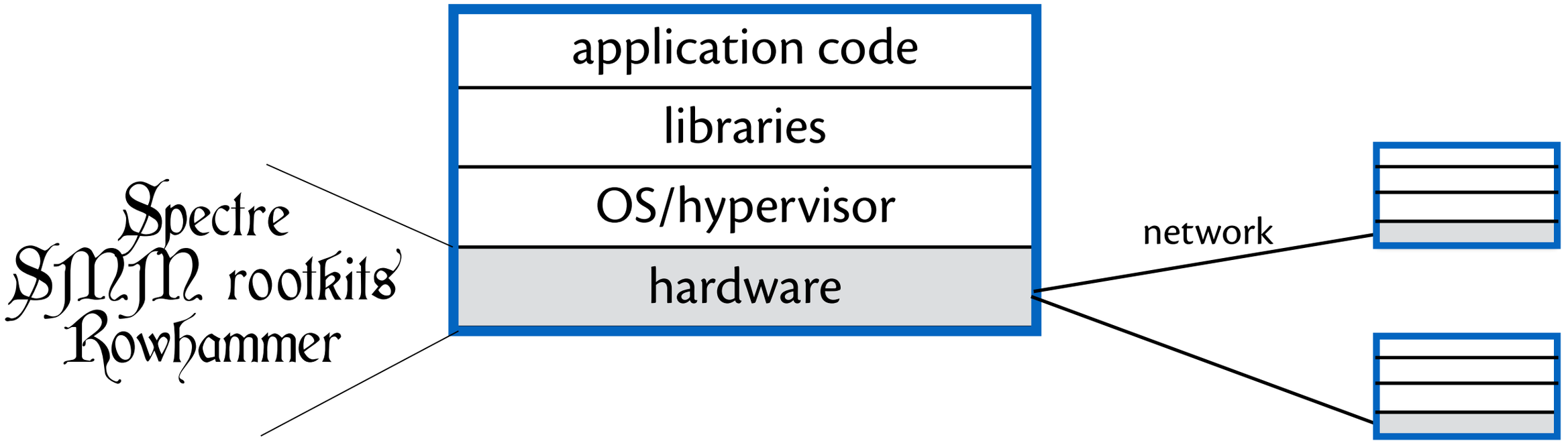
Designing hardware to be free of covert channels by construction

Andrew Myers
Cornell University

(with Ed Suh, Danfeng Zhang, Yao Wang and Andrew Ferraiuolo)

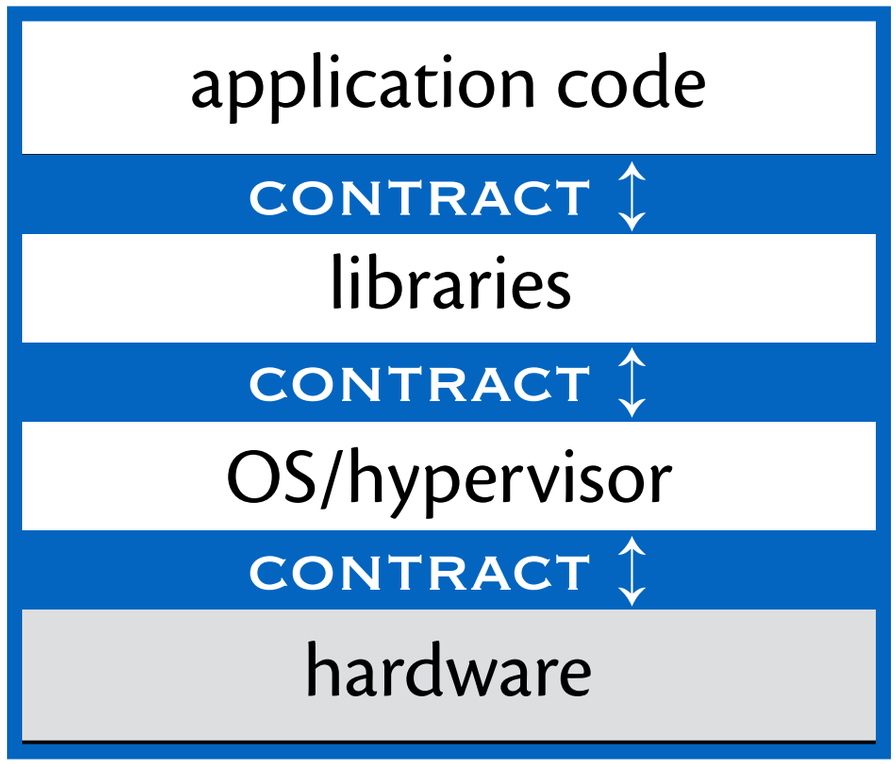
Can we trust the stack?



Got security mechanisms, but:

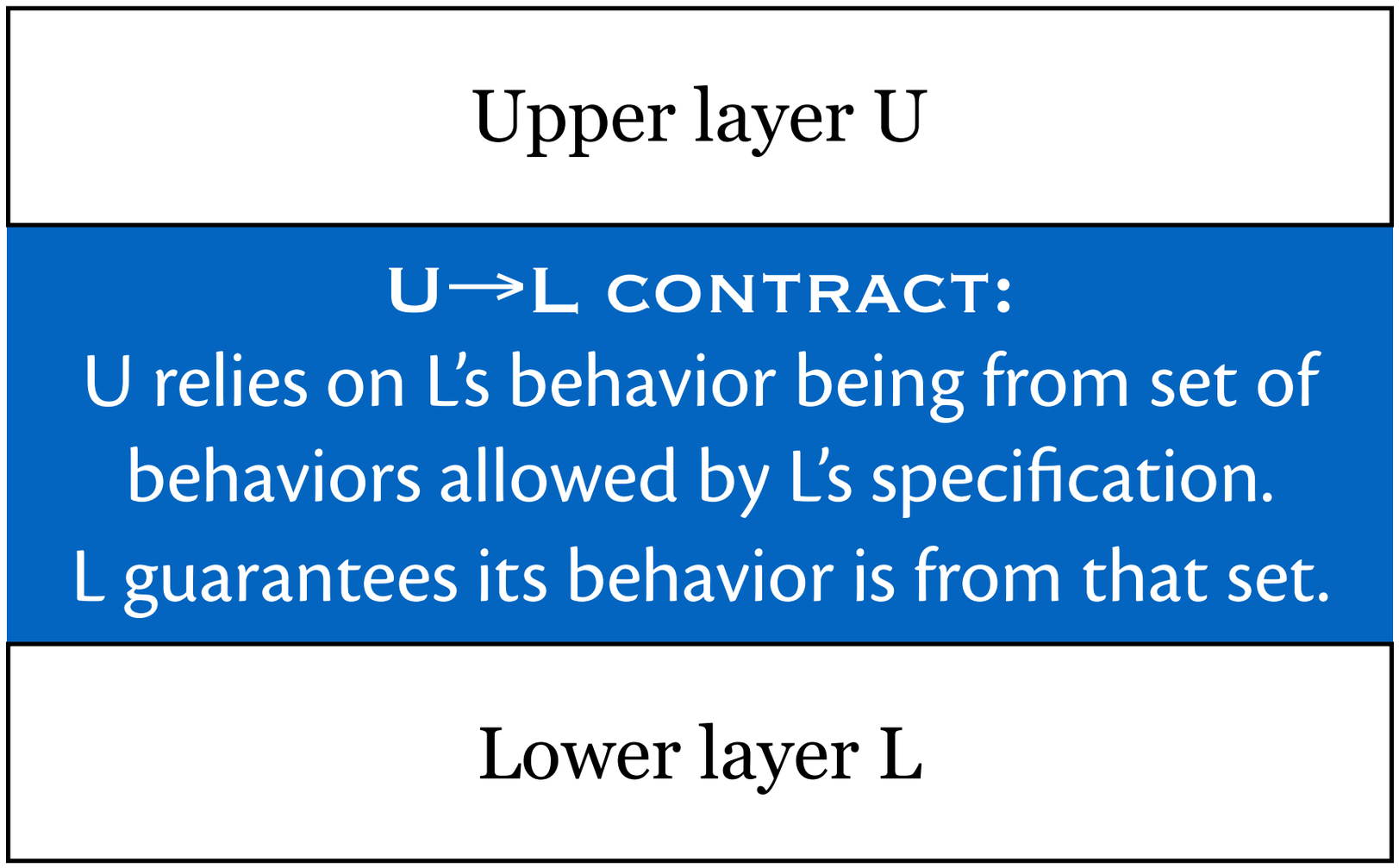
- Modern systems are compositions of complex software & hardware.
- Buggy or malicious code and adversarial data can break security at every level, including hardware

Compositional security?



