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

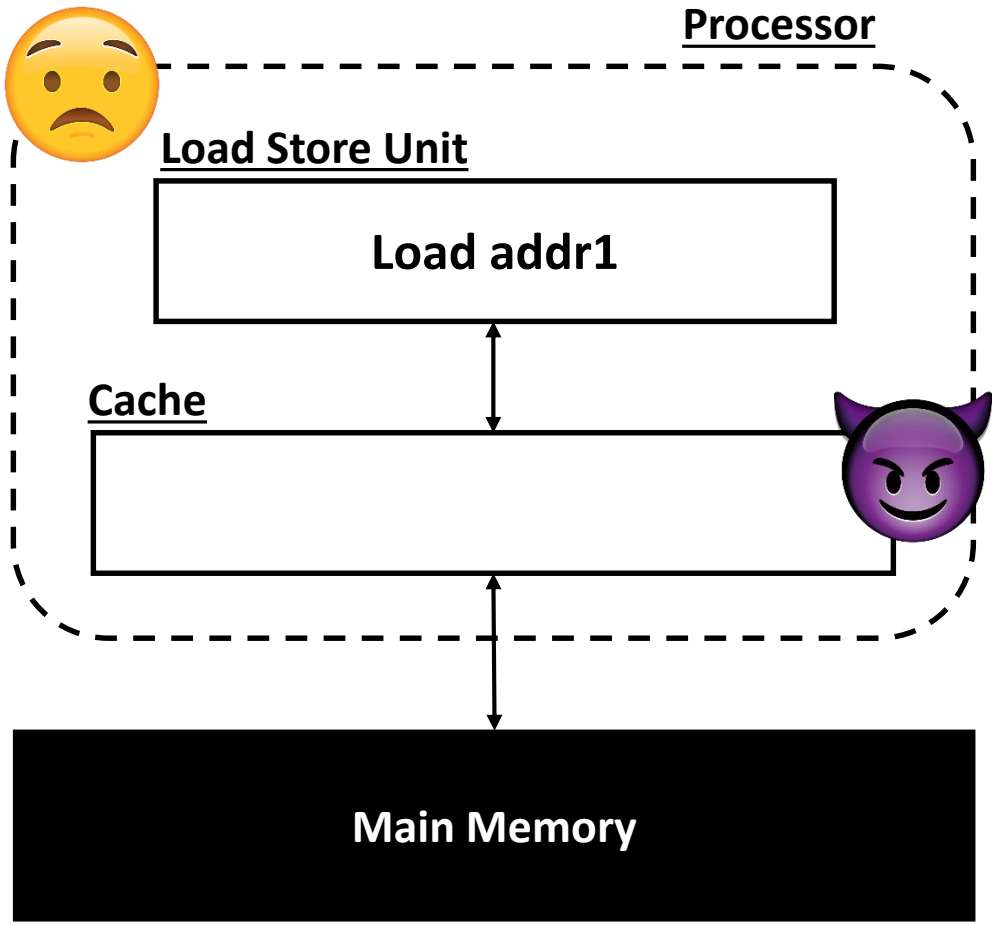
Network and Distributed System Security Symposium (NDSS), San Diego, 2019

Outline

- Introduction
- Data Oblivious ISA (OISA) Extension
- Hardware Implementation
- Security Analysis
- Evaluation
- Conclusion

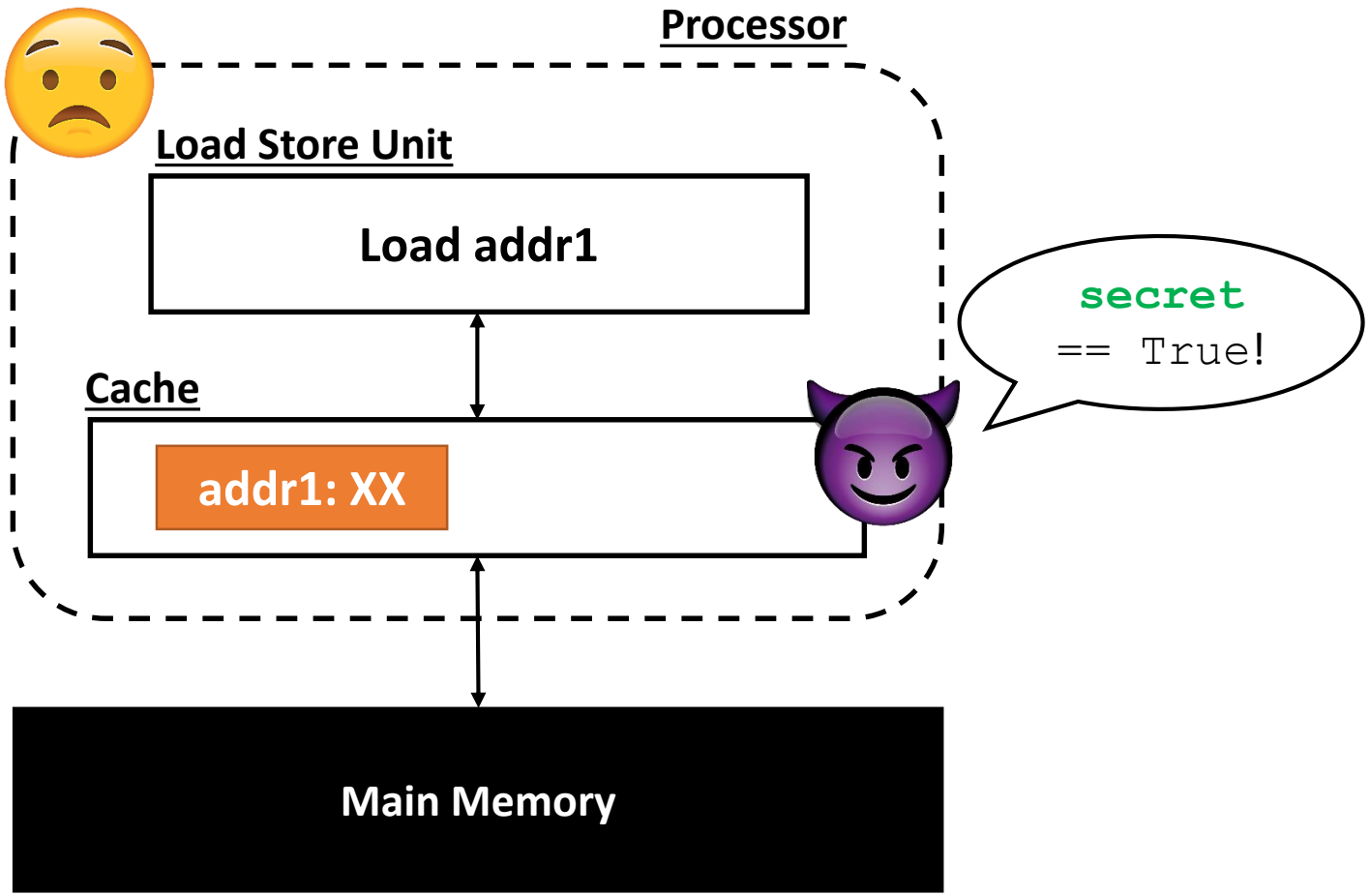
Microarchitectural Side Channels Attacks

