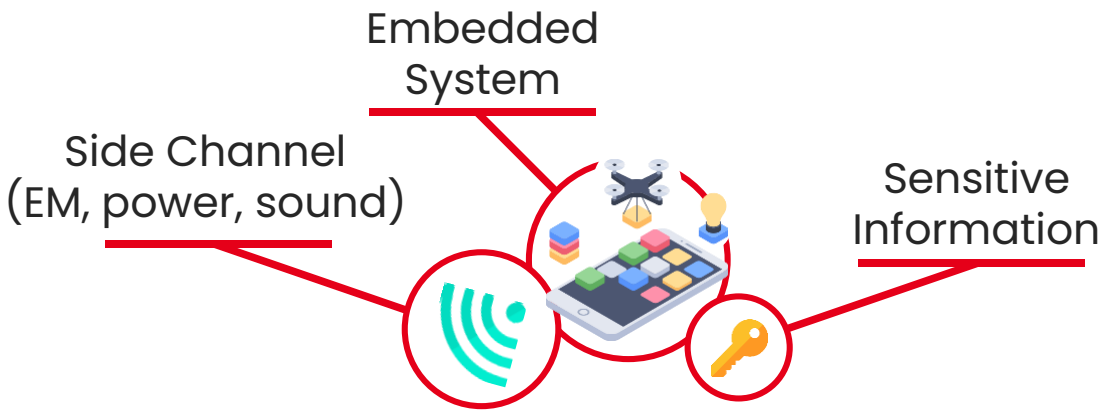
(On) The Impact of the Micro-architecture on Countermeasures against Side-Channel Attacks

PhD Student:
Lorenzo Casalino

Supervisors:
Nicolas Belleville
Damien Couroussé
Karine Heydemann

30/01/2024

Embedded Systems and Side Channels



Embedded systems have *observable* effects (side channels) on the surrounding environment

The processed information Influences the side channel

Step #1

Observe Side Channel

Step #2

Statistically Analyse Side Channel

Step #3

Recover Information

An attacker can exploit side channels to recover information

Masking to the Rescue!

Break
Statistical Link

Remove link between sensitive information and side-channel

Masking *encodes* Information with random variables

X → f → $X_0, X_1, \dots, X_N$

Masking Order

Sensitive Information

Masking Function

Random Variables (*shares*)

$X_0, X_1, \dots, X_N$

Statistical Link

Random variables influence side channel.
Attacker recovers random values.
Sensitive information protected

Masking: a Software Example

$C = A \; and \; B$

Algorithm

Input to

A

B

Sensitive Information

f

Boolean Masking

A_0, A_1

B_0, B_1

Random Variables (*shares*)

Masking: a Software Example

$C = A \; and \; B$

Algorithm

f

Boolean Masking

```
SEC-AND2(A0, A1, B0, B1):
C0  = (A0 and B0) xor R
X01 = A0 and B1
X10 = A1 and B0
X11 = A1 and B1
C1  = (((X01 xor R) xor X10)) xor X11)
return <C0, C1>
```

C-like Masked Algorithm

Input to

Input to

A

B

Sensitive Information

f

Boolean Masking

A_0, A_1

B_0, B_1

Random Variables (*shares*)

Independent Leakage Assumption (ILA)

$X_0, X_1, \ldots, X_N$

DO NOT RECOMBINE

```
SEC-AND2(A0, A1, B0, B1):
  C0  = (A0 and B0) xor R
  X01 = A0 and B1
  X10 = A1 and B0
  X11 = A1 and B1
  C1  = (((X01 xor R) xor X10)) xor X11)
  return <C0, C1>
```

Each (sub-)computation must not recombine the shares

IS IT ENOUGH ?

Violation of the ILA

- A CPU executes an ***implementation*** of an **algorithm**
- An Implementation employs ***architectural registers***
  - Memory elements to save temporary values
- The *re-use* of registers recombines the shares
  - We call it ***transition-based leakages***
- From the compiled <secAnd2> example:
  - Register $R0$ and $R2$
  - $R0$'s re-use: $A0 \rightarrow A1$
  - $R2$'s re-use: $B1 \rightarrow B0$

```
SEC-AND2(A0, A1, B0, B1):
  C0  = (A0 and B0) xor R
  X01 = A0 and B1
  X10 = A1 and B0
  X11 = A1 and B1
  C1  = (((X01 xor R) xor X10)) xor X11)
  return <C0, C1>
```

Compile

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R2 = read B0
R0 = read A1
R2 = and R0,R2
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R2,R0
        ...
```

Violation of the ILA

- A CPU executes an ***implementation*** of an **algorithm**
- An Implementation employs ***architectural registers***
  - Memory elements to save temporary values
- The *re-use* of registers violates the ILA
  - We call it ***transition-based leakages***
- From the compiled <secAnd2> example:
  - Register $R0$ and $R2$
  - $R0$'s re-use: $A0 \rightarrow A1$
  - $R2$'s re-use: $B1 \rightarrow B0$
- Solutions:
  1. Avoid register re-uses

```
SEC-AND2(A0, A1, B0, B1):
C0  = (A0 and B0) xor R
X01 = A0 and B1
X10 = A1 and B0
X11 = A1 and B1
C1  = (((X01 xor R) xor X10)) xor X11)
return <C0, C1>
```

Compile

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R2 = read B0
R0 = read A1
R2 = and R0,R2
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R2,R0
        ...
```

Avoid register re-uses

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R3,R0
        ...
```

Violation of the ILA

- A CPU executes an ***implementation*** of an **algorithm**
- An Implementation employs ***architectural registers***
  - Memory elements to save temporary values
- The *re-use* of registers violates the ILA
  - We call it ***transition-based leakages***
- From the compiled <secAnd2> example:
  - Register $R0$ and $R2$
  - $R0$'s re-use: $A0 \rightarrow A1$
  - $R2$'s re-use: $B1 \rightarrow B0$
- Solutions:
  1. Avoid register re-uses
  2. *Flush* (i.e., overwrite) the leaking registers

```
SEC-AND2(A0, A1, B0, B1):
C0  = (A0 and B0) xor R
X01 = A0 and B1
X10 = A1 and B0
X11 = A1 and B1
C1  = (((X01 xor R) xor X10)) xor X11)
return <C0, C1>
```

Compile

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R2 = read B0
R0 = read A1
R2 = and R0,R2
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R2,R0
        ...
```

Avoid register re-uses

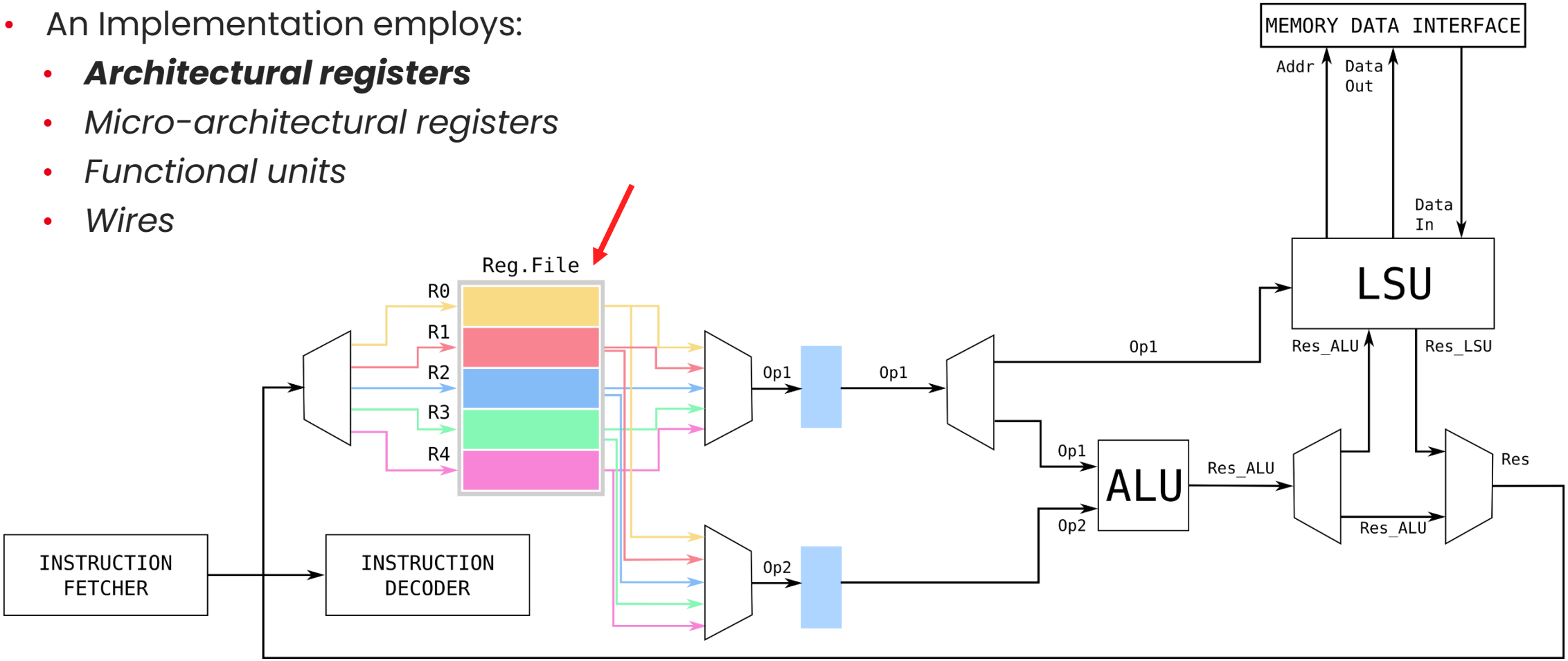
```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R2,R0
        ...
```

Flushing

```
<secAnd2>:
        ...
R2 = mov #42
R2 = read B0
R0 = mov #0
R0 = read A1
        ...
```

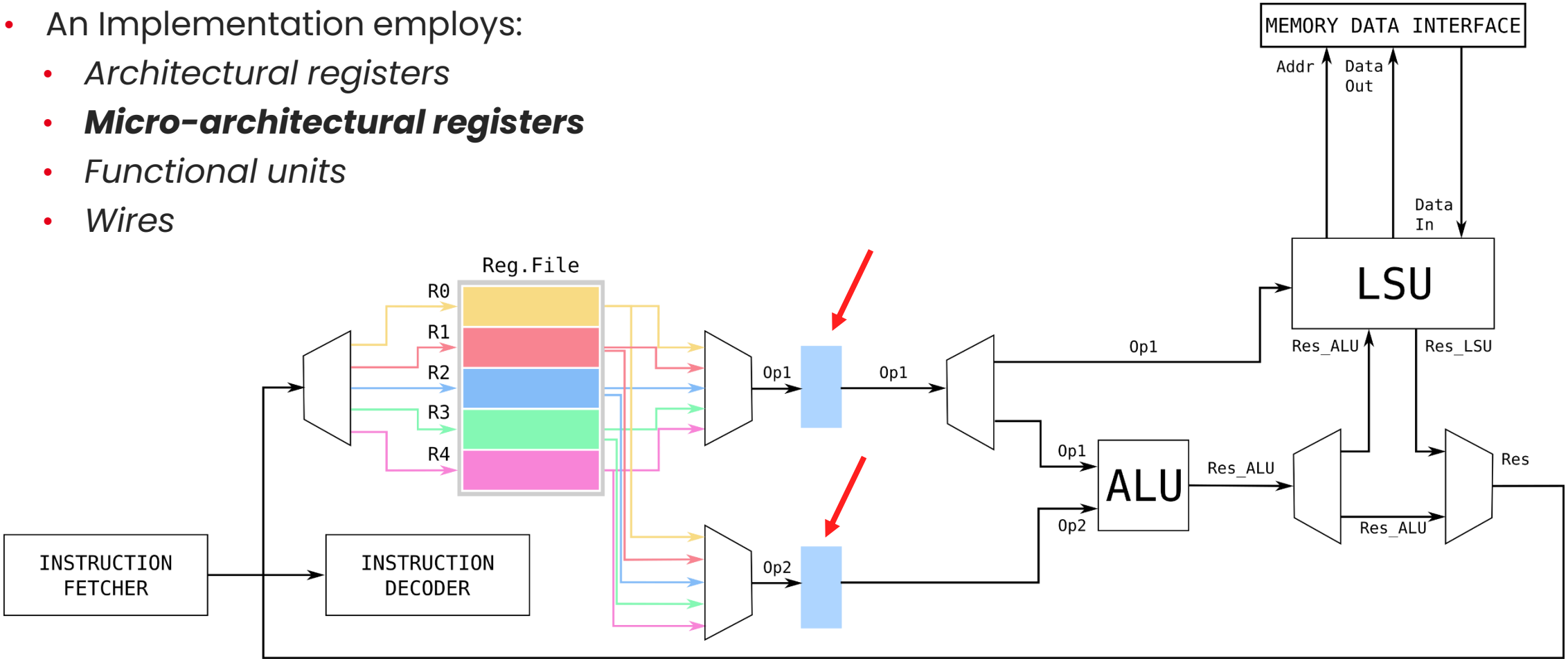
The Elephant in the Room

- A CPU *hides* more complex structures: *micro-architectures*
- An Implementation employs:
  - ***Architectural registers***
  - *Micro-architectural registers*
  - *Functional units*
  - *Wires*



