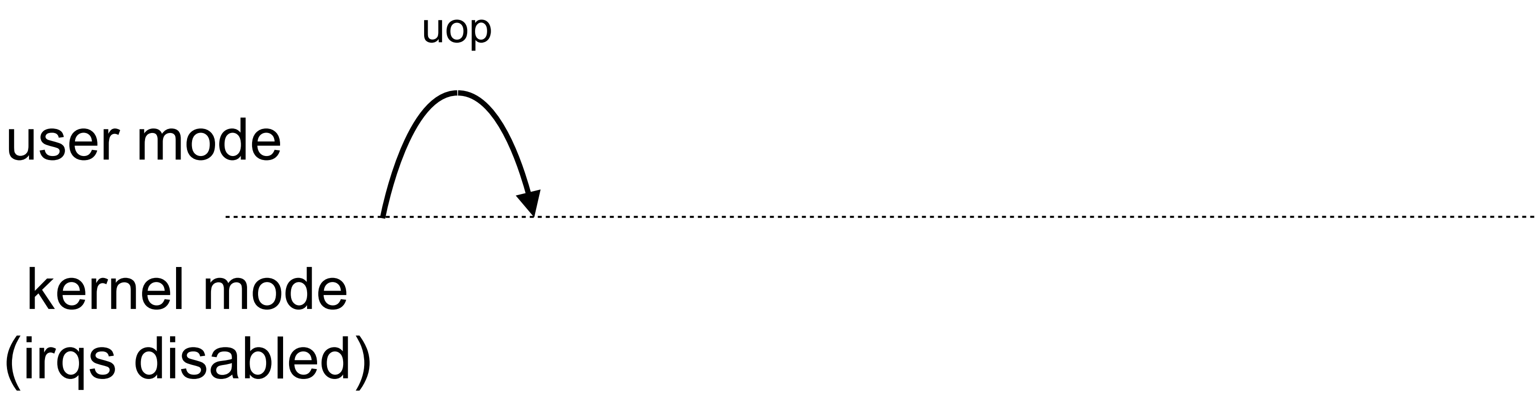


user mode

kernel mode
(irqs disabled)

Covert Channels

- Infoflow 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



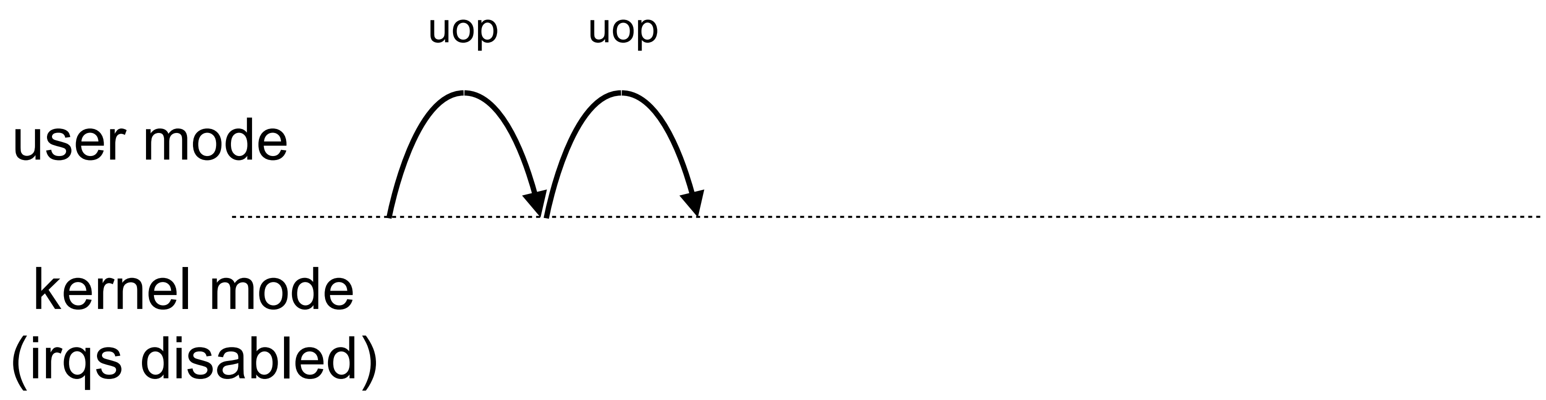
uop

user mode

kernel mode
(irqs disabled)