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

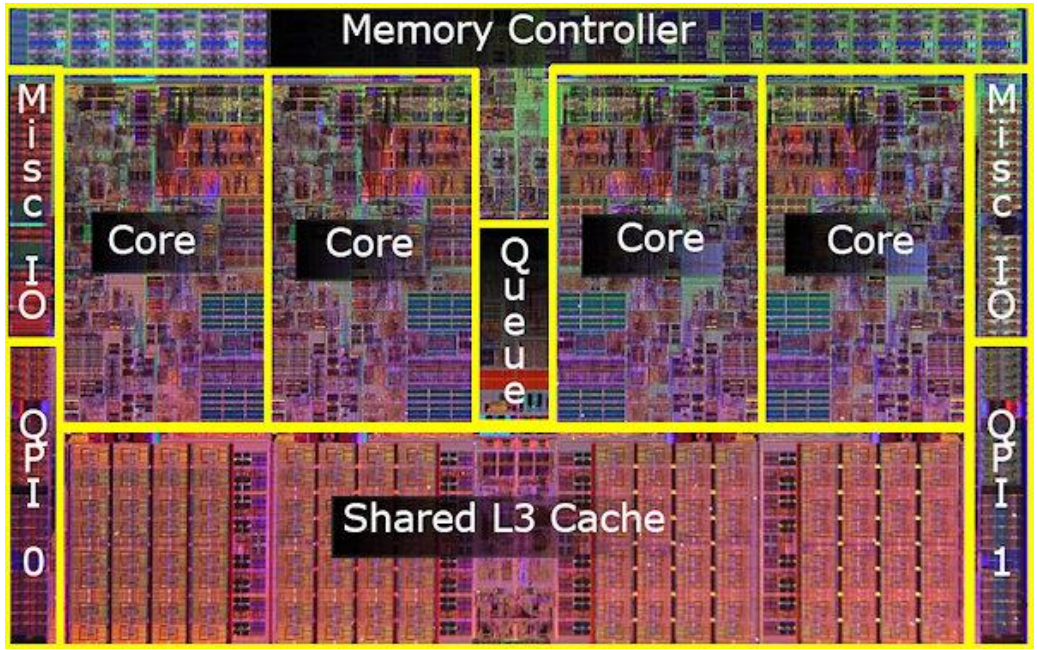


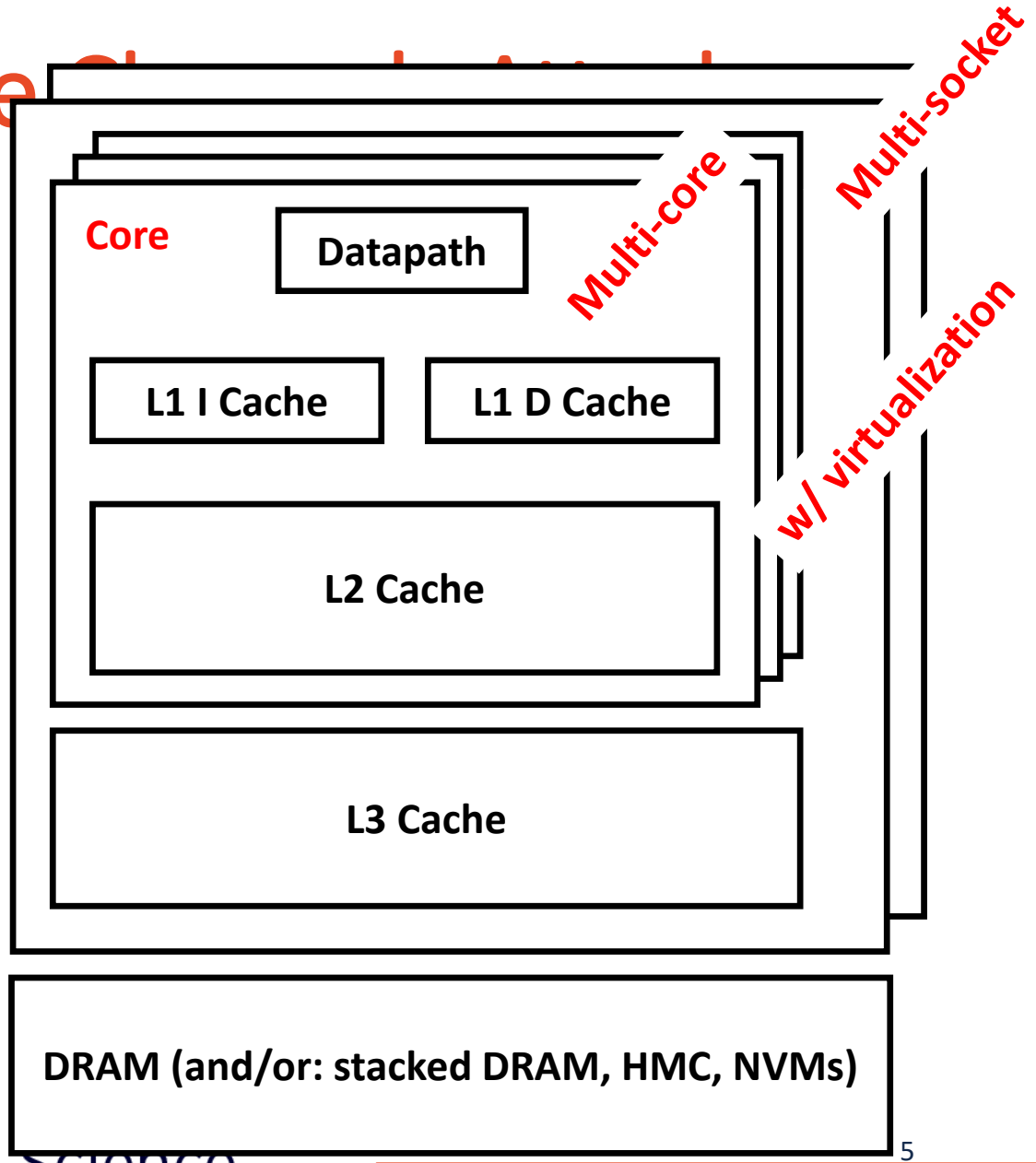
Microarchitectural Side Channels Attacks

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

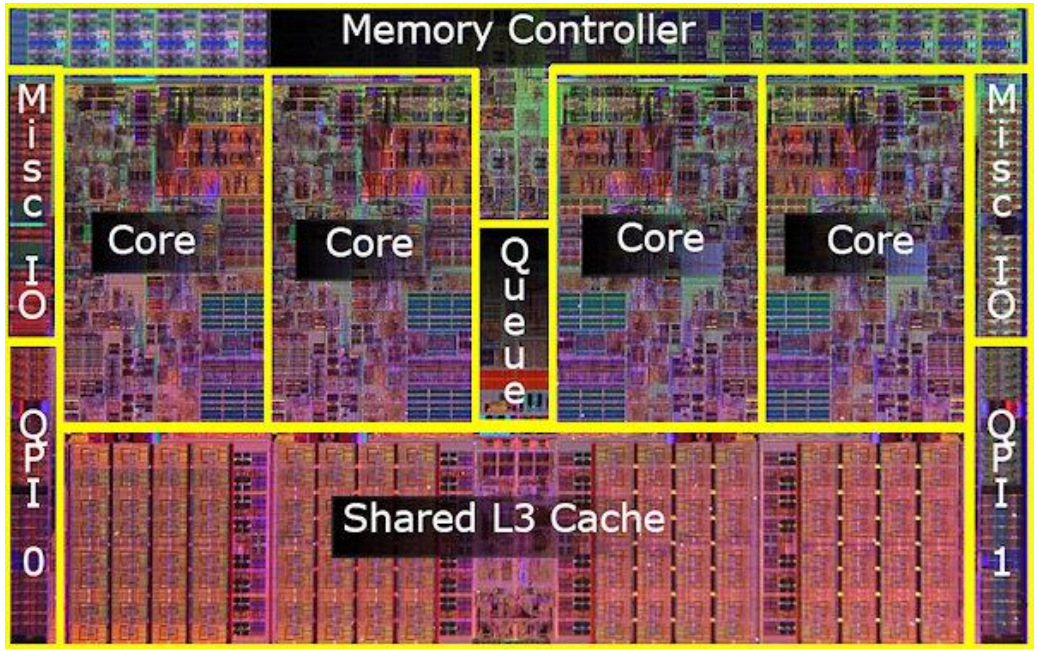


Microarchitectural Side Channel Attacks





Microarchitectural Side-Channel Attacks



Speculative execution [Spectre'18]

