Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing

Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, Christopher W. Fletcher

University of Illinois at Urbana-Champaign

Intel Research Tech Talk, January 22nd 2019

To appear in NDSS'19

https://eprint.iacr.org/2018/808

Outline

- Introduction
- Threat Model & Security Definition
- Oblivious ISA Design
- Microarchitecture
- Security Argument
- Evaluation
- Conclusion

Microarchitectural side channel attacks = huge privacy threat

- Many HW structures leak
  - Branch predictor
  - Arithmetic units
  - 4K Aliasing
  - Speculative execution
  - …

New ones everyday it seems.

How can programmers write software to block *all* side channels?

Observation

- Many parallel efforts try to achieve holistic protection by writing programs ***data obliviously***
- *A.K.A.:*
  - *"constant time programming"* — systems community
  - *"data oblivious programming"* — applied crypto community
  - *"writing programs in the circuit abstraction"* — pure crypto community
  - …

Data Oblivious Programming

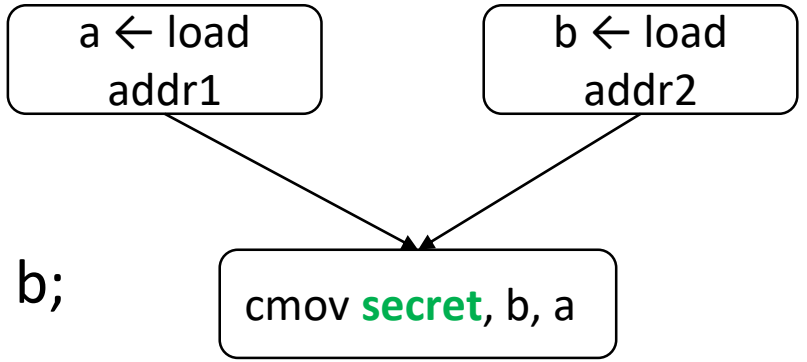
- Run a program "as a circuit" w/ "static topology"
- Security
  - Evaluating each gate (instruction) is data-independent
  - The order of evaluating gates is data-independent

```
if (secret)
    a = *(addr1);
else
    a = *(addr2);
```

Program level

```
a ← load (addr1);
b ← load (addr2);
cmov a = (secret) ? a : b;
```

Machine code



"circuit"
instructions = gates
dependencies = wires

In principle, data oblivious programming can block all side channels

Problems w/ Data Oblivious + Processors Today

Security

- Data dependent HW optimizations break correctly written data oblivious code

  Example: cmov micro-coded into branch+mov (paper has 11 examples)

Portability

- Code not portable between processors

Efficiency

- Simple operations → many instructions; **E.g.,** scan memory to perform a read
- Transformation to circuit is expensive; **E.g.,** execute both sides of branch

Solution: Data Oblivious ISA (OISA) Extensions

Security

- Security guarantee per-instruction at ISA level
- Security requirements visible to HW architects. → can disable leaky optimizations

Portability

- ISA fixed across implementations

Efficiency

- Key insight: Specify security at ISA-level → Asymptotically more efficient code
- Idea is related to CISC: Instead of scanning memory, give me a “secure load”

Major theme:

Decouple security guarantee/threat model from implementation

Threat Model & Security Definition

Threat Model

- Victim/attacker processes co-locate to same machine
  - Victim code is public, HW is trusted
- Attacker = Ring-3 or Ring-0 (e.g., SGX attacker)
  - Attacker anywhere on chip (SMT context, another core, within same thread)
- Attacker's goal: learn Victim data
- Security goal (informal):
  - Victim written in OISA mitigates all microarchitectural/digital side channels
  - Speculative and non-speculative channels
- Non-goals: integrity, physical side channels, Rowhammer

Security Definition: What does it mean to mitigate all side channels?

Goal: "Computational indistinguishability/non-interference" of program traces

- For ∀ public input x, ∀ confidential input y, y':

$$Obs(uArch(P(x, y))) \cong Obs(uArch(P(x, y')))$$

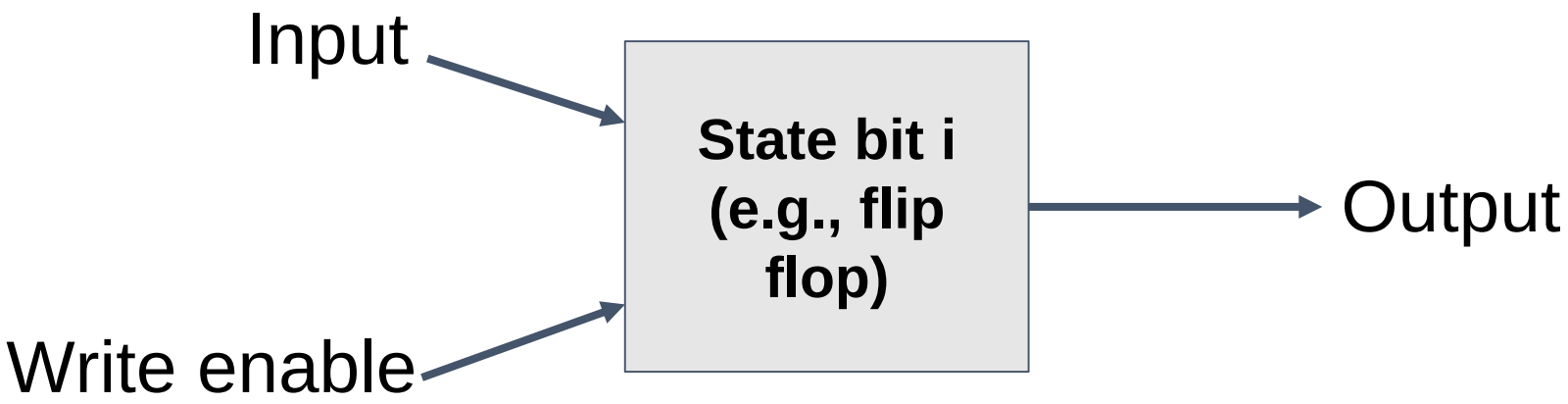
- P() is the program
- uArch() is a specific uArch implementation
- Obs() models what attacker can observe

What is the Observability Function Obs?

- We consider Obs = *visibility at bit granularity, in each clock cycle ("BitCycle")*
  - Let program run for t cycles
  - We have: BitCycle(uArch(P(x, y))) = $\{X_0, X_1, \ldots X_t\}$
  - X_t^i = does i-th processor memory cell contains "resource pressure" in cycle t
  - Resource pressure = "explicit flow of confidential data"

- Note: Implicit flows accounted for when we quantify over y

What is Explicit Flow at the Hardware Gate Level?



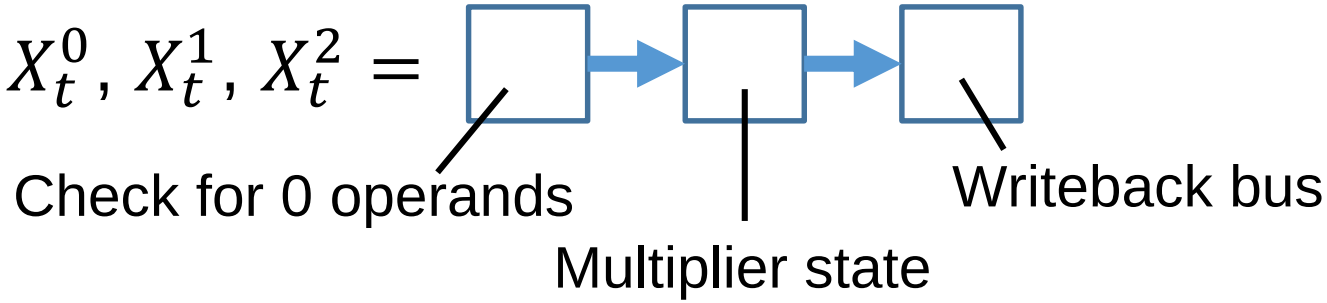
State bit i has explicit flow in cycle **t** if (a) or (b) holds

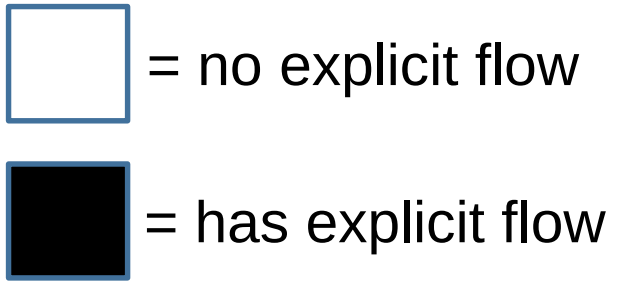
a.) Write enable is **false** in **t-1** and there was explicit flow in previous cycle

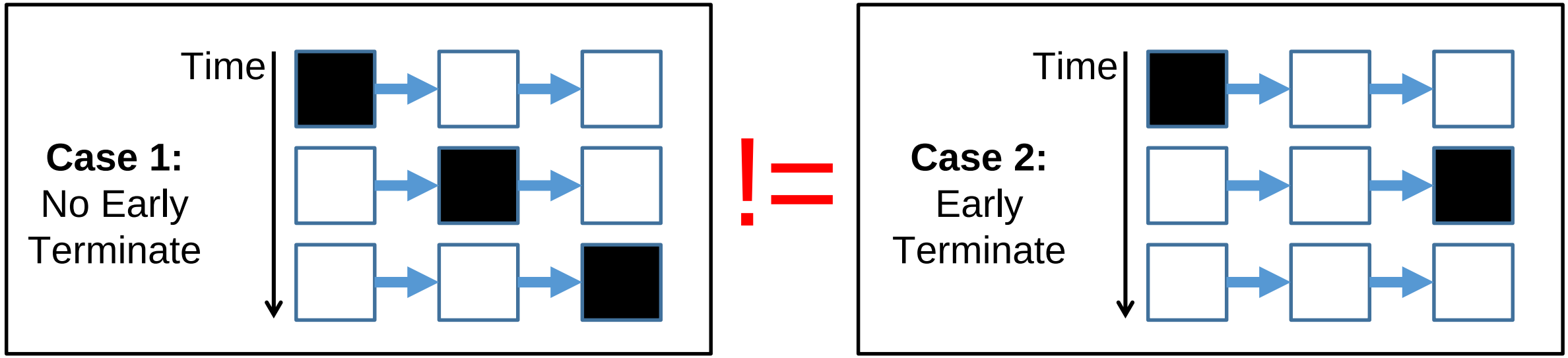
b.) Write enable is **true** in **t-1** and either Input/Write enable contain explicit flow

Also see GLIFT papers

Example: Early terminating multiplier







Data Oblivious ISA Extensions

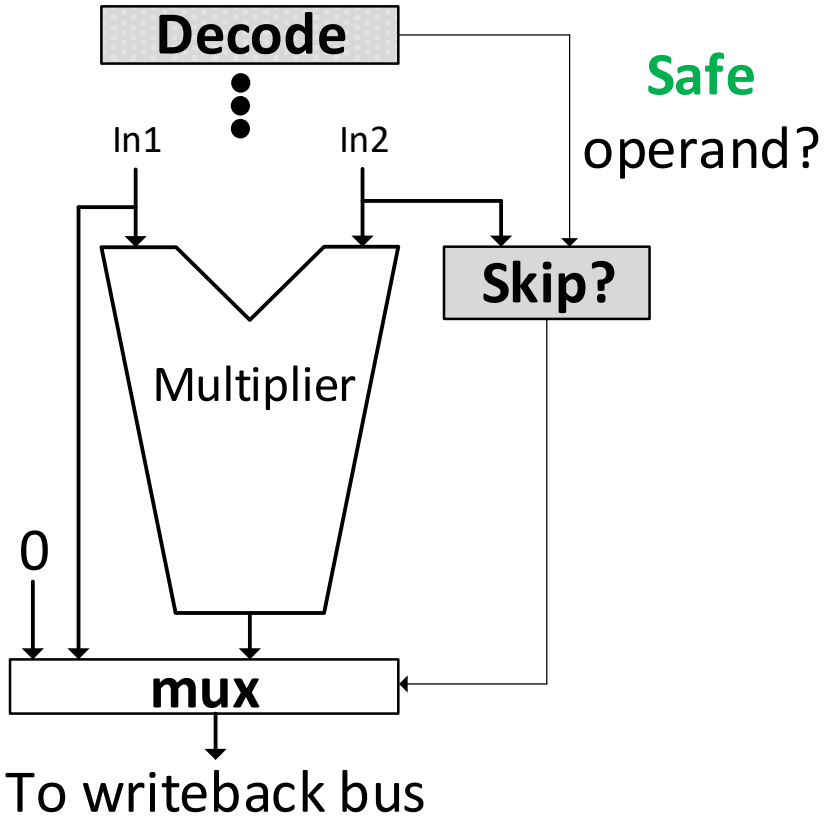
Starting Point: mark instruction operands "Safe" or "Unsafe" at ISA level

- **Safe operand:** block side channels stemming from that operand
  = Result in indistinguishable traces (e.g., with BitCycle)
- **Unsafe operand:** no protection
- Safe/Unsafe doesn't specify implementation strategy
  - HW people happy
- Safe/Unsafe doesn't expose microarchitectural details
  - SW people happy

Example: Early terminating multiplier

- Two flavors:
  - Mul (original instr): Unsafe operand
  - OMul (oblivious): Safe operand

- Decode logic tells Multiplier HW to disable optimizations



Which instructions can have Safe operands?

Strawman proposal:

- **Arithmetic** w/ ***Safe*** operands
  - Implement by disabling data-dependent optimizations
- **Branches/jumps** w/ ***Unsafe*** operands
  - Cannot implement *Safe* predicate w/o solving the halting problem
- **Load/Stores** with ***Unsafe*** address, ***Safe*** data
  - Implement by disabling data-centric optimizations (e.g., silent stores)

Safe Operands → Performance Improvement

Leaky program:

```
if (secret)
    a = *(addr1);
else
    a = *(addr2);
```

Strawman Oblivious ISA:

Name	Specification
ORload	(Unsafe) address
Ocmov	(Safe) pred, (Safe) src, dst

```
a ← ORload (addr1);
b ← ORload (addr2);
Ocmov a = (secret) ? a : b;
```

Optimized Oblivious ISA:

Name	Specification
OCload	(Safe) address
Ocmov	(Safe) pred, (Safe) src, dst

```
Ocmov addr =
    (secret) ? addr1 : addr2;
a ← OCload (addr)
```

How to Implement OCload w/ *Safe* address

- Multiple implementation options; different trade-offs
- Array size N

Implementation	Efficiency
Micro-code into simpler oblivious instructions (e.g., loads w/ **Unsafe** address)	O(N)
Hardware partitioning (e.g., cache partitioning, private scratchpads)	O(1), size restricted
Cryptographic techniques (e.g., Oblivious RAM)	O(log N) or O($\log^2$ N)

- Detailed design in paper

More Opportunities for Oblivious "CISC" Instructions

- Sort = common data oblivious kernel

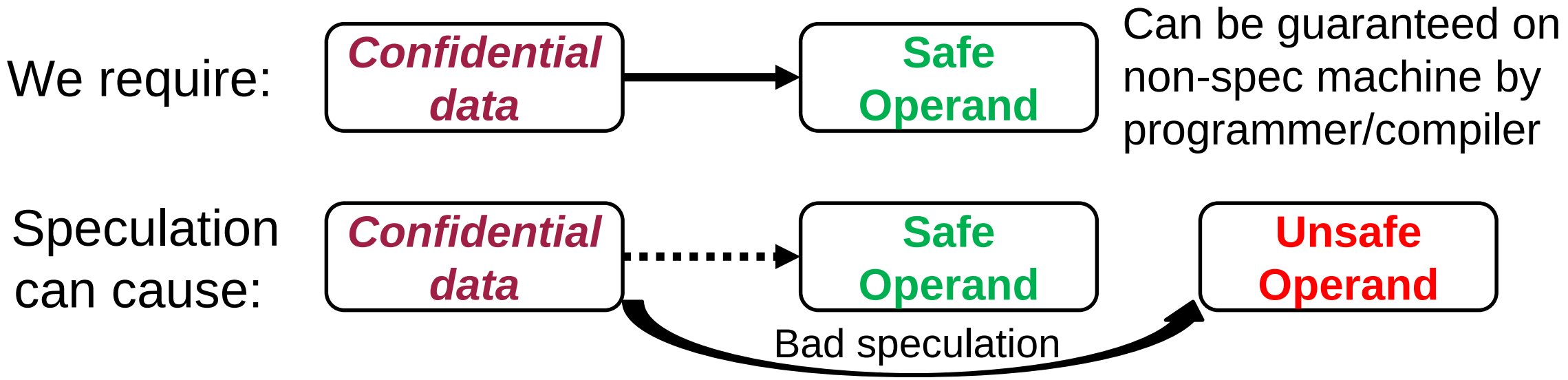
- Array size N
- Strawman data oblivious = $O(N * \log^2 N)$ (bitonic sort)
- Sort instruction, **Safe** operand = $O(N * \log N)$

(constant time merge sort)

Are Safe Operands Enough?

Almost, but no.

Issue 1: Speculative execution



Issue 2: Data-centric optimization (e.g., cache compression)

Adding Dynamic Information Flow Tracking

- Hardware tracks confidential data dynamically **pre**&post-retirement

  Example Policy:

  - ***Confidential*** **OP** ***Confidential/Public*** → result is ***Confidential***
  - ***Public*** **OP** ***Public*** → result is ***Public***

- **Speculation:** Hardware blocks leakage dynamically

- **Data-centric optimizations:** Tags follow data at rest

Complete Proposal: Safe/Unsafe + DIFT

1. ISA design time: ISA designers decide whether each instruction operand is ***Safe***/***Unsafe***.

2. Programming time: Programmers annotate program inputs/static data ***Public***/***Confidential***.

3. Runtime: Hardware tracks flows using DIFT. Processor implements transition rules during execution:

- ***Public*** data → ***Safe*** operand: Execute w/o protection
- ***Public*** data → ***Unsafe*** operand: Execute w/o protection
- ***Confidential*** data → ***Safe*** operand: Execute w/ protection
- ***Confidential*** data → ***Unsafe*** operand: HW exception

Hardware Prototyping

Prototype on RISC-V BOOM v2

- OoO, super-scalar, speculative RISC-V
- Threat vectors in BOOM v2
  - Branch/jump speculation
  - Lazy memory disambiguation
  - Data-dependent arithmetic
  - Banked caches

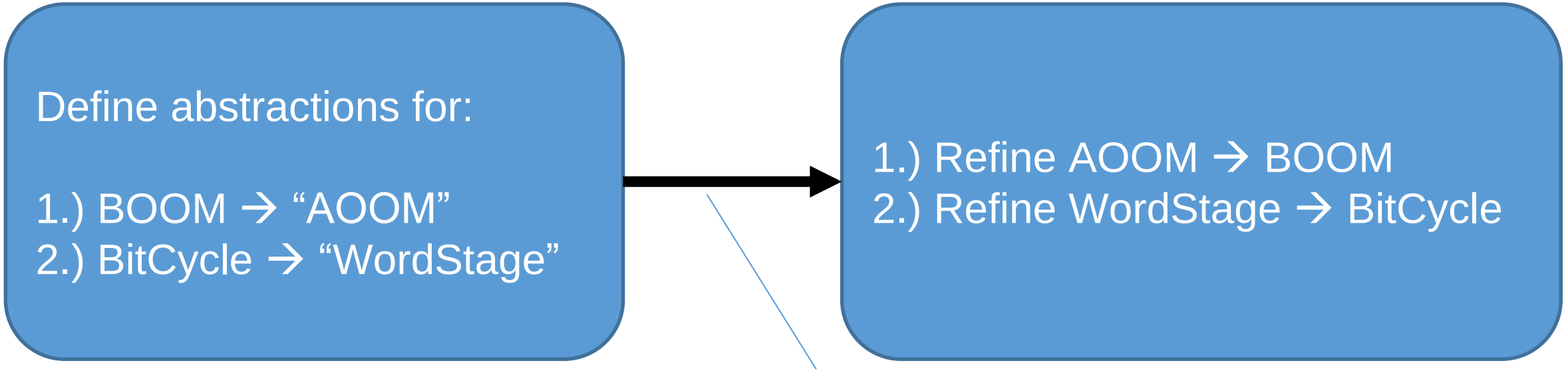
- Currently supported OISA
  - Int/FP arithmetic w/ **Safe** operands
  - Branches/Jumps w/ **Unsafe** operands
  - Two flavors of loads/stores
    - **Safe** data, **Unsafe** address
    - **Safe** data, **Safe** address
  - Instructions to set data as ***Confidential***/*Public*

We are open-sourcing implementation in late February

Security Argument

Approach to showing security

Goal: prove BitCycle(BOOM(P(x, y))) ≅ BitCycle(BOOM(P(x, y'))) over x,y,y'



Prove security of WordStage(AOOM(x, y)) on paper.

Best effort analysis right now. Need formal methods

Key Insight:
(Local checks for *Confidential* → Unsafe)
→ Whole-program security

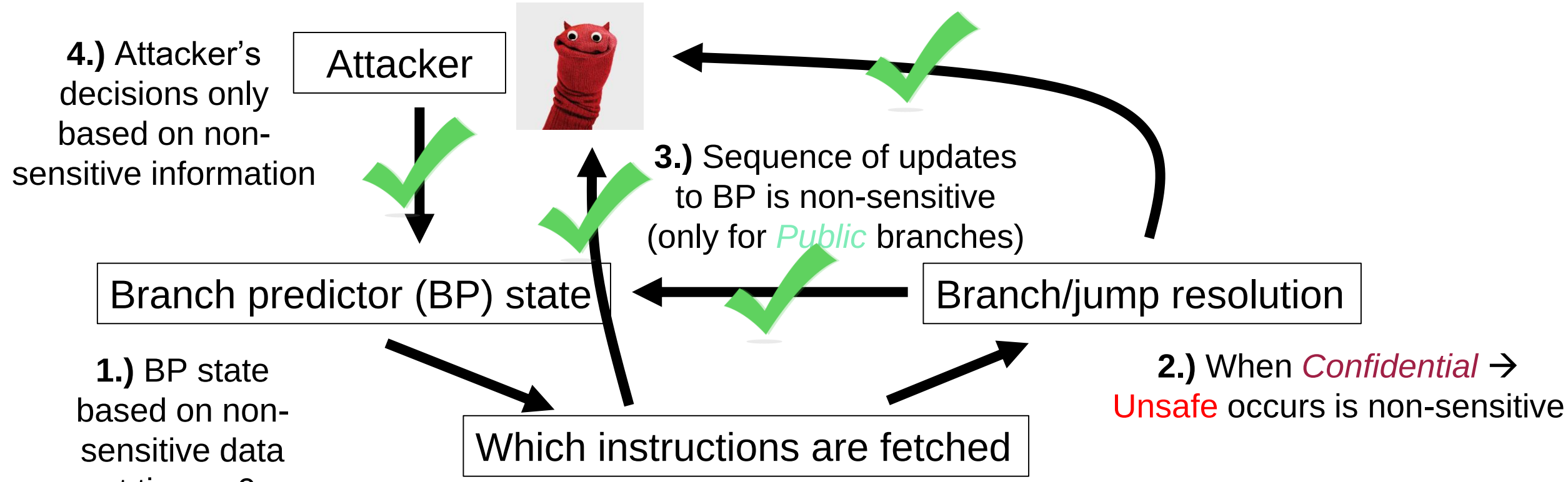
- Recall, for security we need:
  1. Each instruction's execution is data-independent
  2. Sequence of instruction's fetched & issued is data-independent

We get (1) from correctly implemented OISA.

How to get (2) given speculative execution?

Key Insight:
(Local checks for *Confidential* → Unsafe)
→ Whole-program security

- Sequence of instruction's fetched & issued doesn't leak privacy



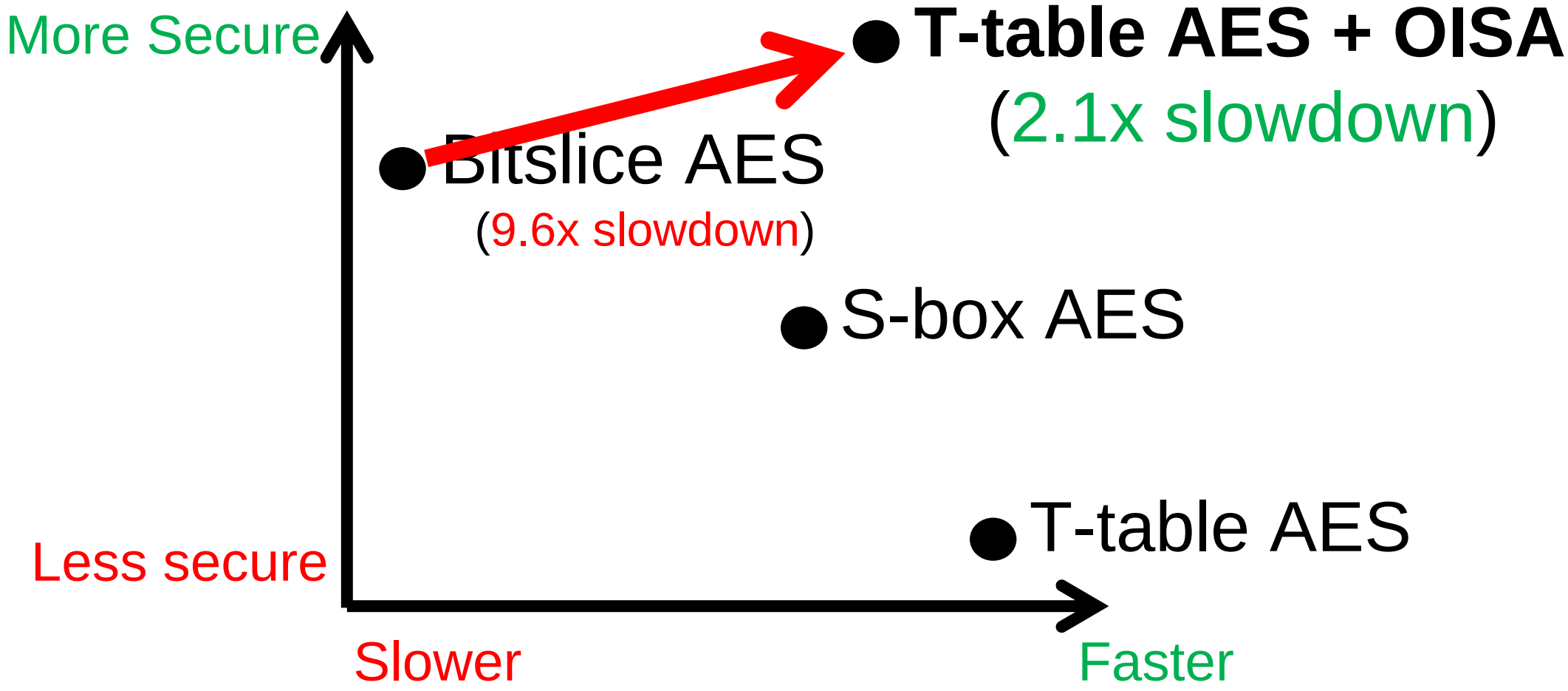
Evaluation and Conclusion

Evaluation

- Performance Overhead:
  - Data oblivious benchmarks (e.g., GEMM, graph SSSP)
    - Either match or significantly speedup
  - Case studies:
    1. Constant time AES
    2. Memory oblivious library

- Area overhead < 5%

Case study: Constant time AES



Data Oblivious ISA

Decouple security from functionality/implementation

SW receives portable security guarantee
HW not constrained to specific implementation

Applies to both speculative & non-speculative side channels

Thank you!

Backup

Why is this the right design? / Key Insights

- *Safe* does not specify an implementation
  - HW architects are free to implement in a way efficient for individual machine
  - SW designers don't need to know about processor implementations

- *"Confidential* data → *Safe* operand" implies advanced HW opts (e.g., OoO speculative execution) can remain enabled in common case

- *"Confidential* data → *Unsafe* operand" prevents spec. execution attacks and non-spec. execution attacks ("badly typed programs")

Why will this block present and future speculative/transient attacks?

- How can this apply to types of speculation you haven't thought of?

- *"Confidential* data → *Unsafe* operand" doesn't distinguish between whether *or how* an instruction is speculative.