Covert Channels

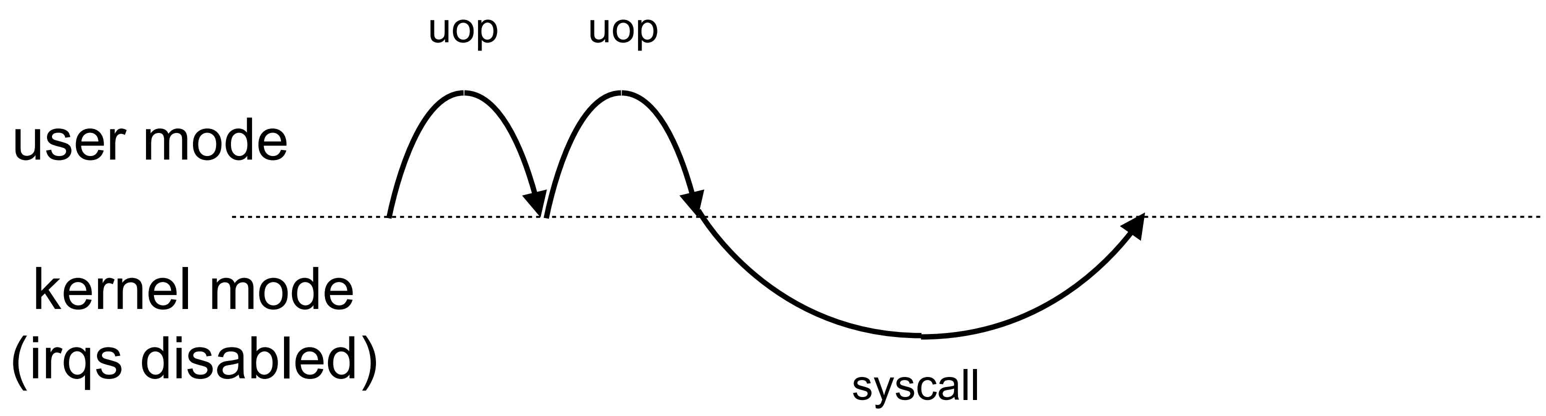
- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be **delayed** by syscall



NICTA Copyright 2010

Covert Channels

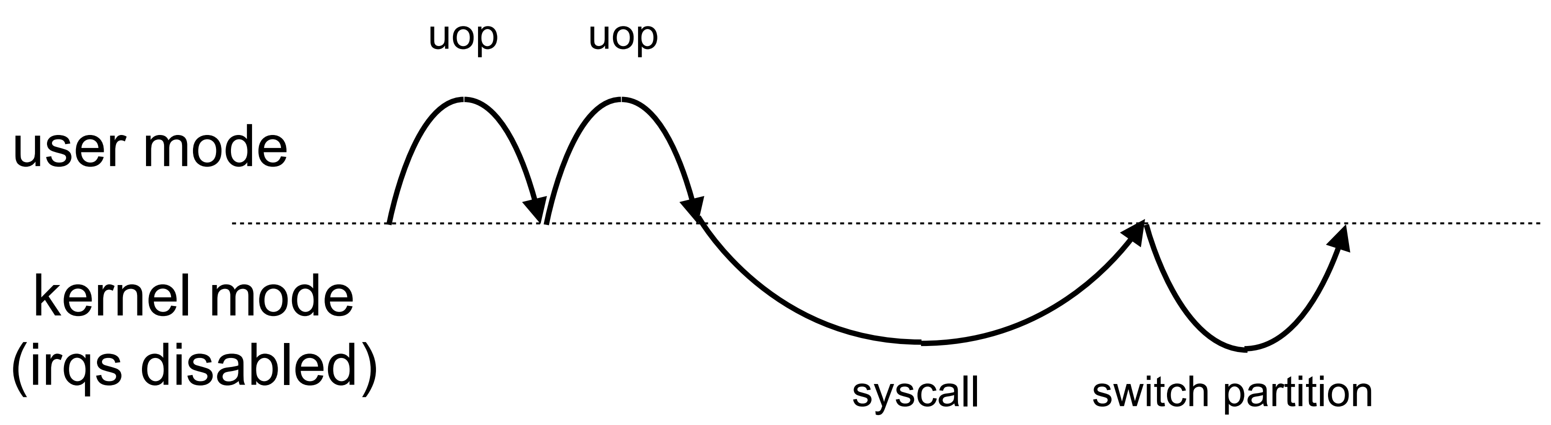
- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be **delayed** by syscall



