Nickel

A Framework for Design and Verification of Information Flow Control Systems

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang

UNIVERSITY of WASHINGTON

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

UNSAT.org

Enforcing information flow control is critical

U.S. » Crime + Justice | Energy + Environment | Extreme Weather | Space + Science   Live TV   U.S. Edition +

FBI: Hacker claimed to have taken over flight's engine controls

By **Evan Perez**, CNN
Updated 9:19 PM ET, Mon May 18, 2015

Man claims entertainment system helped him hack plane 02:09

The Washington Post
Democracy Dies in Darkness

Morning Mix

Hacker Chris Roberts told FBI he took control of United plane, FBI claims

By **Justin Wm. Moyer**
May 18, 2015

Covert channels through error codes

Eddie Kohler @xexd · Aug 8

I spent many years after Asbestos/HiStar down on information flow, because it makes things too hard to program for too little gain. Still think that! But this keeps happening.

noreply@hotcrp.com 2:35 AM (6 hours ago)

to me

2018/08/08 06:30:07 h.asplos19: bad doc 403 Forbidden You aren't allowed to view submission #500. [] @/asplos19-paper500.pdf xxx@stanford.edu
2018/08/08 06:30:13 h.asplos19: bad doc 403 Forbidden You aren't allowed to view submission #600. [] @/asplos19-paper600.pdf xxx@stanford.edu
2018/08/08 06:30:18 h.asplos19: bad doc 403 Forbidden You aren't allowed to view submission #1000. [] @/asplos19-paper1000.pdf xxx@stanford.edu
2018/08/08 06:30:24 h.asplos19: bad doc 403 Forbidden You aren't allowed to view submission #10000. [] @/asplos19-paper10000.pdf xxx@stanford.edu

4 11

Eliminating unintended flows is difficult

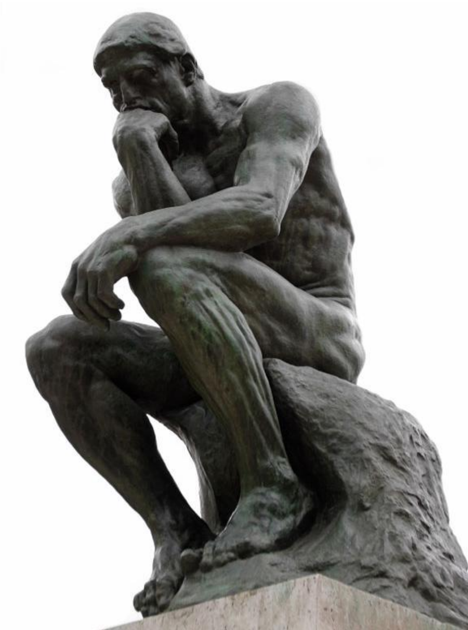
- **Covert channels:** A channel not intended for information flow [Lampson '73]

- Covert channels are often inherent in **interface design**

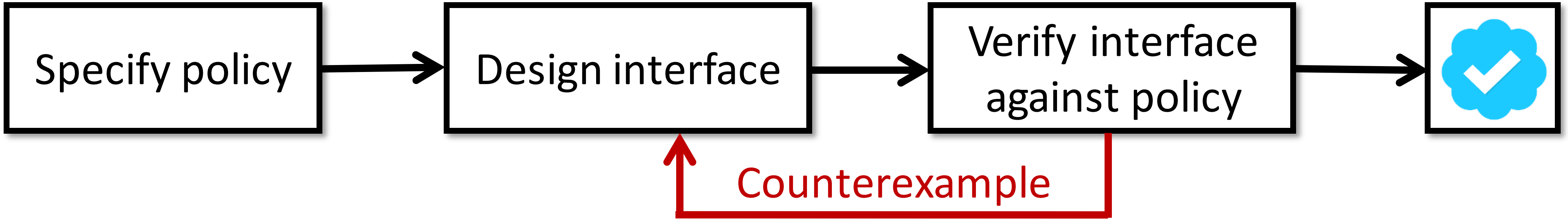
- Examples of covert channels in interfaces:
  - ARINC 653 avionics standard [TACAS '16]
  - Floating labels in Asbestos [Oakland '09, OSDI '06]

Eliminating unintended flows is difficult

- **Covert channels:** A channel not intended for information flow [Lampson ‘73]
- Covert channels are often inherent in **interface design**
- Examples of covert channels in interfaces:
  - ARINC 653 avionics standard [TACAS ‘16]
  - Floating labels in Asbestos [Oakland ‘09, OSDI ‘06]



Our approach: Verification-driven interface design



- Extends prior work of push-button verification:
  - Yggdrasil [OSDI '16] & Hyperkernel [SOSP '17]

- Limitations
  - Finite interface, expressible using SMT.
  - Hardware-based side channels not in scope and no concurrency.

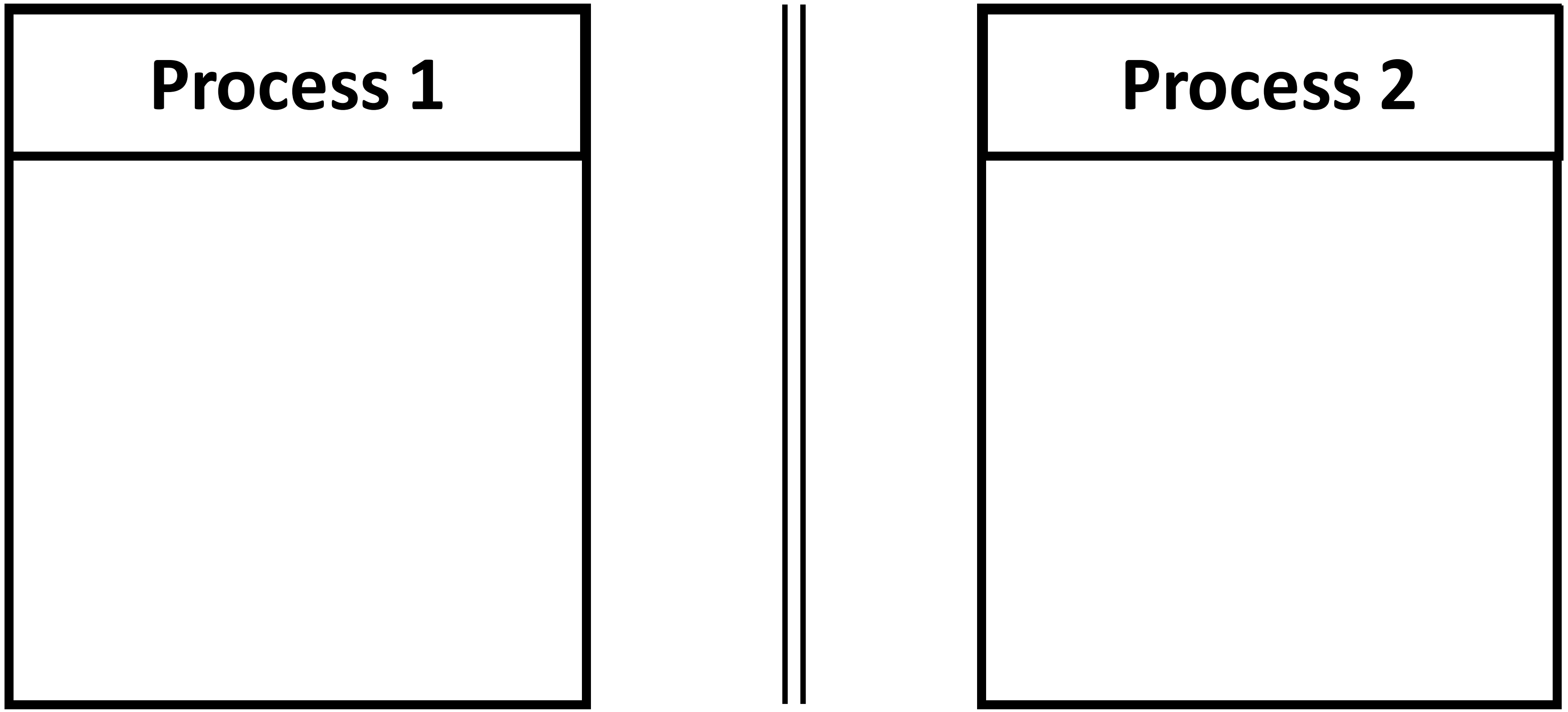
Contributions

- New formulation and proof strategy for **noninterference**
- **Nickel**: A framework for design and verification of information flow control (IFC) systems
- Experience building three systems using Nickel
  - First formally verified decentralized IFC OS kernel
  - Low proof burden: order of weeks

Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

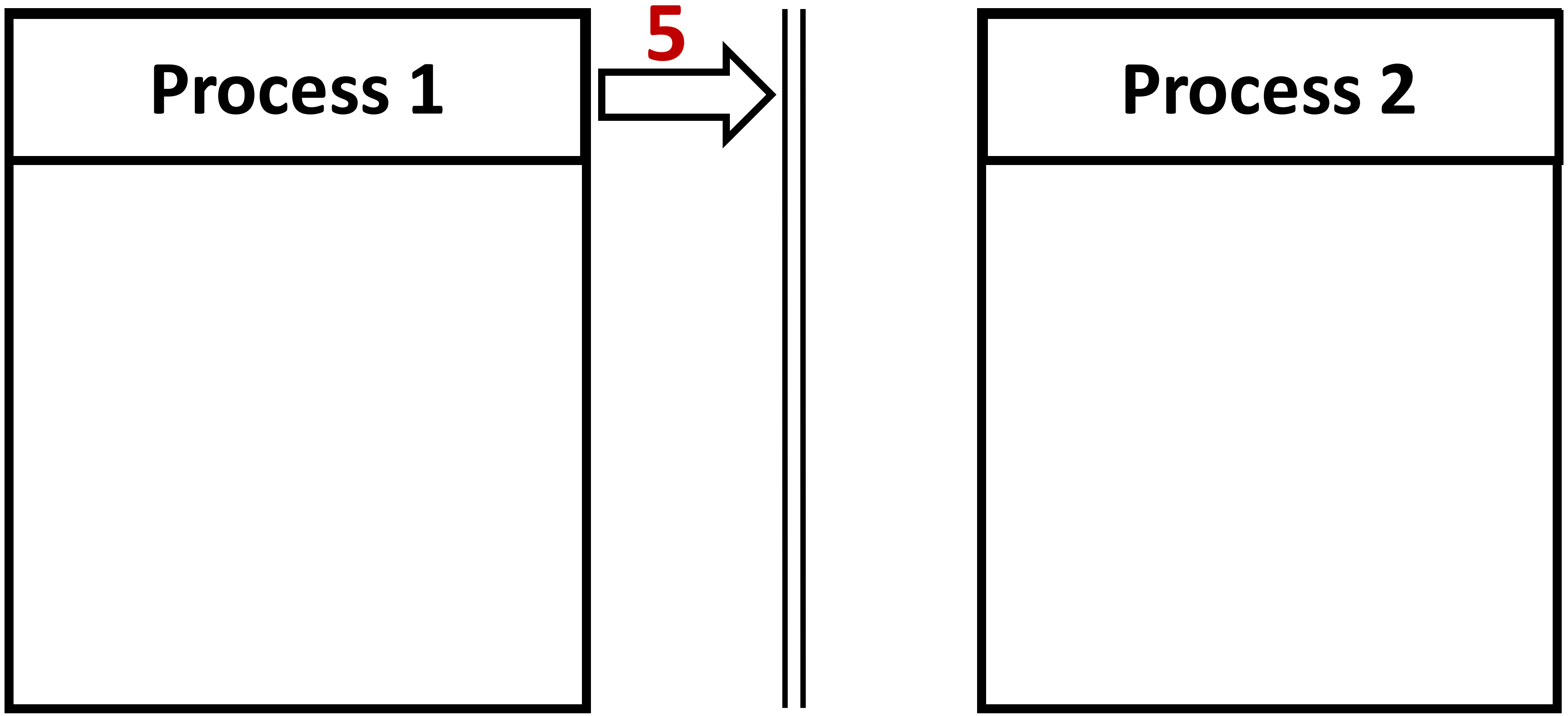
Design: Spawn with sequential PID allocation



Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

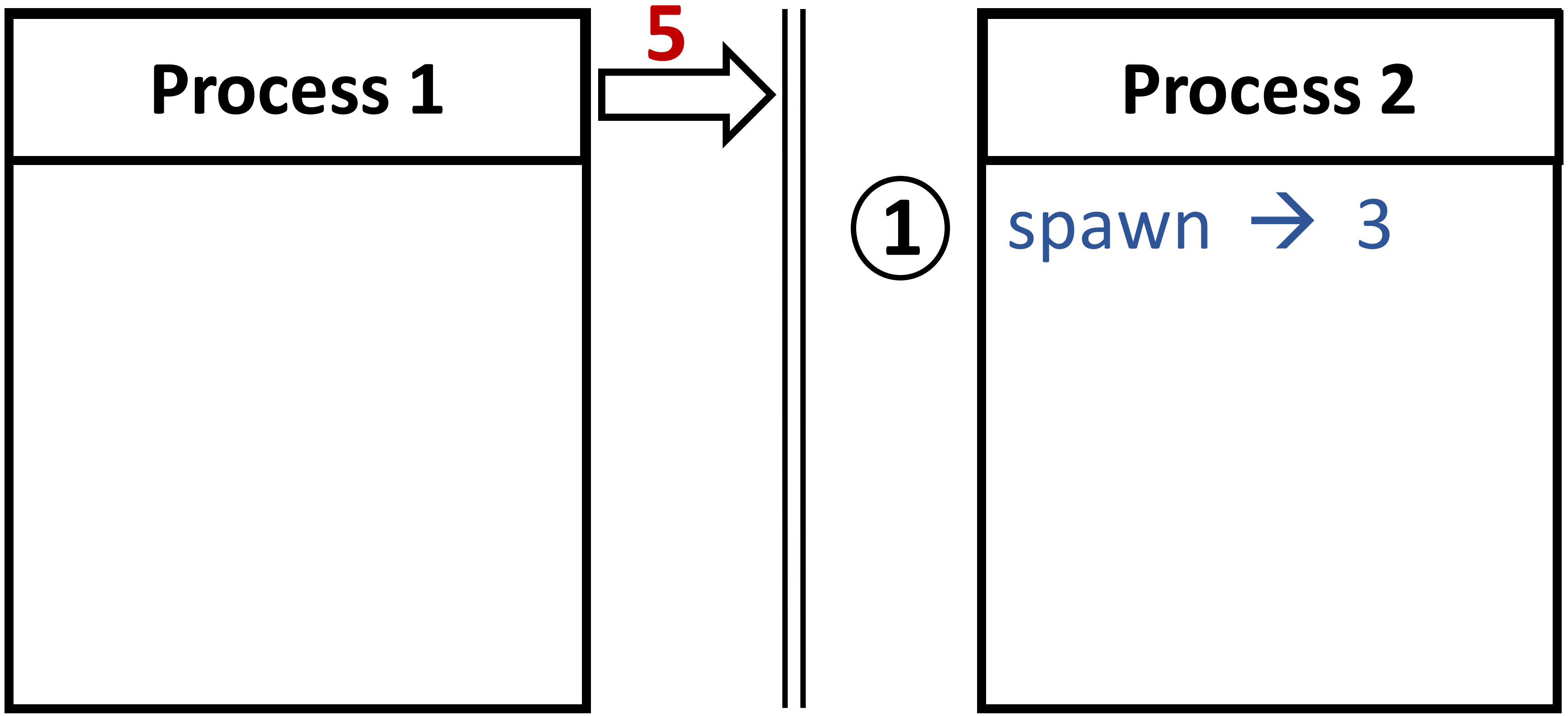
Design: Spawn with sequential PID allocation



Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

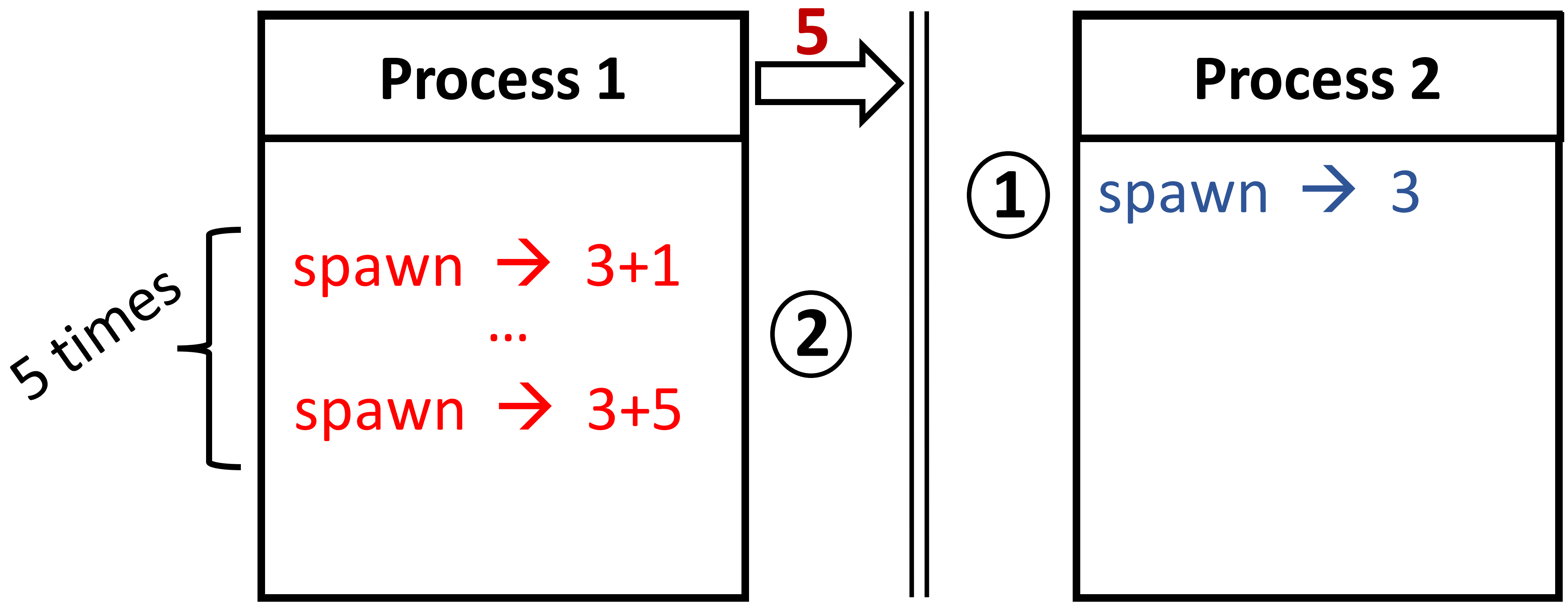
Design: Spawn with sequential PID allocation



Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

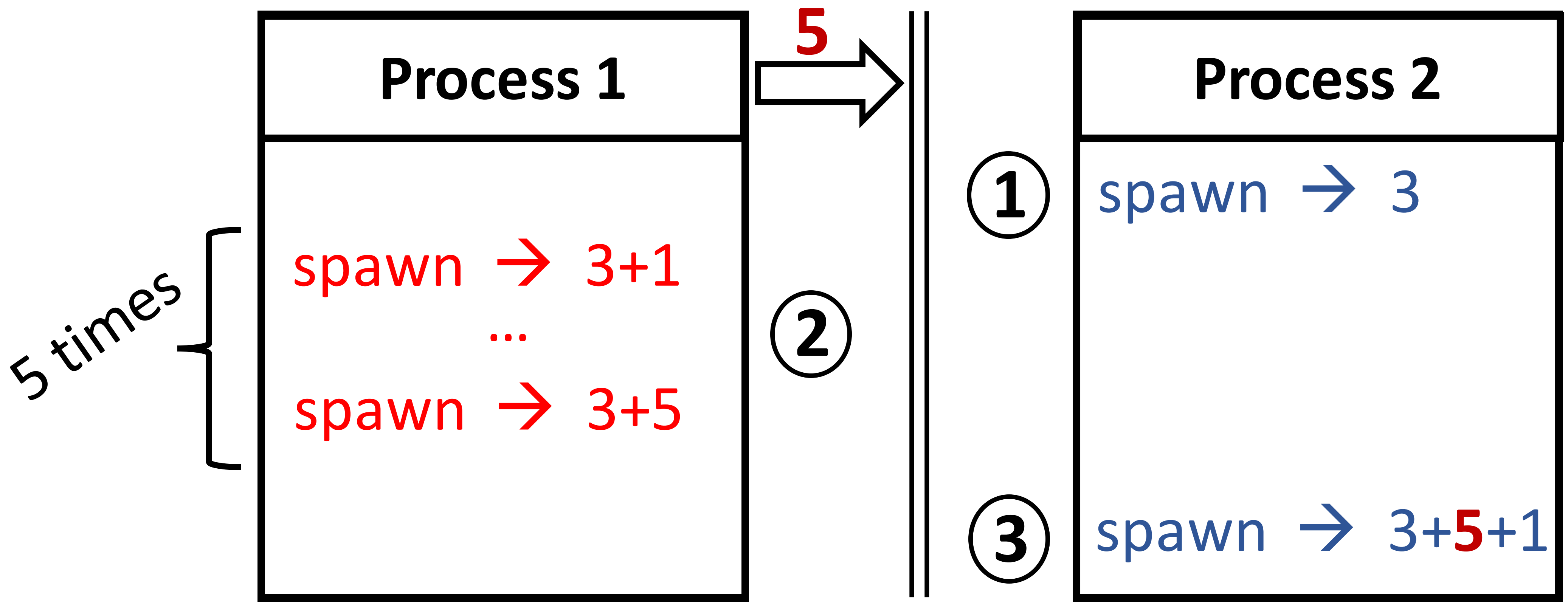
Design: Spawn with sequential PID allocation



Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

Design: Spawn with sequential PID allocation

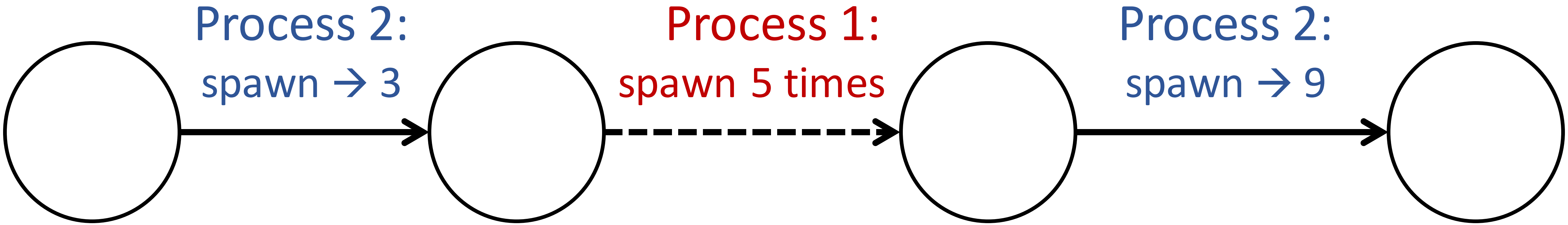


Covert channel in resource names

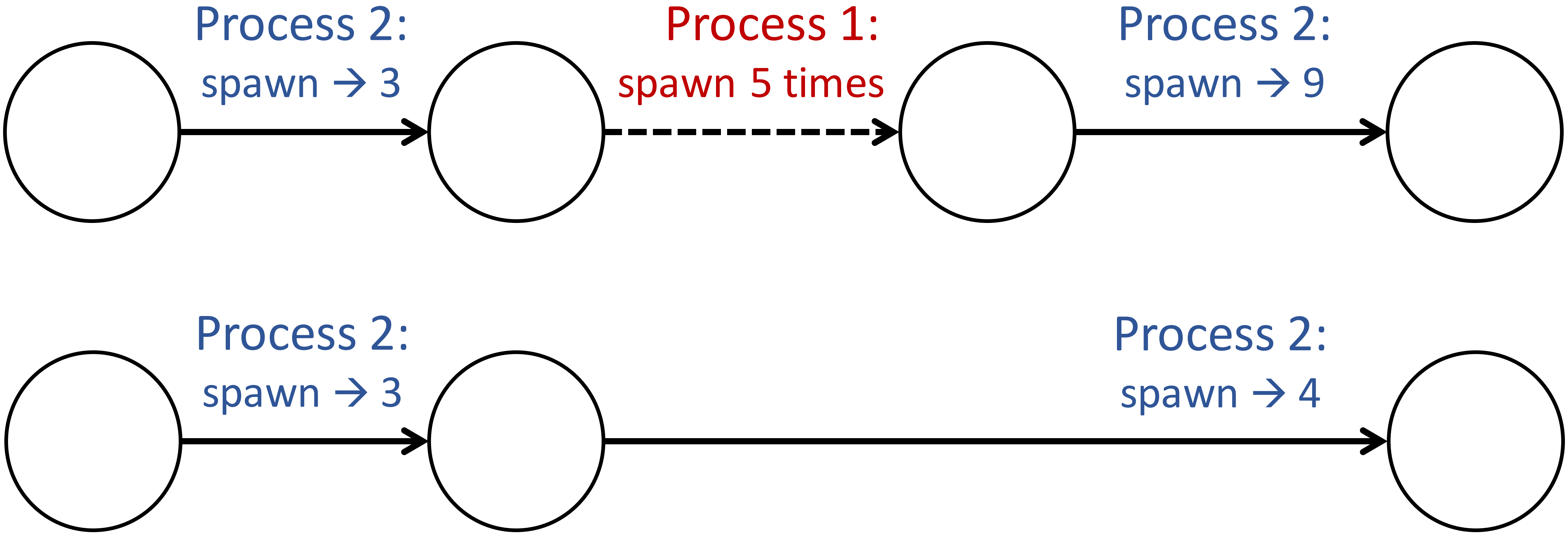
Examples of covert channels

- **Resource names**
- Resource exhaustion
- Statistical information
- Error handling
- Scheduling
- Devices and services