Port contention [CBHPT'18]

Datapath

4K aliasing [MES'17]

L1 I Cache

L1 D Cache

Cache banking [YGH'16]

L2 Cache

Non-inclusive LLC [YSGFCT'19]

L3 Cache

DRAM [PGMSM'16]

DRAM (and/or: stacked DRAM, HMC, NVMs)

Multi-core

Multi-socket

w/ virtualization

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.

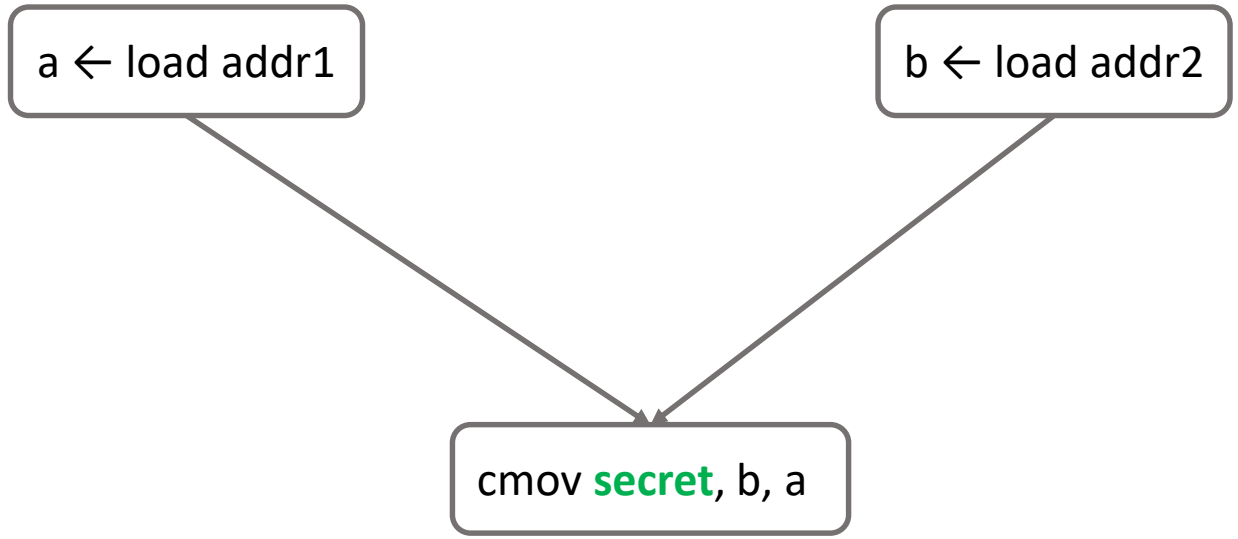
- Different names:
  - "constant time programming" (system community)
  - "data oblivious programming" (applied crypto community)
  - "writing programs in the circuit abstraction" (pure crypto community)

- Remove data-dependent behaviors from programs

Data Oblivious Programming: An Example

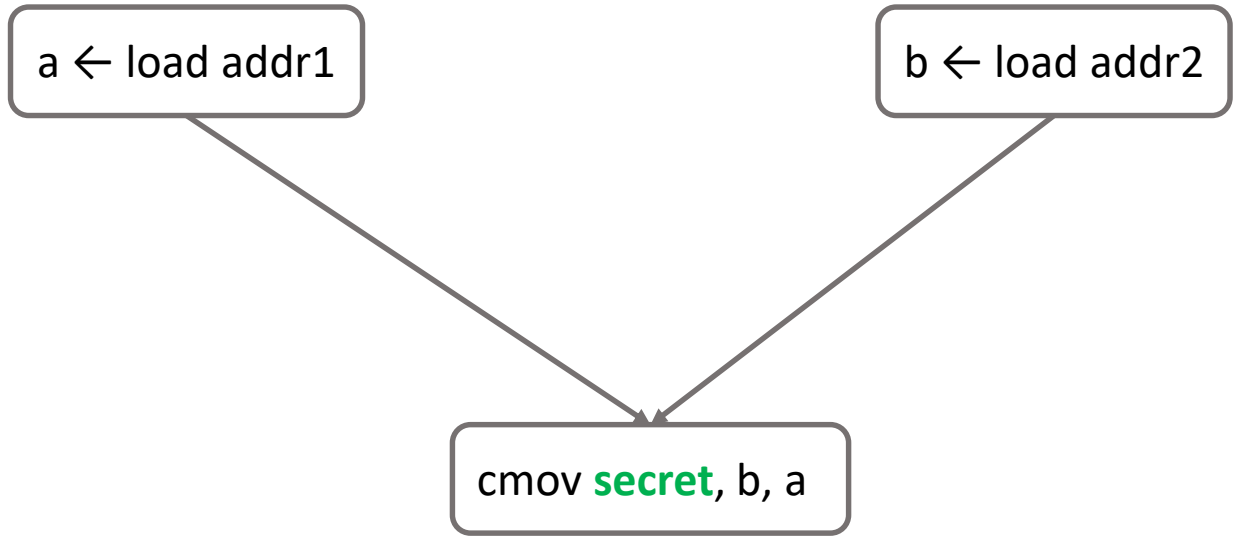
```
/* Source program */
if (secret)
      a = *(addr1);
else
      a = *(addr2);
```

```
/* machine code */
a ← load (addr1);
b ← load (addr2);
cmov secret, a, b;
    // a = secret? b : a
```