NICTA Copyright 2010

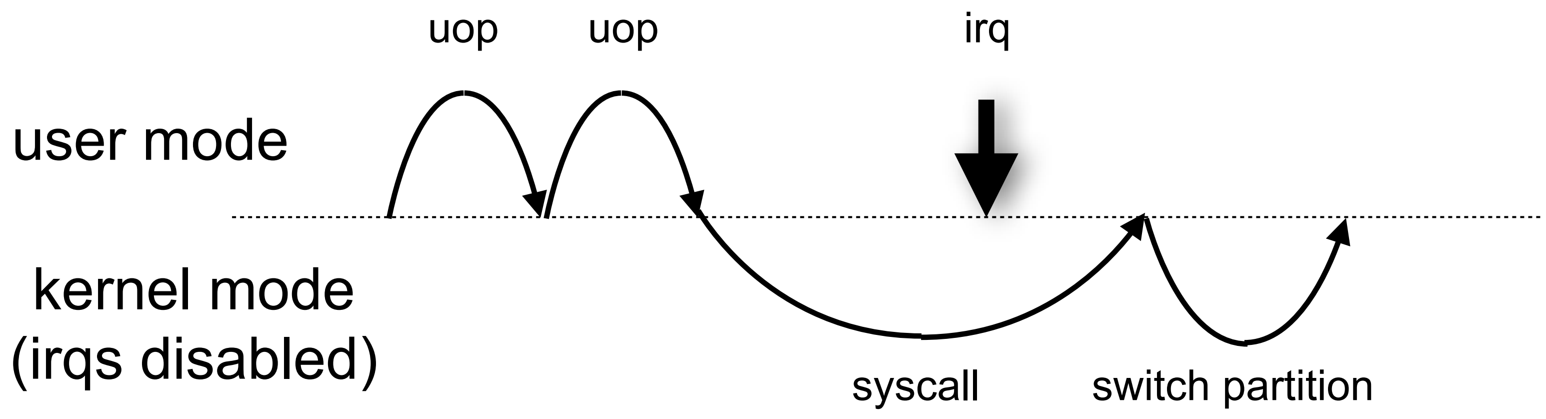
Covert Channels

- Infoflow 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



Covert Channels

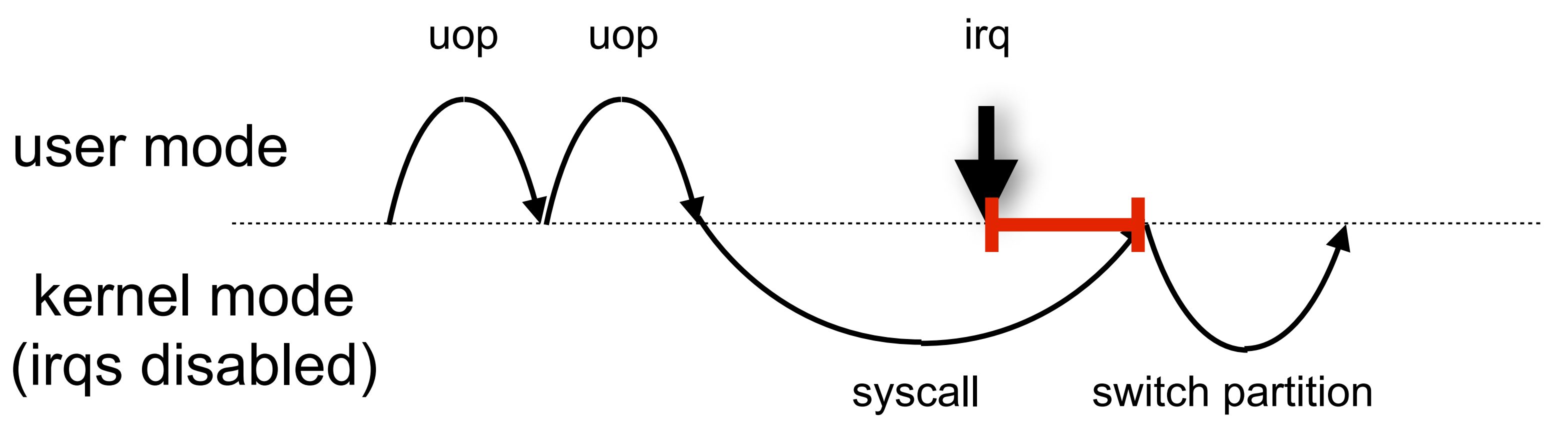
- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be **delayed** by syscall



NICTA Copyright 2010

Covert Channels

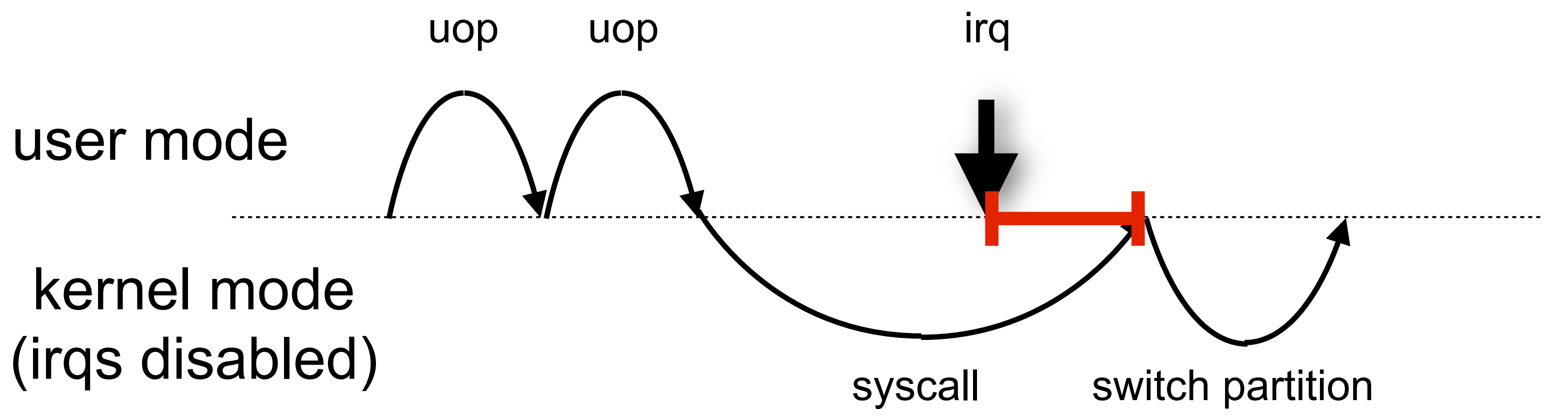
- Infoflow proof says nothing about timing channels
- e.g. jitter in scheduler
  - seL4 syscalls are generally non-preemptible
    - except at well-defined points during long-running calls e.g. Revoke()
  - partition switch can be **delayed** by syscall