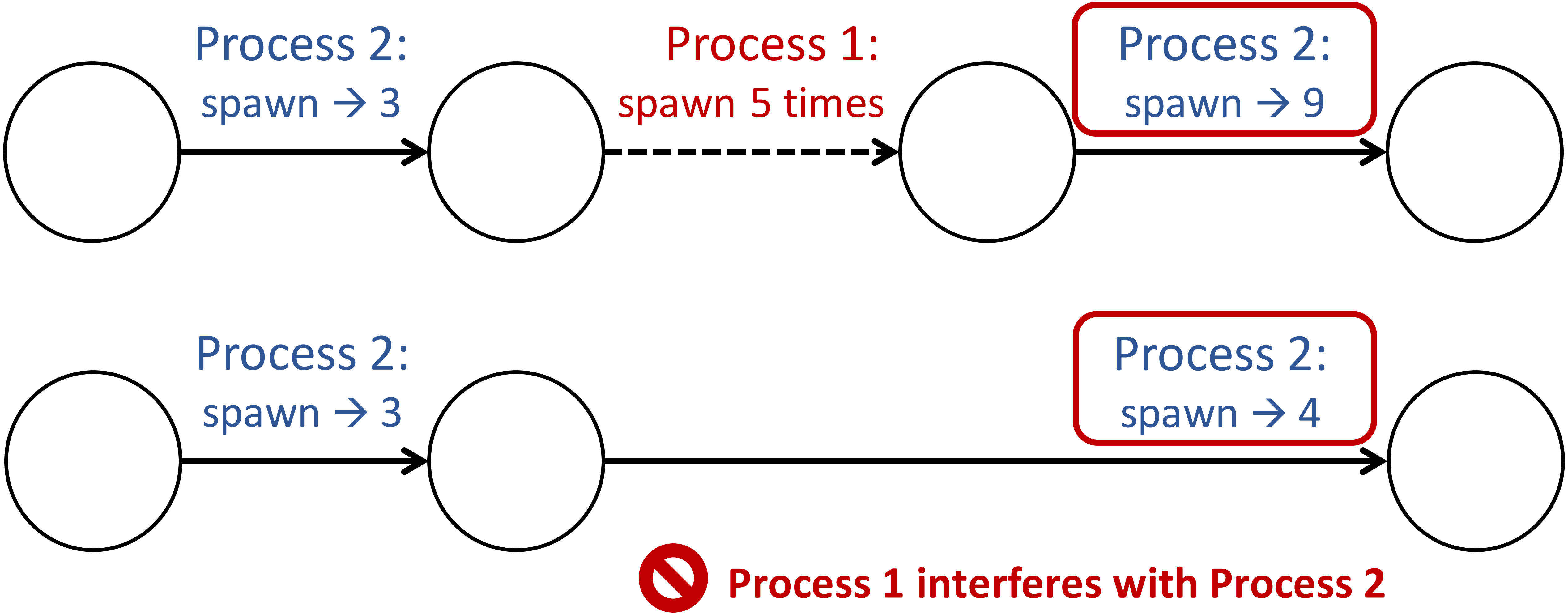
Noninterference intuition



Noninterference intuition

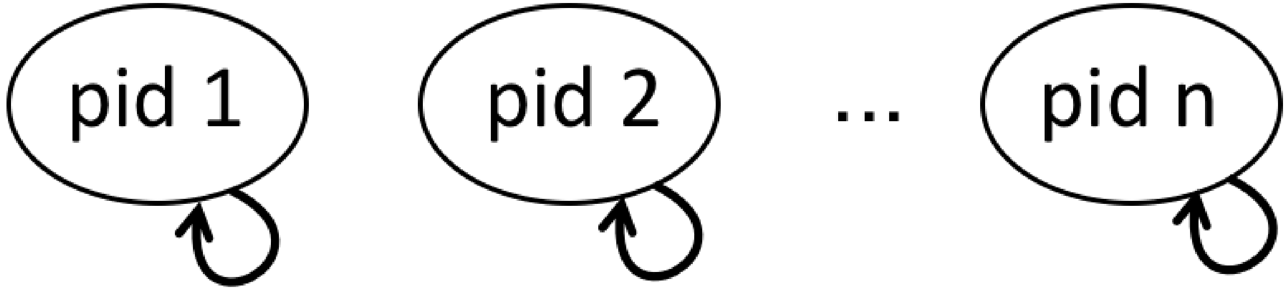


Noninterference intuition



Information flow policies in Nickel

- Set of domains $\mathcal{D}$: e.g., processes
- Can-flow-to relation $\rightsquigarrow \subseteq (\mathcal{D} \times \mathcal{D})$: permitted flow between domains
- Function $\mathrm{dom}: (A \times S) \rightarrow \mathcal{D}$: maps an action and state to a domain

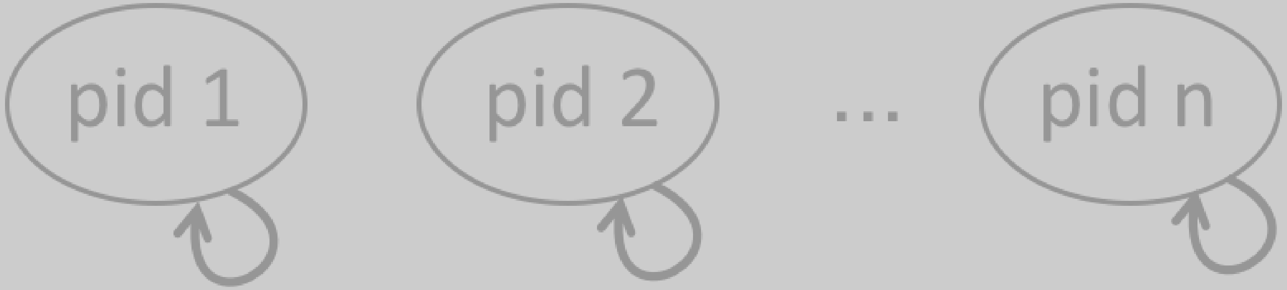


Information flow policies in Nickel

- Se
- Ca ... ains
- Fu ... n

Flexible definition enables broad set of policies

- Can-flow-to relation can be intransitive
- State dependent dom



Noninterference definition

$$\text{sources}(\epsilon, u, s) \coloneqq \{u\}$$

$$\text{sources}(a \circ tr, u, s) \coloneqq \begin{cases} \text{sources}\big(tr, u, \text{step}(s,a)\big) \cup \{\text{dom}(a,s)\} & \text{if } \exists v \in \text{sources}(tr, u, \text{step}(s,a)).\ \text{dom}(a,s) \rightsquigarrow u \\ \text{sources}\big(tr, u, \text{step}(s,a)\big) & \text{otherwise} \end{cases}$$

$$\text{purge}(\epsilon, u, s) \coloneqq \{\epsilon\}$$

$$\text{purge}(a \circ tr, u, s) \coloneqq \begin{cases} \{a \circ \text{tr}' \mid tr' \in \text{purge}\big(tr, u, \text{step}(s,a)\big)\} & \text{if dom}(a,s) \in \text{sources}(a \circ tr, u, s) \\ \{a \circ \text{tr}' \mid tr' \in \text{purge}\big(tr, u, \text{step}(s,a)\big)\} \cup \text{purge}(tr, u, s) & \text{otherwise} \end{cases}$$

$$\forall\, tr' \in \text{purge}\Big(tr, \text{dom}\big(a, \text{run}(\text{init}, tr)\big), \text{init}\Big).\ \text{output}(\text{run}(\text{init}, tr), a) = \text{output}(\text{run}(\text{init}, tr'), a)$$

Noninterference definition

$\mathrm{sources}(\epsilon, u, s) \coloneqq \{u\}$

Given a policy, purging actions “irrelevant” to a domain should not affect the output of the actions for that domain

$$\forall\, tr' \in \mathrm{purge}\big(tr, \mathrm{dom}\big(a, \mathrm{run}(\mathrm{init}, tr)\big), \mathrm{init}\big).\ \mathrm{output}(\mathrm{run}(\mathrm{init}, tr), a) = \mathrm{output}(\mathrm{run}(\mathrm{init}, tr'), a)$$

Automated verification of noninterference

- $\mathcal{I}(\text{init}) \wedge \mathcal{I}(s) \Rightarrow \mathcal{I}\big(\text{step}(s, a)\big)$
- $\overset{u}{\approx}$ is reflexive, symmetric, and transitive
- $\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\text{dom}(a,s)}{\approx} t \Rightarrow \text{dom}(a, s) = \text{dom}(a, t)$
- $\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \Rightarrow (\text{dom}(a, s) \rightsquigarrow u \Leftrightarrow \text{dom}(a, t) \rightsquigarrow u)$
- $\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\text{dom}(a,s)}{\approx} t \Rightarrow \text{output}(s, a) = \text{output}(t, a)$
- $\mathcal{I}(s) \wedge \text{dom}(a, s) \not\rightsquigarrow u \Rightarrow \text{s} \overset{u}{\approx} \text{step}(s, a)$
- $\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge s \overset{\text{dom}(a,s)}{\approx} \text{t} \Rightarrow \text{step}(s, a) \overset{u}{\approx} \text{step}(t, a)$

Automated verification of noninterference

- $\mathcal{I}(\text{init}) \wedge \mathcal{I}(s) \Rightarrow \mathcal{I}(\text{step}(s,a))$

Proof strategy: unwinding conditions

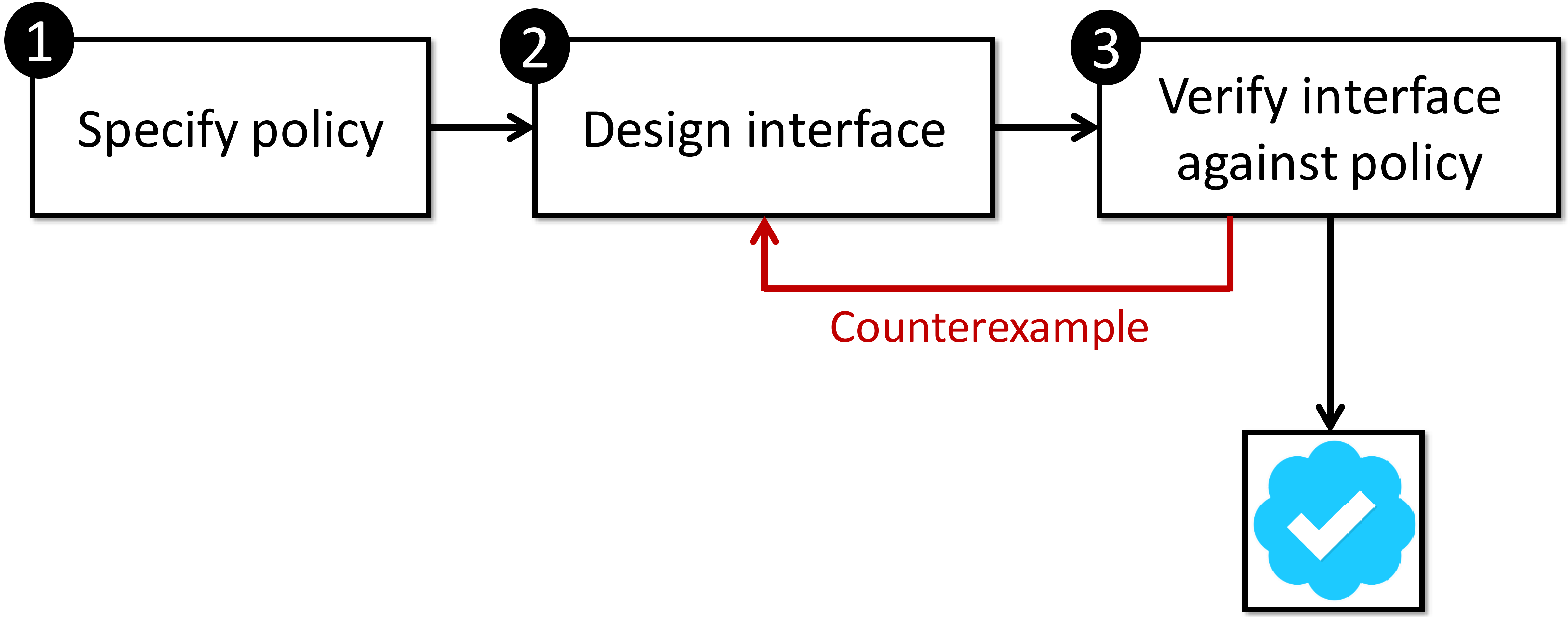
- Together imply noninterference
- Requires reasoning only about individual actions
- Amenable to automated reasoning using SMT

- $\mathcal{I}(s) \wedge \text{dom}(a,s) \not\rightsquigarrow u \Rightarrow s \overset{u}{\approx} \text{step}(s,a)$
- $\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge s \overset{\text{dom}(a,s)}{\approx} t \Rightarrow \text{step}(s,a) \overset{u}{\approx} \text{step}(t,a)$

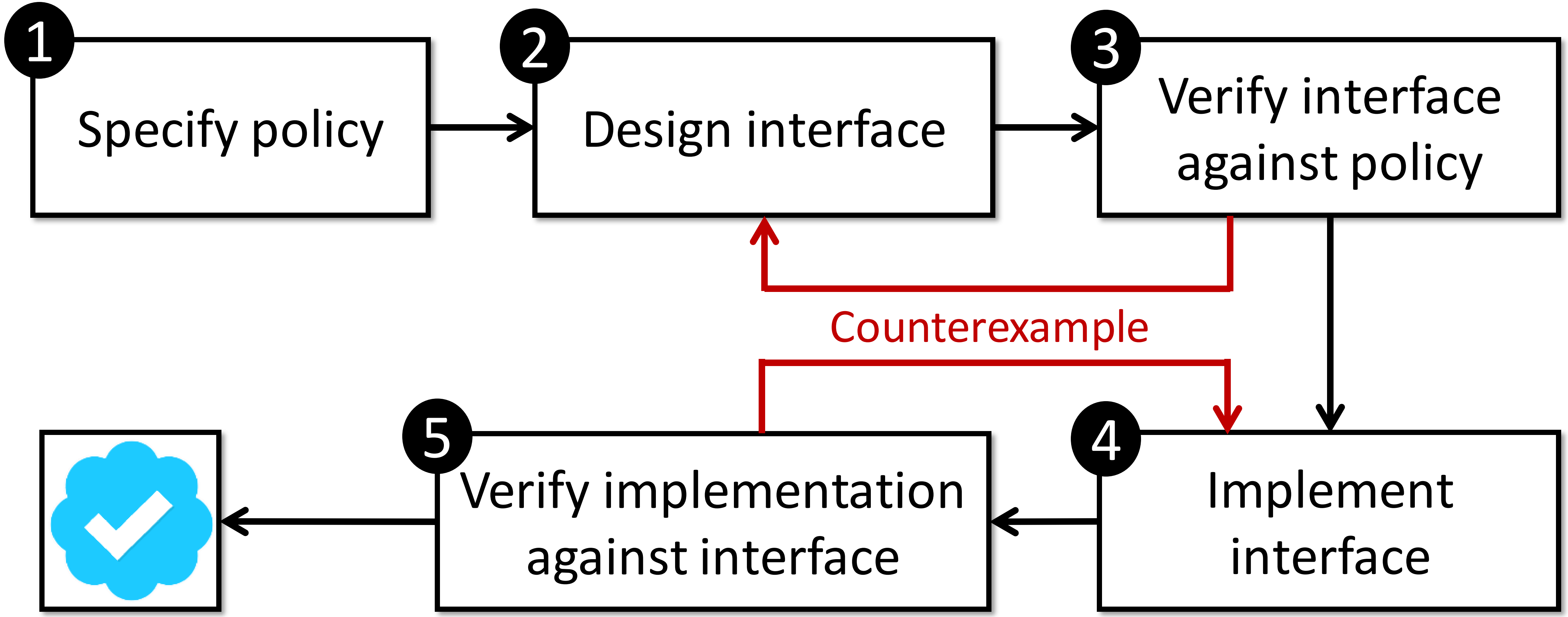
Outline

- New formulation and proof strategy for noninterference
- **Nickel: A framework for design and verification of information flow control (IFC) systems**
- Experience building three systems using Nickel
  - First formally verified decentralized IFC OS kernel
  - Low proof burden: order of weeks

Verification-driven interface design in Nickel



Verification-driven interface design in Nickel

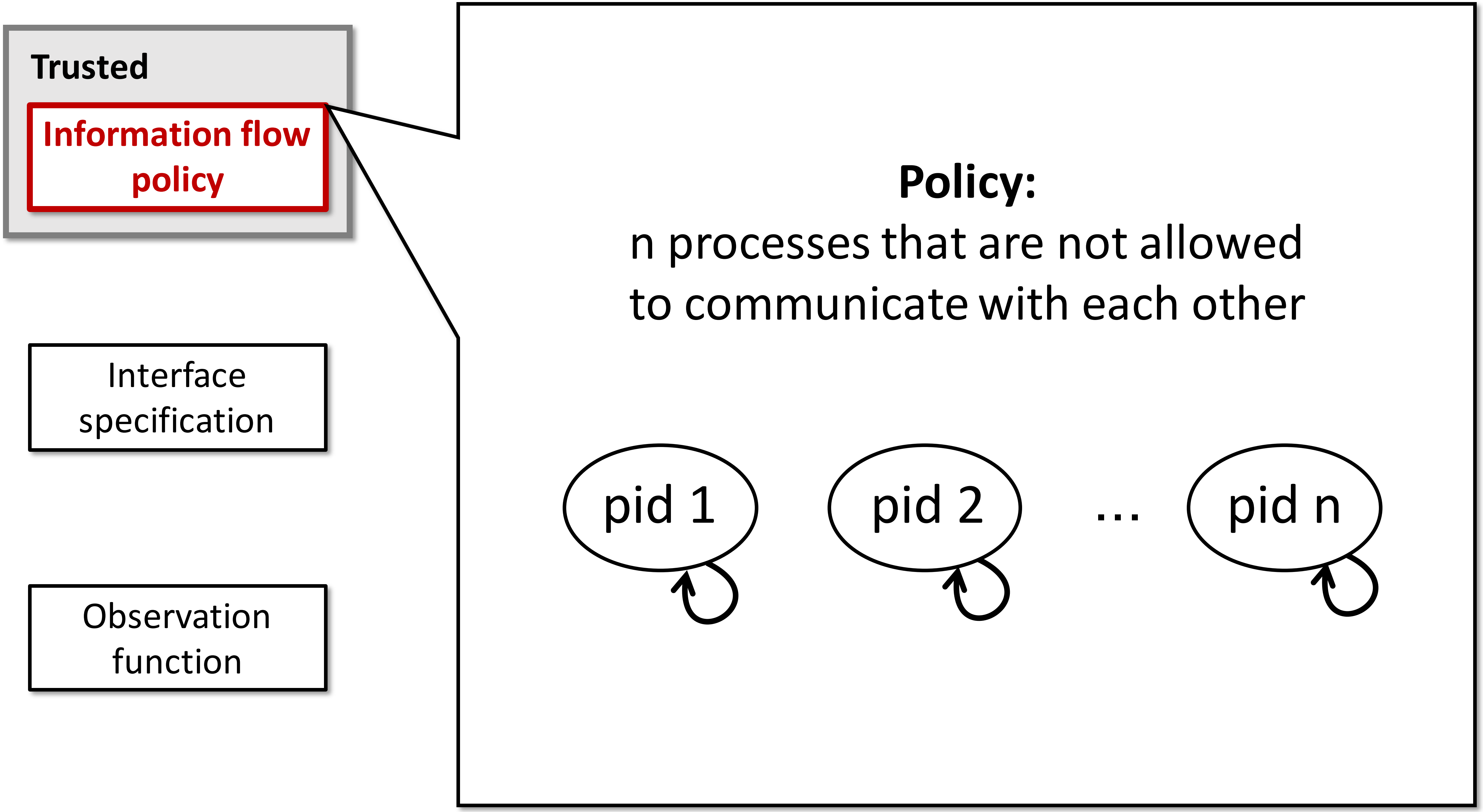


Trusted

Information flow policy

Interface specification

Observation function


Trusted
Information flow policy
Interface specification
Observation function
Policy:
n processes that are not allowed to communicate with each other
pid 1
pid 2
…
pid n

Trusted

Information flow policy

Interface specification

Observation function

```
class State:
    current     = PidT()
    nr_procs    = SizeT()
    proc_status = Map(PidT, StatusT)
```

Trusted

Information flow policy

Interface specification

Observation function

```python
class State:
    current      = PidT()
    nr_procs     = SizeT()
    proc_status  = Map(PidT, StatusT)

def can_flow_to(domain1, domain2):
    # Flow is only permitted,
    # if they are the same domain
    return domain1 == domain2
```

Trusted

Information flow policy

Interface specification

Observation function

```python
class State:
    current      = PidT()
    nr_procs     = SizeT()
    proc_status = Map(PidT, StatusT)

def can_flow_to(domain1, domain2):
    # Flow is only permitted,
    # if they are the same domain
    return domain1 == domain2

def dom(action, state):
    # Domain of every action
    # is just the current process
    return state.current
```

Trusted

Information flow policy

Interface specification

Observation function

```
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```python
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```python
def sys_spawn(old):
    # compute child pid
    child_pid = old.nr_procs + 1

    # Check if there are too many processes
    pre = child_pid <= NR_PROCS

    # clone old state
    new = old.copy()

    # bump the number of processes
    new.nr_procs += 1

    # initialize the child process
    new.procs_status[child_pid] = RUNNABLE

    # return the new state and condition and
    # the child's pid
    return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```python
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```python
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```
def sys_spawn(old):
  # compute child pid
  child_pid = old.nr_procs + 1

  # Check if there are too many processes
  pre = child_pid <= NR_PROCS

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.nr_procs += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

Trusted

Information flow policy

Interface specification

Observation function

```
class State:
    current     = PidT()
    nr_procs    = SizeT()
    proc_status = Map(PidT, StatusT)
```

Trusted

Information flow policy

Interface specification

Observation function

```python
class State:
    current     = PidT()
    nr_procs    = SizeT()
    proc_status = Map(PidT, StatusT)

def observable_state(state, pid):
    return [
        state.current,
        state.nr_procs,
        state.procs_status[pid]
    ]
```

Trusted

Information flow policy

Interface specification

Observation function

```
class State:
    current     = PidT()
    nr_procs    = SizeT()
    proc_status = Map(PidT, StatusT)

def observable_state(state, pid):
    return [
        state.current,
        state.nr_procs,
        state.procs_status[pid]
    ]
```

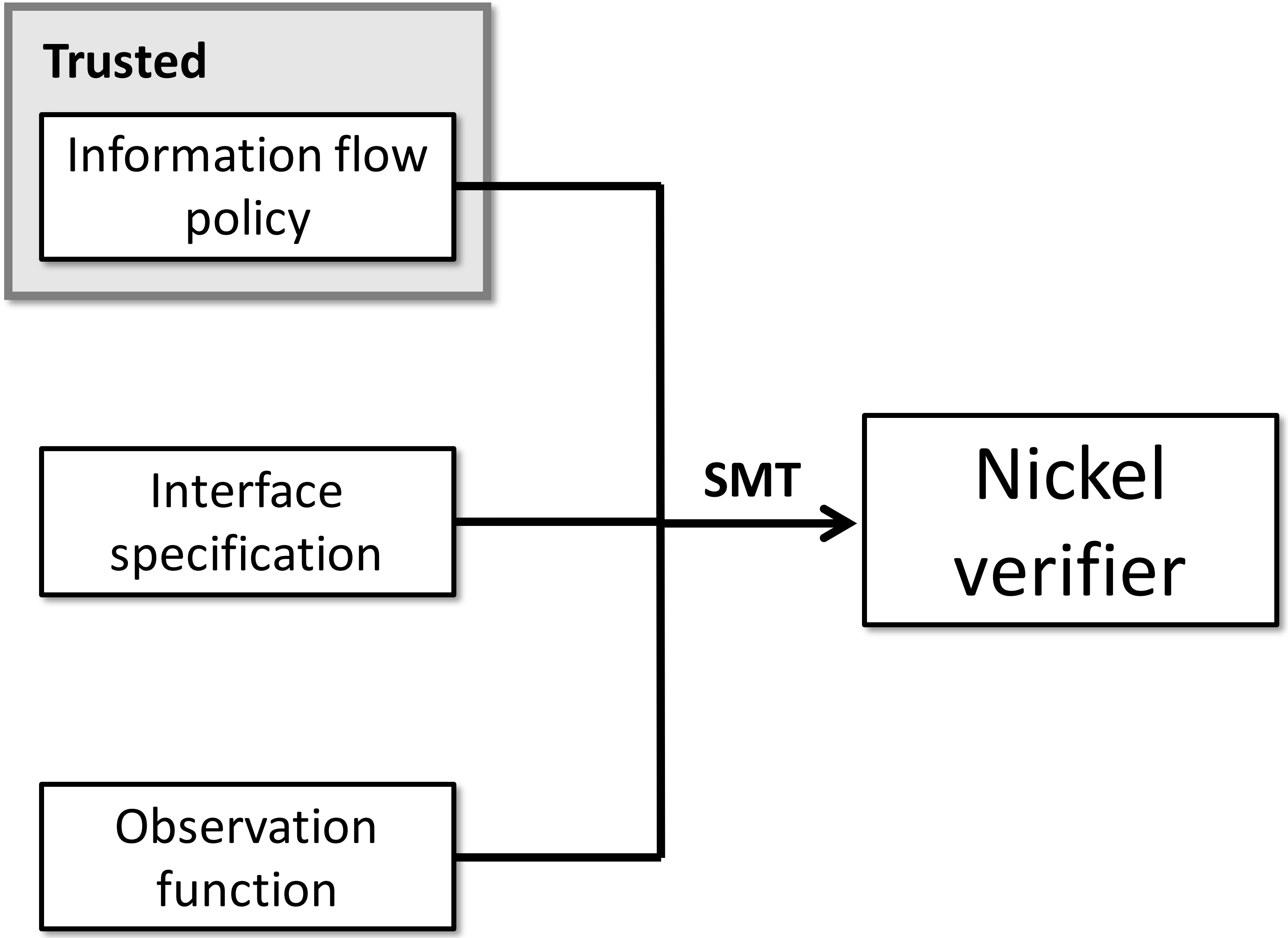
Trusted

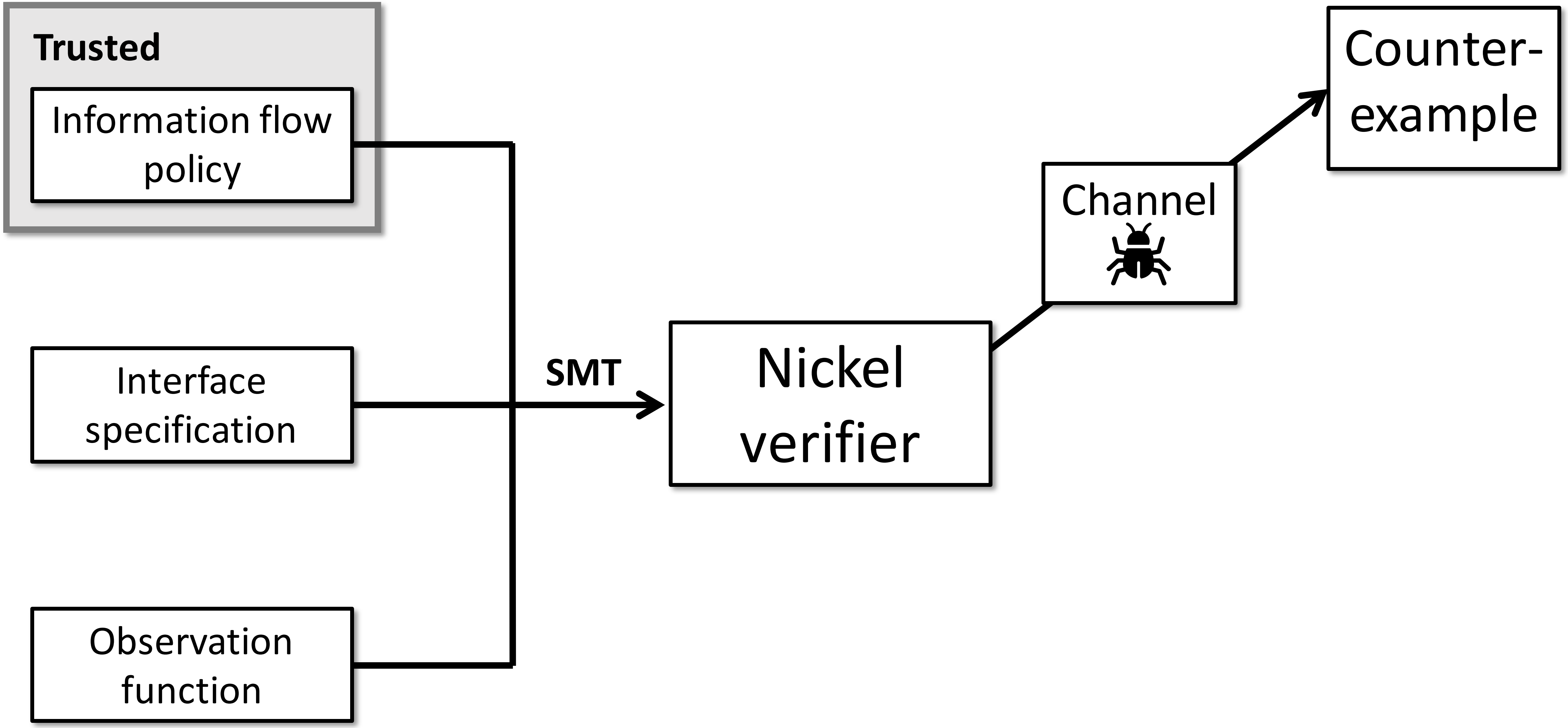
Information flow policy

Interface specification

Observation function

