Nickel: A framework for design and verification of information flow control systems

Luke Nelson
Joint work with Helgi Sigurbjarnarson, Bruno Castro-Karney, James Bornholt, Emina Torlak, Xi Wang

2018 New England Systems Verification Day
Motivation: high verification burden

• Verification is effective at eliminating bugs
• Requires expertise
• Large time investment
Approach: push-button verification

- Yggdrasil
  - Crash-safe filesystems (Python)
  - OSDI 2016
- Hyperkernel
  - Small OS kernel (C, memory isolation)
  - SOSP 2017
- Nickel
  - Information flow control systems
  - OSDI 2018
Information flow control systems
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
Eddie Kohler
@xedx

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

5:43 AM - 8 Aug 2018
Goal: eliminate covert channels from systems

- **Covert channel** *(Lampson ’73)*: unintended flow between system components

- **Approach**: verification-driven development
  - Verify noninterference for interface specification
  - Verify refinement for implementation

- **Limitations**: no physical channels; no concurrency
Contributions

• Formulation of noninterference amenable to automated verification

• Nickel is a framework for verifying IFC systems.

• Applied Nickel to verify systems including
  • NiStar: first formally verified DIFC OS kernel
  • ARINC 653 communication interface: avionics kernel standard
Example covert channel: resource names

**Policy:** process A and process B should not communicate

**Interface:** spawn system call returns sequential PIDs

Try to violate policy by sending a secret (in this case, 2) to process B.
Example covert channel: resource names

**Policy:** process A and process B should not communicate

**Interface:** spawn system call returns sequential PIDs

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn → 4</td>
<td>spawn → 3</td>
</tr>
<tr>
<td>spawn → 5</td>
<td>spawn → 6</td>
</tr>
</tbody>
</table>

6−3−1=2
Noninterference intuition

Process B
spawn $\rightarrow$ 3

Process A
spawn $\rightarrow$ 4

Process A
spawn $\rightarrow$ 5

Process B
spawn $\rightarrow$ 6

Process B
spawn $\rightarrow$ 3

Process B
spawn $\rightarrow$ 4
Noninterference intuition

Many kinds of covert channels

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

For any trace $tr$, action $a$, removing “irrelevant” actions should not affect the output of $a$.

\[
\text{output}(\text{run}(\text{init}, tr), a) = \\
\text{output}(\text{run}(\text{init}, \text{purge}^*(tr, a)), a)
\]
Information flow policies in Nickel

A set of domains

A can-flow-to relation specifying permitted flows among domains

A function mapping an action in a state to a domain

\[ D : \text{Set} \]

\[ \sim \subseteq (D \times D) \]

\[ \text{dom} : (A \times S) \rightarrow D \]

pid 1

pid 2

... pid n
Automated verification of noninterference

**Proof strategy**: unwinding conditions

- Together imply noninterference
- Reason about one action at a time
- Amenable to SMT solving using Z3

Local respect

\[ I(s) \land \neg(\text{dom}(a, s) \not\sim v) \rightarrow s \approx \text{step}(s, a) \]

Output consistency

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

Weak step consistency

\[ I(s) \land I(t) \land s \overset{u}{\approx} t \land s \overset{\text{dom}(a,s)}{\approx} t \rightarrow \text{step}(s, a) \overset{u}{\approx} \text{step}(t, a) \]
Nickel workflow

Specify policy → Design interface → Verify interface against policy → Interface noninterference

Implement interface → Verify implementation against interface → Implementation noninterference and functional correctness

Counterexample
Programmer inputs

- Information flow policy
- Interface specification
- Observational equivalence
\( n \) processes that are not allowed to communicate
$n$ processes that are not allowed to communicate

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

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

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

pid 1 → pid 2 → … → pid $n$
```python
def sys_spawn(old):
    child_pid = old.nr_procs + 1
    pre = child_pid <= NR_PROCS
    new = old.copy()
    new.nr_procs += 1
    new.proc_status[child_pid] = RUNNABLE
    return pre, If(pre, new, old)
```
class State:
    current = PidT()
    nr_procs = SizeT()
    proc_status = Map(PidT, StatusT)

def obs_eqv(domain, state1, state2):
    return And(
        state1.current == state2.current,
        state1.nr_procs == state2.nr_procs,
        state1.proc_status[domain.pid] ==
        state2.proc_status[domain.pid]
    )
Systems verified using Nickel

<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>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 (shared by NiStar/NiKOS)</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Demo
spawn example
tainting example

send(B, 12)
tainting example

send(B, 12)

Process A
Value: 0
Level: tainted

Process B
Value: 12
Level: tainted

Process C
Value: 3
Level: untainted

Process D
Value: 5
Level: tainted
tainting example

if Value == 0:
    send(B, 0)

wait(500)
if Level != tainted:
    send(C, 1)

wait(1000)
if Value == 0:
    # secret is 0
else:
    # secret is 1
tainting example

if Value == 0:
    send(B, 0)

wait(500)
if Level != tainted:
    send(C, 1)

wait(1000)
if Value == 0:
    # secret is 0
else:
    # secret is 1

send(C, 1)
tainting example

if Value == 0:
    send(B, 0)

wait(500)
if Level != tainted:
    send(C, 1)

wait(1000)
if Value == 0:
    # secret is 0
else:
    # secret is 1

if Value == 0:
    send(B, 0)
Thanks!

https://nickel.unsat.systems