NICTA Copyright 2010

Covert Channels

- Infoflow 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

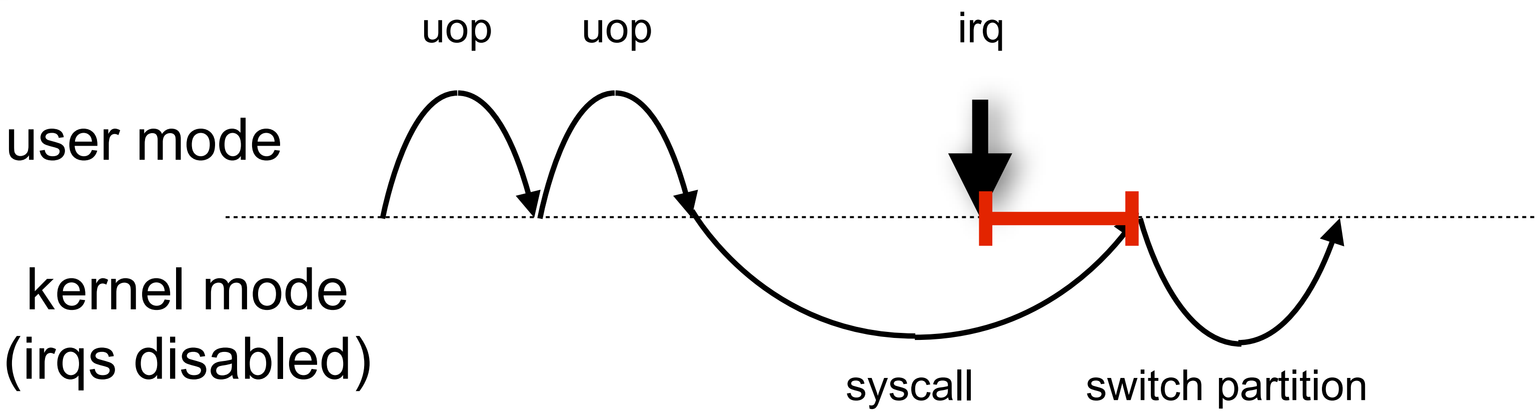


- Others: caches, CPU temp. etc.

Covert Channels

- Infoflow proof … ng channels
- e.g
  - se … ble
    - … e.g. Revoke()
  - pa … syscall

must be mitigated by complementary techniques



- Others: caches, CPU temp. etc.

Covert Channels

- Infoflow proof ng channels
- e.g
  – se ble
    - e.g. Revoke()
  – pa syscall

use

ke

(irqs disabled)

