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 22\textsuperscript{nd} 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”
  • “data oblivious programming”
  • “writing programs in the circuit abstraction”
  • ...

  systems community
  applied crypto community
  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);
```

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

Program level 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;
• Transformation to circuit is expensive;
  **E.g.,** scan memory to perform a read
  **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

Computer Science
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':

\[
\text{Obs}(\text{uArch}(P(x, y))) \cong \text{Obs}(\text{uArch}(P(x, y'))) 
\]

- 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\}
  - \(X^i_t\) = 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 \textbf{false} in \( t-1 \) and there was explicit flow in previous cycle
b.) Write enable is \textbf{true} in \( t-1 \) and either Input/Write enable contain explicit flow

Also see GLIFT papers
Example: Early terminating multiplier

\[ X_t^0, X_t^1, X_t^2 = \]

Check for 0 operands
Multiplier state
Writeback bus

Case 1:
No Early Terminate

Case 2:
Early Terminate

\( \neq \)
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

Diagram:
- Decode
- Skip?
- Safe operand?
- Multiplier
- To writeback bus
- mux
- In1
- In2
- 0
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  \(\rightarrow\) Performance Improvement

Leaky program:

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

**Strawman Oblivious ISA:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORload</td>
<td>(Unsafe) address</td>
</tr>
<tr>
<td>Ocmov</td>
<td>(Safe) pred, (Safe) src, dst</td>
</tr>
</tbody>
</table>

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

**Optimized Oblivious ISA:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLoad</td>
<td>(Safe) address</td>
</tr>
<tr>
<td>Ocmov</td>
<td>(Safe) pred, (Safe) src, dst</td>
</tr>
</tbody>
</table>

```
Ocmov addr =
    (secret) ? addr1 : addr2;
a ← OLoad (addr)
```
How to Implement OLoad with Safe address

- Multiple implementation options; different trade-offs
- Array size \( N \)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-code into simpler oblivious instructions (e.g., loads w/ <strong>Unsafe</strong> address)</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Hardware partitioning (e.g., cache partitioning, private scratchpads)</td>
<td>( O(1), ) size restricted</td>
</tr>
<tr>
<td>Cryptographic techniques (e.g., Oblivious RAM)</td>
<td>( O(\log N) ) or ( O(\log^2 N) )</td>
</tr>
</tbody>
</table>

- Detailed design in paper
More Opportunities for Oblivious “CISC” Instructions

• Sort = common data oblivious kernel

• Array size N
• Strawman data oblivious = O(N * log² 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**

We require: Confidential data → Safe Operand

Speculation can cause: Confidential data → Safe Operand (Bad speculation)

Can be guaranteed on non-spec machine by programmer/compiler

**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 $\text{BitCycle}(\text{BOOM}(P(x, y))) \cong \text{BitCycle}(\text{BOOM}(P(x, y'))) \text{ over } x, y, y'$

Define abstractions for:
1.) $\text{BOOM} \rightarrow \text{"AOOM"}$
2.) $\text{BitCycle} \rightarrow \text{"WordStage"}$

Prove security of $\text{WordStage}(\text{AOOM}(x, y))$ on paper.

1.) Refine $\text{AOOM} \rightarrow \text{BOOM}$
2.) Refine $\text{WordStage} \rightarrow \text{BitCycle}$

Best effort analysis right now. Need formal methods.
Key Insight:
(Local checks for Confidential \(\rightarrow\) Unsafe) \(\rightarrow\) 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

1.) BP state based on non-sensitive data at time = 0.

2.) When Confidential → Unsafe occurs is non-sensitive

3.) Sequence of updates to BP is non-sensitive (only for Public branches)

4.) Attacker’s decisions only based on non-sensitive information
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

- **Bitslice AES** (9.6x slowdown)
- **S-box AES**
- **T-table AES + OISA** (2.1x slowdown)

- More Secure
- Less secure
- Faster
- Slower

Computer Science
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 $\rightarrow$ *Safe* operand” implies advanced HW opts (e.g., OoO speculative execution) can remain enabled in common case

• “*Confidential* data $\rightarrow$ *Unsafe* operand” prevents spec. execution attacks and non-spec. execution attacks (“badly typed programs”)

Computer Science
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.