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

Network and Distributed System Security Symposium (NDSS), San Diego, 2019
Outline

• Introduction
• Data Oblivious ISA (OISA) Extension
• Hardware Implementation
• Security Analysis
• Evaluation
• Conclusion
if (secret)
    a = *(addr1);
else
    a = *(addr2);
Microarchitectural Side Channels Attacks

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

secret == True!
Microarchitectural Side Channels Attacks

Introduction

OISA Extension

Hardware

Security

Evaluation

Conclusion

Computer Science
Microarchitectural Side Channels

- L1 I Cache
- L1 D Cache
- L2 Cache
- L3 Cache
- DRAM (and/or: stacked DRAM, HMC, NVMs)
- Core
- Speculative execution [Spectre’18]
- Port contention [CBHPT’18]
- Datapath
- 4K aliasing [MES’17]
- Cache banking [YGH’16]
- Non-inclusive LLC [YSGFCT’19]
- DRAM [PGMSM’16]
- DRAM (and/or: stacked DRAM, HMC, NVMs)

Introduction
OISA Extension
Hardware
Security
Evaluation
Conclusion
Threat Model

• How to block all privacy threats from microarchitectural side channels.

• Software adversary is monitoring resource contention/program timing
Why Microarchitectural Side Channels are Big Issues

Software does not know what hardware can leak

Hardware does not know what is secret in the software
Why Microarchitectural Side Channels are Big Issues

Software does not know what hardware can leak

Hardware does not know what is secret in the software
Data Oblivious Programming

• A programmer’s solution to block all side channels
  [WNLCSSH’14], [NWIWTS’15], [SDSCFRYD’13], [RLT’15], [DJB’06], etc.

• Different names:
  • “constant time programming” (system community)
  • “data oblivious programming” (applied crypto community)
  • “writing programs in the circuit abstraction” (pure crypto community)

• Remove data-dependent behaviors from programs
/* Source program */
if (secret)
   a = *(addr1);
else
   a = *(addr2);

/* machine code */
a ← load (addr1);
b ← load (addr2);
cmov secret, a, b;
   // a = secret? b : a
Data Oblivious Programming: Three Assumptions

Security based on 3 assumptions

\[
\begin{align*}
    a & \leftarrow \text{load addr1} \\
    b & \leftarrow \text{load addr2} \\
    \text{cmov secret, b, a} & 
\end{align*}
\]
Data Oblivious Programming: Three Assumptions

Instructions processing data

**Assumption 1:** Every instruction is evaluated in a data-independent manner.

a ← load addr1

b ← load addr2

cmov secret, b, a
Data Oblivious Programming: Three Assumptions

Data transfer within and across hardware structures

Assumption 2: Data transfers in a data-independent manner
Data Oblivious Programming: Three Assumptions

Executed instruction sequence

Assumption 3: Instruction sequence is fixed regardless of program data
Data Oblivious Programming: Problems

security
portability
efficiency
Data Oblivious Programming: Problems

• Security

```
a ← load addr1
b ← load addr2
cmov secret, b, a
```
Data Oblivious Programming: Problems

• Security:

Assumption 1: Instructions are evaluated in a data-independent manner

Violations:
• Input-dependent arithmetic
• Microcode
• Silent stores
• ... ...
Data Oblivious Programming: Problems

• Security:

Assumption 2: Data transfers in a data-independent manner

Violations:
• Data-based compression
• Microop fusion
• ... ...
Data Oblivious Programming: Problems

• Security:

Assumption 3: Instruction sequence is fixed

Violations:
• Speculative execution
Data Oblivious Programming: Problems

• Security:

Assumption 3: Instruction sequence is fixed

Violations:
• Speculative execution
Data Oblivious Programming: Problems

• Security

• Portability

```c
if (condition)
    /* path A */
else
    /* path B */
```
Data Oblivious Programming: Problems

• Security

• Portability

X86 Processor A
without branch prediction

X86 Processor B
with branch prediction

if (condition)
/* path A */
else
/* path B */
Data Oblivious Programming: Problems

• Security
• Portability
• Efficiency
Data Oblivious Programming: Problems

**Conclusion**: data oblivious programing still lacks of a good contract

- **Security**: All assumptions are not in a contract that hardware can see
- **Portability**: No consistent contract across hardware implementations
- **Efficiency**: Software has to use simple instructions
This paper: Augment Instruction Set Architecture (ISA) for Data Oblivious Programming

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>ADD/SLT/SLTU</td>
<td>dest</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>AND/OR/XOR</td>
<td>dest</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>src2</td>
<td>src1</td>
<td>SLL/SRL</td>
<td>dest</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100000</td>
<td>src2</td>
<td>src1</td>
<td>SUB/SRA</td>
<td>dest</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Oblivious ISA: the Right Solution

Security
- ISA tells software what operations leak/do not leak
- ISA tells hardware what data is confidential