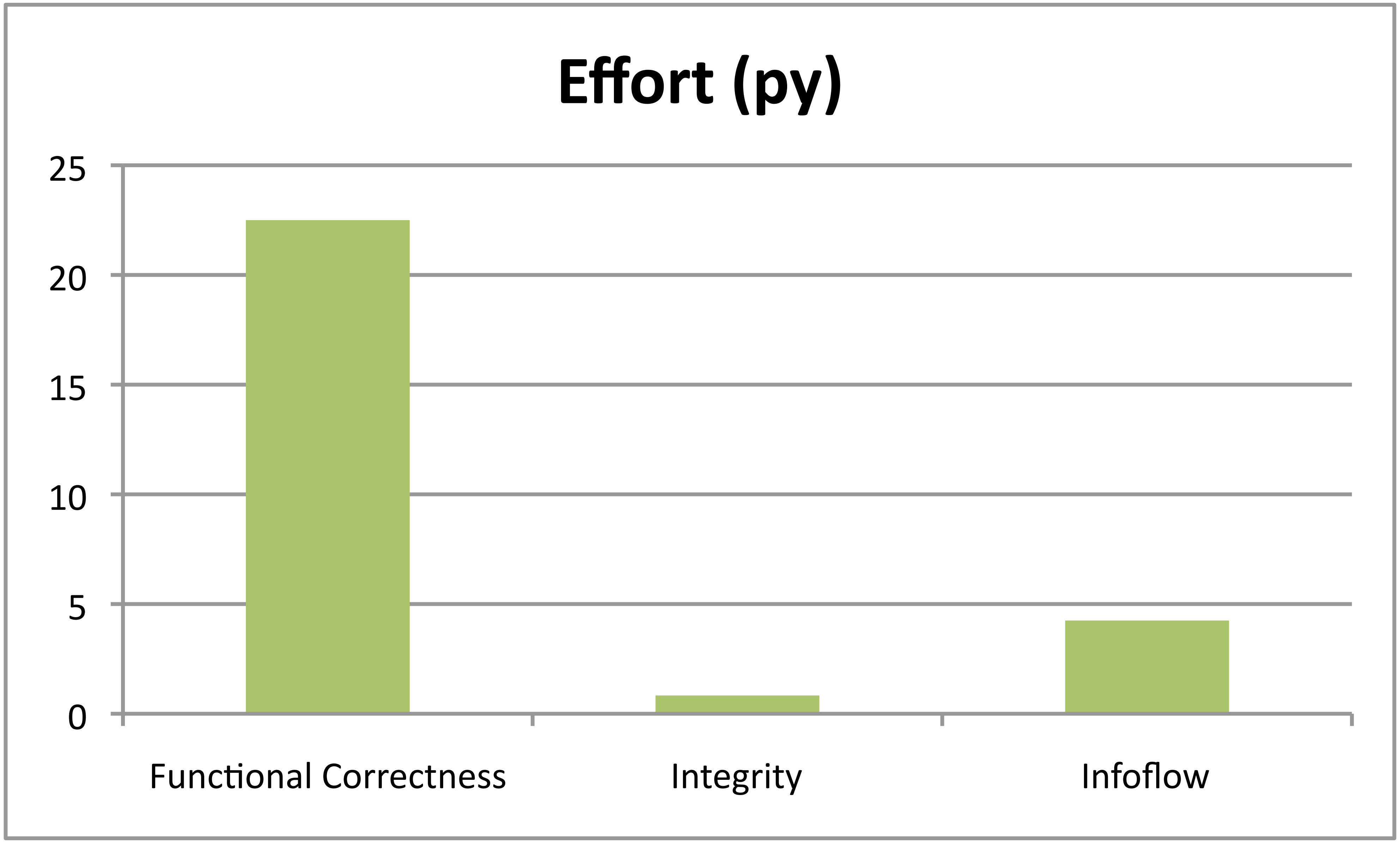
must be mitigated by complementary techniques

mitigation strategy depends on risk profile of deployment

- Others: caches, CPU temp. etc.

Lesson

- Functional correctness enables cheap security proofs



NICTA Copyright 2010

Security Proofs for seL4: Summary

Security Proofs for seL4: Summary

strongest such results ever for a general-purpose kernel

Security Proofs for seL4: Summary

strongest such results ever for a general-purpose kernel

security proofs of operating system kernels are practical.