Data Oblivious Programming: Three Assumptions

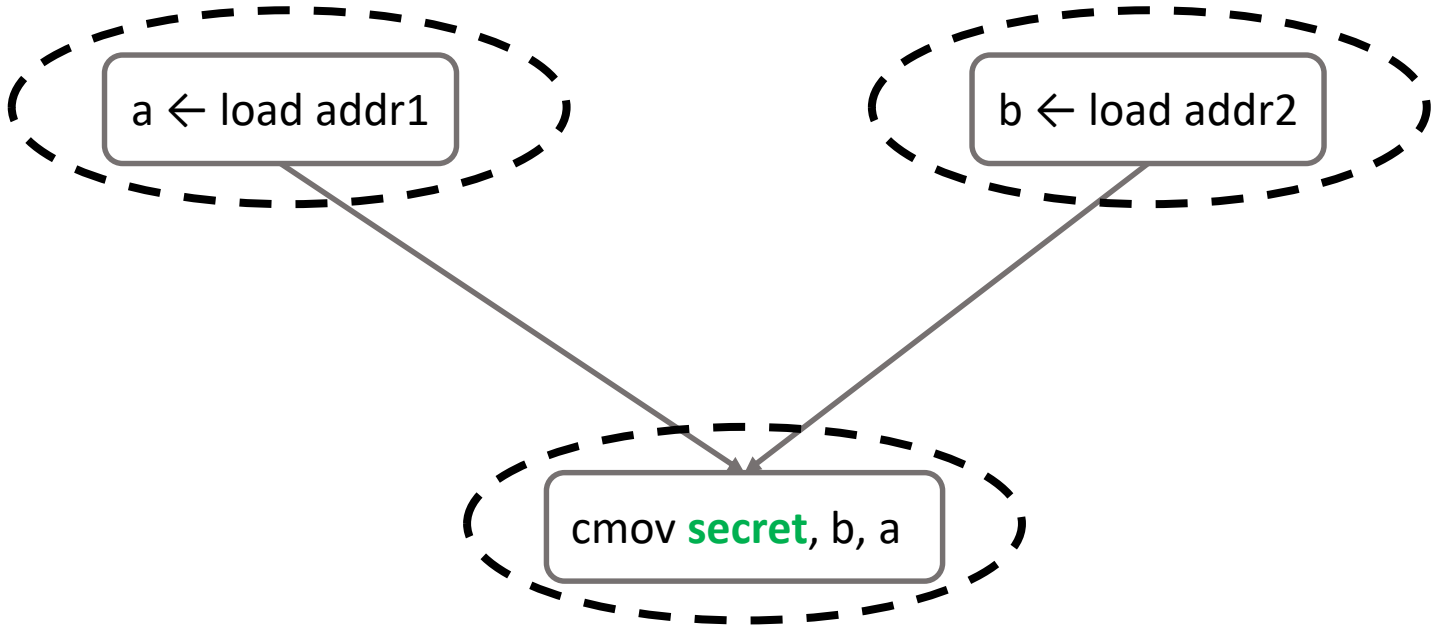
Security based on 3 assumptions



Data Oblivious Programming: Three Assumptions

Instructions processing data

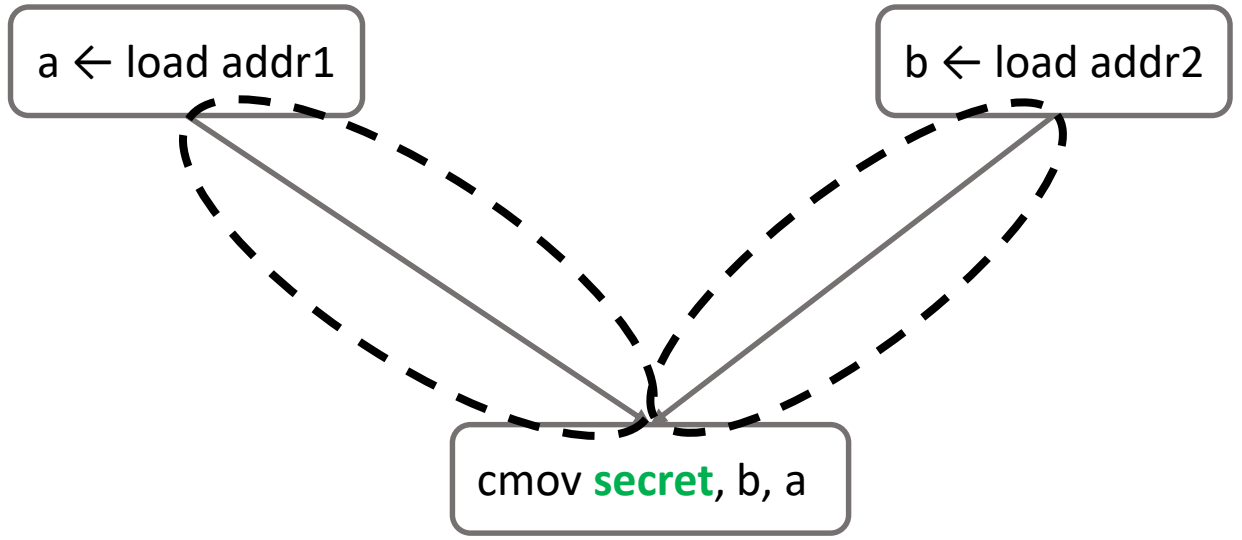
Assumption 1: Every instruction is evaluated in a data-independent manner



Data Oblivious Programming: Three Assumptions

Data transfer within and across hardware structures

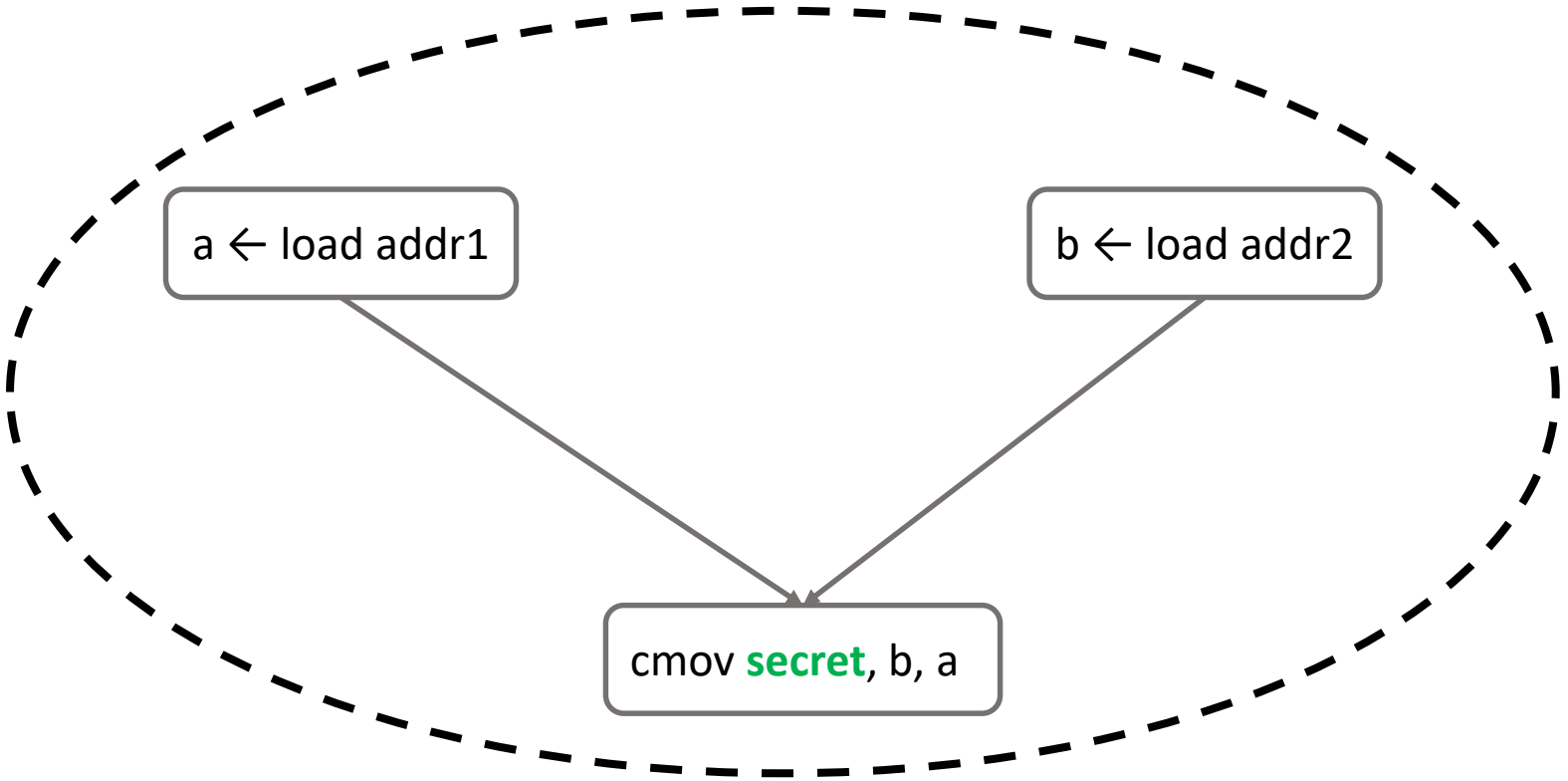
Assumption 2: Data transfers in a data-independent manner