```
class State:
    current     = PidT()
    nr_procs    = SizeT()
    proc_status = Map(PidT, StatusT)

def observable_state(state, pid):
    return [
        state.current,
        state.nr_procs,
        state.procs_status[pid]
    ]
```


Trusted
Information flow policy
Interface specification
Observation function
SMT
Nickel verifier


Trusted
Information flow policy
Interface specification
Observation function
SMT
Nickel verifier
Channel
Counter-example

Trusted

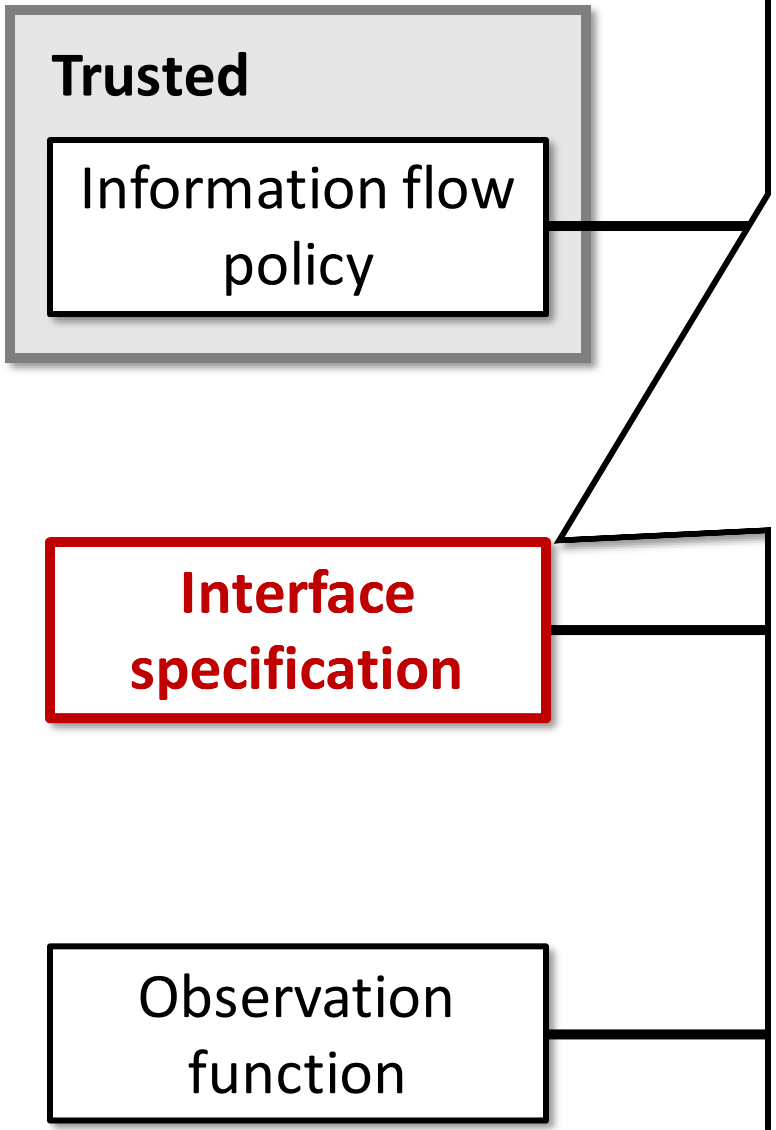
Information flow policy

Interface specification

Observation function

Design patterns

- **Partition names among domains**
- Reduce flows to the scheduler
- Perform flow checks early
- Limit resource usage with quotas
- Encrypt names from a large space
- Expose or enclose nondeterminism




```python
def sys_spawn(old):
  # compute child pid
  child_pid = (old.procs_nr_children[old.current]
               + 1 + old.current * 3)

  # Check if current has too many children
  pre = old.procs_nr_children[new.current] <= 3

  # clone old state
  new = old.copy()

  # bump the number of processes
  new.procs_nr_children[new.current] += 1

  # initialize the child process
  new.procs_status[child_pid] = RUNNABLE
  new.procs_nr_children[child_pid] = 0

  # return the new state and condition and
  # the child's pid
  return new, pre, child_pid
```

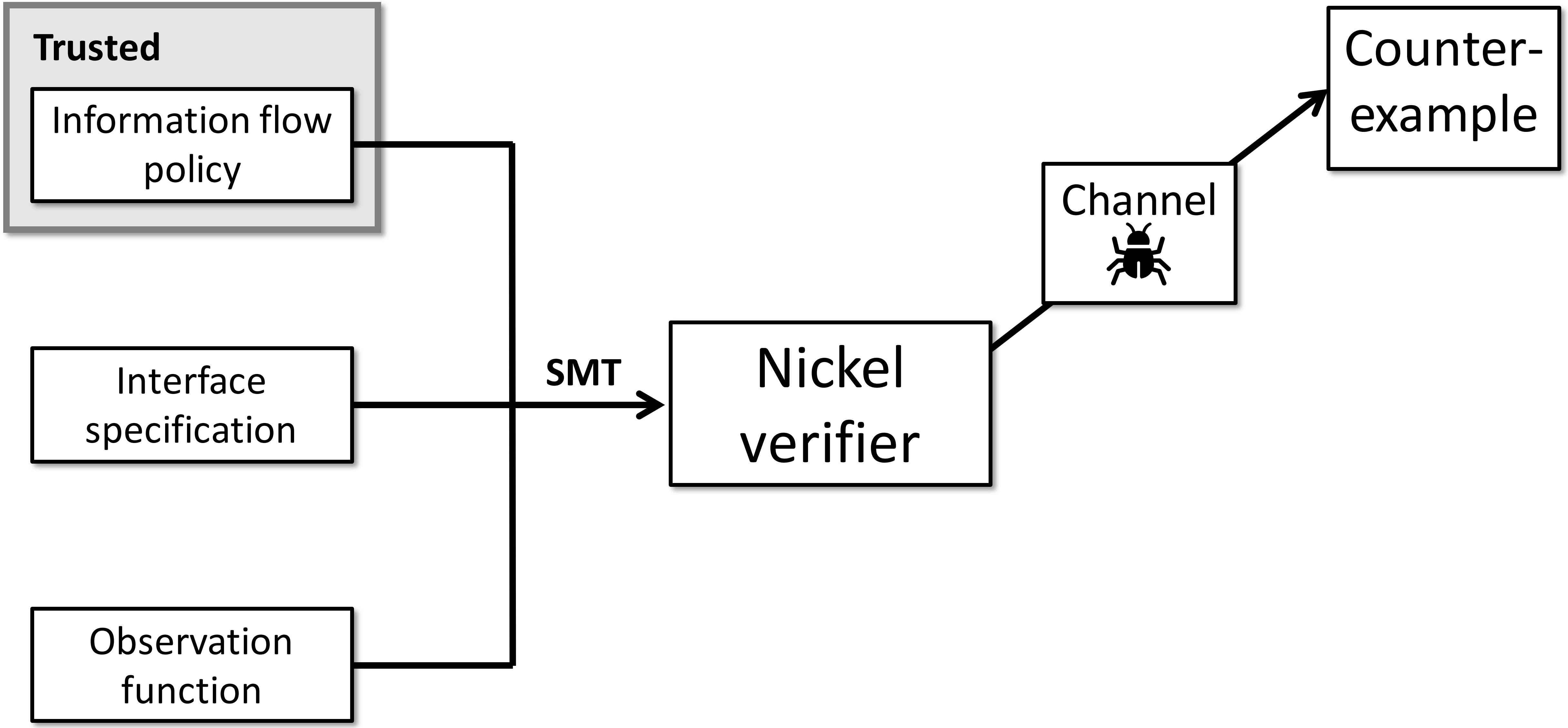
Trusted

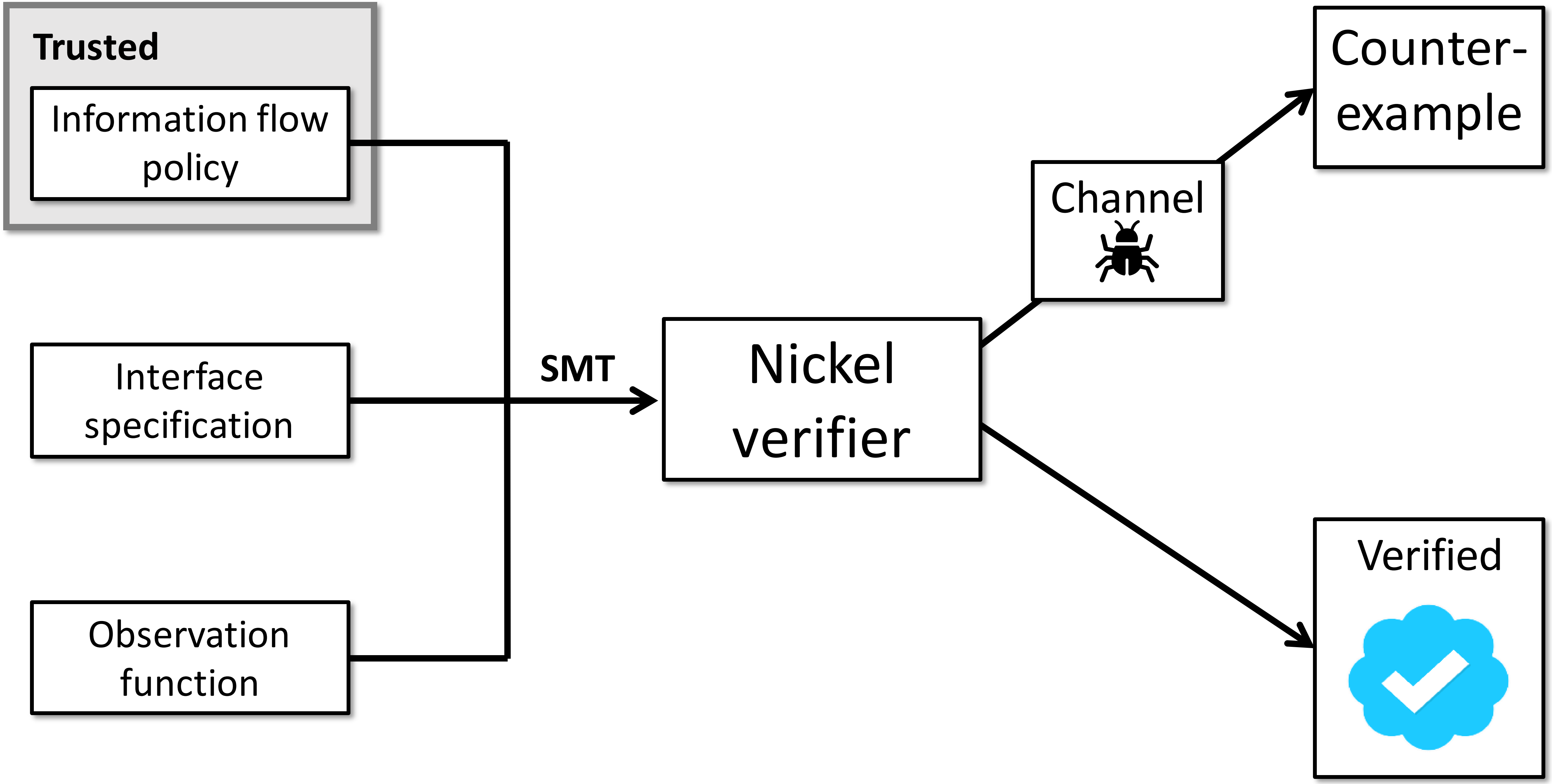
Information flow policy

Interface specification

Observation function

```
def sys_spawn(old):
    # compute child pid
    child_pid = (old.procs_nr_children[old.current]
                 + 1 + old.current * 3)

    # Check if current has too many children
    pre = old.procs_nr_children[new.current] <= 3

    # clone old state
    new = old.copy()

    # bump the number of processes
    new.procs_nr_children[new.current] += 1

    # initialize the child process
    new.procs_status[child_pid] = RUNNABLE
    new.procs_nr_children[child_pid] = 0

    # return the new state and condition and
    # the child's pid
    return new, pre, child_pid
```


Trusted
Information flow policy
Interface specification
Observation function
SMT
Nickel verifier
Channel
Counter-example


Trusted
Information flow policy
Interface specification
Observation function
SMT
Nickel verifier
Channel
Counter-example
Verified

Outline

- New formulation and proof strategy for noninterference
- Nickel: A framework for design and verification of information flow control (IFC) systems
- **Experience building three systems using Nickel**
  - **First formally verified decentralized IFC OS kernel**
  - **Low proof burden: order of weeks**

Decentralized information flow control (DIFC)

- Flexible mechanism to enforce security policies [SOSP ’97]
  - Each object assigned labels for tracking and mediating data access

- Several operating system kernels enforce DIFC:
  - Asbestos [SOSP ’05]
  - HiStar [OSDI ’06]
  - Flume [SOSP ’07]

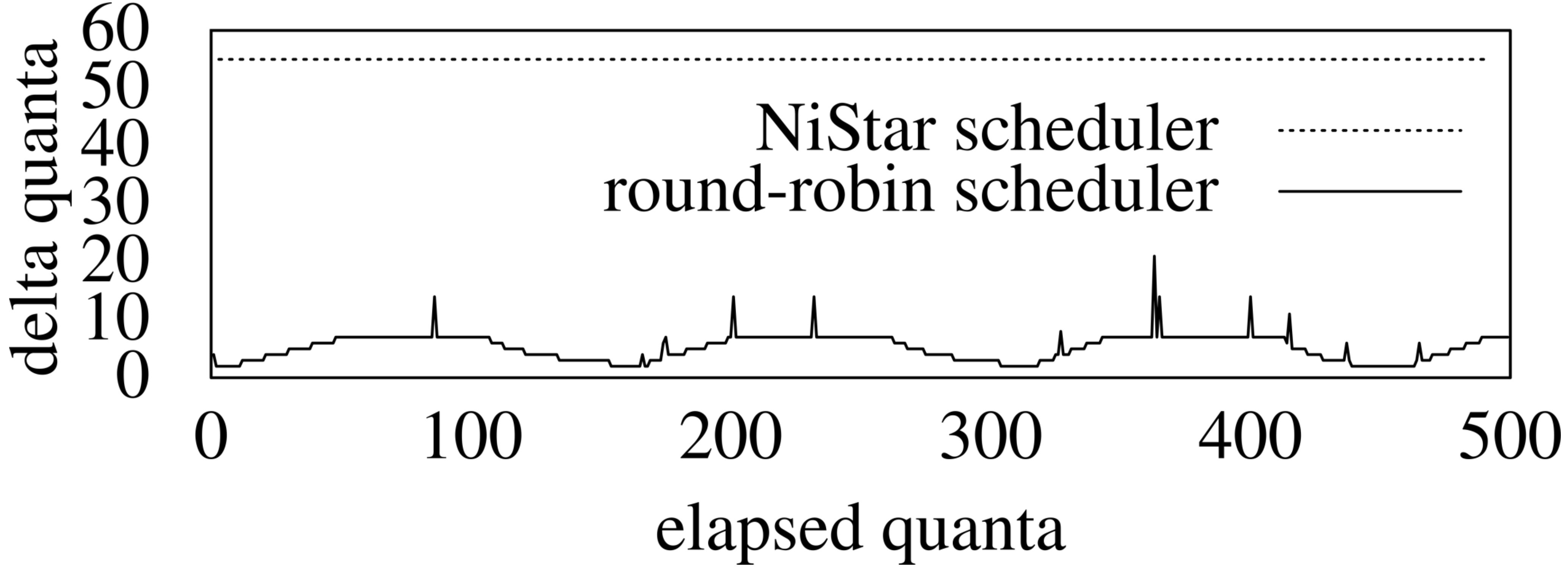
- **Our goal:** Build a DIFC OS kernel without any covert channels

NiStar: First verified DIFC OS

- Resembles an exokernel with finite interface design
  - 46 system calls and exception handlers
  - Supports musl C stdlib using Linux emulation, file system, lwip network service
- Enforces information flow among small number of object types
  - Labels, containers, threads, gates, page-table pages, user pages, quanta
  - Each object is assigned three labels: Secrecy S, integrity I, ownership O
- Simple policy: Given two objects with domains $\mathcal{L}_1$ and $\mathcal{L}_2$:
  - $\mathcal{L}_1 = \langle S_1, I_1, O_1 \rangle, \mathcal{L}_2 = \langle S_2, I_2, O_2 \rangle$
  - $\mathcal{L}_1 \rightsquigarrow \mathcal{L}_2 \coloneqq (S_1 - O_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1 \cup O_1)$

NiStar Scheduler

- New object types to close channel in scheduler



NiStar closes logical time channel in scheduler

Other systems

Subset of ARINC 653

- Industrial standard for avionics systems
- Reproduced three known bugs in the specification

NiKOS:

- Small Unix-like OS kernel mirroring mCertiKOS [PLDI '16]
- Process isolation policy

Implementation

Component	NiStar	NiKOS	ARINC 653
Information flow policy	26	14	33
Interface specification	714	82	240
Observational equivalence	127	56	80
Interface implementation	3,155	343	-
User-space implementation	9,348	389	-
Common kernel infrastructure	4,829	(shared by NiStar / NiKOS)	

Implementation

Component	NiStar	NiKOS	ARINC 653
Information flow policy	**26**	**14**	**33**
Interface specification	714	82	240
Observational equivalence	127	56	80
Interface implementation	3,155	343	-
User-space implementation	9,348	389	-
Common kernel infrastructure	4,829 (shared by NiStar / NiKOS)		

Concise policy

Low proof burden

- **NiStar**:
  - Six months for the first prototype implementation
  - Six weeks on verification
- **NiKOS**: two weeks
- **ARINC 653**: one week

Conclusion

- Verification-driven interface design
  - Systematic way to design secure interfaces
  - Interactive workflow with counterexample-based debugging
- First verified DIFC system
  - Low proof burden

UNSAT

https://nickel.unsat.systems