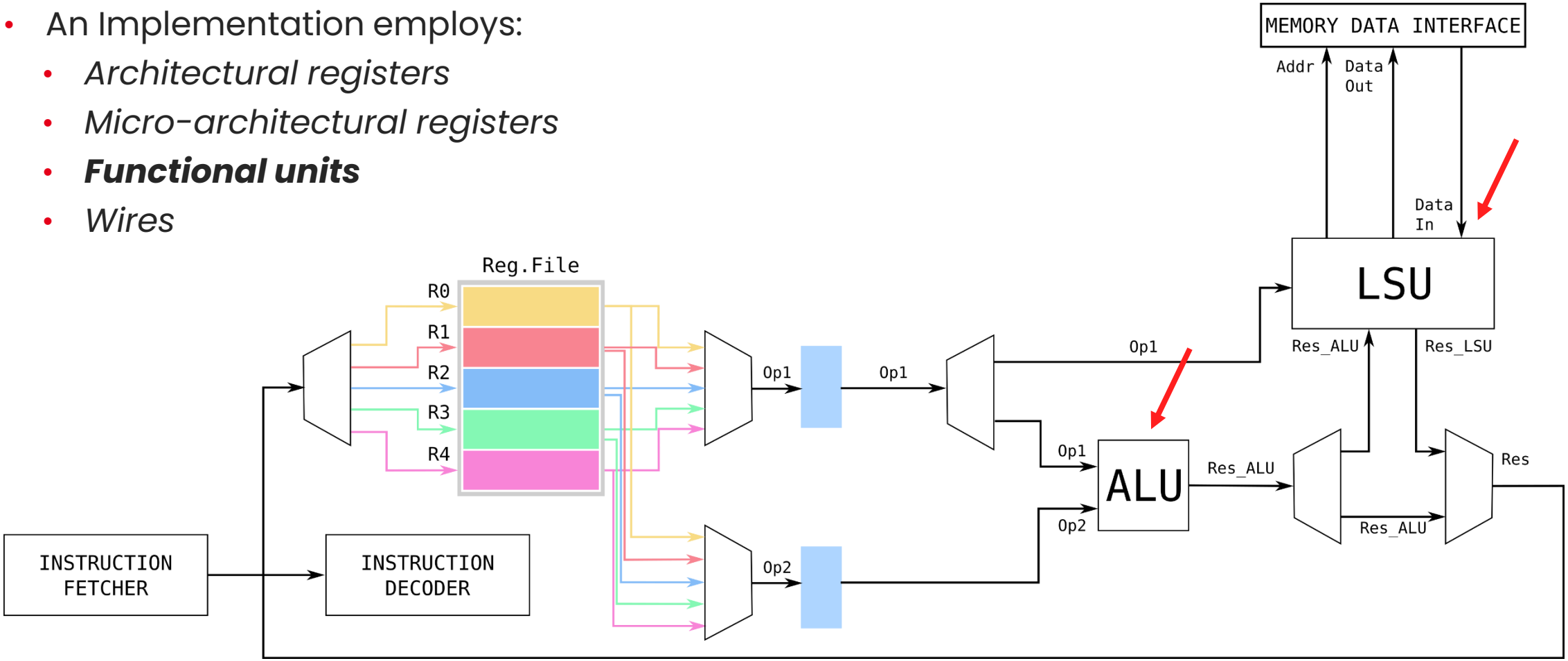
The Elephant in the Room

- A CPU *hides* more complex structures: *micro-architectures*
- An Implementation employs:
  - *Architectural registers*
  - ***Micro-architectural registers***
  - *Functional units*
  - *Wires*



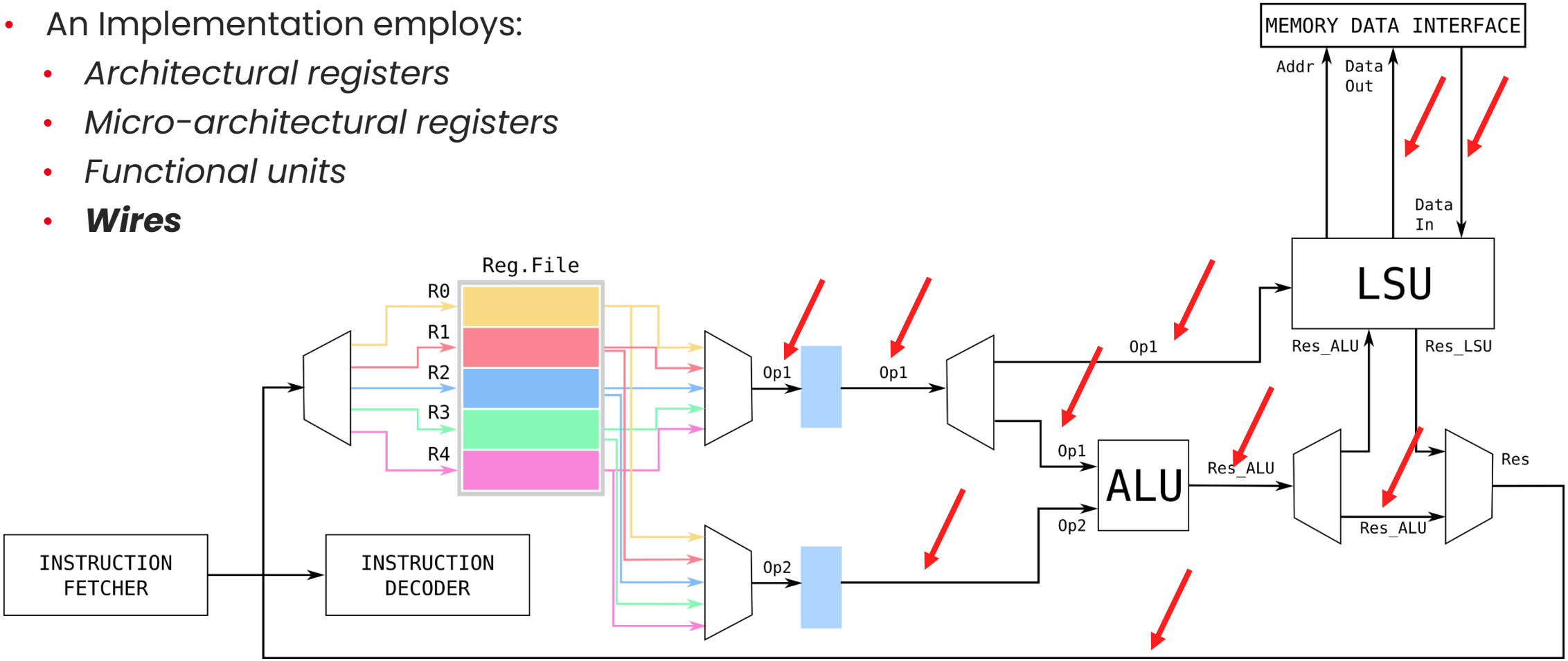
The Elephant in the Room

- A CPU *hides* more complex structures: *micro-architectures*
- An Implementation employs:
  - *Architectural registers*
  - *Micro-architectural registers*
  - ***Functional units***
  - *Wires*



The Elephant in the Room

- A CPU *hides* more complex structures: *micro-architectures*
- An Implementation employs:
  - *Architectural registers*
  - *Micro-architectural registers*
  - *Functional units*
  - ***Wires***



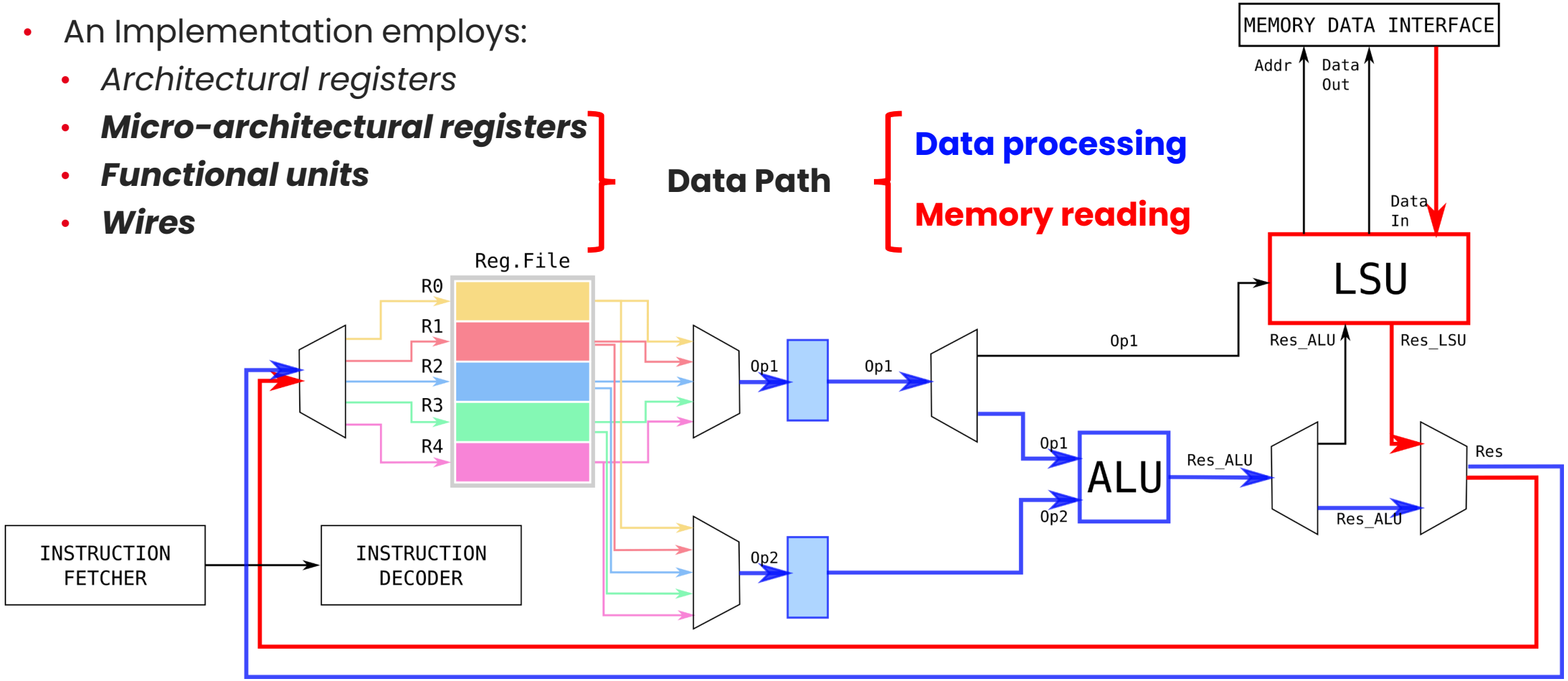
The Elephant in the Room

- A CPU *hides* more complex structures: *micro-architectures*
- An Implementation employs:
  - *Architectural registers*
  - ***Micro-architectural registers***
  - ***Functional units***
  - ***Wires***

Data Path

Data processing

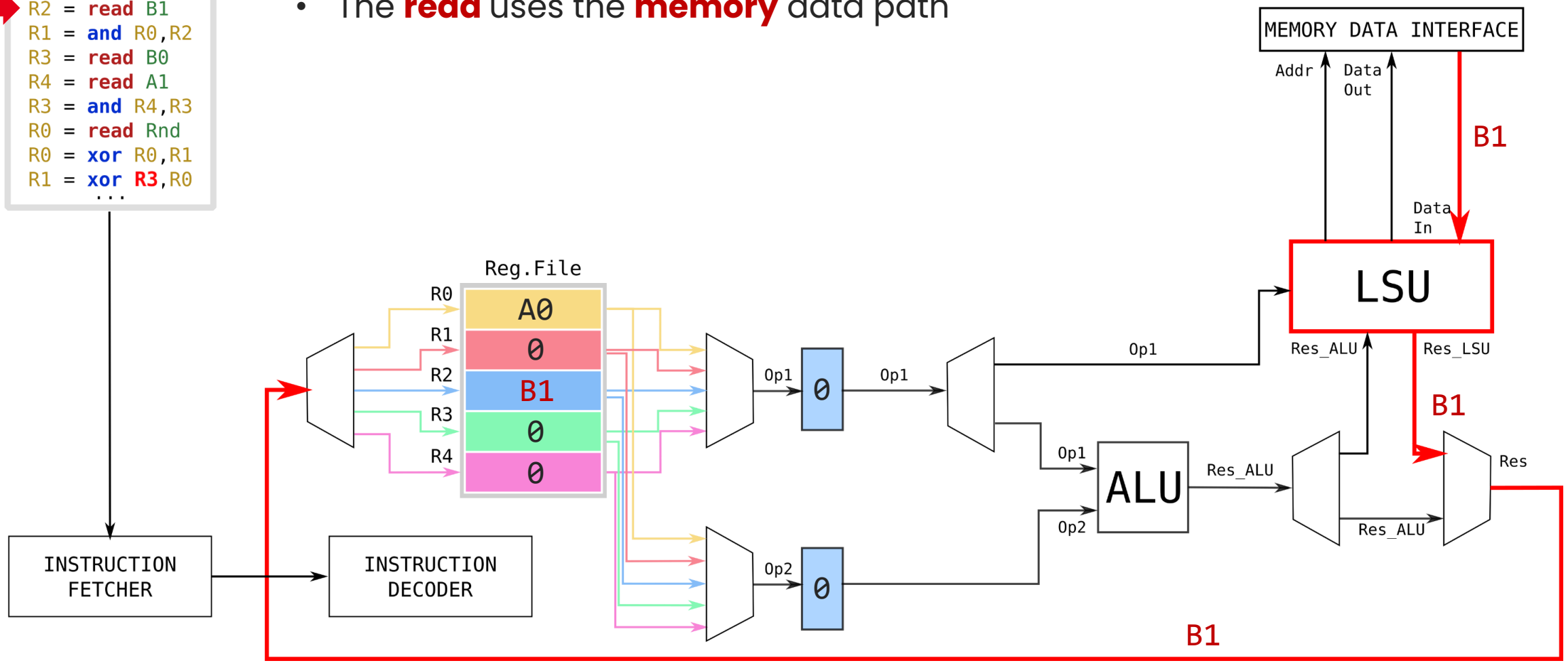
Memory reading



How Does a Micro-architecture Leak?

```
<secAnd2>:
    ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R3,R0
    ...
```

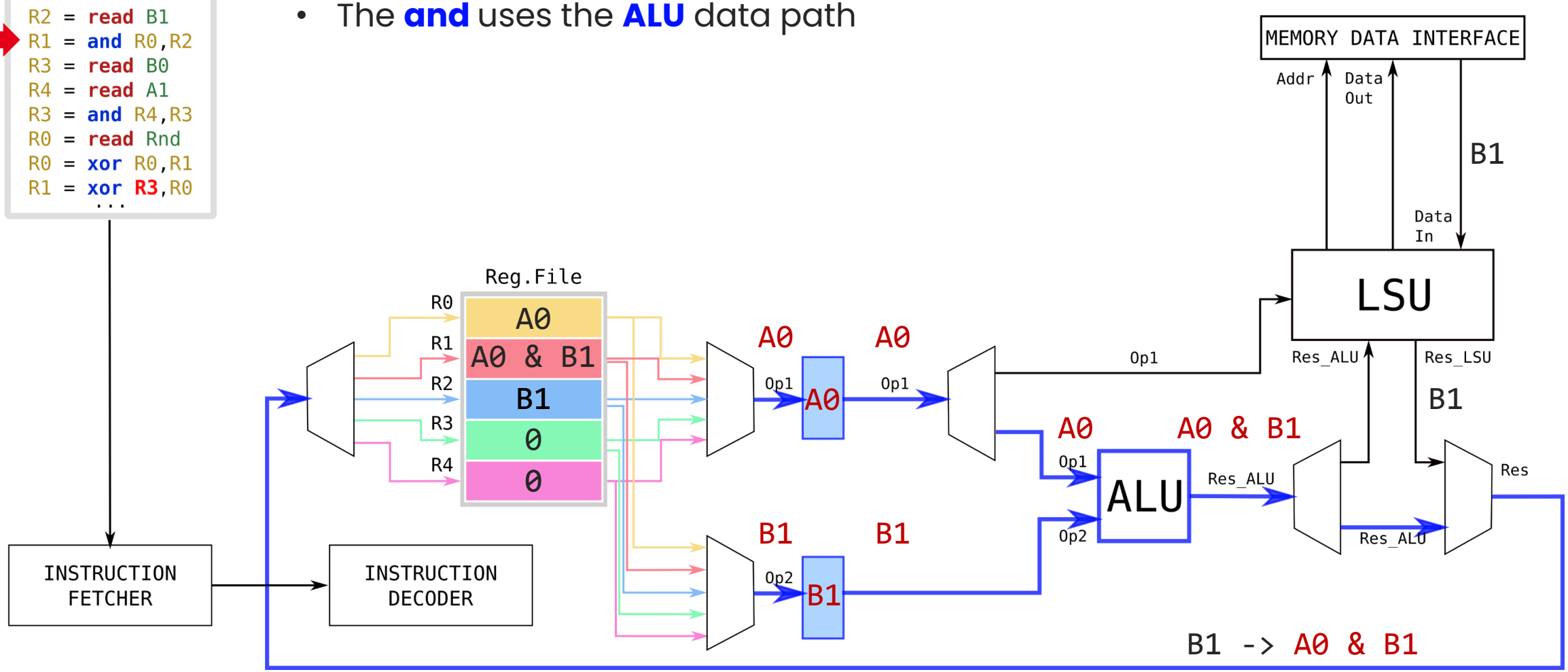
- The CPU read **B1** from the memory
- The **read** uses the **memory** data path



How Does a Micro-architecture Leak?