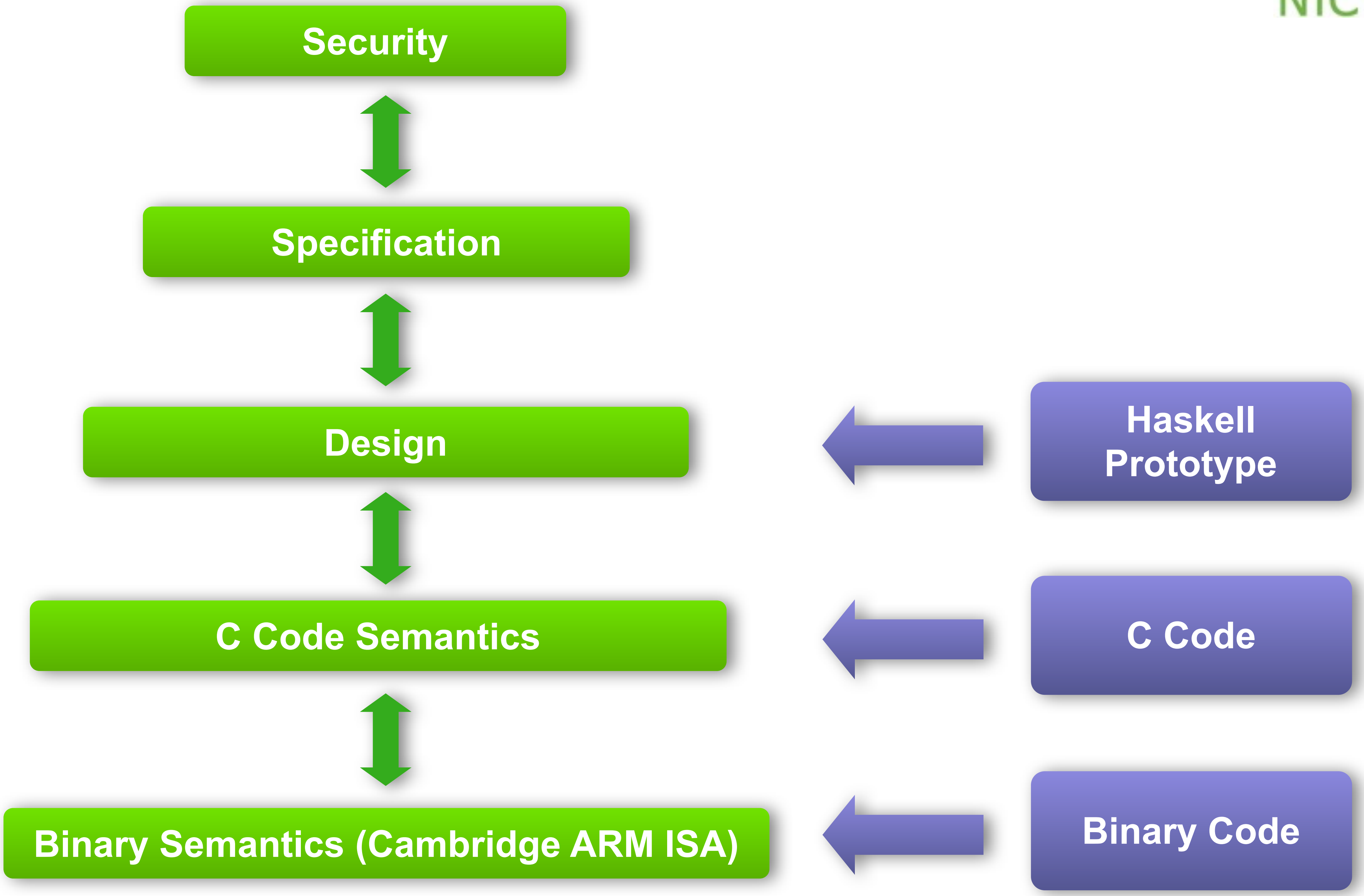
Security Proofs for seL4: Summary

strongest such results ever for a general-purpose kernel

security proofs of operating system kernels are practical.

demand nothing less.

seL4 Verification Stack



Security

Specification

Design

C Code Semantics

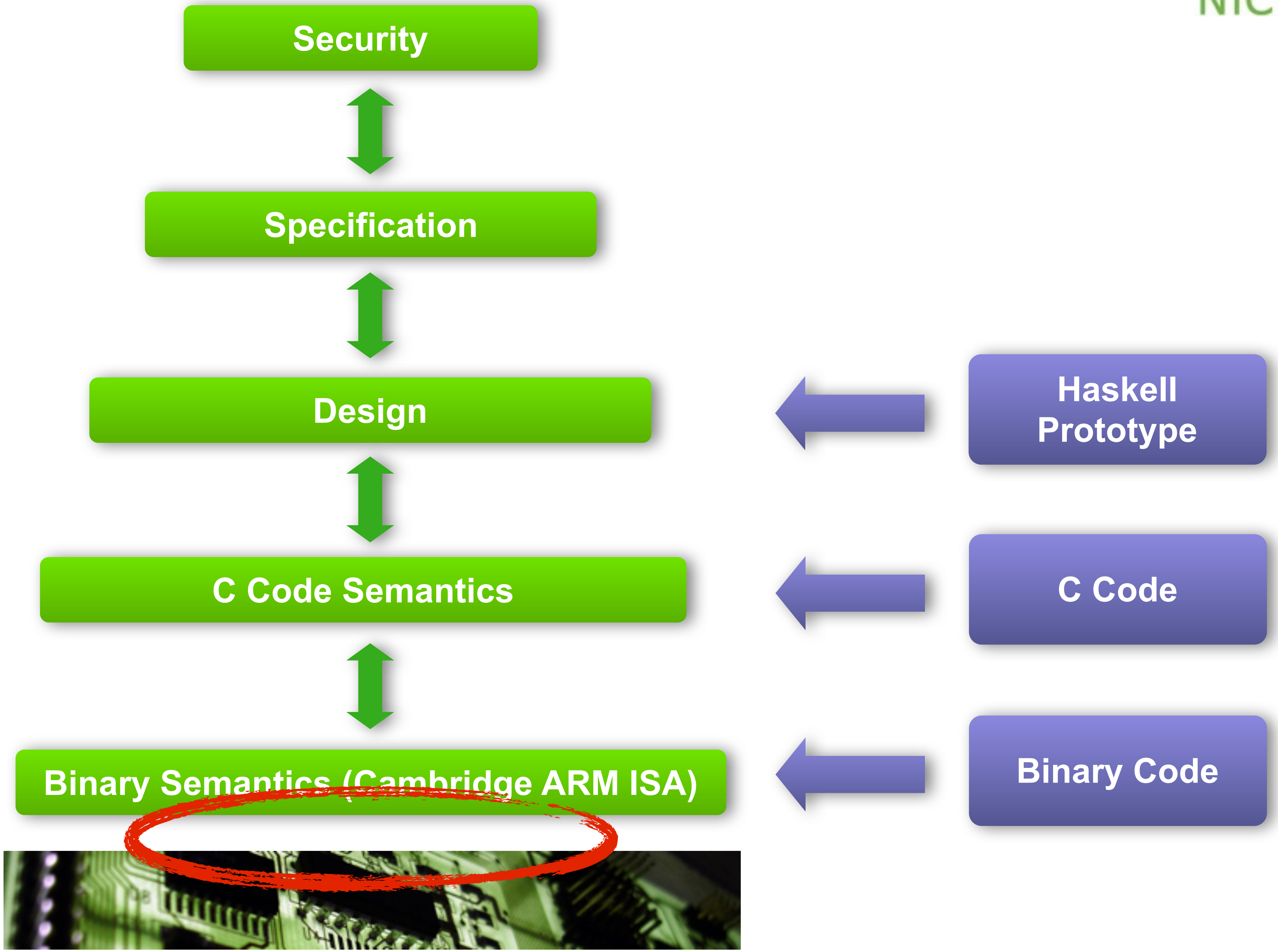
Binary Semantics (Cambridge ARM ISA)

