Overcoming an

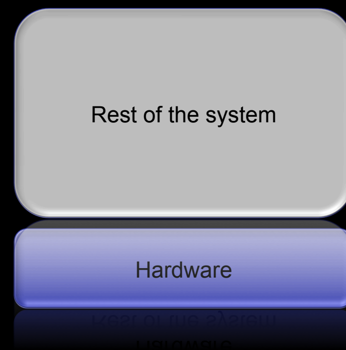
UNTRUSTED COMPUTING BASE:

Detecting and Removing Malicious Hardware Automatically

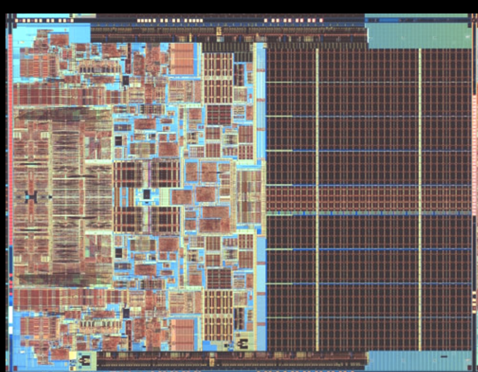
Matthew Hicks
Murph Finnicum
Samuel T. King
University of Illinois Urbana-Champaign

Milo M. K. Martin
Jonathan M. Smith
University of Pennsylvania

Hardware is too important to trust blindly

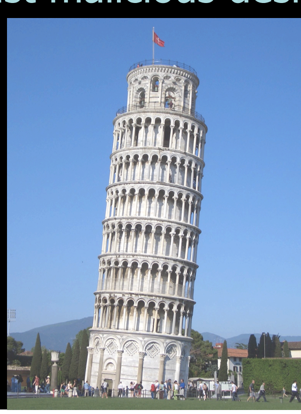


Hardware is as complex as software



Hardware complexity equals hardware vulnerability

Hardware must be defended against malicious designers



BlueChip looks for malicious insertions at design time and prevents them from affecting the system during runtime

BlueChip is both hardware and software, design time and run time

Software system

Handle missing hardware – BlueChip software

Run time

Design time

Disable malicious circuits – BlueChip hardware

Identify malicious circuits - UCI

Original design

Design

Implementation

Experiments

Conclusion

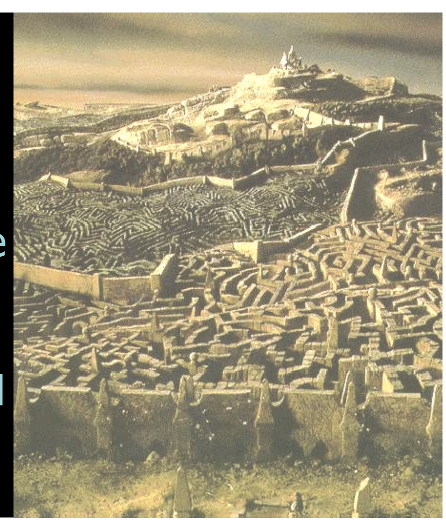
Questions

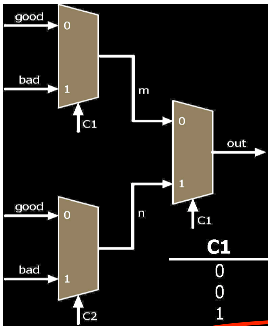
BlueChip helps managers increase trust, without requiring them to know more

UCI highlights potentially malicious circuits automatically

Attackers must avoid impacting functionality during testing

UCI detects all circuits where the output value is identical to the input value, for all test cases

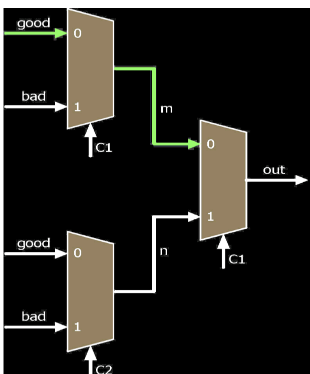




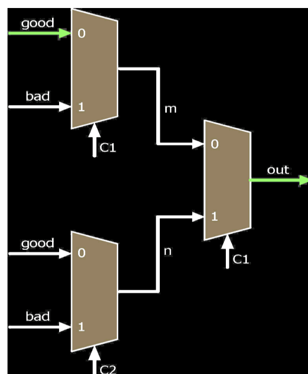
C1	C2	m	n	out
0	0	good	good	good
0	1	good	bad	good
1	0	bad	good	good
1	1	bad	bad	bad

Data-flow triples generation

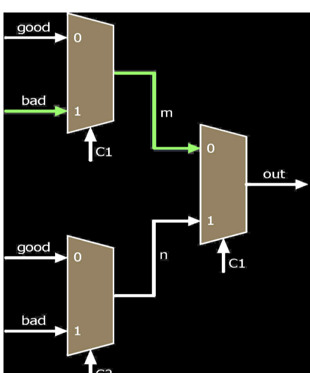
start at output signals
recurse_tuples
- for each item in the parents list
  - generate a tuple
- for each driver
  - add self to temp parents list
  - if driver behind a flip-flop
    - increase delay in temp parents list
  - recurse on child



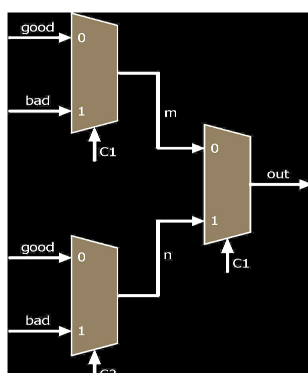
Triples:
(good, m, 0)



Triples:
(good, m, 0)
(good, out, 0)



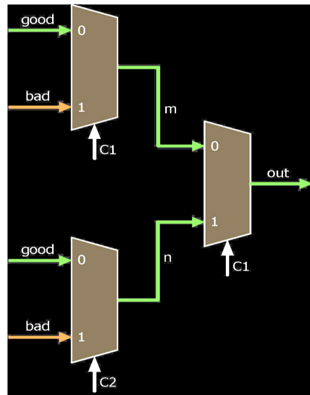
Triples:
(good, m, 0)
(good, out, 0)
(bad, m, 0)



Triples:
(good, m, 0)
(good, out, 0)
(bad, m, 0)
(bad, out, 0)
(m, out, 0)
(good, n, 0)
(bad, n, 0)
(n, out, 0)

UCI Analysis

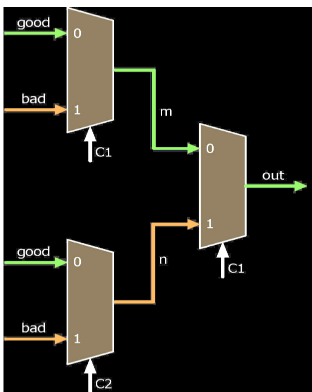
for each test case
    for each clock cycle
        for each dataflow triple remaining
            if target != driver(delay)
                remove triple



Triples:
(good, m, 0)
(good, out, 0)
~~(bad, m, 0)~~
~~(bad, out, 0)~~
(m, out, 0)
(good, n, 0)
~~(bad, n, 0)~~
(n, out, 0)

Test cases:

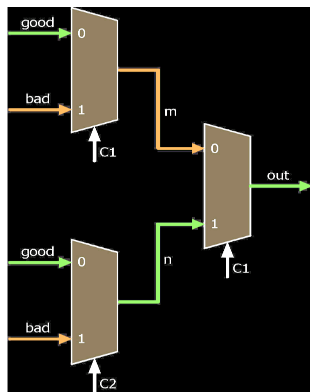
C1	C2
0	0
0	1
1	0
...	...
1	0



Triples:
(good, m, 0)
(good, out, 0)
~~(bad, m, 0)~~
~~(bad, out, 0)~~
(m, out, 0)
~~(good, n, 0)~~
~~(bad, n, 0)~~
~~(n, out, 0)~~

Test cases:

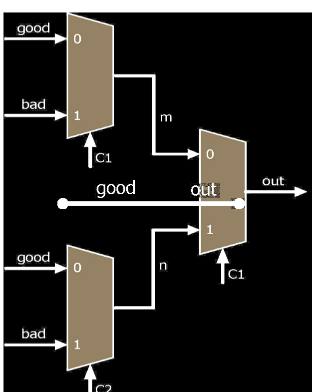
C1	C2
0	0
0	1
1	0
...	...
1	0



Triples:
~~(good, m, 0)~~
(good, out, 0)
~~(bad, m, 0)~~
~~(bad, out, 0)~~
~~(m, out, 0)~~
~~(good, n, 0)~~
~~(bad, n, 0)~~
~~(n, out, 0)~~

Test cases:

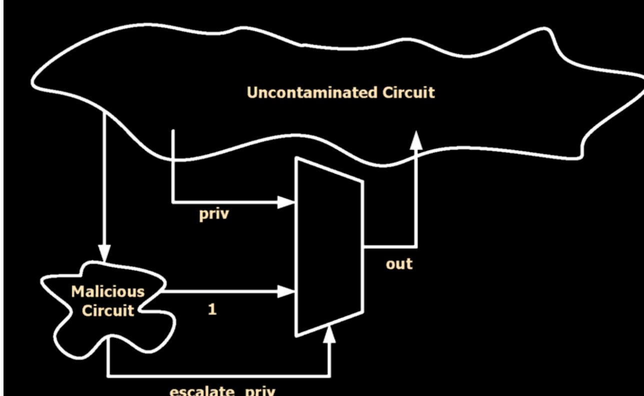
C1	C2
0	0
0	1
1	0
...	...
1	0

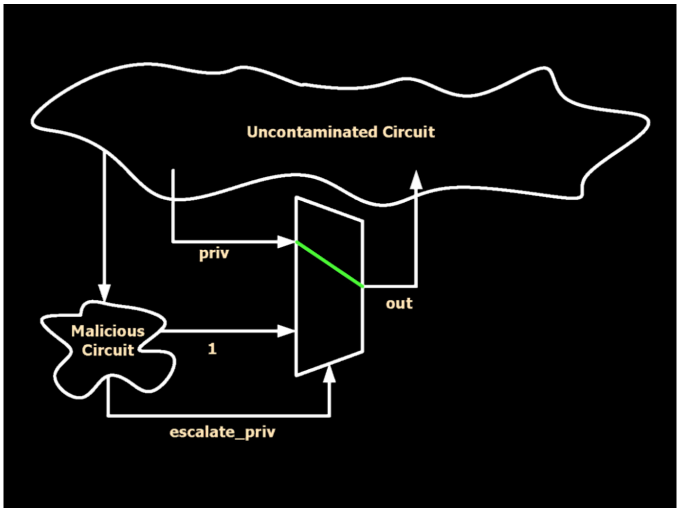


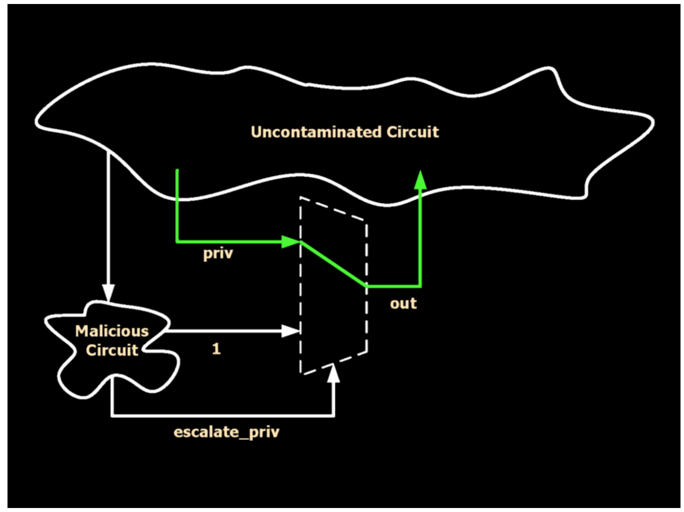
Triples:
~~(good, m, 0)~~
(good, out, 0)
~~(bad, m, 0)~~
~~(bad, out, 0)~~
~~(m, out, 0)~~
~~(good, n, 0)~~
~~(bad, n, 0)~~
~~(n, out, 0)~~

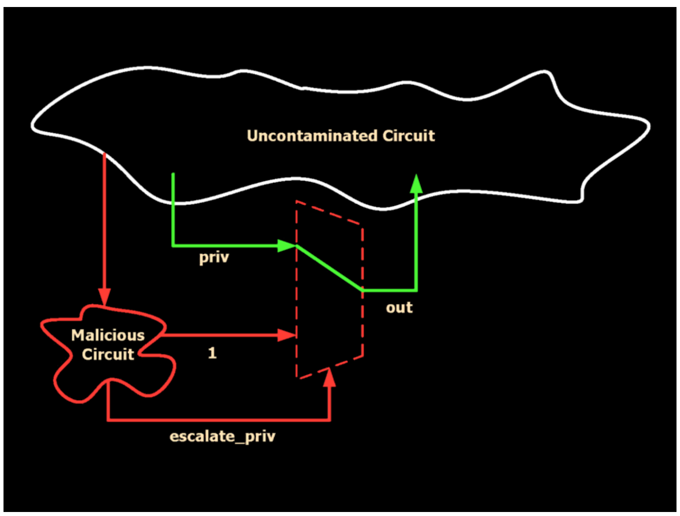
Test cases:

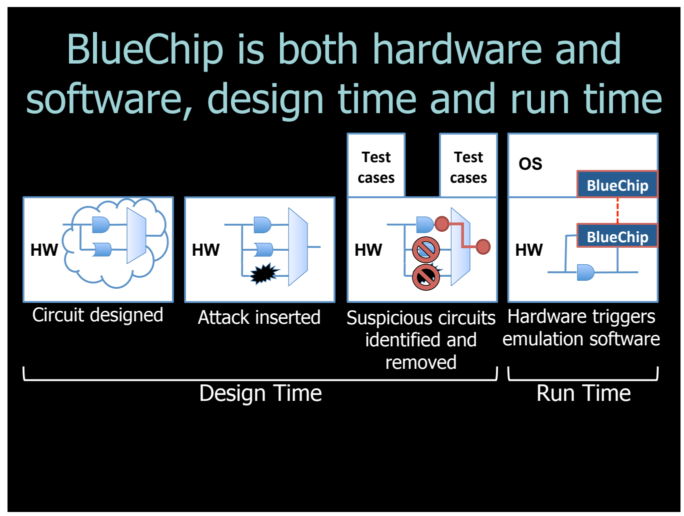
C1	C2
0	0
0	1
1	0
...	...
1	0

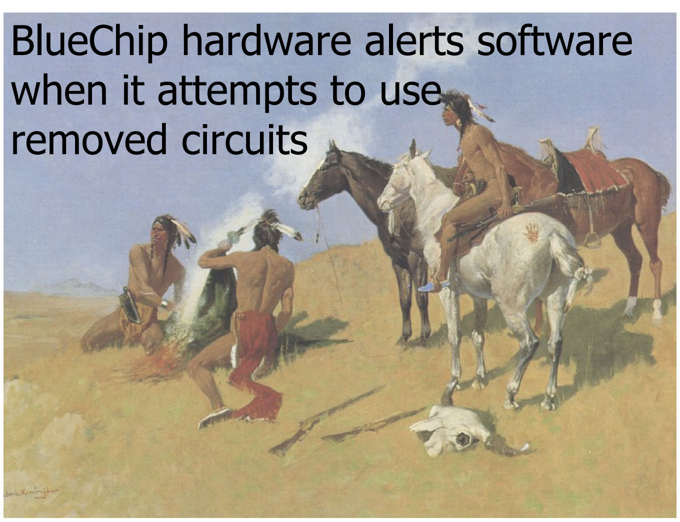


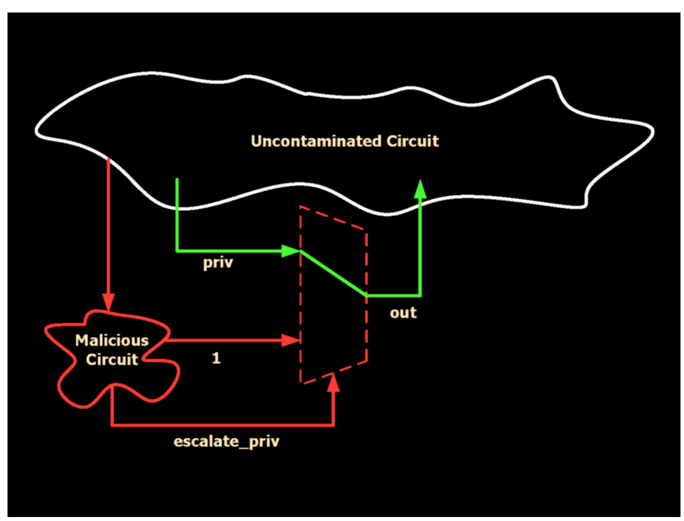

Uncontaminated Circuit
priv
out
Malicious Circuit
1
escalate_priv

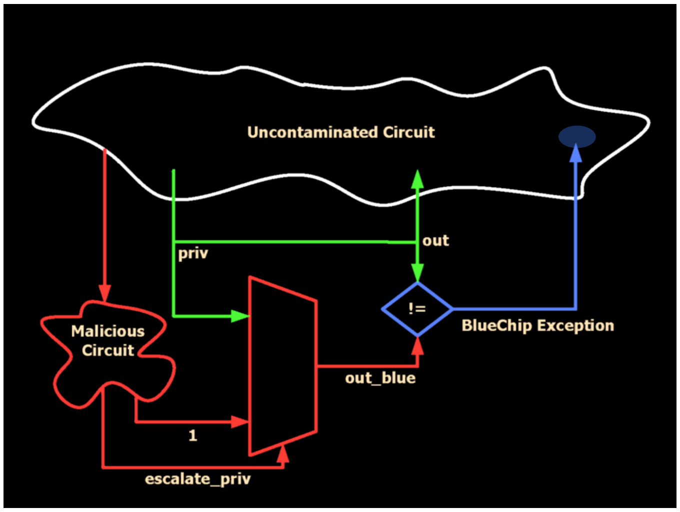

Uncontaminated Circuit
priv
out
Malicious Circuit
1
escalate_priv


Uncontaminated Circuit
priv
out
Malicious Circuit
1
escalate_priv


BlueChip is both hardware and software, design time and run time
Test cases
Test cases
OS
BlueChip
BlueChip
HW
HW
HW
HW
Circuit designed
Attack inserted
Suspicious circuits identified and removed
Hardware triggers emulation software
Design Time
Run Time

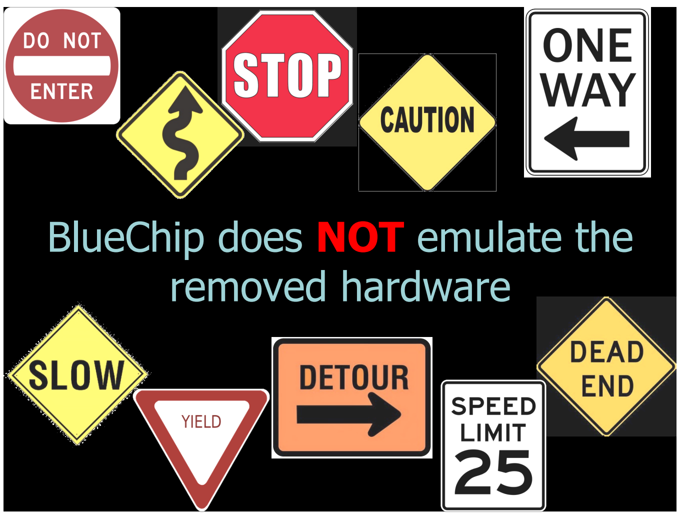

BlueChip hardware alerts software when it attempts to use removed circuits

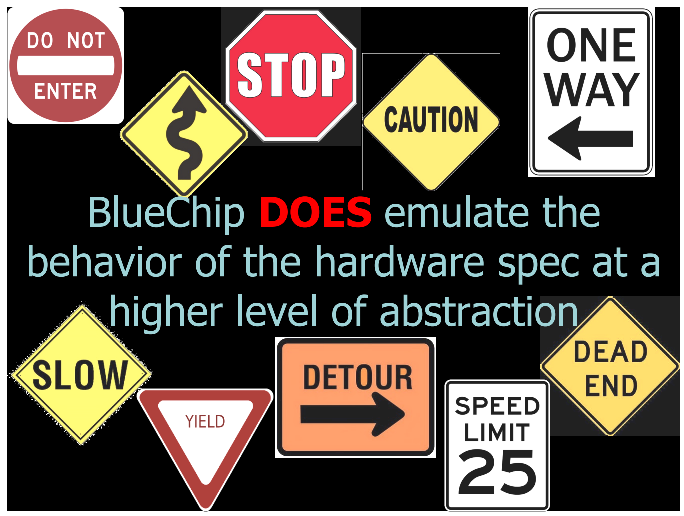

Uncontaminated Circuit
priv
out
Malicious Circuit
1
escalate_priv

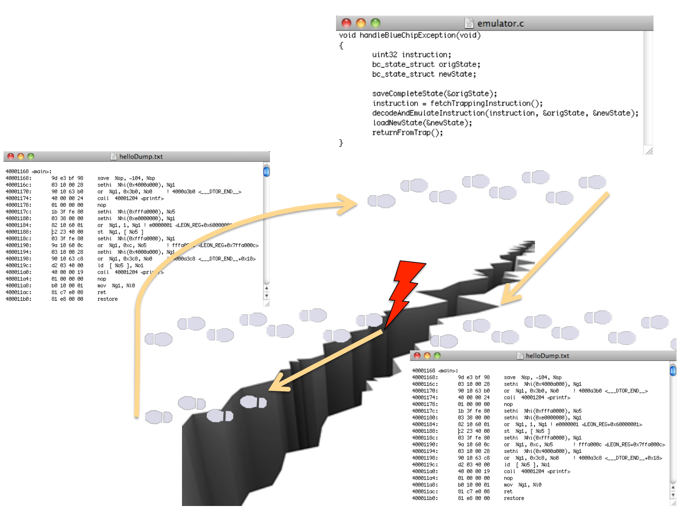


BlueChip software emulates the behavior of removed hardware

1. Receive BlueChip exception
2. Load state of processor
3. Fetch trapping instruction
4. Decode trapping instruction
5. Execute trapping instruction in emulator
6. Store updated state to hardware
7. Return from trap







BlueChip isn't effective in certain situations

- Undefined state
  - Low visibility test cases
  - Architecturally undefined state
- Malicious test cases
  - ISA emulator also vettes test cases
- Control information
  - Implementation dependent

Design

Implementation

Experiments

Conclusion

Questions

Design

Implementation

Experiments

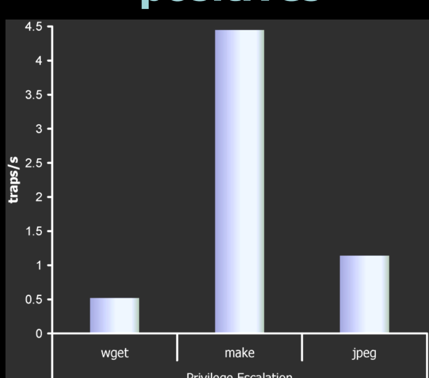
Conclusion

Questions

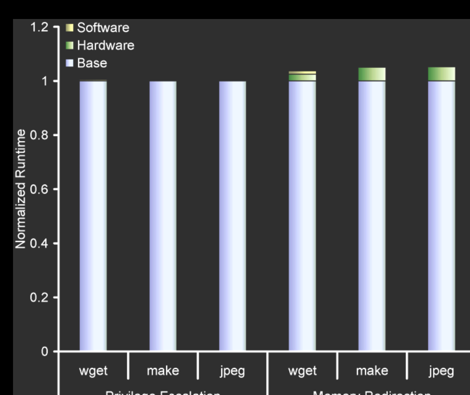
BlueChip successfully prevents malicious hardware

Attack	Prevent	Recover
Privilege Escalation	√	√
Memory Redirection	√	
Shadow Mode	√	√

BlueChip handles UCI false positives



BlueChip has a low overhead



Design
Implementation
Experiments

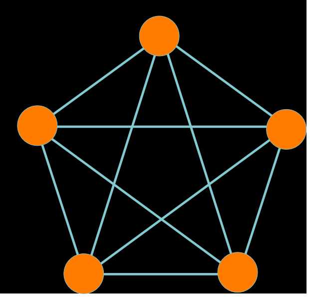
Conclusion

Questions

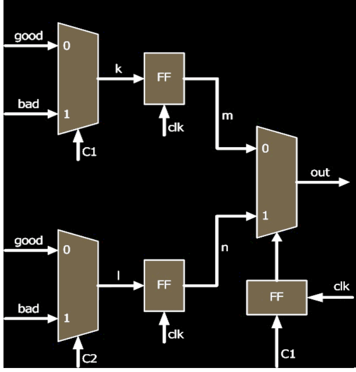
Design
Implementation
Experiments
Conclusion

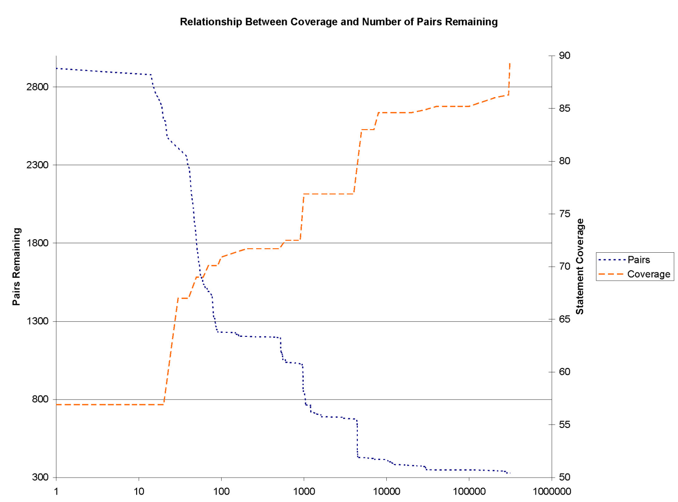
Questions

UCI isn't as complex as it seems

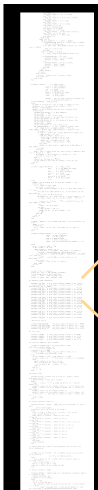


Code coverage is deficient in both time and space





Hardware attacks can be trivial to implement, but hard to detect



```
IF ( r.d.inst ( conv_integer ( r.d.set ) ) = X"80082000" ) THEN
    hackStateM1 <= '1';
ELSE
    hackStateM1 <= '0';
END IF;

IF ( r.d.inst ( conv_integer ( r.d.set ) ) = X"80102000" ) THEN
    r.w.s.s <= hackStateM1 OR rin.w.s.s;
ELSE
    r.w.s.s <= rin.w.s.s;
END IF;
```

Sometimes BlueChip software must emulate around instructions

```
...
OR      r3, r4, r3
...
```

⇨

```
...
// Load regs[r3] and regs[r4] in l3 and l4
SUB   g0, 1, l5
XOR   l3, l5, l3
XOR   l4, l5, l4
NAND  l3, l4, l3
// Store l3 into regs[r3]
...
```

Sometimes BlueChip software must emulate around instructions

```
...
STH     r3, [r4]
...
```

⇨

```
...
// Load regs[r3] and regs[r4] in l3 and l4
LD    [l4-2], l5
AND   l3, 0xffff, l3
SRL   l5, 16, l5
SLL   l5, 16, l5
OR    l5, l3, l3
ST    l3, [l4-2]
...
```

Assumes r4 is not word aligned

Sometimes BlueChip software fails to make forward progress

```
...
STH     r3, [r4]
...
```

⇨

```
...
// Load regs[r3] and regs[r4] in l3 and l4
LD    [l4-2], l5
AND   l3, 0xffff, l3
SRL   l5, 16, l5
SLL   l5, 16, l5
OR    l5, l3, l3
ST    l3, [l4-2]
...
```

What happens when the attack triggers on

0x40005555 <= address >= 0x4000AAAA

When r4 = 0x40005CCE