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R3,R0
        ...
```

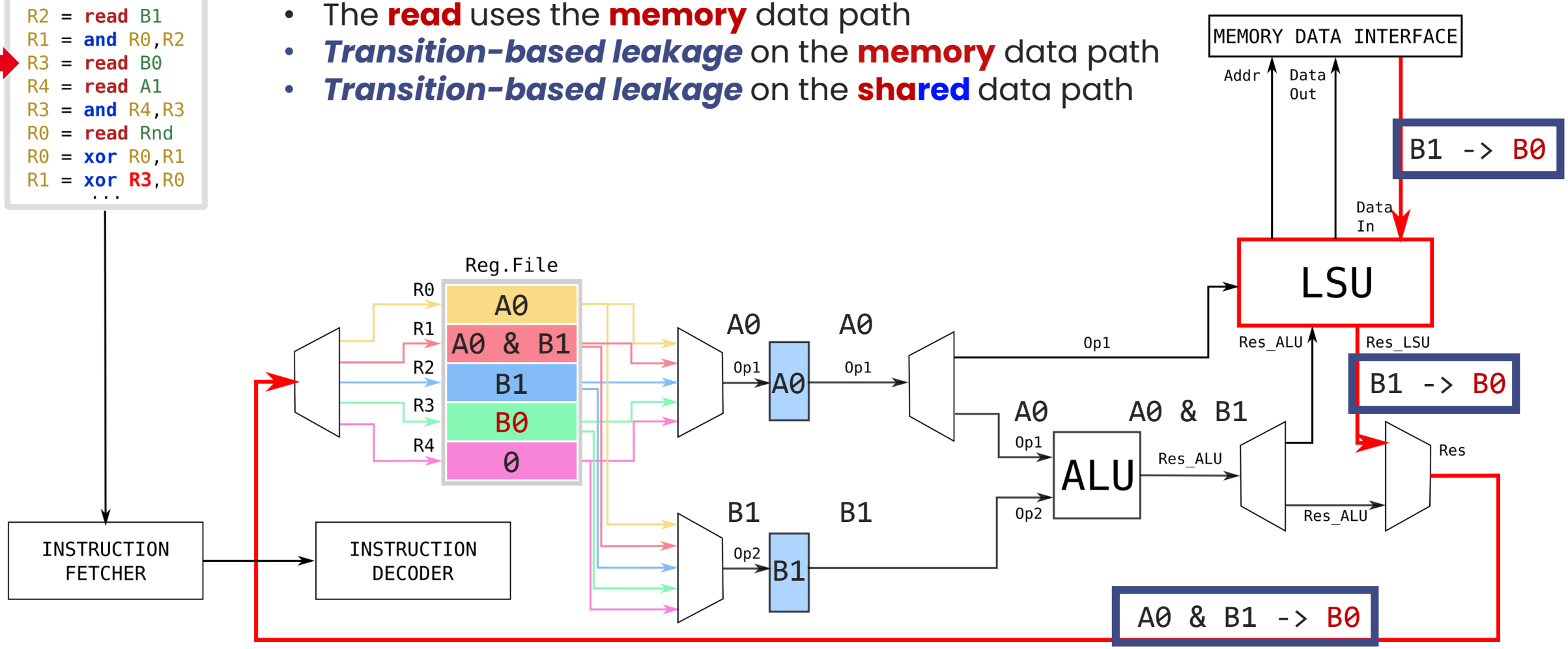
- The CPU read **A0** and **B1** from the Reg. File
- The **and** uses the **ALU** data path



How Does a Micro-architecture Leak?

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R3,R0
        ...
```

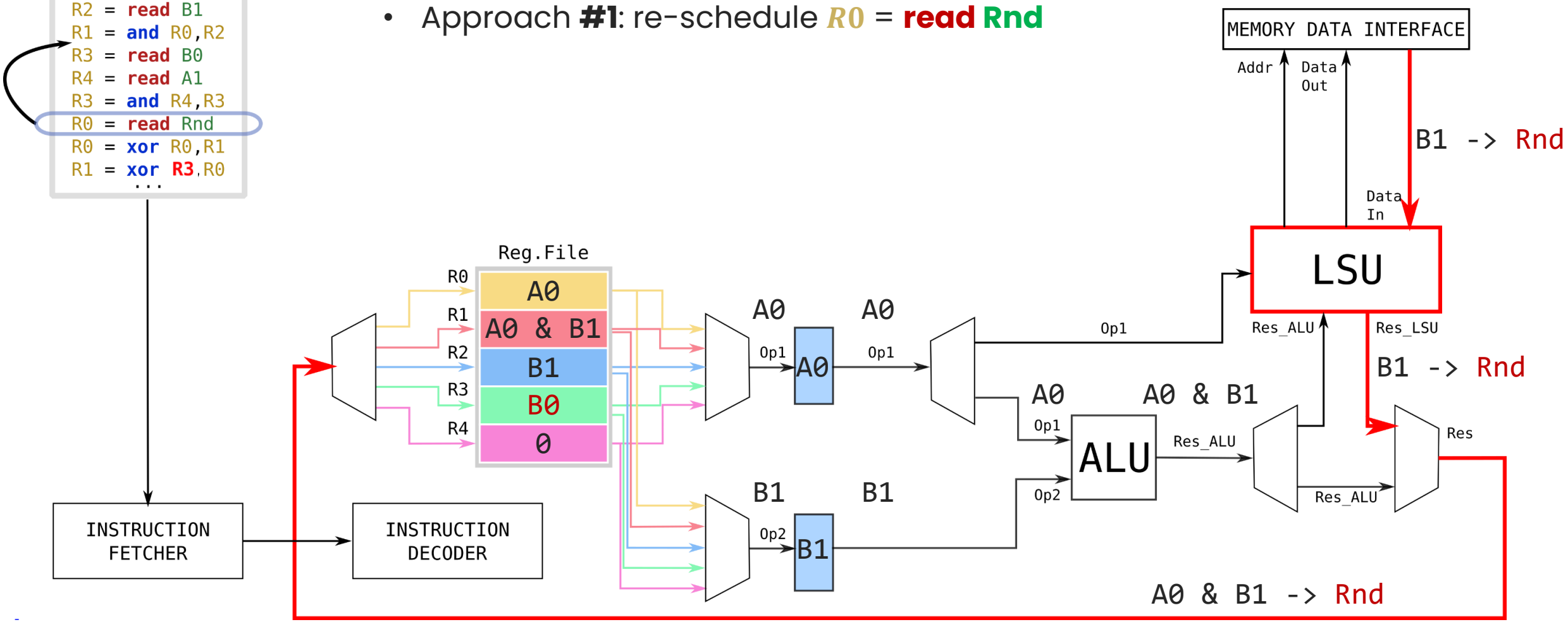
- The CPU reads **B0** from the memory
- The **read** uses the **memory** data path
- ***Transition-based leakage*** on the **memory** data path
- ***Transition-based leakage*** on the **shared** data path



How Do We Handle the Micro-architectural Leakage?

```
<secAnd2>:
    ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
R1 = xor R3,R0
    ...
```

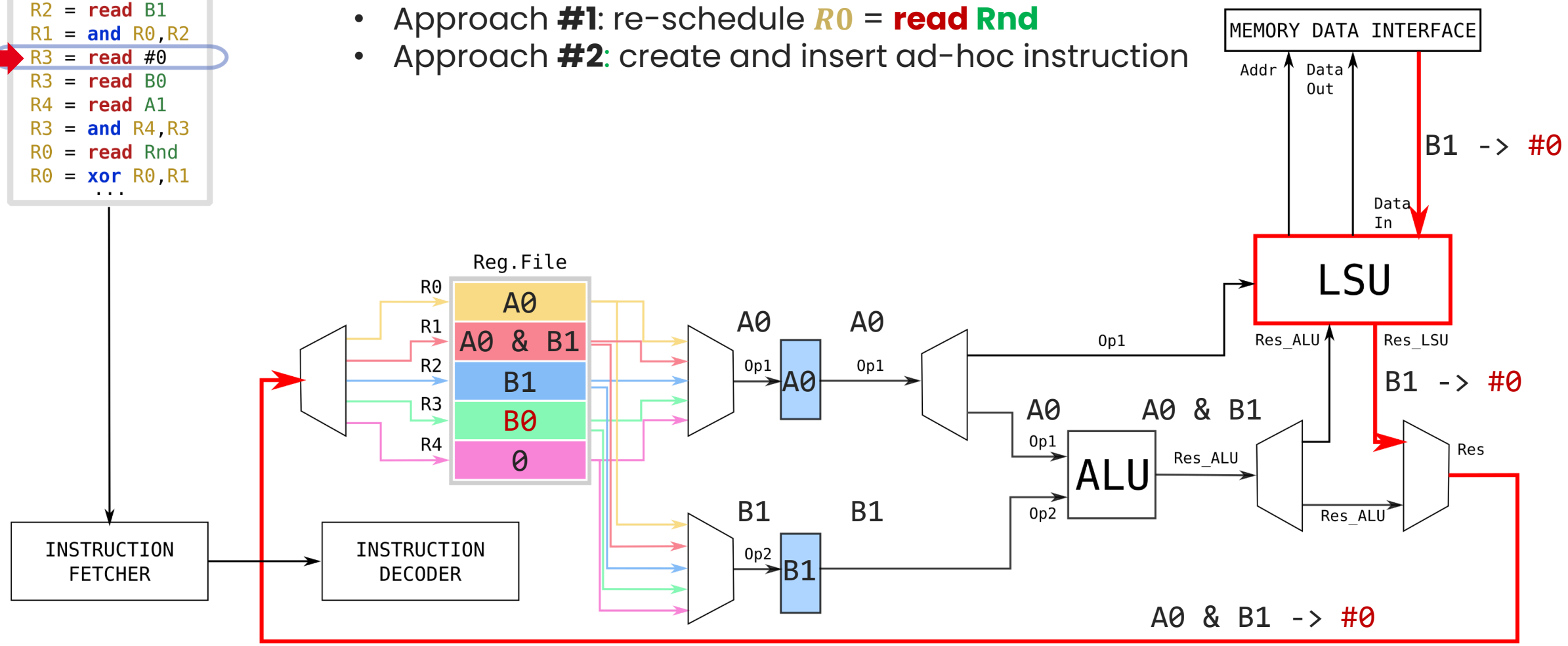
- **Solution:** flush (overwrite) data path
  - Approach **#1**: re-schedule $R0$ = **read Rnd**



How Do We Handle the Micro-architectural Leakage?

```
<secAnd2>:
    ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read #0
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
    ...
```

- **Solution:** flush (overwrite) data path
  - Approach **#1**: re-schedule *R0* = **read Rnd**
  - Approach **#2**: create and insert ad-hoc instruction



Research Question and Thesis Contributions

- We can prove security of masked *algorithms* (ILA satisfied)
- Yet, the security proofs does not immediately translate to *implementations*
- What **solutions** can we provide?

1st Solution

Software Implementation Modification

+

Micro-architecture Information

→

1st Contribution

Automated Mitigation Transition-based Leakages

2nd Solution

Employment of *alternative* masking schemes

→

2nd Contribution

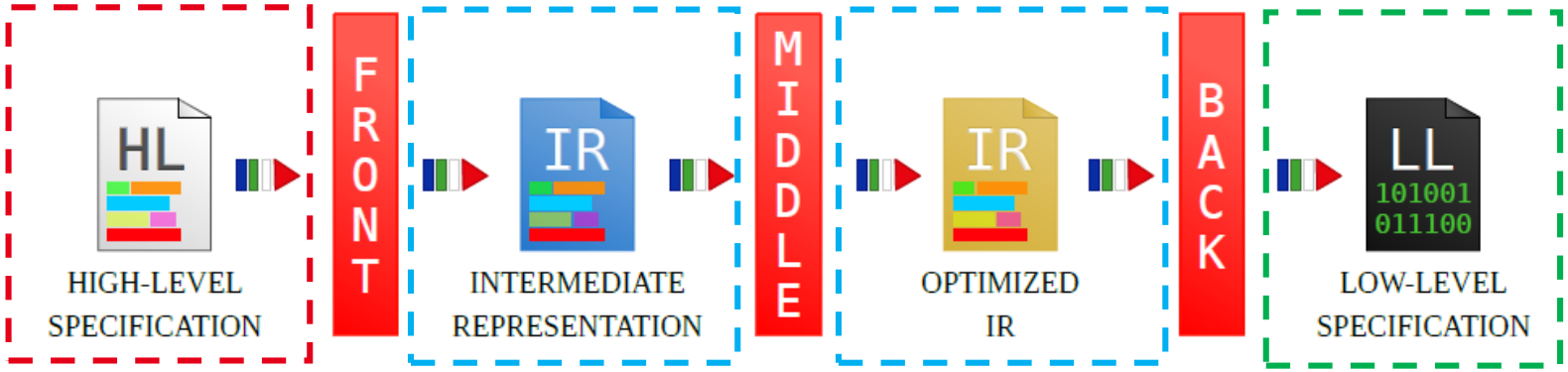
The Impact of the Micro-architecture on Masking Schemes

1. Automated Methodology to Mitigate Transition-based Leakages

SORBONNE UNIVERSITÉ + cea + POLITECNICO MILANO 1863

Overview of the Compilation Process

COMPILER MODULAR ORGANIZATION



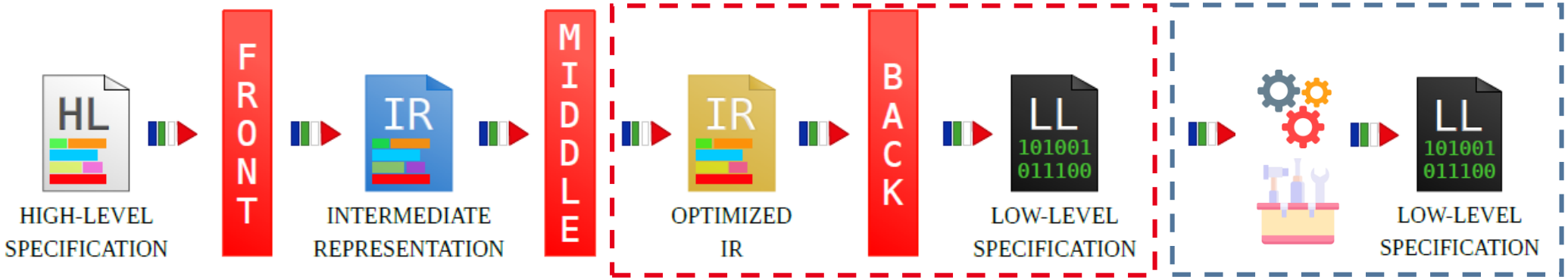

```
SEC-AND2(A0, A1, B0, B1):
  C0  = (A0 and B0) xor R
  X01 = A0 and B1
  X10 = A1 and B0
  X11 = A1 and B1
  C1  = (((X01 xor R) xor X10)) xor X11)
  return <C0, C1>
```

LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

```
<secAnd2>:
  ...
  R0 = read A0
  R2 = read B1
  R1 = and R0,R2
  R3 = read B0
  R4 = read A1
  R3 = and R4,R3
  R0 = read Rnd
  R0 = xor R0,R1
  R1 = xor R2,R0
  ...
```

Overview of Mitigation Approaches



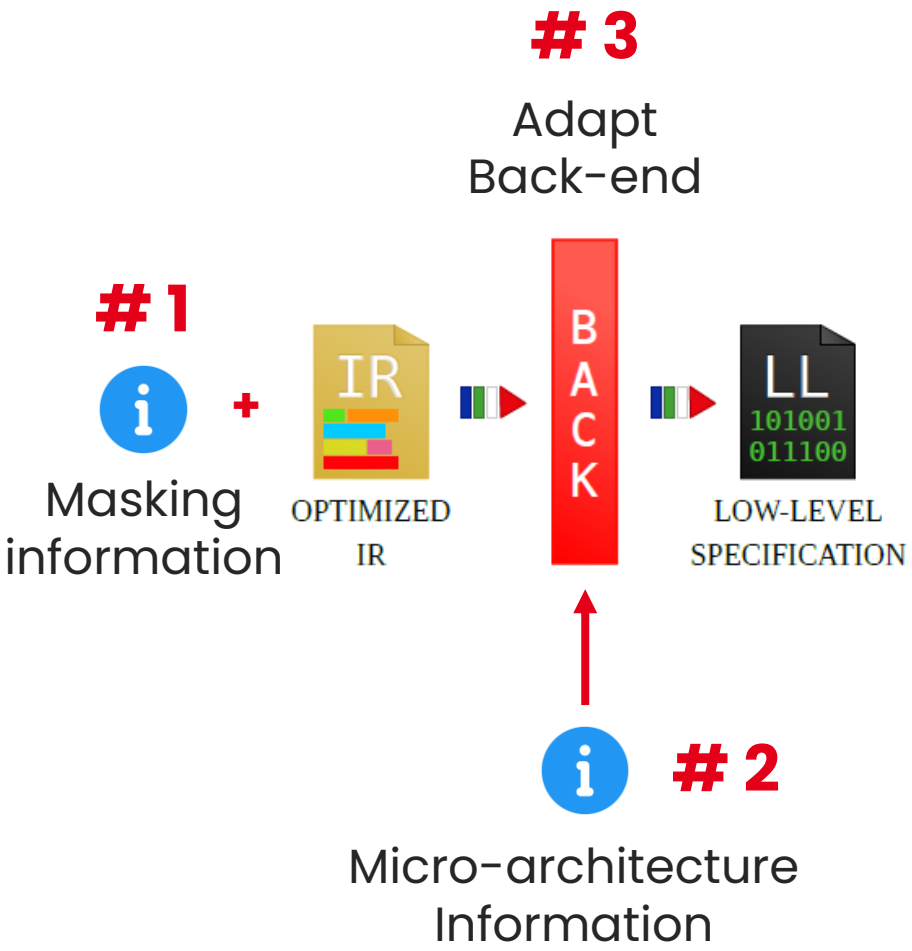
Pro-Active Approach **Reactive Approach**

- **Pro-Active Approach:**
  - Mitigation ***during*** compilation
  - Exploit information on the program
    - Retrieved by the compiler
  - More effective leakage mitigation

- **Reactive Approach:**
  - Mitigation ***after*** compilation
  - No information on the program
  - Less effective leakage mitigation

Requirements

- **Goal:** produce a leakage-free implementation
- **Requirements:**
  1. Identify intermediate variables to keep apart
  2. Specify micro-architectural details
  3. Adapt back-end to avoid transition-based leakages



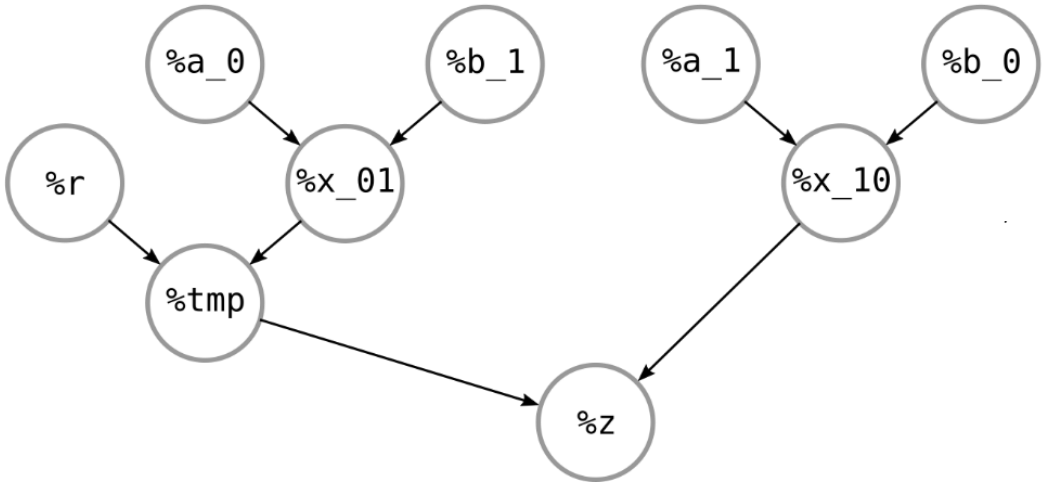
Requirement #1: Masking Information

Compiler works on

LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

DATA-DEPENDECY GRAPH



Goal: identify **intermediate variables** to keep apart

Requirement #1: Masking Information
– Share Information –

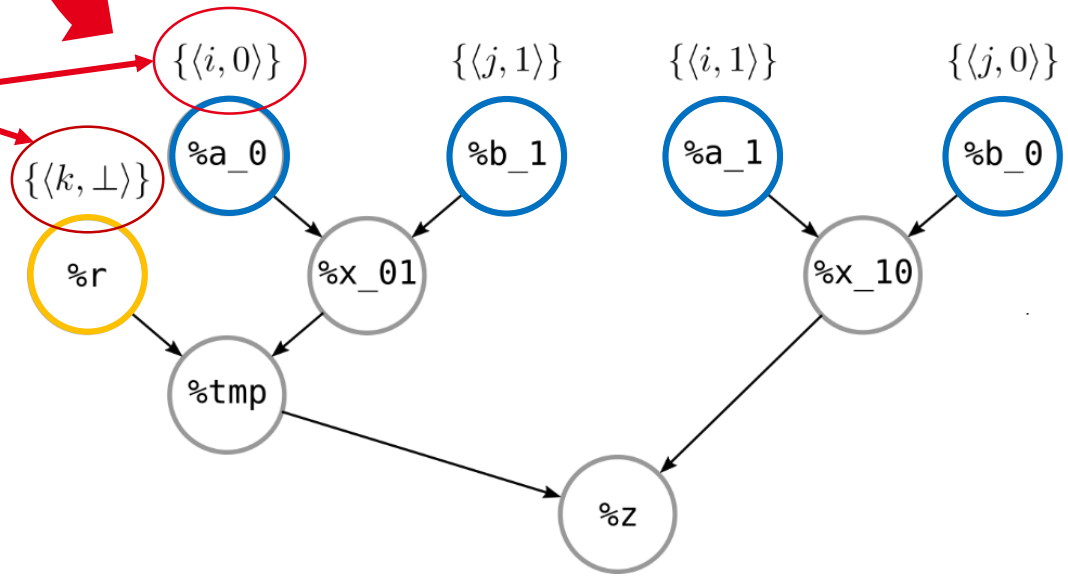
LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

Compiler works on

DATA-DEPENDECY GRAPH

Encoding Tag



Goal: identify **intermediate variables** to keep apart

Identify **input shares** and **random** variables

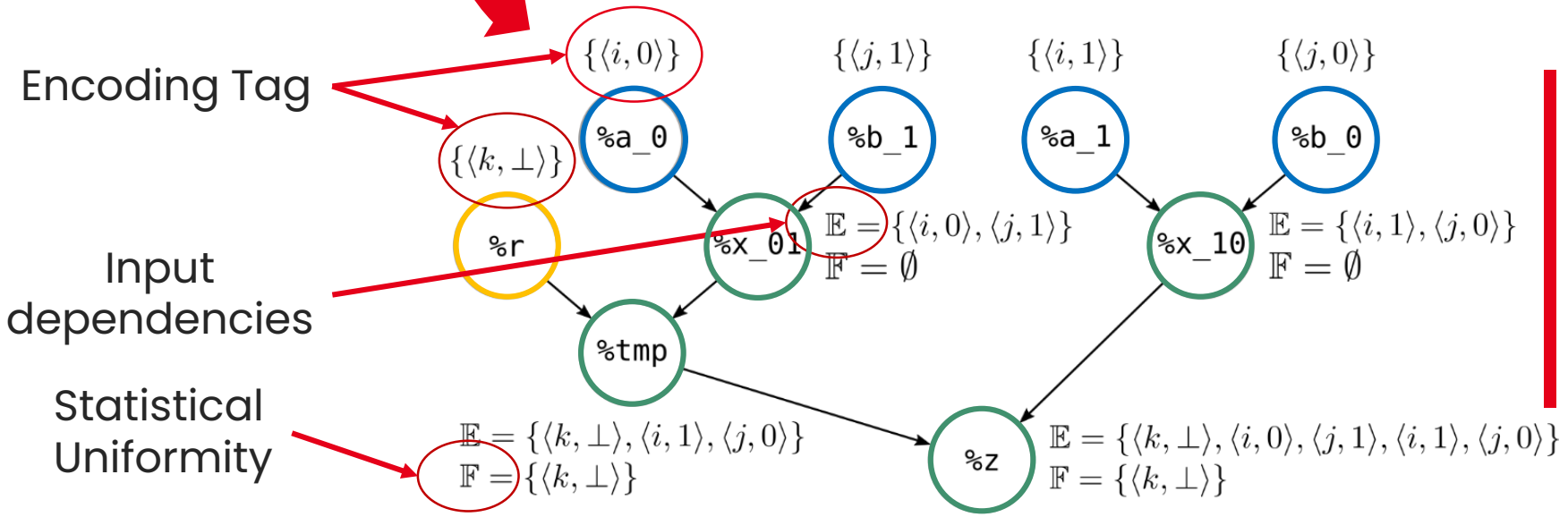
Requirement #1: Masking Information
– Share Propagation –

LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

Compiler works on

DATA-DEPENDECY GRAPH



Goal: identify **intermediate variables** to keep apart

Propagate **input dependencies** and **statistical uniformity** to **intermediate variables**

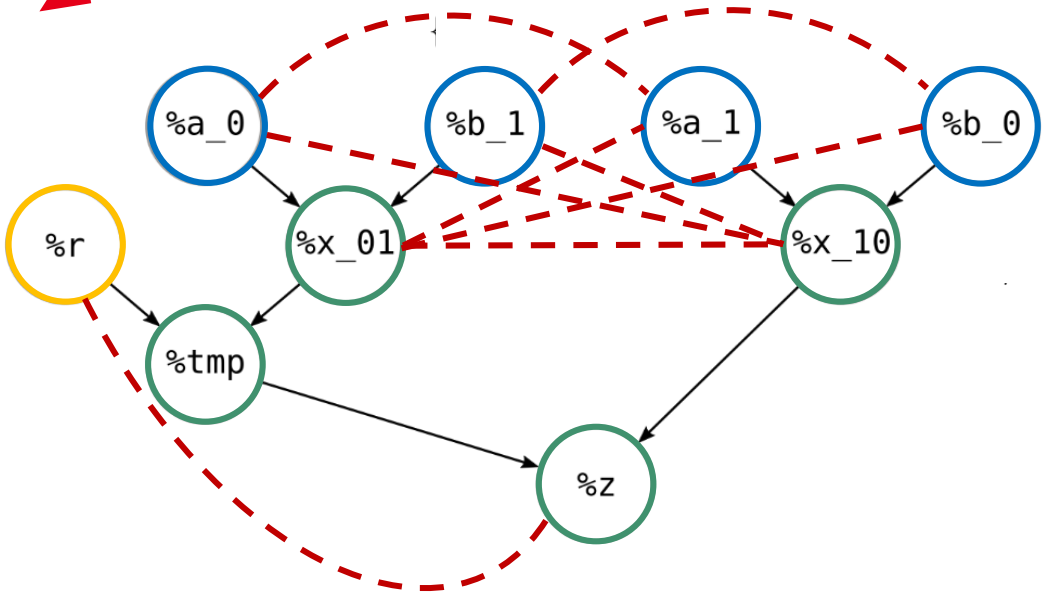
Requirement #1: Masking Information
– Leakage Relation –

LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

Compiler works on

DATA-DEPENDECY GRAPH



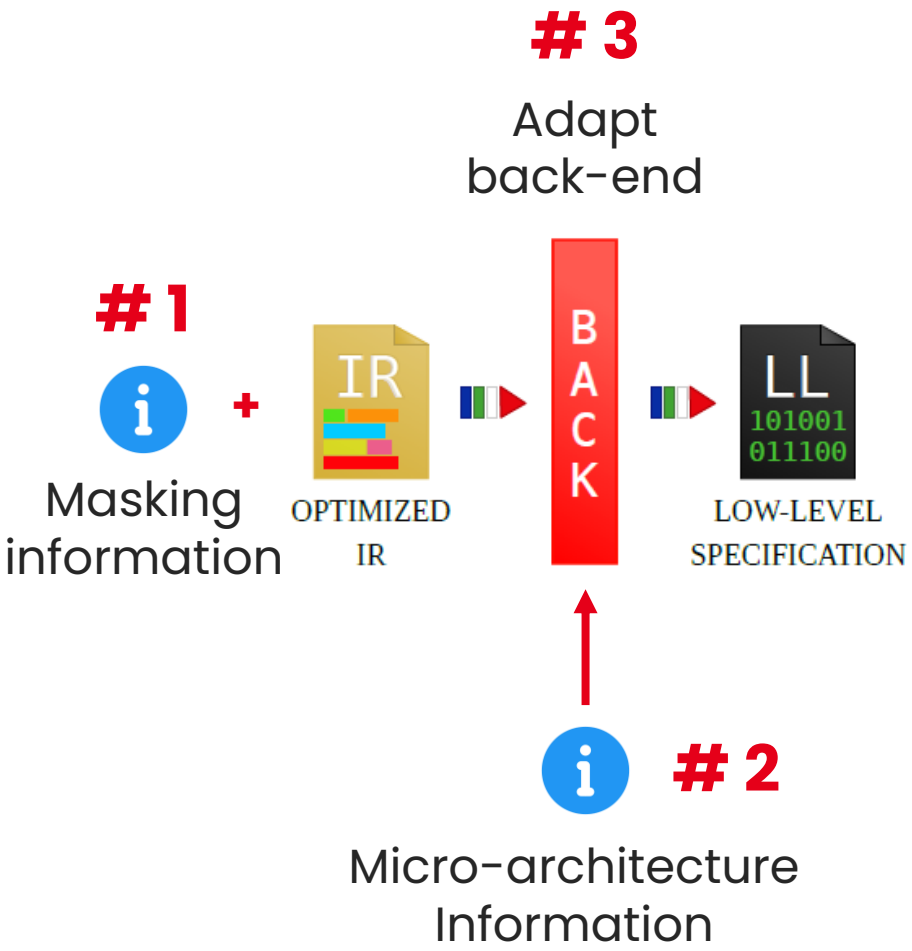
Goal: identify **intermediate variables** to keep apart

Which allows

Compiler computes a **leakage relation** expressing which recombinations leaks information

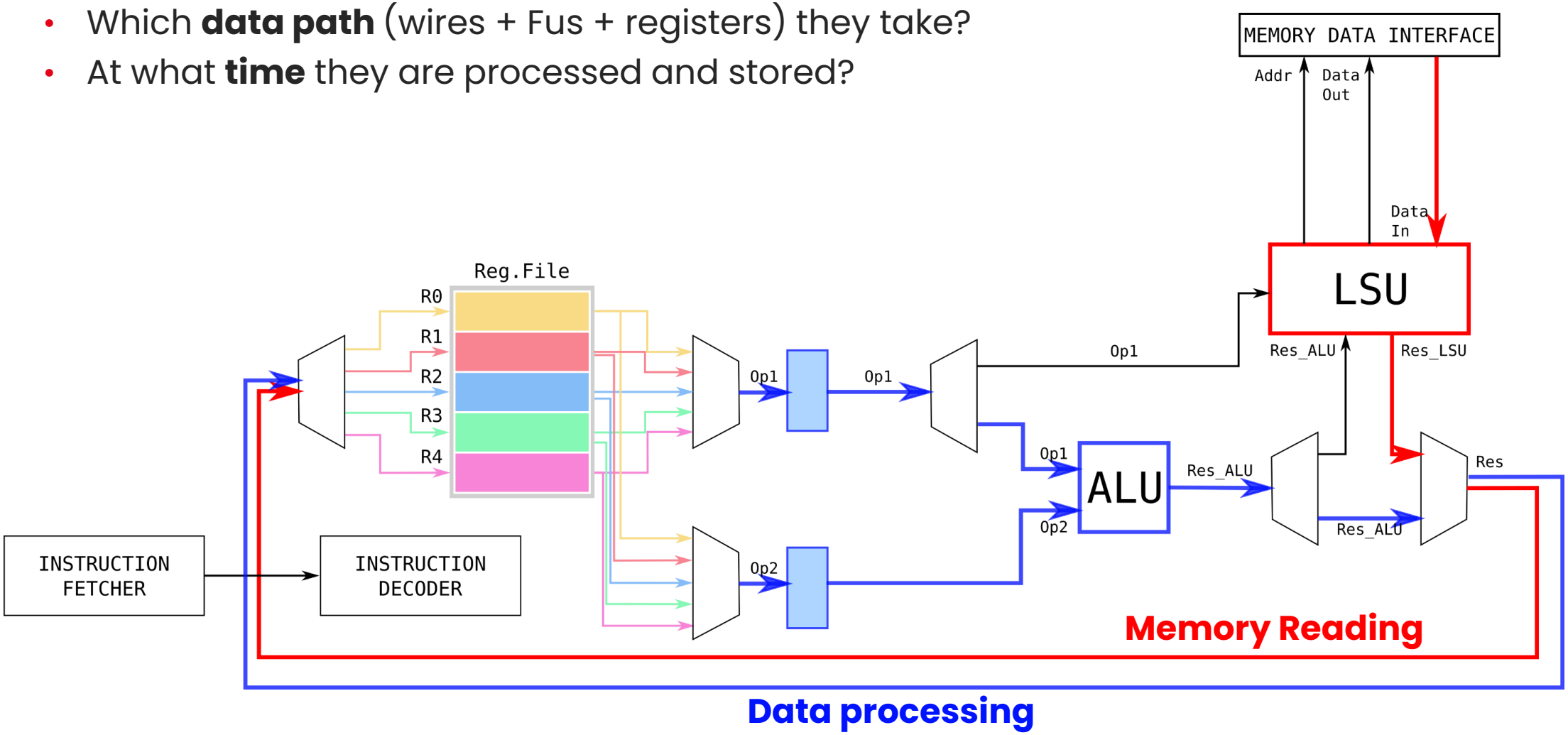
Requirements

- **Goal:** produce a leakage-free implementation
- **Requirements:**
  1. Identify intermediate variables to keep apart
  2. Specify micro-architectural details
  3. Adapt back-end to avoid transition-based leakages



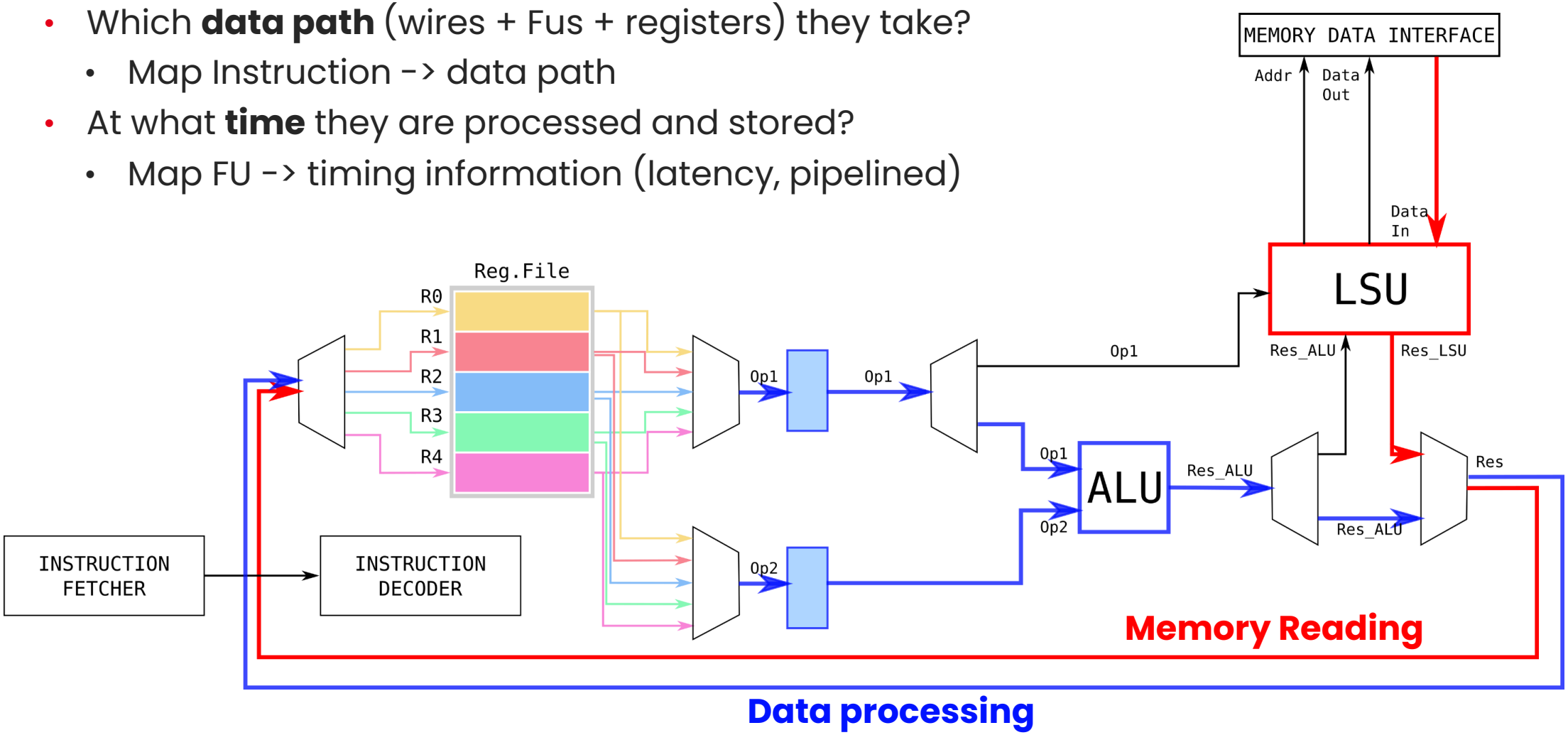
Requirement #2: Micro-architectural Information

- **Question**: how data flow in the micro-architecture?
  - Which **data path** (wires + Fus + registers) they take?
  - At what **time** they are processed and stored?



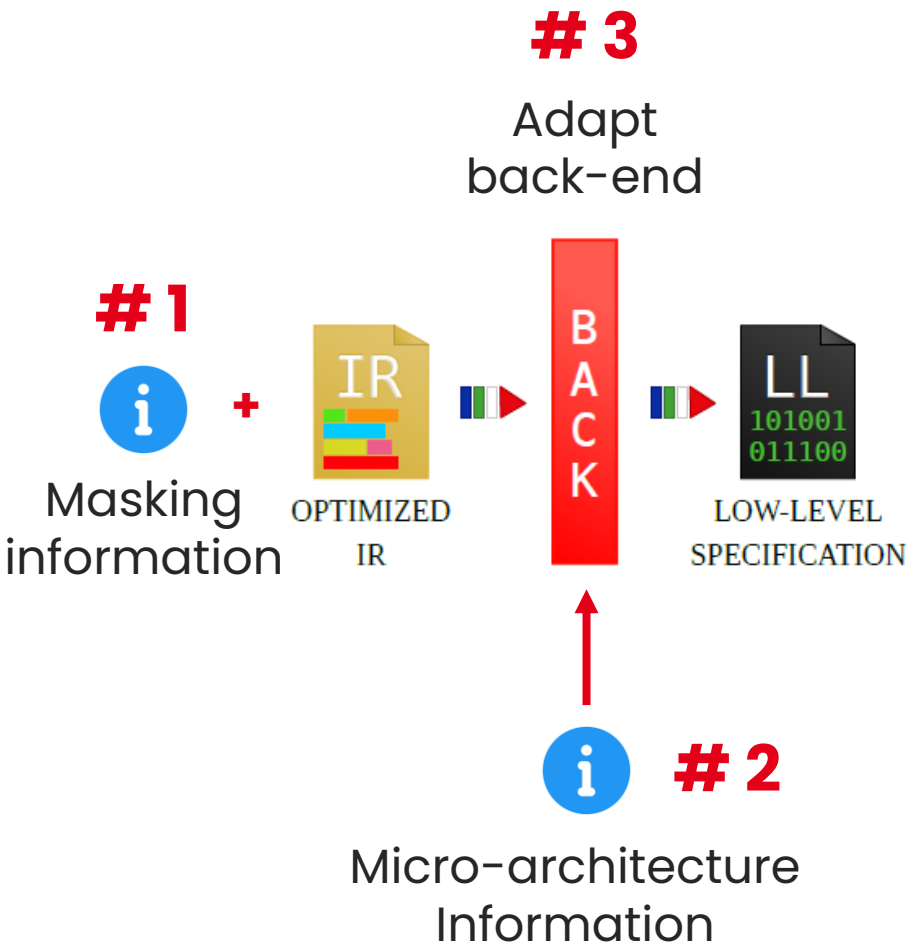
Requirement #2: Micro-architectural Information

- **Question**: how data flow in the micro-architecture?
  - Which **data path** (wires + Fus + registers) they take?
    - Map Instruction -> data path
  - At what **time** they are processed and stored?
    - Map FU -> timing information (latency, pipelined)



Requirements

- **Goal:** produce a leakage-free implementation
- **Requirements:**
  1. Identify intermediate variables to keep apart
  2. Specify micro-architectural details
  3. Adapt back-end to avoid transition-based leakages

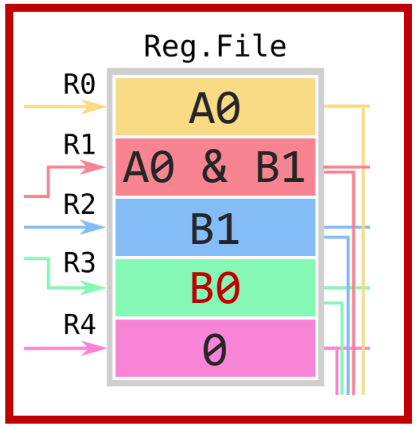


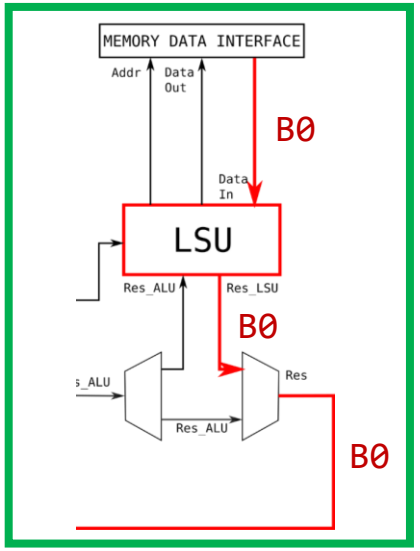
Requirement #3: Adapt Compiler's Back-end

- **Question:** how to mitigate transition-based leakages?
  - Careful **Register Allocation**
  - Careful **Instruction Scheduling**

  } Code Generation Algorithms

- **Adaptation steps:**
  1. **Introduce** concept of ***state*** S_μ **:**
     - **Register allocation**: architectural registers content
     - **Instruction scheduling**:
       - Data on the data path
       - FU execution state (ready, busy, ready in T time instants)
  2. **Simulate** state evolution: update heuristic to update S_μ with each choice
  3. **Leakage constraint:** transition-based leakage cannot occur in S_μ
  4. **Choice selection**: check leakage constraint