Haskell Prototype

C Code

Binary Code

seL4 Verification Stack



Security

Specification

Design

C Code Semantics

Binary Semantics (Cambridge ARM ISA)

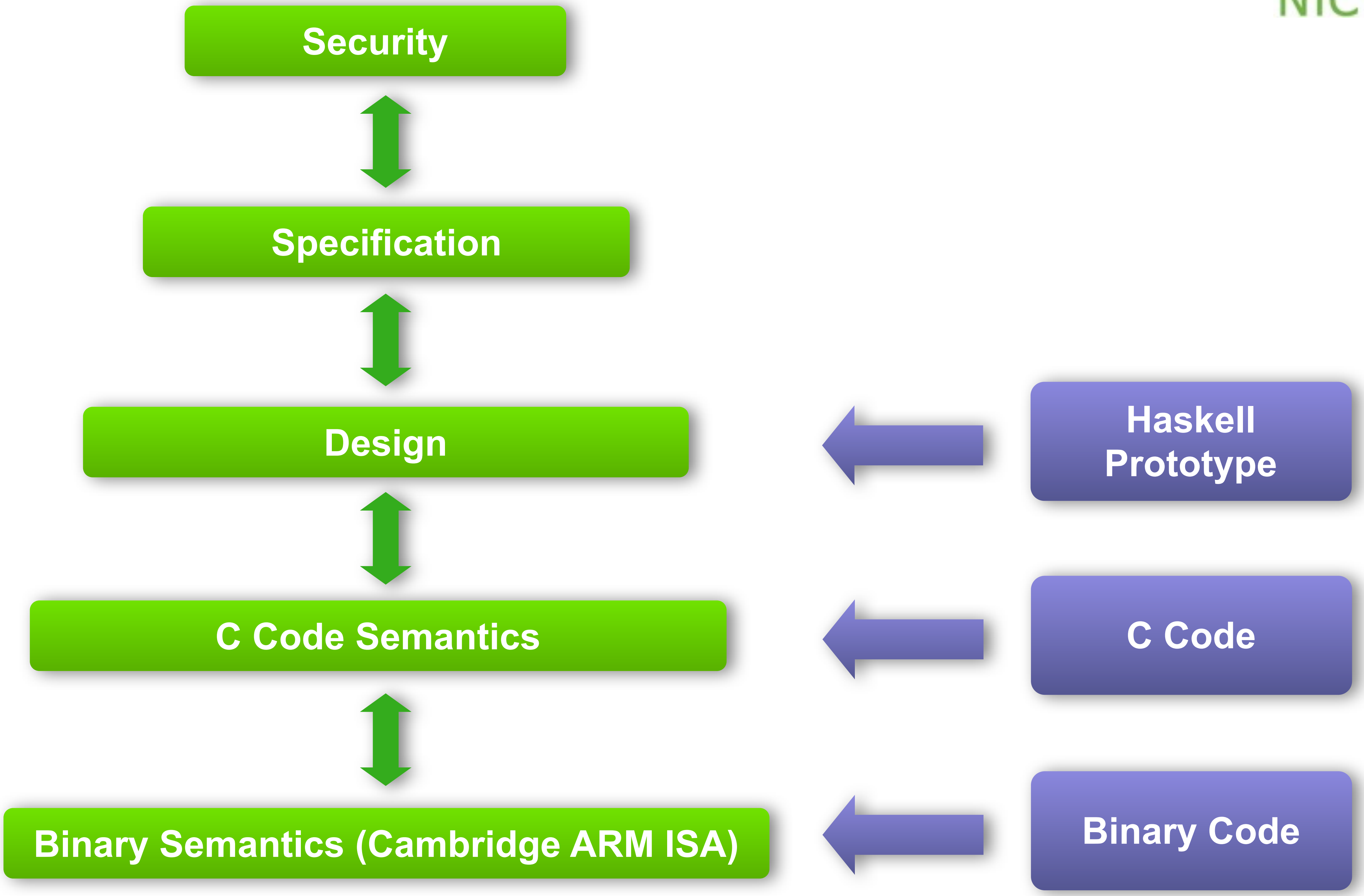
Haskell Prototype

C Code

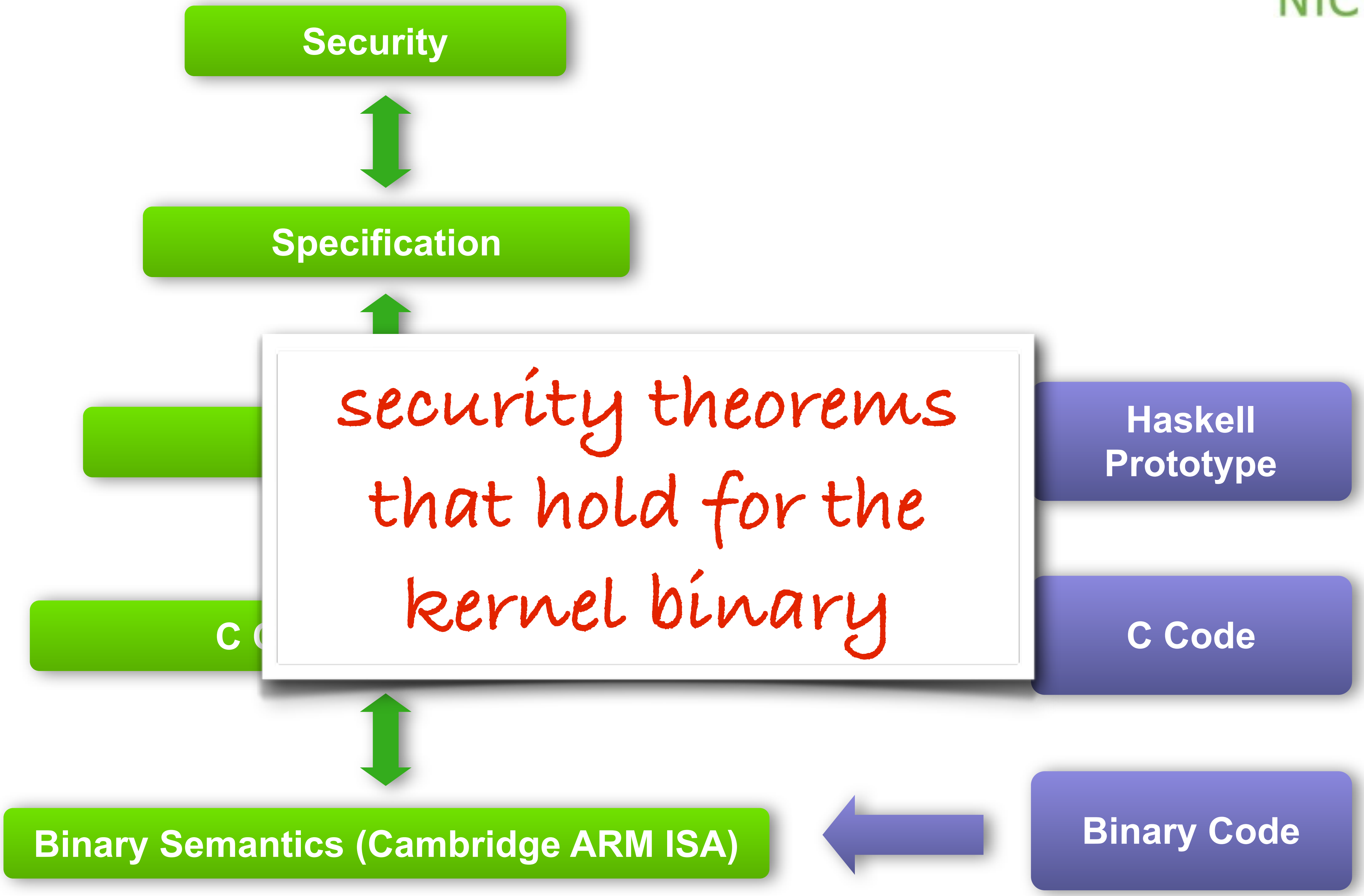
Binary Code

NICTA Copyright 2013

seL4 Verification Stack



seL4 Verification Stack



Security

Specification

C C

Haskell Prototype

C Code

Binary Semantics (Cambridge ARM ISA)

Binary Code

security theorems that hold for the kernel binary