Portability
- ISA is fixed across hardware implementations

Efficiency
- Hardware can optimize expensive data oblivious operations since security semantics is clear at ISA level
Data Oblivious ISA Extensions

Two mechanisms for:

1. Differentiate between Confidential/Public data
   • New type of Dynamic information flow tracking

2. Indicate which operations are Safe to leak Confidential data
   • New notion of Safe instruction operands

Security specifications added to the contract
New Dynamic Information Flow Tracking (DIFT)

• Programmer declares data as Public or Confidential

• Confidential data is tracked in hardware using DIFT
  • Traditional DIFT only tracks retired data
  • Our DIFT tracks data at all instruction stages

• At a high level:
  • Public data needs no protection
  • Confidential data must be protected
Instruction with Safe Operands

• Each instruction’s input operand is defined as *Unsafe* or *Safe*
  • *Safe* operand: Block side channels stemming from that operand if necessary
  • *Unsafe* operand: No protection

• Example: multiplier
  Zero-skipping → input dependent timing
Instruction with Safe Operands

• Each instruction’s input operand is defined as *Unsafe* or *Safe*
  • *Safe* operand: Block side channels stemming from that operand if necessary
  • *Unsafe* operand: No protection

• Example: multiplier
  Zero-skipping →
  input dependent timing

**Example:**
- Processor core
  - Register 1: 1 → <safe> (tag: Public)
  - Register 2: 0 → <safe> (tag: Confidential)
  - Register 3: 0 ← slow, without Zero-skipping (tag: Confidential)

Processor core
Safe Operands + DIFT: Transition Rules

- **Public** data → *Safe* operand: No protection needed
- **Public** data → *Unsafe* operand: No protection needed
- **Confidential** data → *Safe* operand: Execute with protection
- **Confidential** data → *Unsafe* operand: Stop speculation*

---

```
[tag: Public] Register 1: 1 → <safe>
[tag: Confidential] Register 2: 0 → <safe>
[tag: Confidential] Register 3: 0

multiplier

Processor core
```

---

Computer Science
Complete Proposal: *Safe* Operands + DIFT

1. ISA Design time:
   ISA designers decide instructions with *Safe/Unsafe* operands

2. Hardware Design time:
   Hardware designers augment processors with logic to enable/disable optimizations

3. Programming time:
   Programmers annotate some program inputs and static data *Public/Confidential*

4. Runtime
   Processor implements transition rules and taint propagation during execution.
Key Benefits

1. Simple portable guarantee for programmers across implementations

2. Hardware & Data-oblivious-programming co-design

3. Defense against non-speculative and speculative execution attacks
Key Benefit: HW-Algorithm Co-design

- Problem: Sensitive loads are performance bottlenecks
- Solution: add load with *Safe* address

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Efficiency (object with size N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-code into loads w/ <em>Unsafe</em> address</td>
<td>O(N)</td>
</tr>
<tr>
<td>Cryptographic techniques (e.g., Oblivious RAM)</td>
<td>O(log N) or O(log^2 N)</td>
</tr>
<tr>
<td>Hardware partitioning (e.g., cache partitioning, private scratchpads)</td>
<td>O(1), size restricted</td>
</tr>
</tbody>
</table>
Key Benefit: HW-Algorithm Co-design

• Problem: Sensitive loads are performance bottlenecks
• Solution: add load with *Safe* address

• More opportunities for complex instructions
  • Oblivious shuffle instruction
  • Oblivious sort instruction
  • ... ...
Key Benefit: Defense Against Non-spec & Spec Attacks

Defends against Non-speculative attacks

Defends against Speculative attacks
Hardware Implementation

• Hardware prototyping on RISC-V BOOM processor
  • Enumerate potential threat vectors of BOOM
  • Propose an OISA extension for RISC-V ISA
  • Implement new instructions with safe operand and DIFT on BOOM

• Design open sourced at github (see paper)

Current OISA Extension:
• Int/FP arithmetic w/ \textit{Safe} operands
• Branches/Jumps w/ \textit{Unsafe} operands
• Two flavors of loads/stores
  • \textit{Safe} data, \textit{Unsafe} address
  • \textit{Safe} data, \textit{Safe} address
• Instructions to set data as \textit{Confidential}/\textit{Public}
Security Analysis

• Formalize the security of data oblivious ISA extension

• Goal: prove for different confidential data, the trace of observable processor states is invariant.

• Two challenges:
  • How to formalize attacker’s capability of observing processor states
  • How to model modern processors -> designed an abstract BOOM machine
Evaluation

• Achieve a speedup of up to 8.8x over baseline data oblivious programming

• Case studies:
  • Constant time AES: 4.4x speedup over bitslice AES
  • Memory oblivious library: more than 4.6x speedup over ZeroTrace [SGF’18]
Conclusion

Data Oblivious ISA decouples security from functionality and implementation

- Software receives consistent, portable security guarantee
- Hardware is not constrained to specific implementation
- Applies to both speculative & non-speculative side channels
Questions?

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
Thank you for listening to our talk!