Requirement #3: Guarantee Convergence

- **All intermediate choice leaks**:
  - **Register allocation**: cannot change register
  - **Instruction scheduling**: cannot change instructions order
- **Flushing**: add instructions to:
  - **Register allocation** : overwrite leaking register
  - **Instruction scheduling** : overwrite leaking data path
- **Remarks:** add an instruction -> increase exec. time
  - Flush only if needed
  - Overwrite with *constant values*

} **Reduce Performance Impacts**

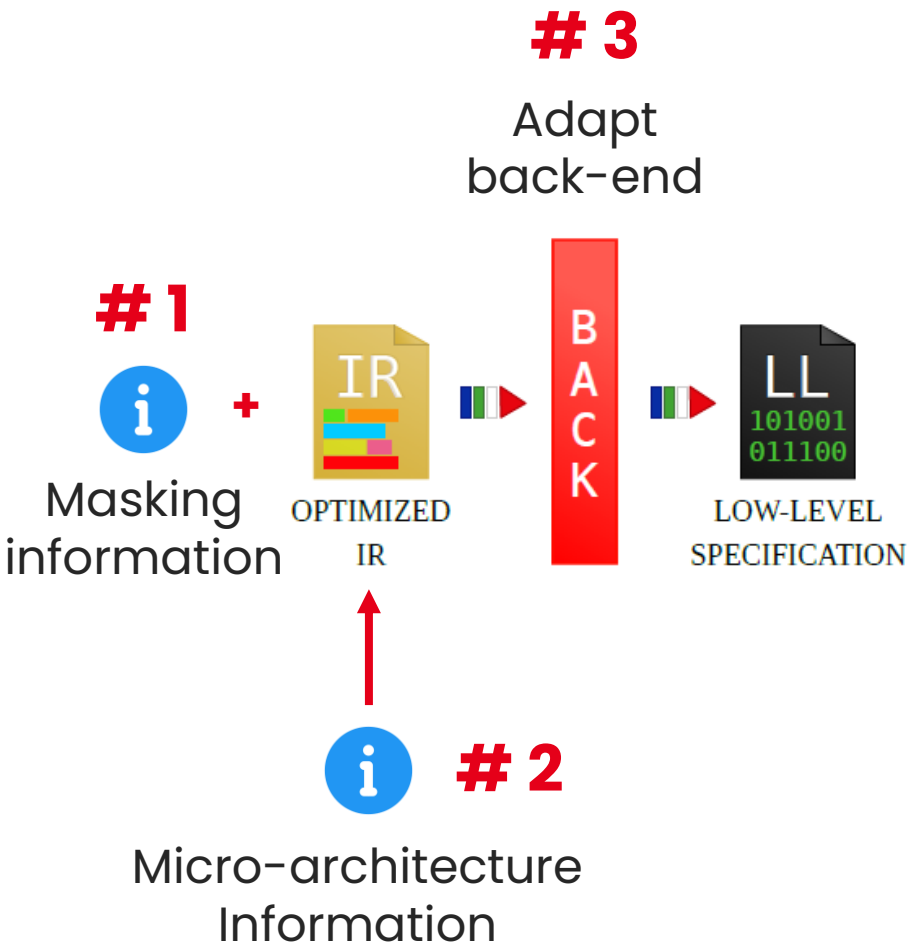
Flushing examples

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R2 = mov #42
R2 = read B0
R0 = mov #0
R0 = read A1
R2 = and R0,R2
R0 = read Rnd
        ...
```

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R1 = and R0,R2
R3 = read #0
R3 = read B0
R4 = read A1
R3 = and R4,R3
R0 = read Rnd
R0 = xor R0,R1
        ...
```

Requirements

- **Goal:** produce a leakage-free implementation
- **Requirements:**
  1. Identify intermediate variables to keep apart
  2. Specify micro-architectural details
  3. Adapt back-end to avoid transition-based leakages



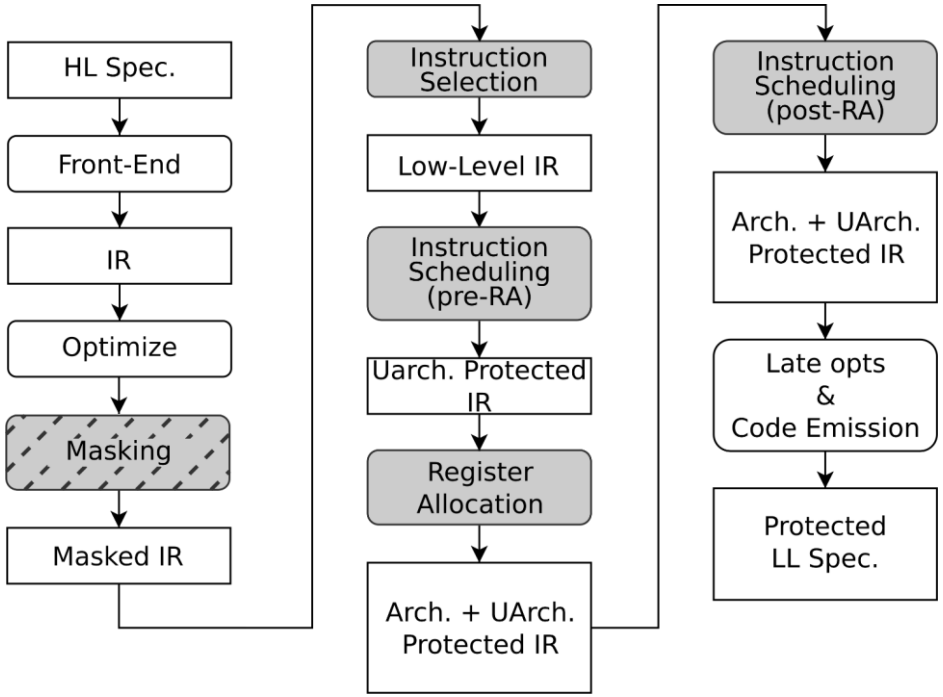
Experimental Evaluation

1. Methodology Implementation
   1. Modification of LLVM-based Compiler
   2. Modified passes in **grey** boxes

2. Experimental Setup
   1. **Benchmark**: SIMON-128/128
      - First and second order **Boolean** masked
      - Verified correct under ILA assumption
   2. **CPU**: Cortex-M4 (STM32F303)
      - Micro-arch. model inferred by public knowledge
   3. **Acquisition**: Chipwhisperer-1200
   4. **Side-channel**: power consumption

3. Evaluation axes
   1. Security
   2. Performance

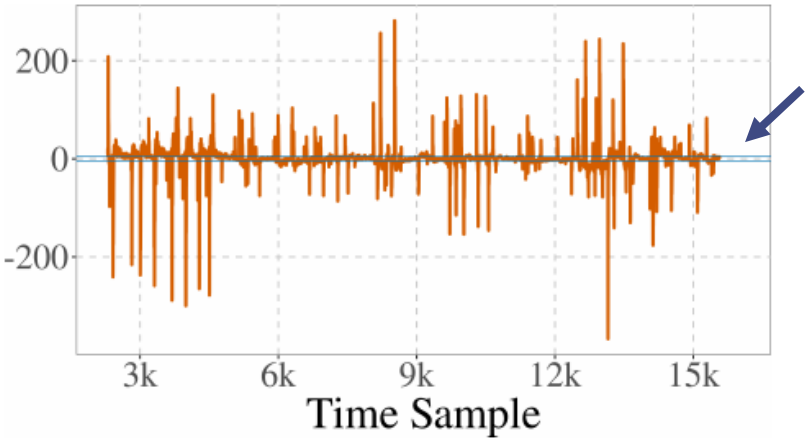
Enhanced LLVM Compiler Pipeline



Security Evaluation

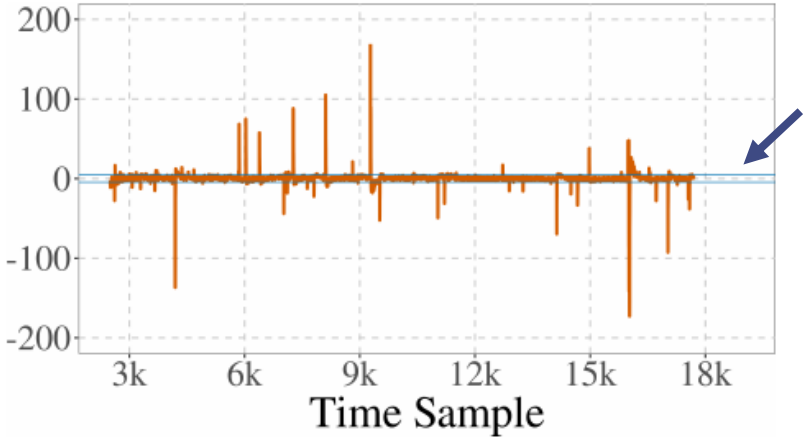
- **Methodology**: detect information leakage along execution of SIMON-128/128 implementation
  - **Blue** lines: borders of the leakage-free area
  - **Orange** peaks: variation of the information leakage metric

1st-order

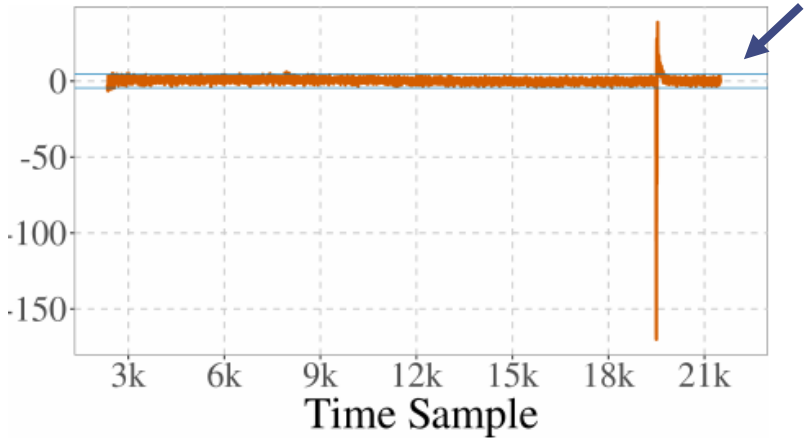


This Work

1st-order



2nd-order



x6 reduction
of leakage peaks

Security Evaluation—Root Cause Analysis

- **(One) Root cause**: new memory-related transition-based leakage
  - **Interaction** between **Z** and **Y**
  - **Observation**: Interleaving **write** and **read** with a **nop**

- **(Potential) Explanation:**
  - Memory optimization
  - If **write** and **read** back-to-back: **read** served first

Conclusion:

- Micro-architectural information still incomplete

<UB-ST-LD-LD>:

```
write Z
R0 = read X
nop
nop
nop
nop
R1 = read Y
```

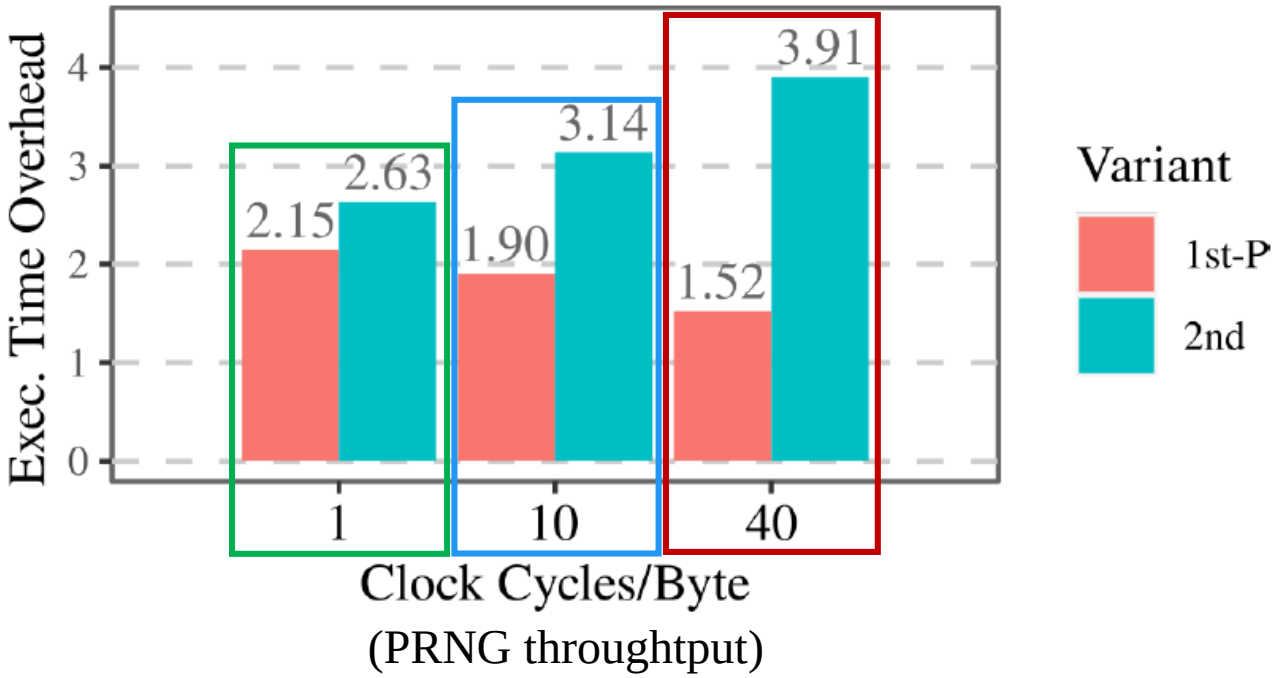
<UB-ST-NOP-LD-LD>:

```
write Z
nop
R0 = read X
nop
nop
nop
nop
R1 = read Y
```

Execution Time Overhead Evaluation

Remarks:

- Masking requires ***randomness***
- PRNG throughtput ***impacts*** on execution time
- Randomness ***exponentially*** increase with masking order
- **Our** methodology (**1st-P**) requires **same** randomness of **naïve 1st-order** implementation



Micro-architectural model incomplete

- More transition-based leakage to handle
- Potentially, worse performance figures for 1st-P
- Yet, we won't invert the plotted trend in real use cases

Considered **3** PRNG throughputs:

- **Ideal**: 1 clock cycle per byte
- **Real #1**: 10 clock cycles per byte
- **Real #2**: 40 clock cycles per byte

Summary and Conclusion

- **Contribution**: automated methodology to mitigate transition-based leakages
  - **Goal:** investigating employment of fine-grained micro-architectural details
  - **How:** adapting compilation tools
  - **Results**: unexpected leakage sources prevent fair assessment of the approach

- **Related Work:**
  - **Pro-active** [Seuschek17][Wang19][Tsoupidi23]:
    - Show how to guarantee convergence to a leakage-free solution
    - Show which micro-architectural information to consider and how to integrate It

- **We need further investigation:**
  - **How**:
    - Full micro-architectural model
    - Open-source micro-architecture designs

2. The Impact of the Micro-architecture on Masking Schemes

Micro-architecture and Alternative Masking Schemes

Literature on micro-architecture's impact

main focus

Boolean masking
(particularly sensitive to transition-based leakages)

(Again) Literature on micro-architecture's impact, e.g., [Meyer20]

suggest to use

Alternative Masking

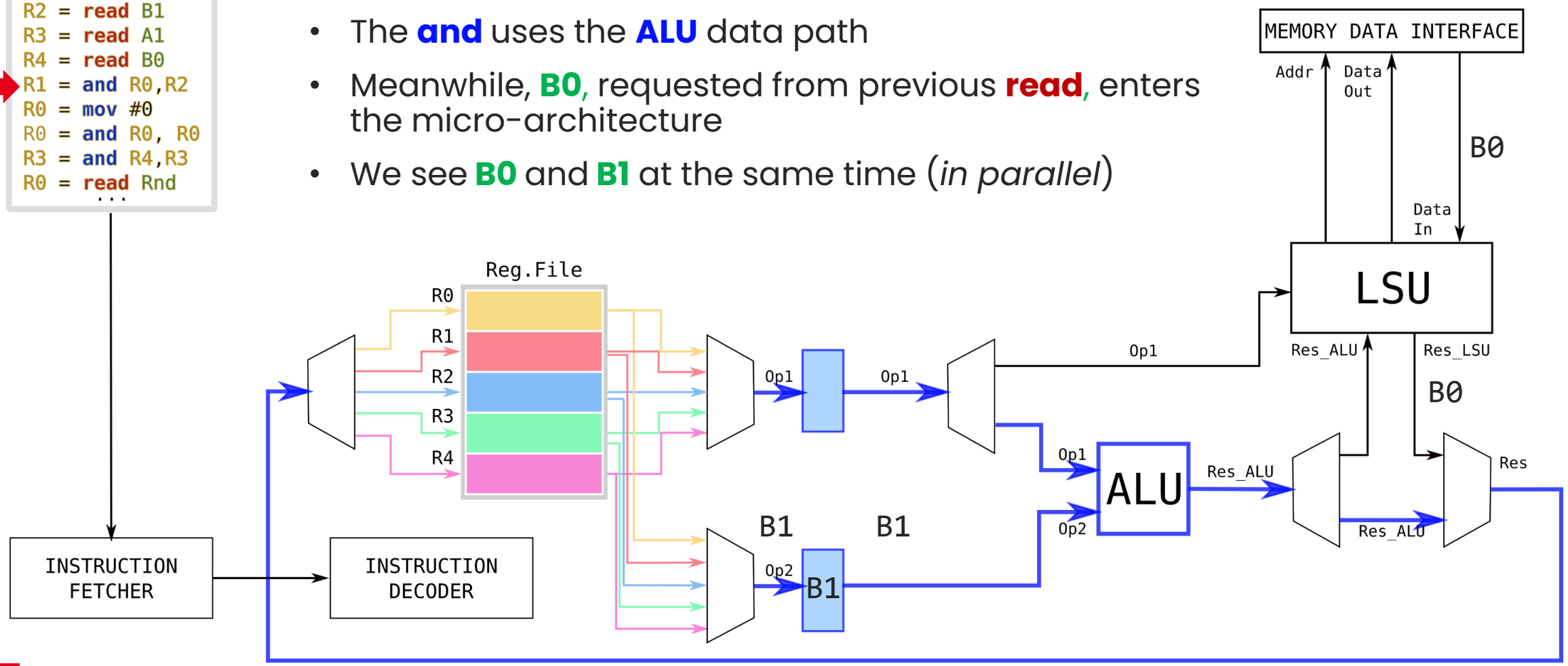
Arithmetic-Sum Masking

Inner-Product Masking
(*immune to transition-based leakages*)

Only Transition-based Leakages are a Threat?

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R3 = read A1
R4 = read B0
R1 = and R0,R2
R0 = mov #0
R0 = and R0, R0
R3 = and R4,R3
R0 = read Rnd
        ...
```

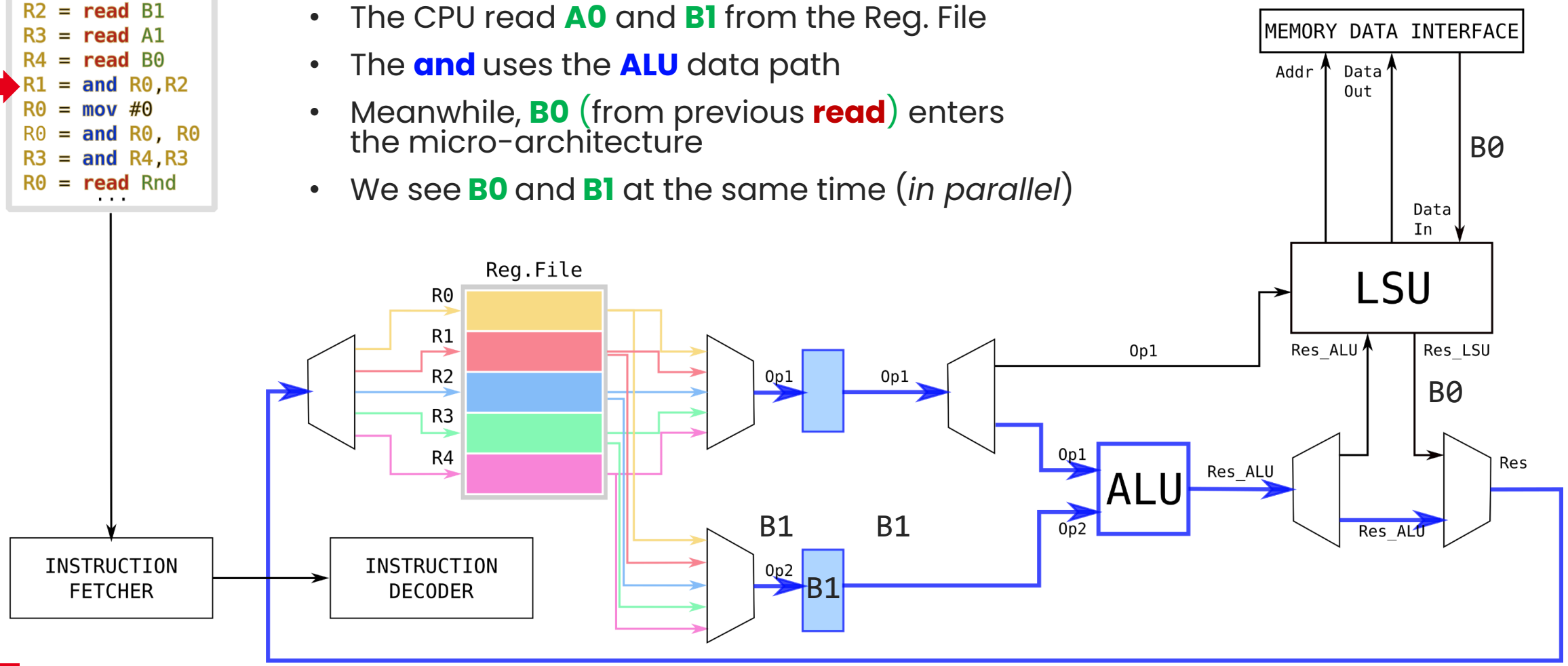
- Modern micro-architectures exhibit data parallelism
- The CPU read **A0** and **B1** from the Reg. File
- The **and** uses the **ALU** data path
- Meanwhile, **B0**, requested from previous **read**, enters the micro-architecture
- We see **B0** and **B1** at the same time (*in parallel*)



Only Transition-based Leakages are a Threat?

```
<secAnd2>:
        ...
R0 = read A0
R2 = read B1
R3 = read A1
R4 = read B0
R1 = and R0,R2
R0 = mov #0
R0 = and R0, R0
R3 = and R4,R3
R0 = read Rnd
        ...
```

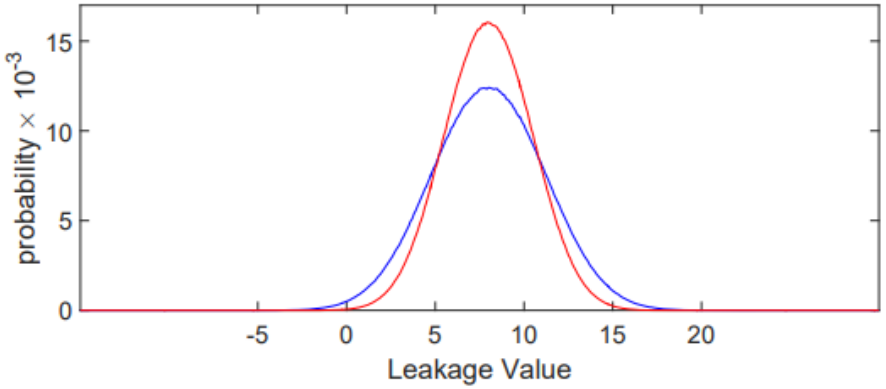
- **Assumption**: CPU processes one share per clock cycle
- **Actually**: micro-architectures exhibit *data parallelism*
- The CPU read **A0** and **B1** from the Reg. File
- The **and** uses the **ALU** data path
- Meanwhile, **B0** (from previous **read**) enters the micro-architecture
- We see **B0** and **B1** at the same time (*in parallel*)



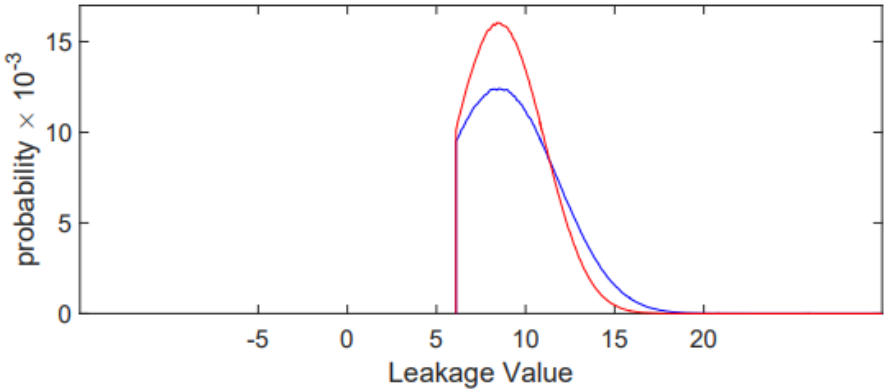
Masked Hardware and Data Parallelism

- We cannot *efficiently* exploit data parallelism as it is
  - We need *higher-order* statistical analyses
  - We need more side channel observations
- Moos and Moradi shown how to efficiently take advantage of these parallelism [Moos17]
  - **How**: filter out certain leakage values (distribution bias)
  - **Target**: ***Boolean*** masked ***hardware*** implementations

Side channel distribution for two sensitive values (in **red** and **blue**)

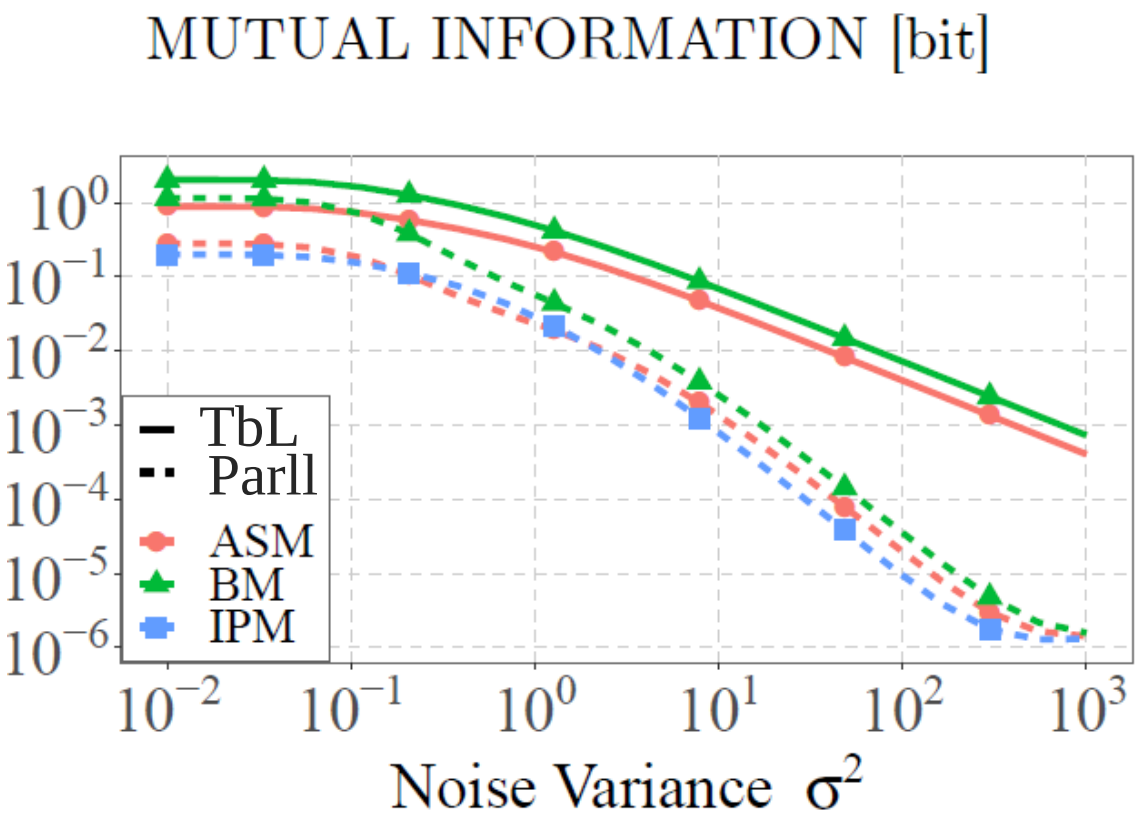


Biased side channel distribution



Masking Schemes: an Observation

Masking	Transition-Based Leakage (TbL)	Data Parallelism (Parll)
Boolean (BM)	Sensitive	Sensitive
Arithmetic (ASM)	Sensitive	Sensitive
Inner-Product (IPM)	**Not Sensitive**	Sensitive



- Data parallelism might be a threat to:
  - Transition-based immune masking schemes (i.e., IPM)
  - Software implementations with all transition-based leakages mitigated

Outline of the Investigation

1. Observation of data parallelism ☐
2. Exploitability of data parallelism ☐
3. Leakage Resilience of Fully Masked Implementations ☐

Correlation-based Analysis 101

- Correlation-based analysis: analyze the dependency between:
  - Observed side channel
  - Leakage model
- Leakage model: side channel ***expected behavior*** when processing an information X
- Correlation coefficient ρ : *quantify* the dependency between two entities

Leakage Model

Observed Side Channel

$$\rho\,[HW(X), T]$$

Observation of data parallelism

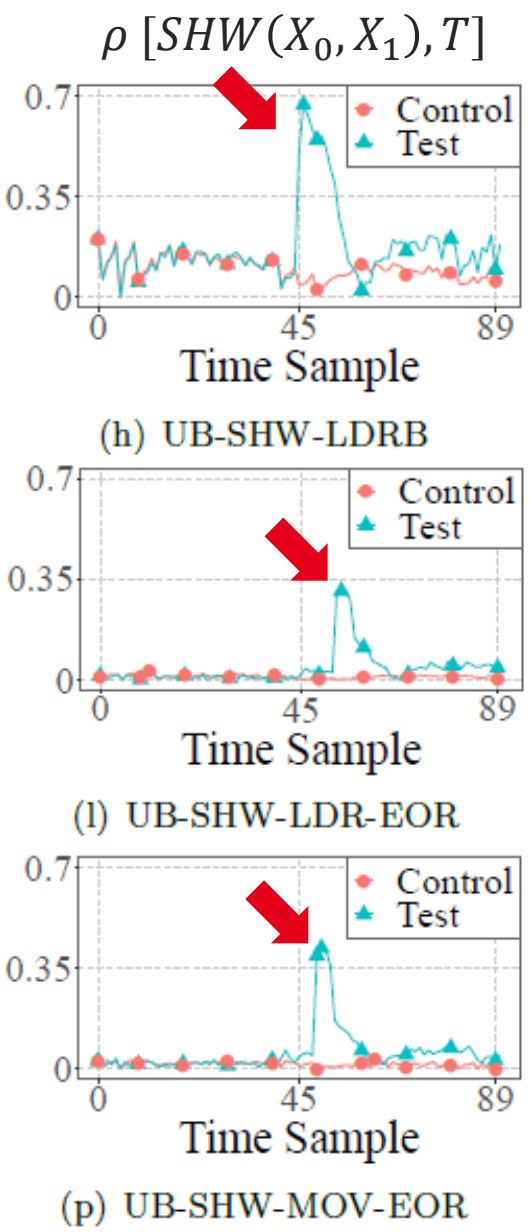
Method:

1. Carefully design code snippet to exhibit data parallelism
2. Run snippet on target CPU
3. Observe side channel behavior T of CPU
4. Choose leakage model for data parallelism

   $SHW(X_0, X_1) = HW(X_0) + HW(X_1)$ ← *share's contribution to side channel*
5. Perform correlation-based analysis

   $\rho\,[SHW(X_0, X_1), T]$

Results: correlation with expected behavior in case of data parallelism



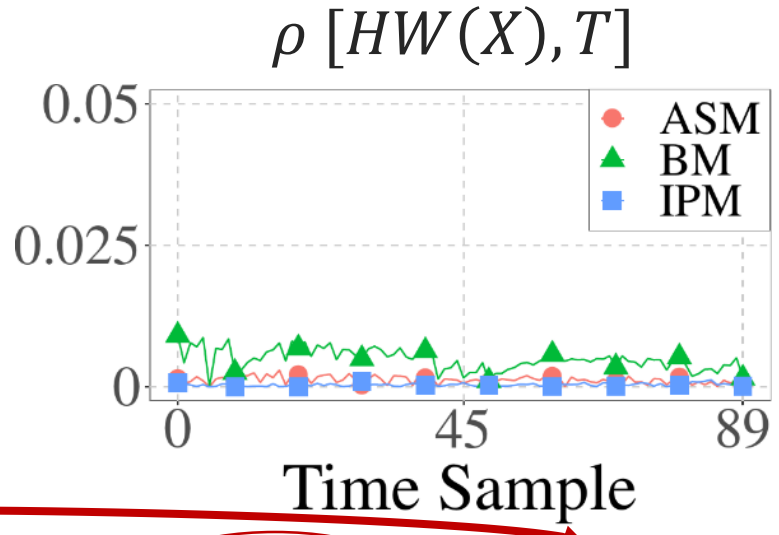
Outline of the Investigation

1. Detect data parallelism ☑
2. Exploitability of data parallelism ☐
3. Leakage Resilience of Fully Masked Implementations ☐

Exploitability of data parallelism

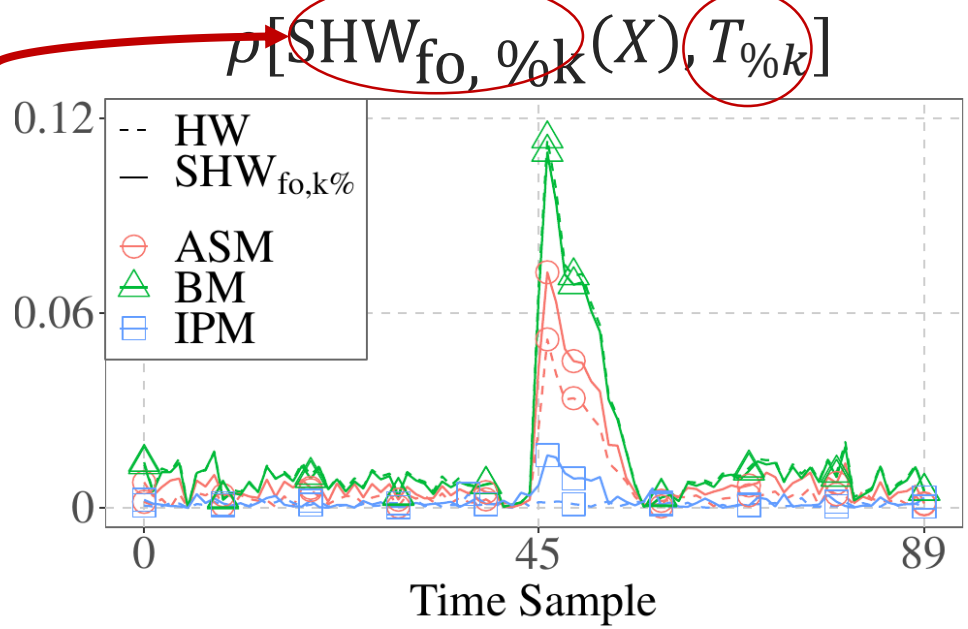
Naïve correlation-based analysis **does not** work

- Correlation-based analysis is a *first-order analysis*
- We need *higher-order* analyses, or…



… biasing the observed leakage behavior [Moos17]
and
custom leakage model

$$\mathrm{SHW}_{\mathrm{fo},\,\%\mathrm{k}}(X) = mean\left(\mathcal{D}_{(HW(X_0)+HW(X_1)),\%k}\right)$$



Results:

- Data-parallelism exploited

Outline of the Investigation

1. Detect data parallelism ☑
2. Exploitability of data parallelism ☑
3. Leakage Resilience of Fully Masked Implementations ☐

Leakage Resilience of Fully Masked Implementations

Use cases:

- Self-implemented 1st order masked AES-128
- Verified correct under ILA assumption

Effects:

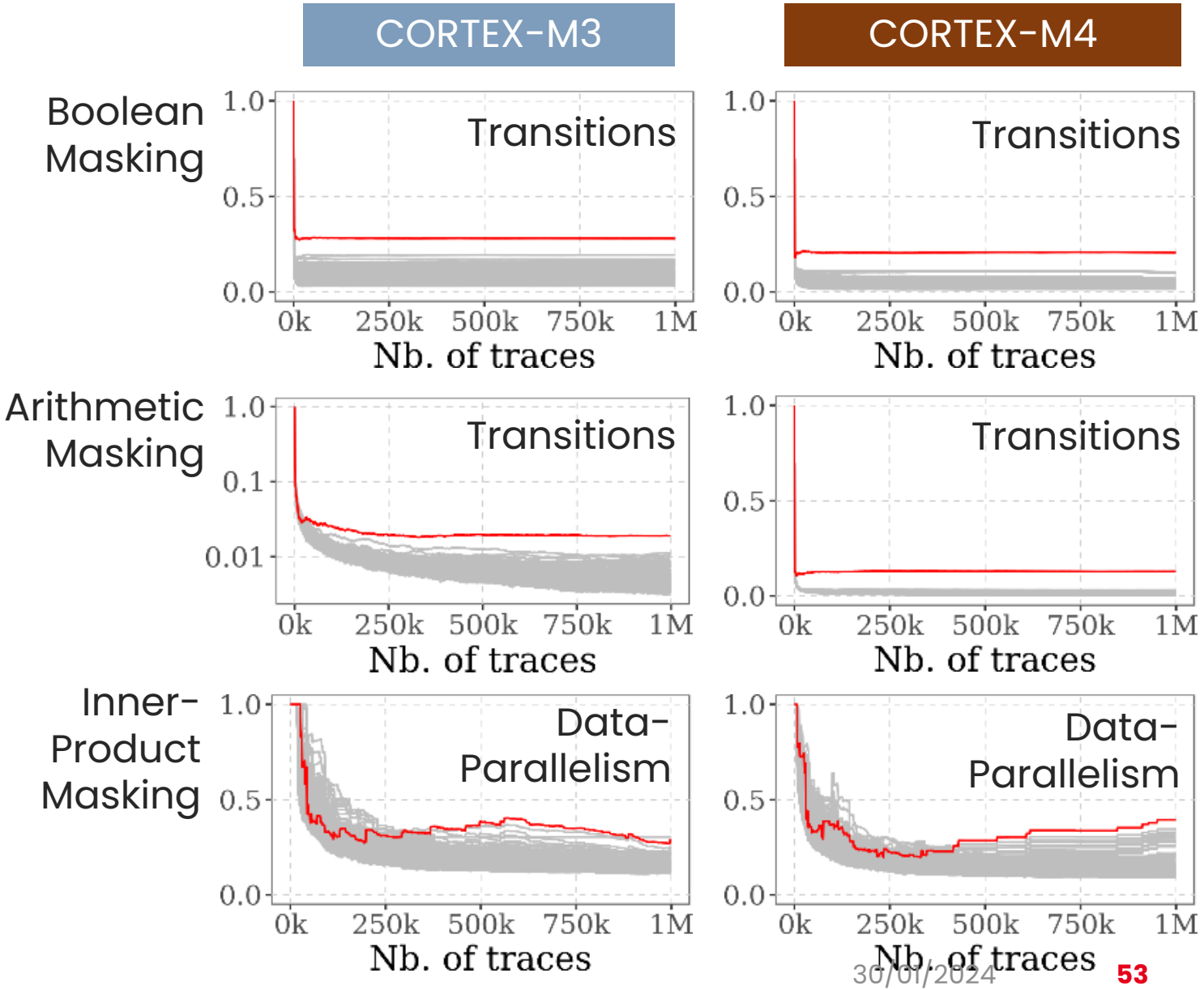
- Transition-based leakages
- Data parallelism

Method:

- Correlation-based analysis

Results:

- Recovered the sensitive information

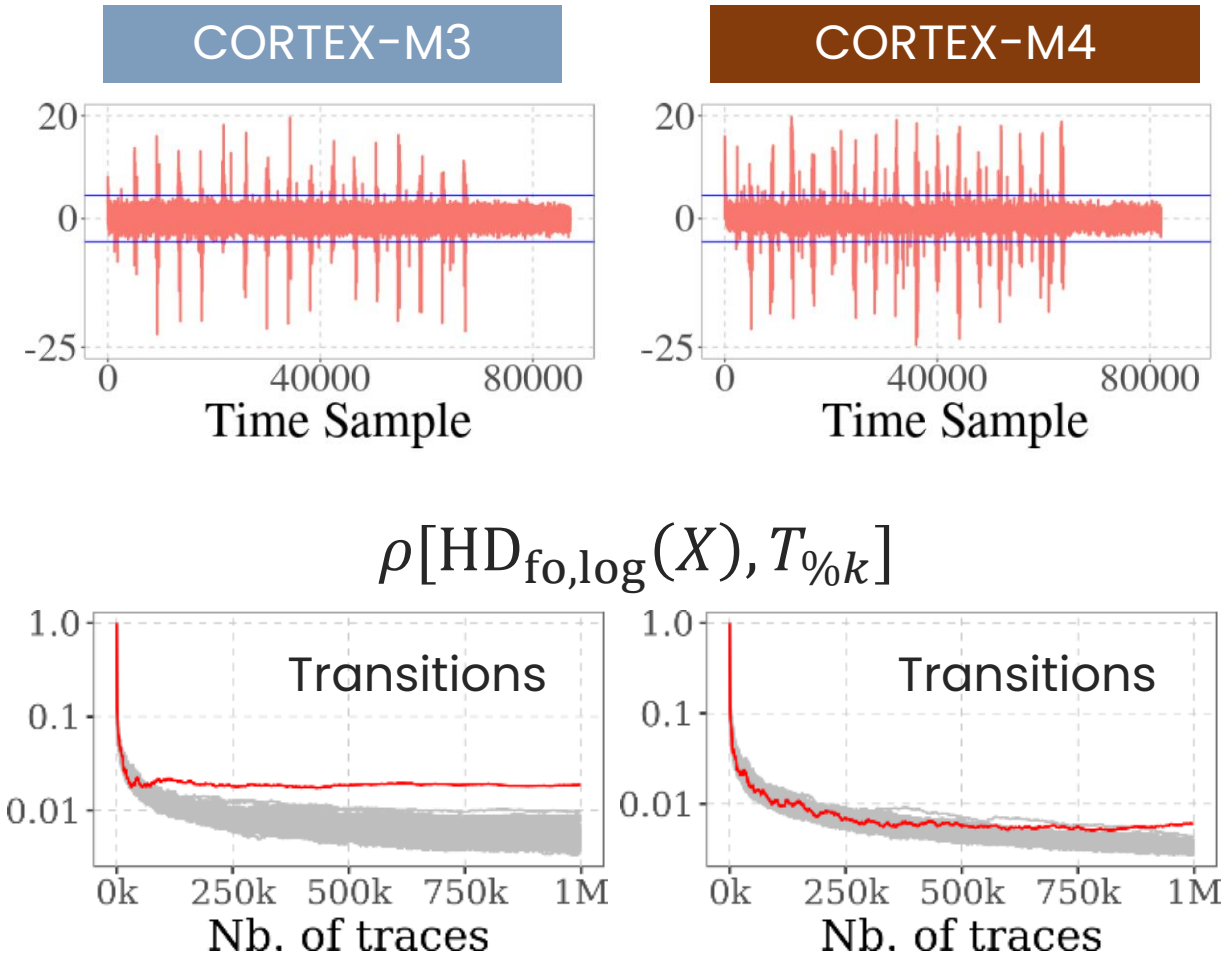


All that Glitters is not Gold, Pt.2

- **Methodology**: detect information leakage along execution of IPM AES-128 implementation
  - **Expected result**: no leakage
    - Implementation correct under ILA
    - IPM immune to transition-based leakage
  - **Actual result**: unexpected leakage
- **Root cause**: log/alog-based multiplication + transition-based leakage
- **Exploitable?**
  - Correlation-based analysis
  - **Result**: yes, it is exploitable ☺

$$\mathrm{HD}_{\mathrm{fo,log}}(X) = mean(\mathrm{HD}(\log_3(X_0), \log_3(X_1)))$$

Inner-Product Masking – AES-128



Outline of the Investigation

1. