Nickel
A Framework for Design and Verification of Information Flow Control Systems


UNIVERSITY of WASHINGTON
PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING
Enforcing information flow control is critical
FBI: Hacker claimed to have taken over flight's engine controls

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

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

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
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:

- 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

![Diagram](image)

- Process 1
- Process 2
- Arrow from Process 1 to Process 2 labeled 5
- Arrow from Process 2 labeled 1
- Process 2 labeled `spawn → 3`
Covert channel in resource names

**Policy:** Process 1 and Process 2 should not communicate

**Design:** Spawn with sequential PID allocation

5 times

<table>
<thead>
<tr>
<th>Process 1</th>
<th>5 times</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn → 3+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spawn → 3+5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. spawn → 3

2. ...
Covert channel in resource names

Policy: Process 1 and Process 2 should not communicate

Design: Spawn with sequential PID allocation

5 times

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn → 3+1</td>
<td>spawn → 3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>spawn → 3+5</td>
<td></td>
</tr>
</tbody>
</table>
Examples of covert channels

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

Process 2:
spawn → 3

Process 1:
spawn 5 times

Process 2:
spawn → 9
Noninterference intuition

Process 2:
spawn → 3

Process 1:
spawn 5 times

Process 2:
spawn → 9

Process 2:
spawn → 3

Process 2:
spawn → 4
Noninterference intuition

Process 2:
- spawn → 3

Process 2:
- spawn → 3
- spawn → 4

Process 1:
- spawn 5 times
- spawn → 9

Process 1 interferes with Process 2
Information flow policies in Nickel

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

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

Flexible definition enables broad set of policies
- Can-flow-to relation can be intransitive
- State dependent $\text{dom}$

pid 1  pid 2  ...  pid n
Noninterference definition

\[
\begin{align*}
\text{sources}(\epsilon, u, s) & := \{u\} \\
\text{sources}(a \circ \text{tr}, u, s) & := \\
& \begin{cases} 
\text{sources}(\text{tr}, u, \text{step}(s, a)) \cup \{\text{dom}(a, s)\} & \text{if } \exists v \in \text{sources}(\text{tr}, u, \text{step}(s, a)). \text{dom}(a, s) \rightarrow u \\
\text{sources}(\text{tr}, u, \text{step}(s, a)) & \text{otherwise}
\end{cases} \\
\text{purge}(\epsilon, u, s) & := \{\epsilon\} \\
\text{purge}(a \circ \text{tr}, u, s) & := \\
& \begin{cases} 
\{a \circ \text{tr}' \mid \text{tr}' \in \text{purge}(\text{tr}, u, \text{step}(s, a))\} & \text{if } \text{dom}(a, s) \in \text{sources}(a \circ \text{tr}, u, s) \\
\{a \circ \text{tr}' \mid \text{tr}' \in \text{purge}(\text{tr}, u, \text{step}(s, a))\} \cup \text{purge}(\text{tr}, u, s) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[\forall \text{tr}' \in \text{purge}(\text{tr}, \text{dom}(a, \text{run}(\text{init}, \text{tr})), \text{init}). \text{output}(\text{run}(\text{init}, \text{tr}), a) = \text{output}(\text{run}(\text{init}, \text{tr}'), a)\]
Given a policy, purging actions “irrelevant” to a domain should not affect the output of the actions for that domain.
Automated verification of noninterference

\[ I(\text{init}) \land I(s) \Rightarrow I(\text{step}(s, a)) \]

\( \approx \) is reflexive, symmetric, and transitive

\[ I(s) \land I(t) \land s \approx t \Rightarrow \text{dom}(a, s) = \text{dom}(a, t) \]

\[ I(s) \land I(t) \land s \approx t \Rightarrow (\text{dom}(a, s) \rightsquigarrow u \Leftrightarrow \text{dom}(a, t) \rightsquigarrow u) \]

\[ I(s) \land I(t) \land s \approx t \Rightarrow \text{output}(s, a) = \text{output}(t, a) \]

\[ I(s) \land \text{dom}(a, s) \not\rightsquigarrow u \Rightarrow s \approx \text{step}(s, a) \]

\[ I(s) \land I(t) \land s \approx t \land s \approx t \Rightarrow \text{step}(s, a) \approx \text{step}(t, a) \]
Automated verification of noninterference

- \( I(\text{init}) \land I(s) \Rightarrow I(\text{step}(s, a)) \)

**Proof strategy:** unwinding conditions
- Together imply noninterference
- Requires reasoning only about individual actions
- Amenable to automated reasoning using SMT

- \( I(s) \land \text{dom}(a, s) \equiv u \Rightarrow s \equiv \text{step}(s, a) \)
- \( I(s) \land I(t) \land s \equiv t \land s \equiv t \Rightarrow \text{step}(s, a) \equiv \text{step}(t, a) \)
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

1. Specify policy
2. Design interface
3. Verify interface against policy

Counterexample
Verification-driven interface design in Nickel

1. Specify policy
2. Design interface
3. Verify interface against policy
4. Implement interface
5. Verify implementation against interface

Counterexample
Trusted

Information flow policy

Interface specification

Observation function
Information flow policy

Policy:

n processes that are not allowed to communicate with each other

pid 1

pid 2

... pid n
class State:
    current = PidT()
    nr_procs = SizeT()
    proc_status = Map(PidT, StatusT)
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
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
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
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
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
```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
```
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
```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
```
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
class State:
    current = PidT()
    nr_procs = SizeT()
    proc_status = Map(PidT, StatusT)
```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]
    ]
```
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]
        ]
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]
    ]
Nickel verifier

- Trusted
  - Information flow policy

- Interface specification

- Observation function

SMT
Nickel verifier

Trusted Interface specification

Information flow policy

Observation function

Counter-example Channel SMT
**Trusted**

- Information flow policy

**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
```
```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
```
Nickel verifier

Counter-example

Channel

Information flow policy

Interface specification

Observation function

Trusted Interface specification

SMT
Nickel verifier

Trusted
- Information flow policy
- Interface specification
- Observation function

SMT

Counter-example

Verified

Channel
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 $L_1$ and $L_2$:
  • $L_1 = \langle S_1, I_1, O_1 \rangle, L_2 = \langle S_2, I_2, O_2 \rangle$
  • $L_1 ~\implies~ L_2 := (S_1 - O_1 \subseteq S_2 \cup O_2) \land (I_2 - O_2 \subseteq I_1 \cup O_1)$
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

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

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

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

https://nickel.unsat.systems