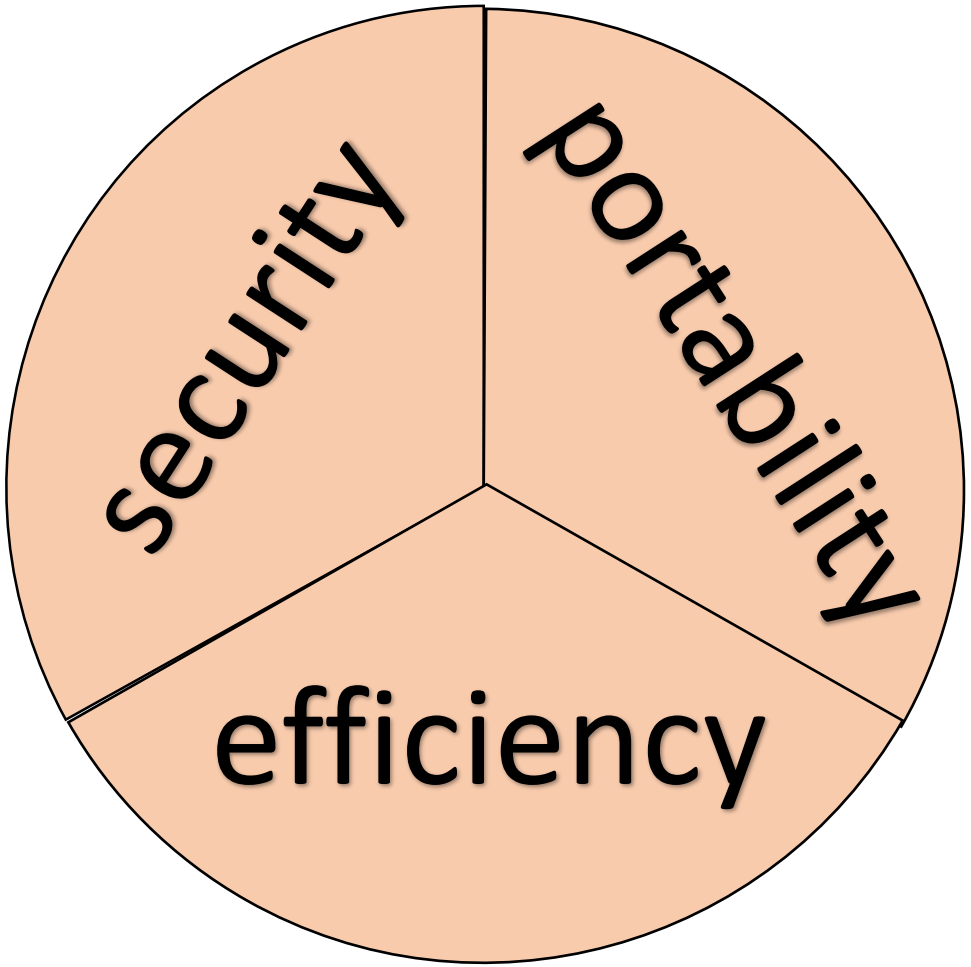
Data Oblivious Programming: Three Assumptions

Executed instruction sequence

Assumption 3: Instruction sequence is fixed regardless of program data

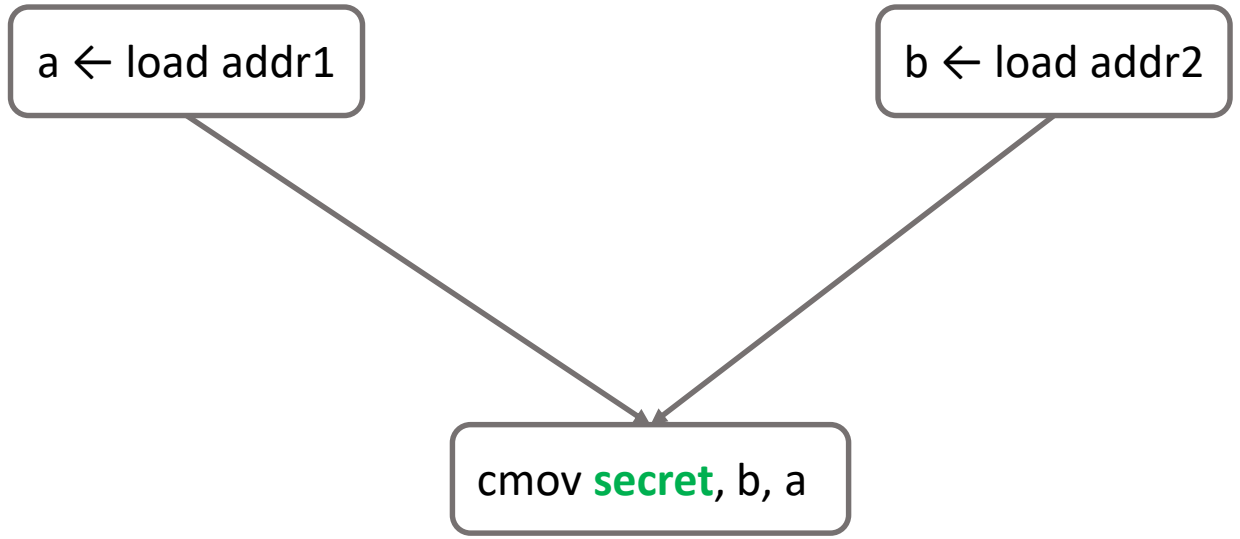


Data Oblivious Programming: Problems



Data Oblivious Programming: Problems

- Security



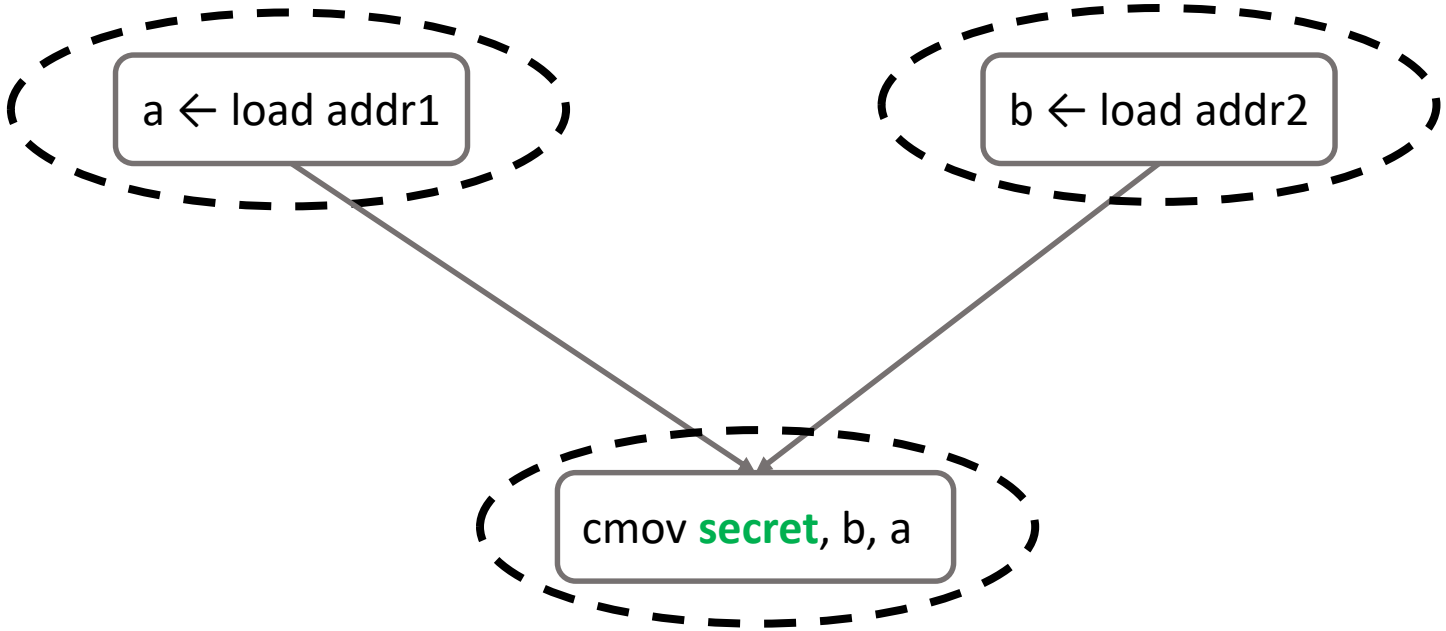
Data Oblivious Programming: Problems

- Security:

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

Violations:

- Input-dependent arithmetic
- Microcode
- Silent stores
- … …



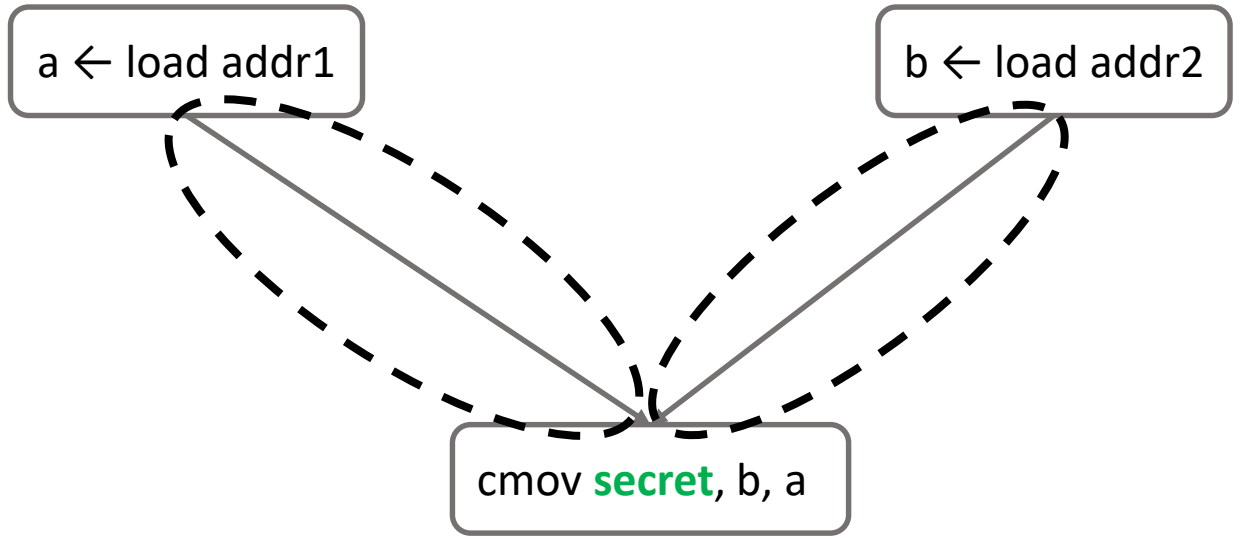
Data Oblivious Programming: Problems

- Security:

Assumption 2: Data transfers in a data-independent manner

Violations:

- Data-based compression
- Microop fusion
- … …



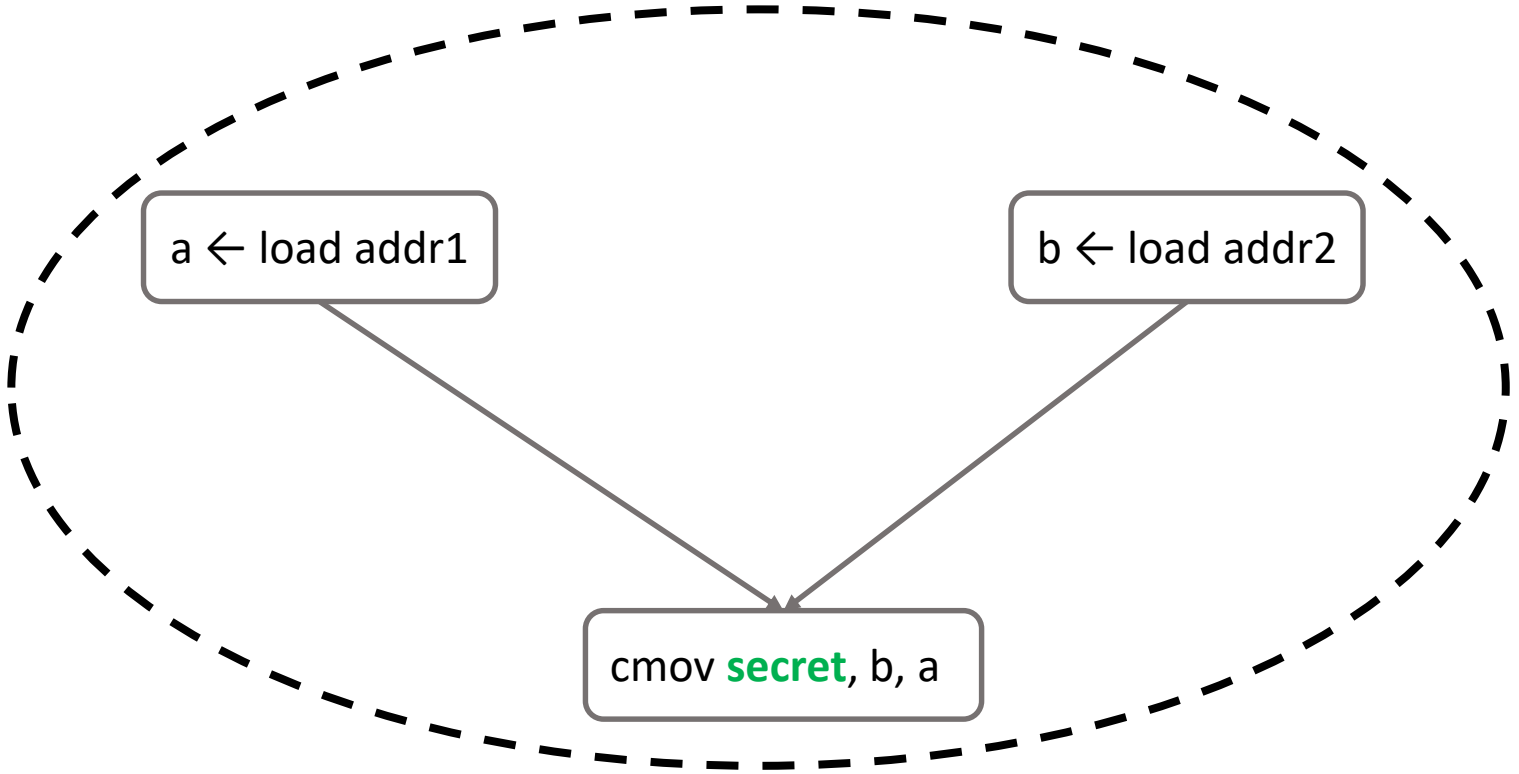
Data Oblivious Programming: Problems

- Security:

Assumption 3: Instruction sequence is fixed

Violations:

- Speculative execution



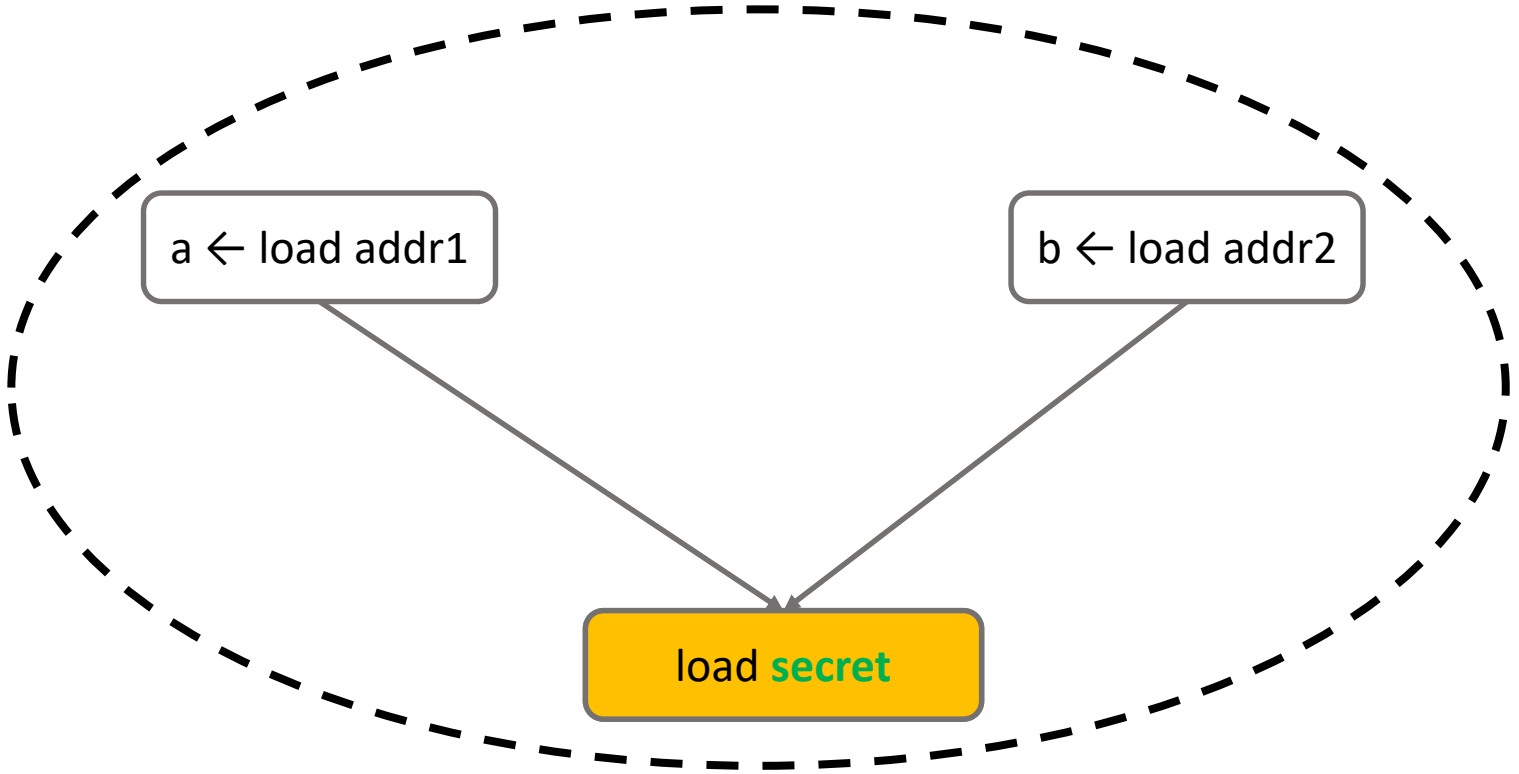
Data Oblivious Programming: Problems

- Security:

Assumption 3: Instruction sequence is fixed

Violations:

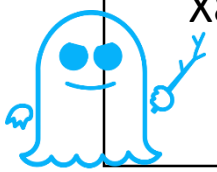
- Speculative execution



Data Oblivious Programming: Problems

- Security
- Portability

X86 Processor A without branch prediction



X86 Processor B with branch prediction

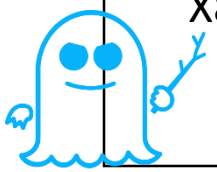
SPECTRE

```
if (condition)
    /* path A */
else
    /* path B */
```

Data Oblivious Programming: Problems

- Security
- Portability

X86 Processor A without branch prediction



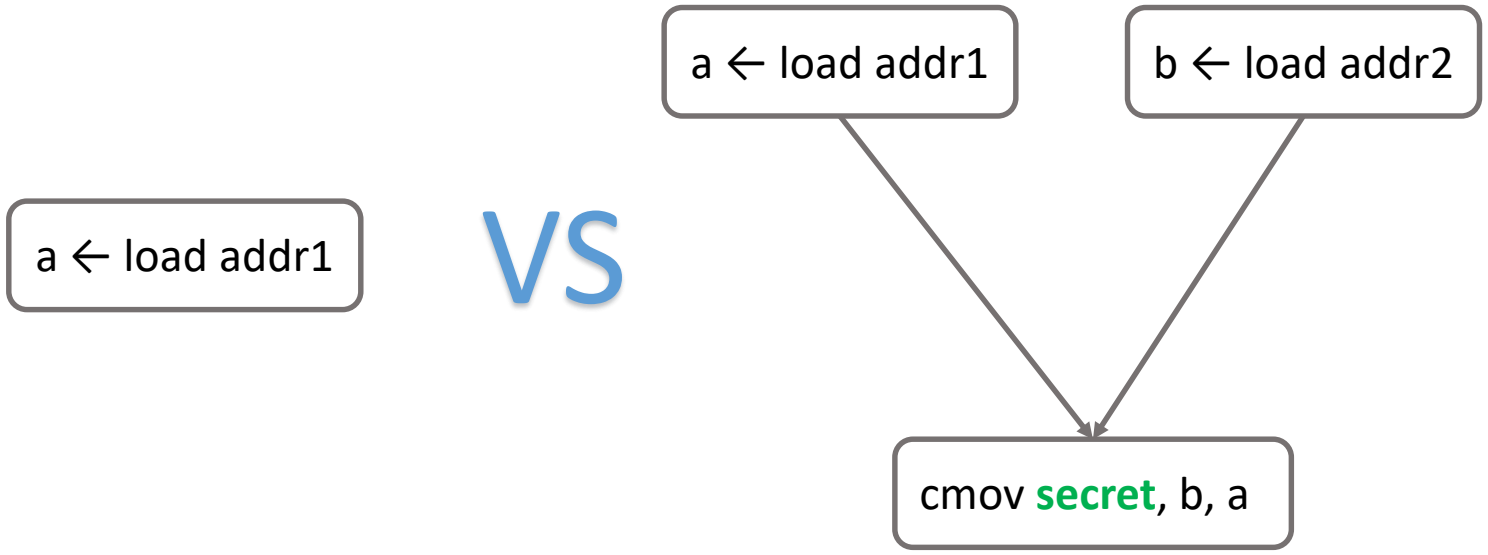
X86 Processor B with branch prediction

SPECTRE

```
if (condition)
    /* path A */
else
    /* path B */
```

Data Oblivious Programming: Problems

- Security
- Portability
- Efficiency



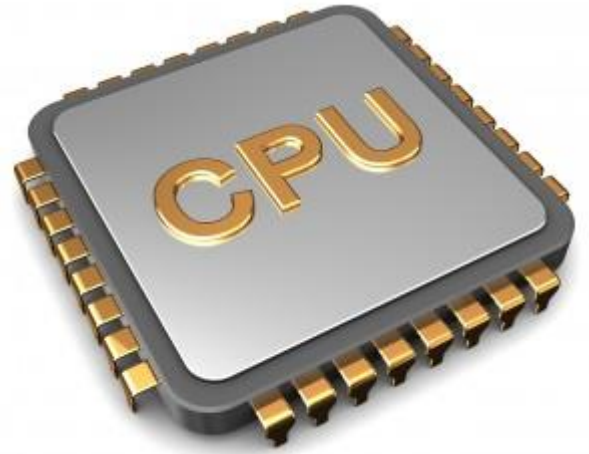
Data Oblivious Programming: Problems

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

```
int add(int a, int b) {
    return a + b;
}
```



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

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:
- telling hardware what data is confidential
- telling software what operations leak/do not leak

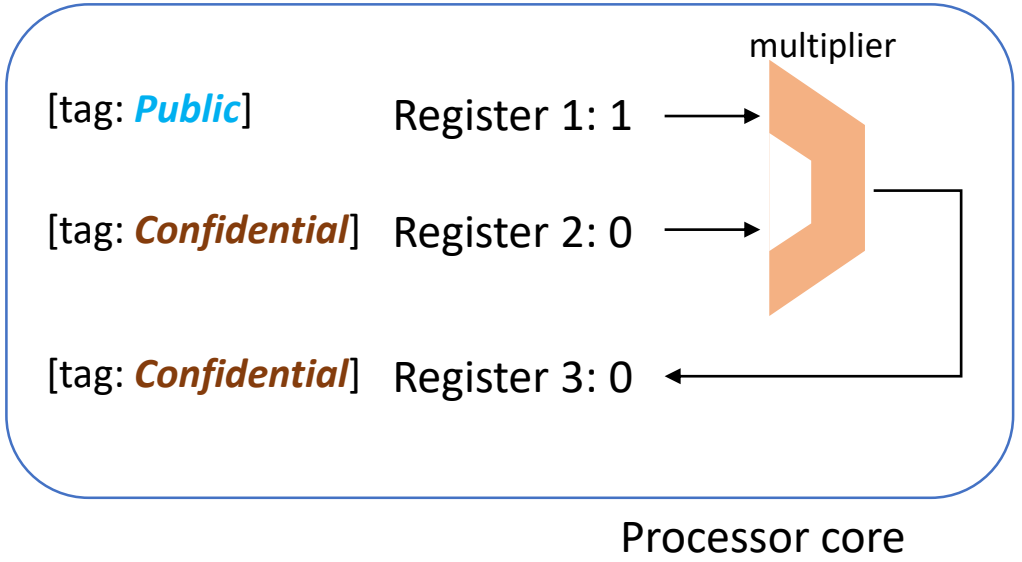
1. Differentiate between *Confidential*/*Public* data
   - New type of Dynamic information flow tracking

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

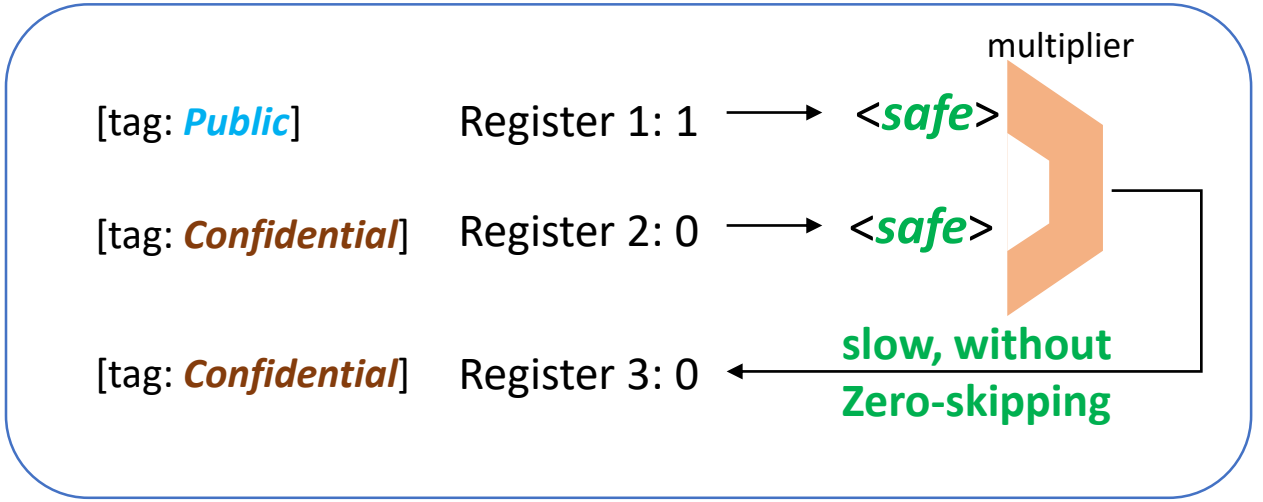
Instruction with Safe Operands

- 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

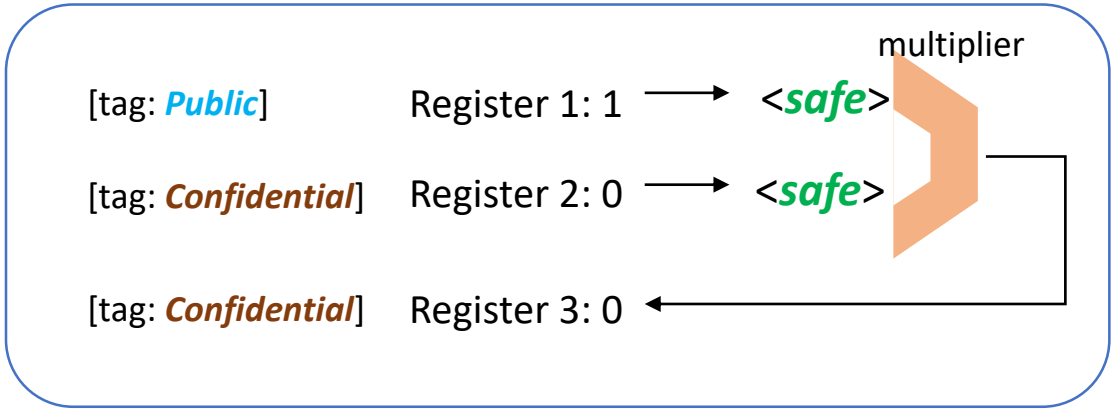
  Zero-skipping →

  input dependent timing



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*



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. Programing 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

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

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/ *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*

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!