- How to build layers so their *composition* is secure?
- Need *contracts* between the layers — but what kind?

Contract = Refinement?



- Correctness: each layer's behavior *refines* its spec.
- Compositional
- Commonly used
- **Not strong enough!**

Example: Meltdown/Spectre

- Attacks completely bypass OS memory protection against reads.

Recent reports that these exploits are caused by a 'bug' … are incorrect

- **Intel was right.**

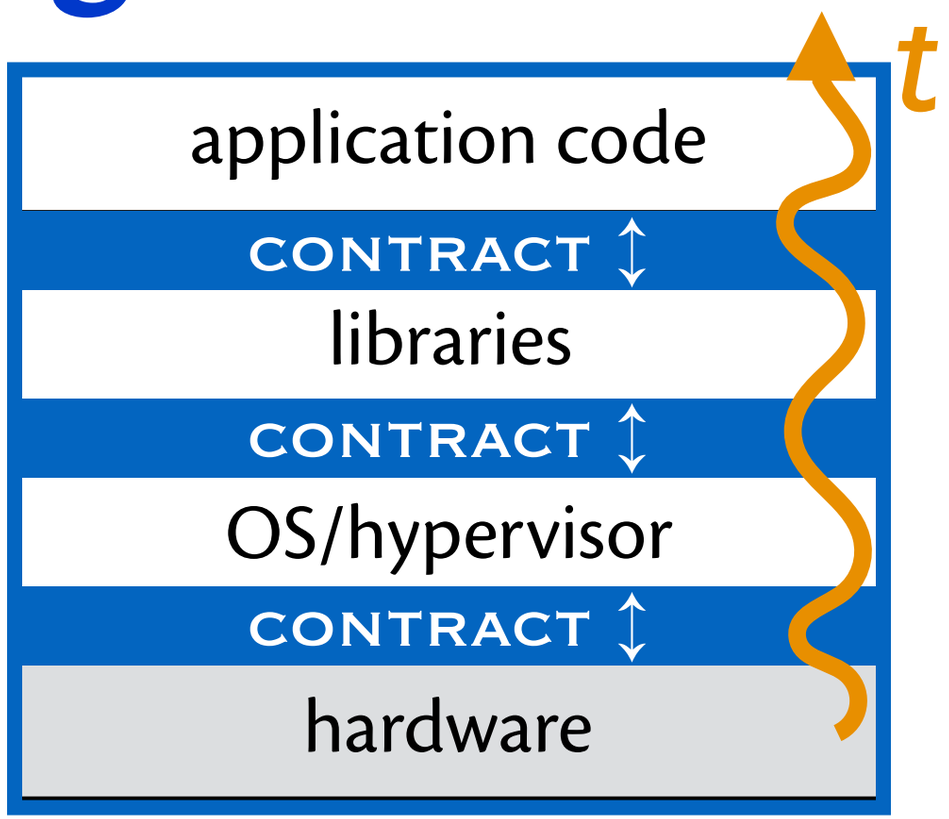
The trouble with refinement

- Processor spec makes no guarantee about time to do a memory read
- Correctness=refinement ⇒ any delay is allowed and is not a ‘bug’.
- But: Meltdown/Spectre *correlate* read delays with contents of inaccessible memory — *a timing channel*

Hyperproperties

- Conventionally, correctness is a **trace property.**
  - Specification gives set of allowed traces; implementation must refine this set
  - Absence of information flow (e.g., on timing channels) is *not* a trace property — it's a **hyperproperty** over sets of possible traces.
- **Spectre shows layer contracts must be — at least — hyperproperties.**

Timing channel control



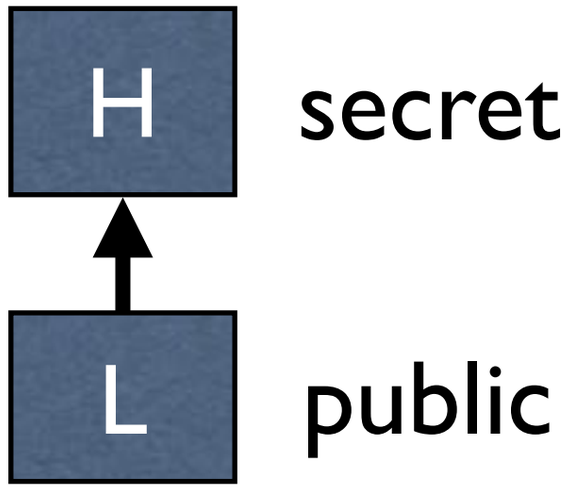
- Abstractions/specs silent about execution time ⇒ vulnerable to timing channels
- How to build layers so that timing channels can't be exploited?

A *language*-based approach

- **Problem:** how can designer know whether there are timing channels?
- **Idea:** static analysis (type system) verifies timing leakage is bounded at every layer

Security lattice

- For now, a simple lattice of security levels:
  - L=public, H=secret, L ⊑ H ⋢ L
- Richer lattices enable multiuser systems and more expressive policies



- Strong adversary ⇒ strong security:
  - Sees everything at level L, e.g., timing of updates to low memory

A timing channel

```
if (h)
  sleep(1);
else
  sleep(2);
```

A subtler example

```
if (h1)
  h2=l1;
else
  h2=l2;
l3=l1;
```

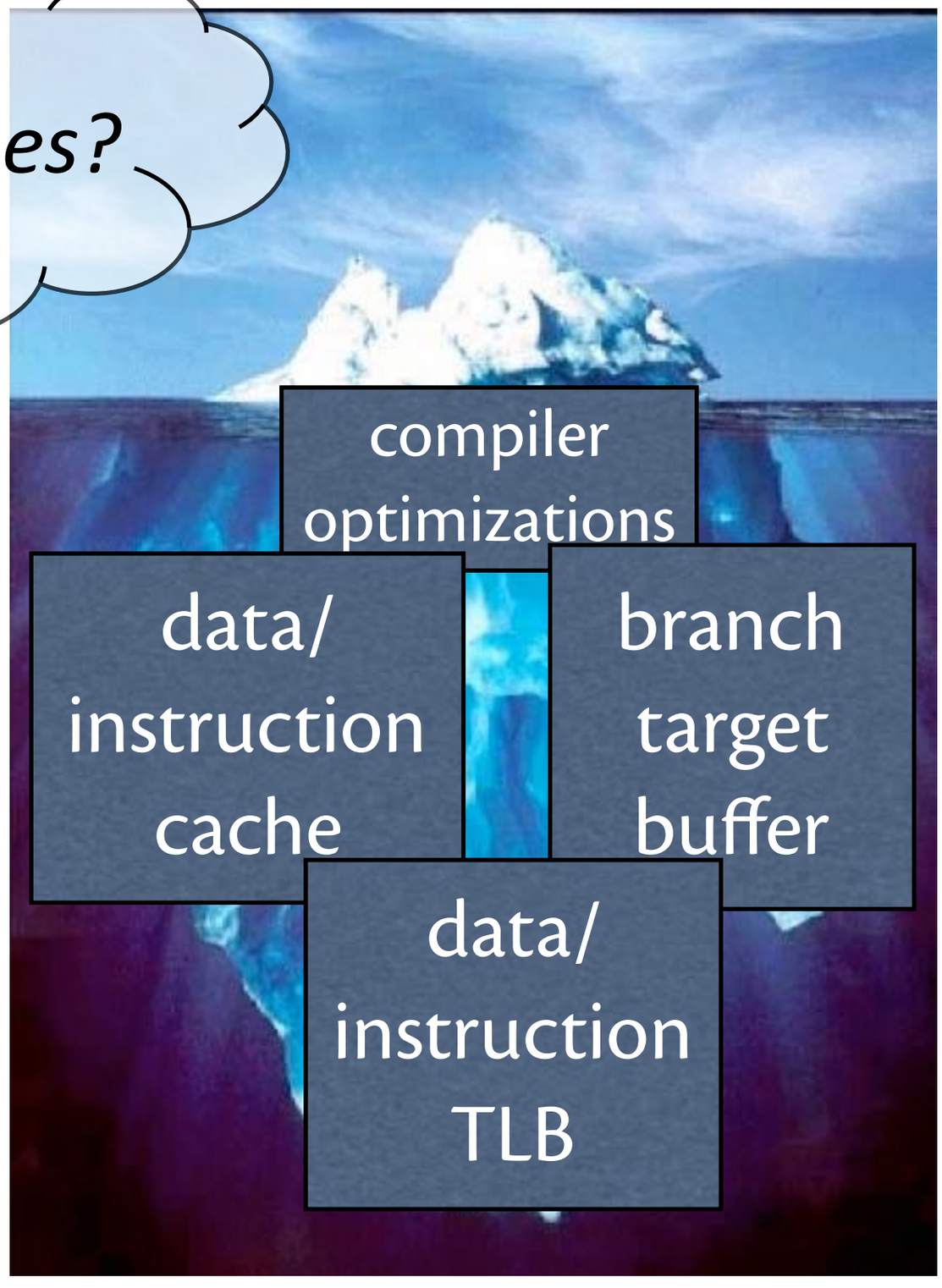
Data cache affects timing!

Beneath the surface

interface?

guarantees?

```
if (h1)
  h2=l1;
else
  h2=l2;
l3=l1;
```



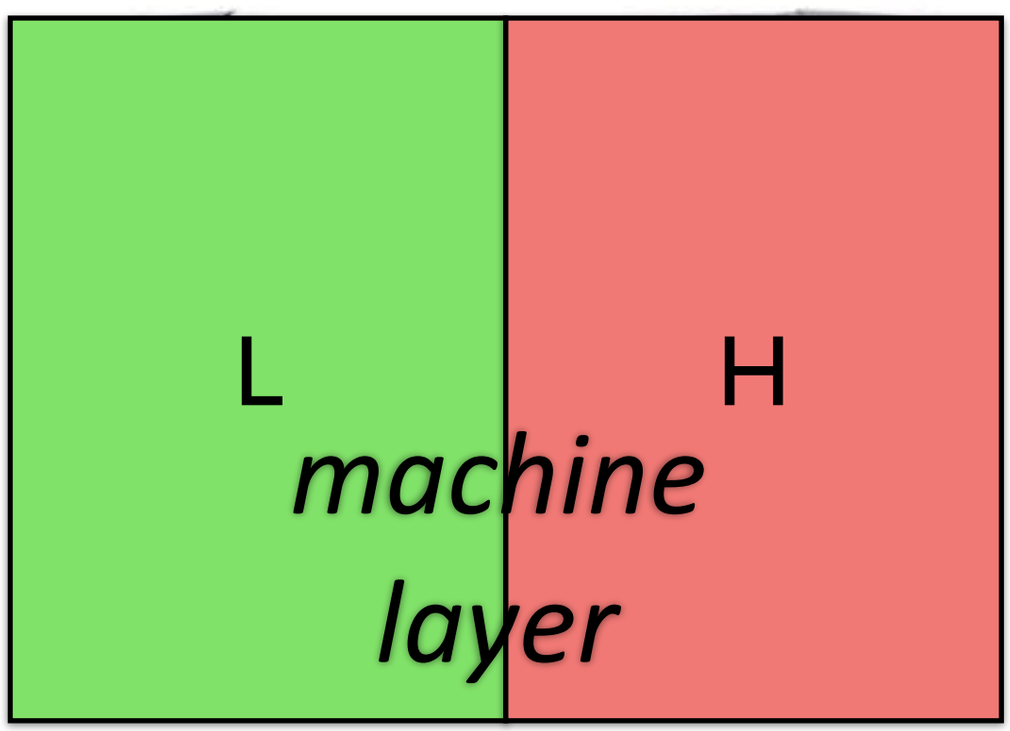
A language-level abstraction

- *Each operation has* **read label**, **write label** *governing interaction with underlying machine*

$$(x := e)_{[\ell_r, \ell_w]}$$

program layer

machine state affecting timing but invisible at language level



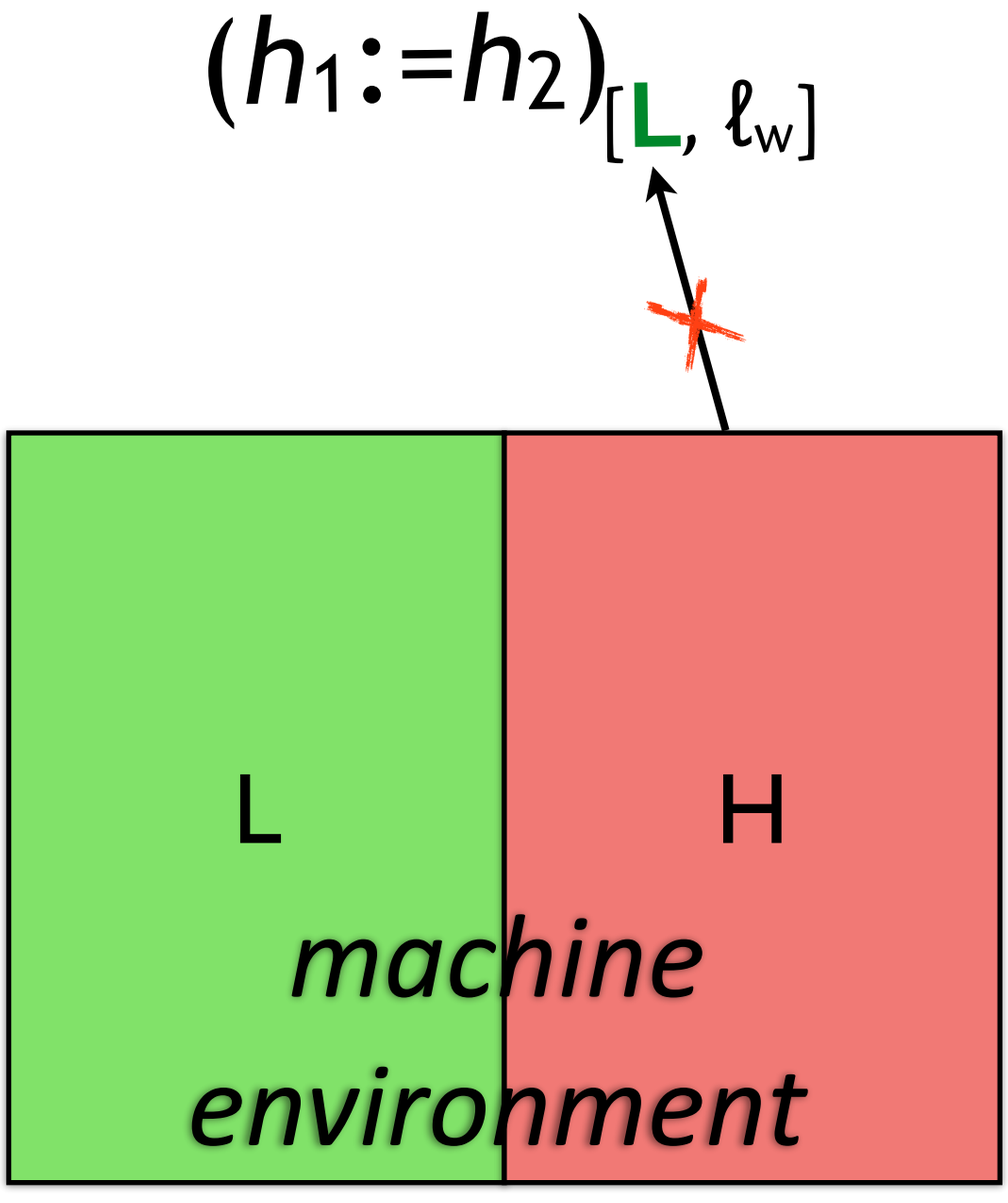
machine layer

Read label

$(x := e)_{[\ell_r, \ell_w]}$

abstracts how machine environment affects time taken by next language-level step.

= upper bound on influence



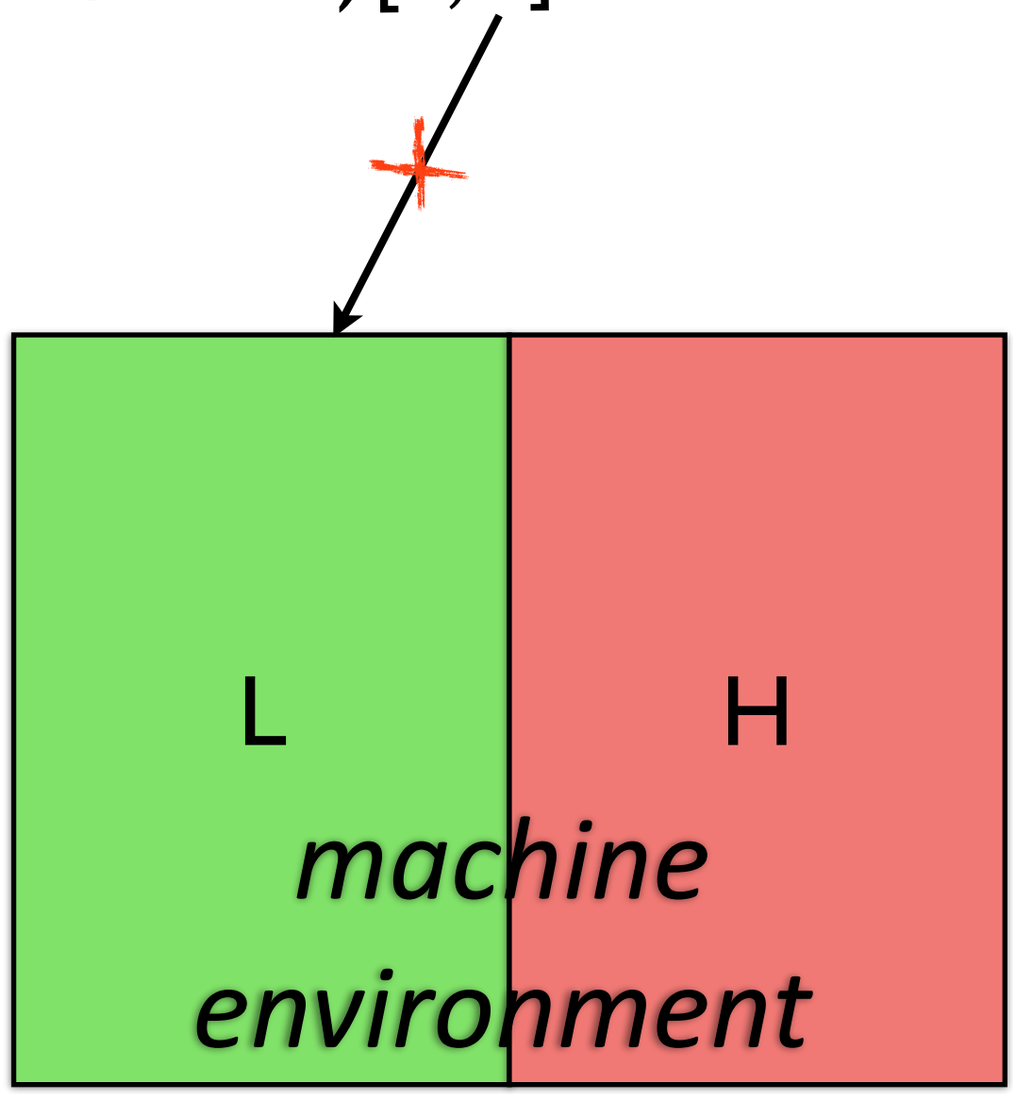
Write label

$(x := e)_{[\ell_r, \ell_w]}$

abstracts how machine environment is affected by next language-level step

= *lower* bound on effects

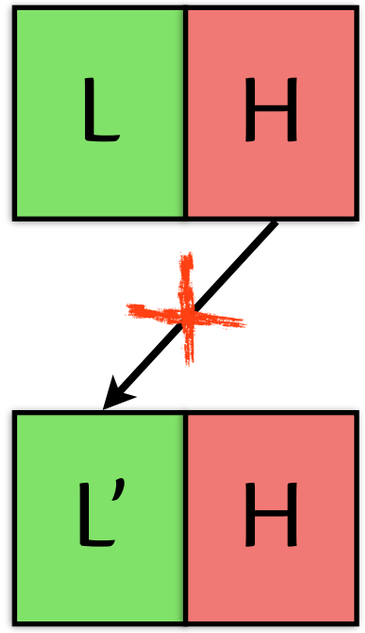
$(h_1 := h_2)_{[L,H]}$



Security properties

- Language implementation must satisfy three (formally defined) properties:

  1. Read label property
  2. Write label property
  3. **Single-step noninterference:** no machine-level leaks from high environment to low



- Provides guidance to compiler writers and designers of future secure architectures

Type system

- We analyze programs using a type system that tracks timing.

  c : T ⇒ time to run c depends on information of (at most) label T

- A "standard" information flow type system, plus read and write labels.
  - Standard part controls data (storage) channels (e.g., forbids l := h)
  - labels can be generated by analysis, inference, programmer...

Examples:

$c_{[H,L]} : H$
$(h_1 := h_2)_{[L,L]} : L$
$sleep(h) : H$

```
if (h1)
  (h2:=l1)[L,H];
else
  (h2:=l2)[L,H];
(l3:=l1)[L,L]
```

low cache read cannot be affected by h_1

Formal results

Memory and machine environment noninterference [PLDI'12]:

Assuming hardware satisfies the contract, a well-typed program* leaks *nothing* via either timing *or* data channels

- Can we express interesting computations as well-typed programs?
- Can we build reasonably efficient hardware that satisfies the contract?

*using no dynamic mitigation

Language-level timing channels

- What about *language-level* timing dependencies?

```
for (i = 0; i < guess.length; i++) {
  if (pwd[i] != guess[i]) return false
}
```

- Sometimes avoidable:

```
for (i = 0; i < MAX_PWD_LEN; i++) {
  count += (pwd[i] == guess[i]);
}
return count == pwd.length;
```

- In general, language-level timing channels cannot be eliminated entirely.

Dynamic timing mitigation

- Idea: *predict* timing to *mitigate* timing leakage [CCS '10, '11]

```
mitigate(l) { s }
```

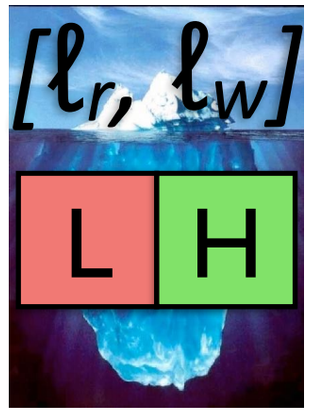
label of running time

mitigated command

- Running time of `mitigate` padded based on predictions using only information at level L.
- Well-typed program running on compliant hardware has bounded leakage (e.g., $O(\log^2 T)$)

Are we done?

- **Read and write labels** provide a contract that controls timing leaks across abstraction layers



- **Information-flow type systems** and **predictive mitigation** can be used to verify that programs don't leak

- **But…** *Can we build hardware that satisfies the contract?*

How to build **efficient** HW that **verifiably** prevents illegal information flows?

Hardware

- Systems increasingly rely on hardware-level protection
  - ARM TrustZone, Intel SGX, IBM SecureBlue

- But are hardware systems trustworthy?
  - Processors are complex and error-prone
  - Hard to spot security issues: e.g., Intel SMM-mode escalation attack [Wojtczuk et al., 2009]

- Need formal security guarantees

Shared HW Leaks Information

- Data cache
  - AES [Osvik et al.'05, Bernstein'05, Gullasch et. al.'11]
  - RSA [Percival'05]
- Instruction cache [Aciiçmez'07]
- Computation unit [Z. Wang&Lee'06]
- Memory controller [Wang&Suh'12]
- On-chip network [Wang et al.'14]

How to prevent the next 700 timing channel attacks?

Secure HDLs

- Idea: add security annotations to hardware description language
  - **SecVerilog** = Verilog + information security annotations [ASPLOS'15, ASPLOS'17, DAC'17]
- Applications:
  - controlling leakage through microarchitectural side channels
  - catching bugs in hardware security architectures (e.g., TrustZone)

SecVerilog

- A general-purpose security-typed hardware description language
  - Lightweight language design (Verilog + labels)
  - Dependent typing ⇒ fine-grained resource sharing
  - Low verification overhead (design-time & run-time)
- Formally proved security guarantees
- Verified MIPS processor and TrustZone implementations

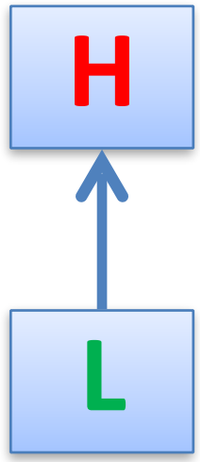
Dynamic vs Static

- **Dynamic enforcement:** propagate labels at run time with information (IX, Asbestos, Histar, Hails, …)
  - for statement $x = y$, where $L_y \not\sqsubseteq L_x$, system halts or assignment is ignored.
    - Weak guidance: security failures⇒run-time failures
    - When coarse-grained ⇒ need to reorganize application
- **Static enforcement:** design verified ahead of time (Jif [POPL'99], FlowCaml, Fabric [SOSP'09], SecVerilog, …)
  - *compiler* checks $L_y \sqsubseteq L_x$
  - but: capturing dynamic behavior may require complex annotations

Security Model

- Attacker sees contents of public HW state at each clock tick

  (synchronous logic)

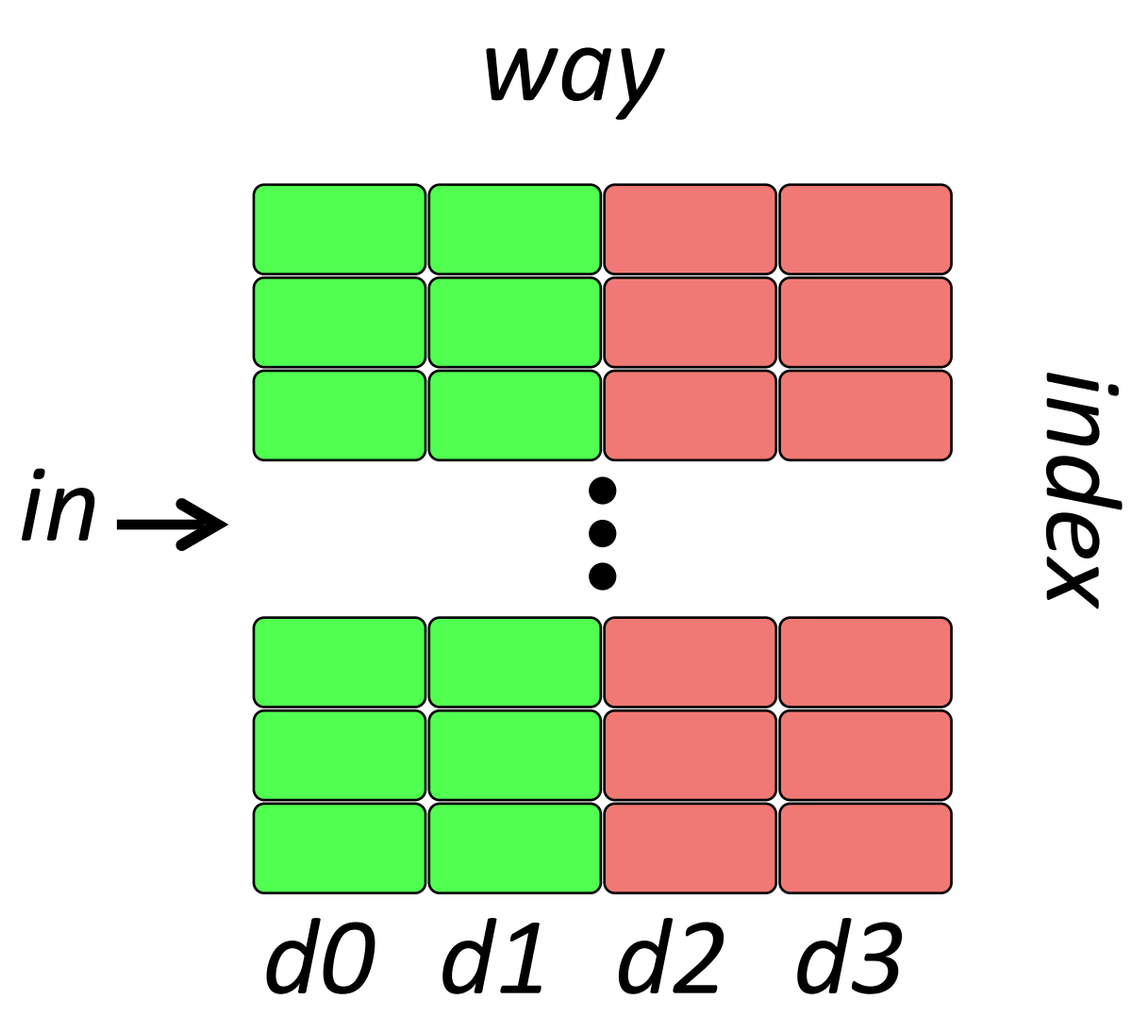


Statically partitioned cache

A 4-way cache in Verilog

```
reg[31:0]      d0[256],d1[256];
reg[31:0]      d2[256],d3[256];
wire[7:0]      index;
wire[1:0]      way;
wire[31:0]     in;

...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```



SecVerilog

= Verilog + security labels

Partitioned cache

```
reg[31:0]{L}   d0[256],d1[256];
reg[31:0]{H}   d2[256],d3[256];
wire[7:0]{L}   index;
wire[1:0]{L}   way;
wire[31:0]     in;

...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```

Annotations on variable declarations

- General
- Few annotations
- Verify HW design as-is

Static labels ⇒ no resource sharing?

```
reg[31:0]{L}   d0[256],d1[256];
reg[31:0]{H}   d2[256],d3[256];
wire[7:0]{L}   index;
wire[1:0]{L}   way;
wire[31:0]     in;

...
case (way)
   0: begin d0[index]=in; end
   1: begin d1[index]=in; end
   2: begin d2[index]=in; end
   3: begin d3[index]=in; end
endcase
...
```

label?

When `way` *= 0 or 1,* `in` *has label* **L**

When `way` *= 2 or 3,* `in` *has label* **H**

SecVerilog

- Verilog + ***dependent*** security labels

An example of partitioned cache

```
reg[31:0]{L}   d0[256],d1[256];
reg[31:0]{H}   d2[256],d3[256];
wire[7:0]{L}   index;
wire[1:0]{L}   way;
wire[31:0] {Par (way)}    in;

...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```

Resource "in" shared across security labels

Using type-level function:

```
Par(0) = Par(1) = L
Par(2) = Par(3) = H
```

Less HW needed for secure designs

A permissive yet sound type system

Soundness

*A well-typed HW design **provably** enforces observational determinism*

L *info. at each clock tick leaks no* **H** *info.*

Permissiveness

Verifies an efficient MIPS processor

Soundness challenges

- Label channels [ASPLOS'15]
- Statically preventing implicit downgrading [DAC'17]
- Enforcing robust declassification and transparent endorsement [CCS'17]

Label Channels

```
reg{L}      p;
reg{H}      s;
reg{LH(x)} x;
if (s) begin x = 1; end
if (x==0) begin
    p = 0;
end
```

Type-level function:

`LH(0)=L` `LH(1)=H`

Change of label leaks information

When p *= 1,*
s *= 0*

p	x
1	0

p	x
1	0

p	x
0	0

When p *= 1,*
s *= 1*

p	x
1	0

p	x
1	1

p	x
1	1

p = s !

No-Sensitive-Upgrade

[Austin&Flanagan'09]

"No update to public variable in secret context"

NSU rejects secure designs

From a real processor design

```
reg{H}      hit2, hit3;
reg[1:0]{Par(way)} way;
if (hit2 || hit3)
  way ⇐ hit2 ? 2 : 3;
else
  way ⇐ 2;
```

(incorrectly) rejected

Insight: Label of `way` is always **H** after branch

Solution: definite assignment

No update to public variable in secret context,
if the variable is not updated in all branches

```
reg{H}      hit2, hit3;
reg[1:0]{Par(way)} way;
if (hit2||hit3)
 way ⇐ hit2 ? 2 : 3;
else
 way ⇐ 2;
```

(correctly) accepted

Also more permissive than
flow-sensitive systems *[Hunt&Sands'06, Russo&Sabelfeld'10]*

Precision of dependent labels

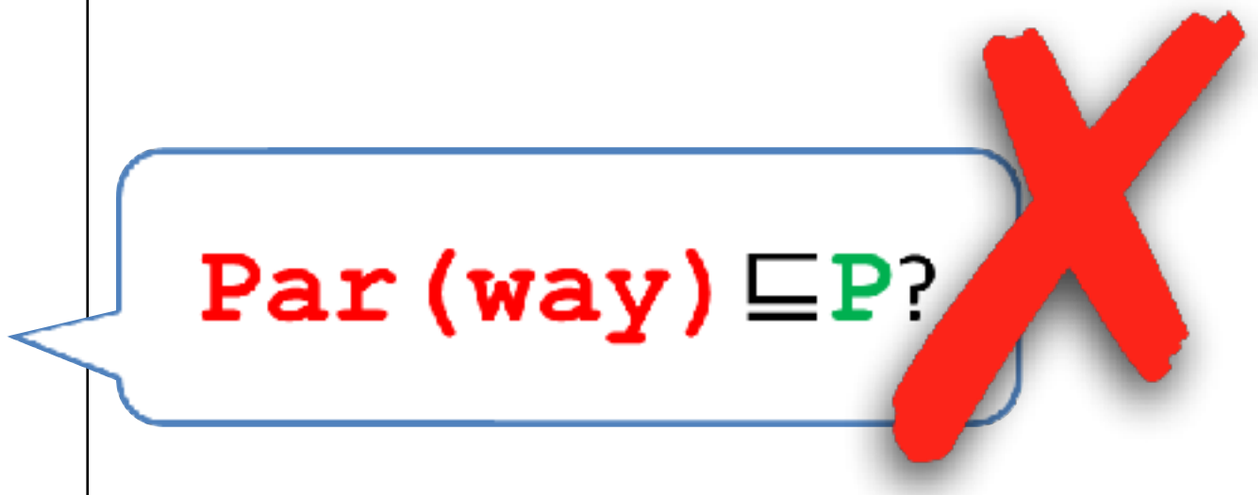
```
reg[31:0]{L}  d0[256],d1[256];
reg[31:0]{H}  d2[256],d3[256];
wire[7:0]{L}  index;
wire[1:0]{L}  way;
wire[31:0] {Par (way)}   in;

...
case (way)
  0: begin d0[index]=in; end
  1: begin d1[index]=in; end
  2: begin d2[index]=in; end
  3: begin d3[index]=in; end
endcase
...
```

Type-level function:

Par(0)=Par(1)=L

Par(2)=Par(3)=H



Predicate generation

P(c) : a predicate that holds before c executes

```
reg[31:0]{L}   d0[256],d1[256];
reg[31:0]{H}   d2[256],d3[256];
wire[7:0]{L}   index;
wire[1:0]{L}   way;
wire[31:0] {Par (way)}    in;

...
case (way)
   0: begin d0[index]=in; end
   1: begin d1[index]=in; end
   2: begin d2[index]=in; end
   3: begin d3[index]=in; end
endcase
...
```

$P(c)$: (way = 0)

Type-level function:

Par(0)=Par(1)=P

Par(2)=Par(3)=S

Par(way) ⊑ L

when way=0?

Approximated by propagating postconditions

Soundness

Permissiveness

Type system

Other analyses

Variables not always updated

Predicate generation

Typing obligations discharged using Z3 SMT solver.

Formally verified MIPS processor

Rich ISA: runs OpenSSL with off-the-shelf GCC

- extended with instruction to set current security level

Classic 5-stage in-order pipeline

- Typical pipelining techniques
  - data hazard detection
  - stalling
  - data bypassing/forwarding

Overhead of SecVerilog

- Verification time:

  2 seconds for complete MIPS processor

- Designer effort
  - Annotation burden:
    one label/variable declaration (mostly inferable, as shown in forthcoming work)
  - Imprecision leads to little extra logic:
    27 LoC to establish necessary invariants

Overhead of secure processor

- Added HW resources
- Performance overhead on SW

Overhead of verification

Believed secure but not verified

	Unverified	Verified	Overhead
Delay w/ FPU (ns)	***4.20***	***4.20***	***0%***
Delay w/o FPU (ns)	***1.67***	***1.66***	***-0.6%***
Area (μ²)	***401420***	***402079***	***0.2%***
Power (mW)	***575.6***	***575.6***	***0%***

Verification overhead is very small!

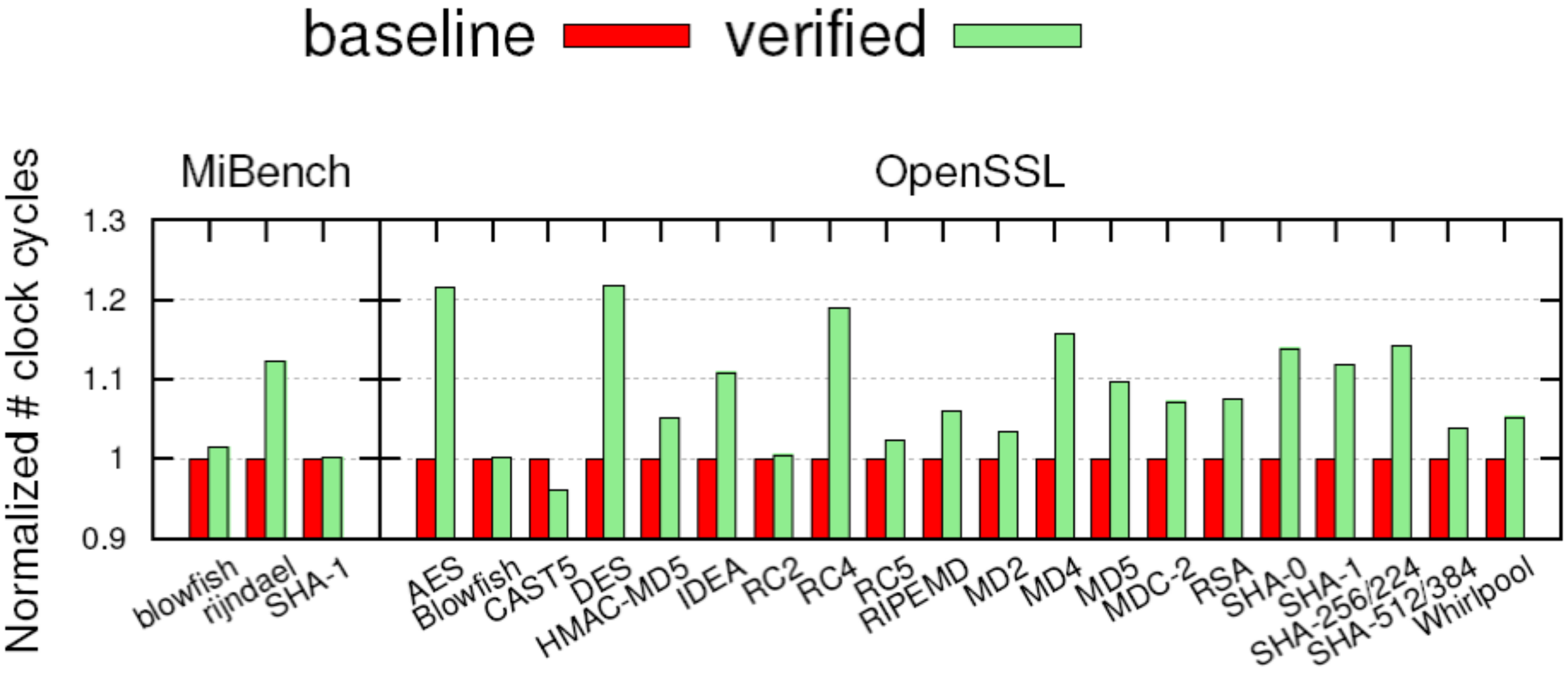
Overhead of secure processor (HW)

unmodified, insecure

	Baseline	Verified	Overhead
Delay w/ FPU (ns)	***4.20***	***4.20***	***0%***
Delay w/o FPU (ns)	***1.64***	***1.66***	***1.21%***
Area (μ²)	***399400***	***402079***	***0.67%***
Power (mW)	***575.5***	***575.6***	***0.02%***

Enabled by the SecVerilog type system

SW-level overhead



9% overhead on average

same cache area ⇒ smaller effective cache

Prior HDL-level info flow control

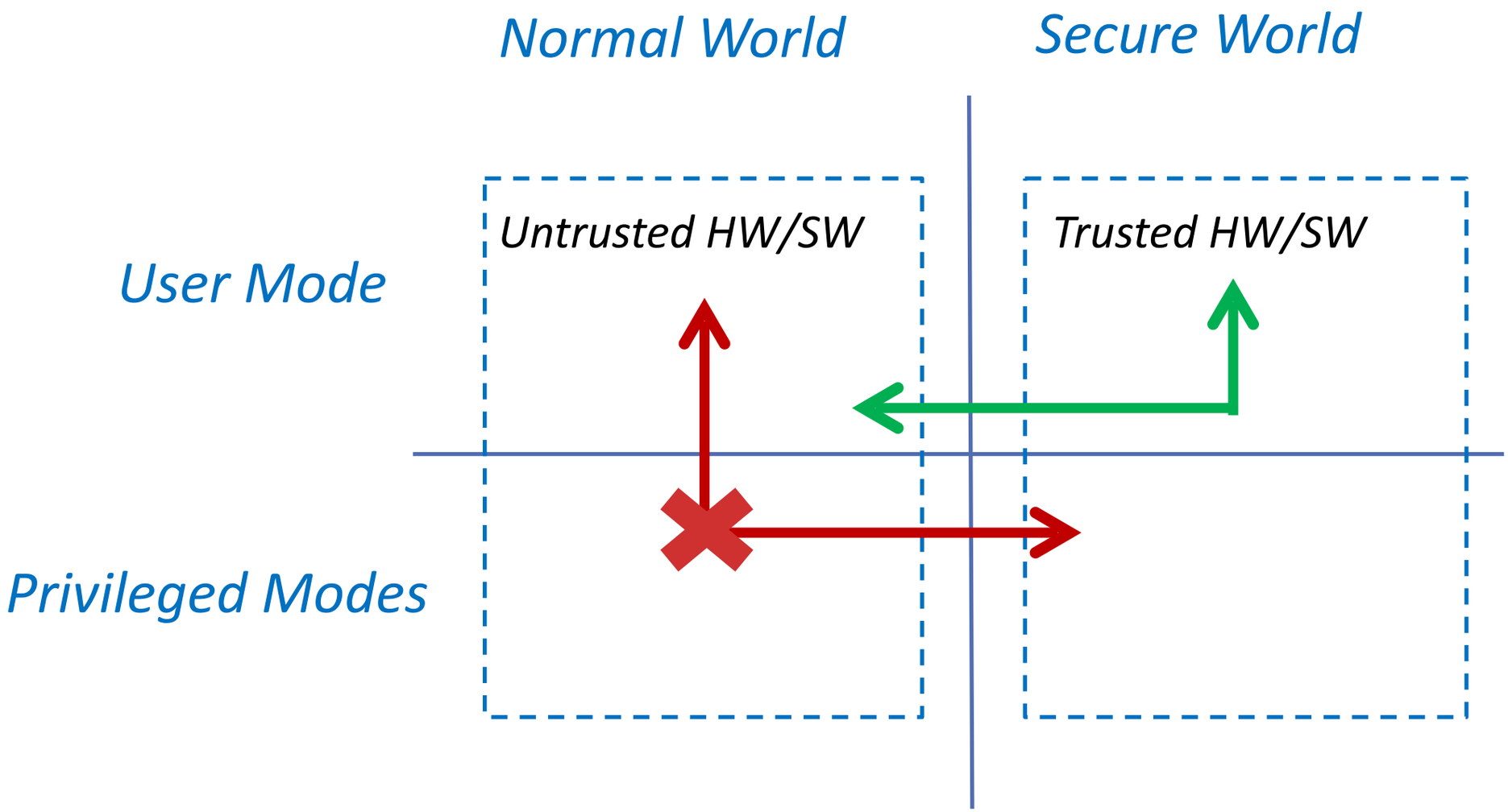
	Lightweight design	Fine-grained resource sharing	Low verification overhead	Security bugs change run-time behavior
Caisson ***[Li et al.'11]***	✗	✗	✓	✓
Sapper ***[Li et al.'14]***	✗	✓	✗	✗
SecVerilog	✓	✓	✓	✓ **[DAC'17]**

Implementing TrustZone

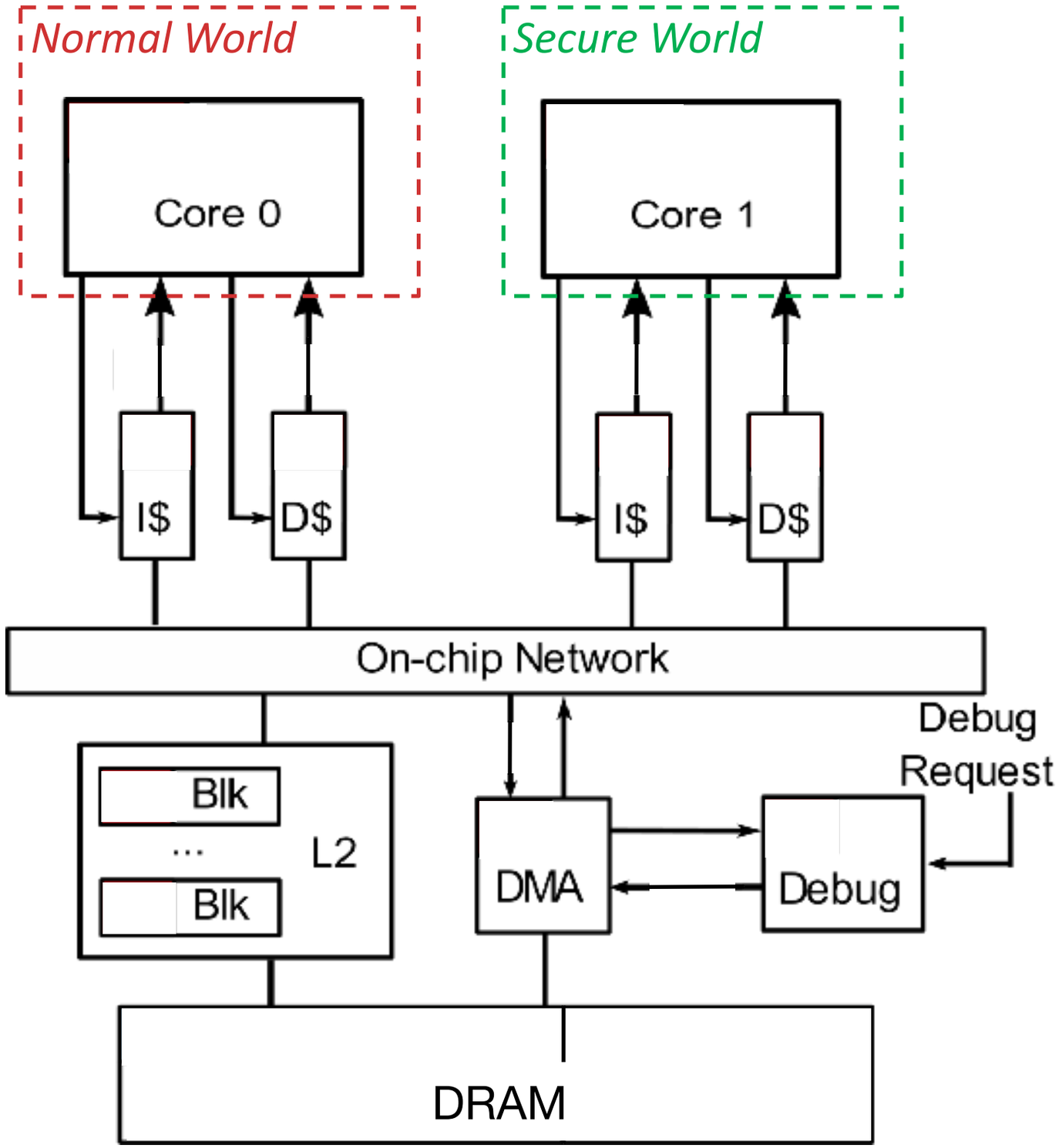
- Goal: map security requirements of a practical processor arch to IFC.
  - Multi-core RTL prototype of ARM TrustZone
  - Demonstrate that security bugs can be caught
  - Low overhead
- HDL type system extensions
  - Heterogeneous security labels for arrays and vectors
  - Downgrading to permit communication

ARM TrustZone

- Normal world: only accesses normal-world data
- Secure world: can access data in either world



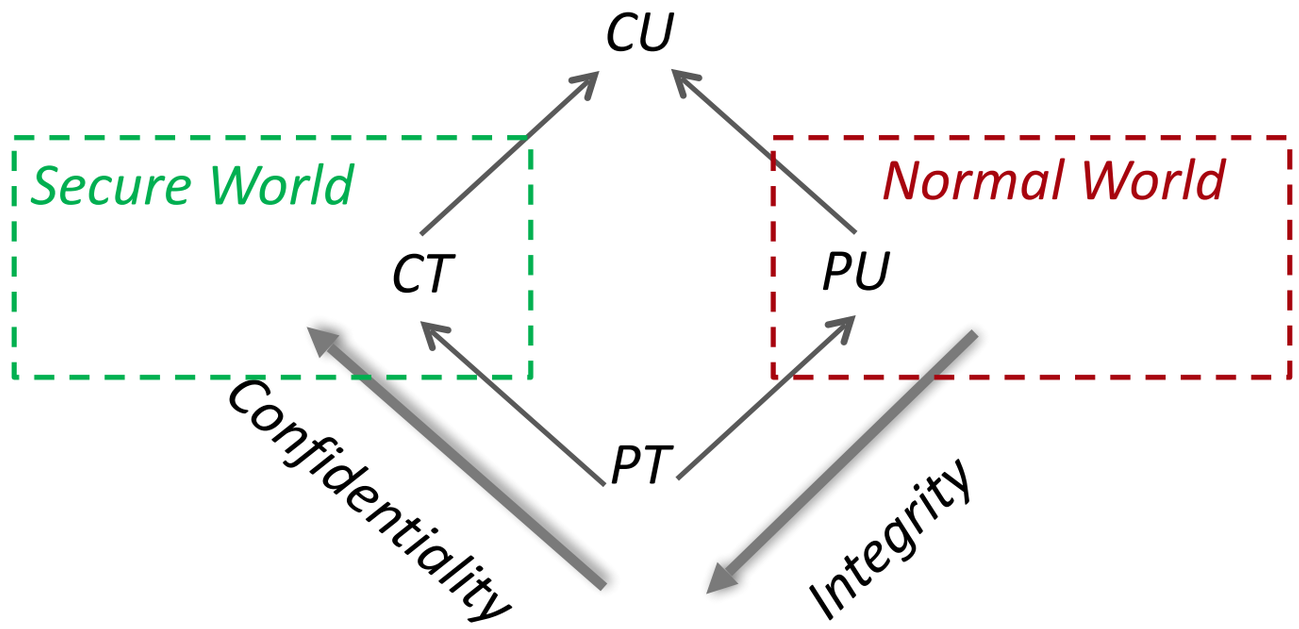
TrustZone Prototype Implementation



- NS bit indicates world
- Cache blocks have an NS bit
- Network transactions are appended with the NS bit
- DRAM is partitioned
- Access control modules enforce security

TrustZone as an Information Flow Policy

- Policy enforces integrity and confidentiality protection



- Secure world is CT, Normal World is PU
- Control registers and the NS bit are labeled PT.
- Policy mismatch with TrustZone spec, where secure world can access normal world (secure-world SW must be careful!)

Language Extension: Bit Vector Types

- Bit vectors are a convenient hardware data structure
  - Security information is lost when bits are grouped
- Solution: types that are functions describing each bit's level

```
1 wire [0:31 ] {CT} sw_data;
2 wire [32:41] {PT} addr;
3 wire [0:41 ] {i -> if (i <= 31) CT PT} packet;
4
5 assign packet = {addr, sw_data};
```

- Type Rules:
  - Precisely capture per-bit label propagation
  - Enforce security policy for each bit

Language Extension: Array Support

- Describing cache blocks: needed to unroll the array

```
145 reg {PT} block_lbl_23;
146 reg {world(block_lbl_24)} block_34;
147 reg {PT} block_lbl_24;
148 reg {world(block_lbl_24)} block_24;
```

```
1 ...
2 reg {PT} block_lbl [0:1023];
3 reg [0:31] { i -> j -> world(block_lbl[i]) } block[0:1023];
4 ...
```

```
153 reg {PT} block_lbl_27;
154 reg {world(block_lbl_27)} block_27;
155 reg {PT} block_lbl_28;
156 reg {world(block_lbl_28)} block_28;
```

- More expressive dependent labels avoid unrolling

Downgrading

- Information flow analysis reveals potentially dangerous flows
  - Secure-world writes to control registers
  - But: overly restrictive

- *Downgrading* – release of information
  - Like typecasts: `downgrade(expr, label)`
  - Potential problems are limited to downgrades

Security Results

- Extended type system – same security as original SecVerilog
- The processor type-checks…
- Downgrading relaxes noninterference
  - …only in the secure world
  - We audit and categorize each use of downgrading.

Security Vulnerability Detection

- Implemented 9 hardware vulnerabilities
  - 3 modeled on real-world vulnerabilities:
    - Backdoor in Actel ProASIC3 [Sergei et al., CHES 2012]
    - Security-critical AMD errata [Hicks et al., ASPLOS 2015]
    - Intel SMM-mode [Wojtczuk et al., 2009]
- Only undetected bug was designed to thwart type system:
  - Uses downgrading incorrectly, adds a nonsensical constant to an address

Overheads

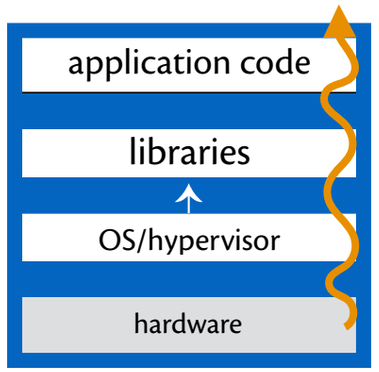
- Programmer effort in lines of code:
  - Unverified: 16234
  - Verified: 16700
  - Overhead: 2.9%
- Hardware overheads:
  - Clock frequency and CPI unchanged
  - The area and power overheads are negligible (0.37% and 0.32%)

HDL information flow?

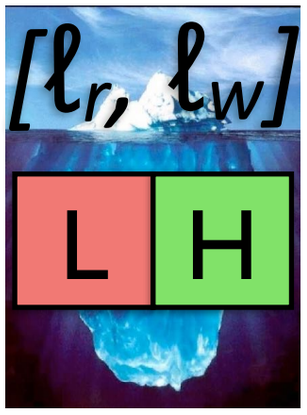
- Seems to be effective way to gain security assurance for hardware designs
  - A lightweight development methodology that allows building efficient hardware with verified properties
  - Implemented a MIPS processor verified to have no timing channels or other leaks
  - Implemented a multicore prototype of ARM TrustZone and detected vulnerabilities found in commercial processors

Conclusions

- Want trustworthy stack of abstractions? Need new kinds of contracts — beyond safety and liveness to hyperproperties



- Timing channels can be controlled with static analysis at the language level — if hardware obeys a contract



- Timing channel contracts and other policies can be enforced at the hardware level by a security-typed HDL, with reasonable overhead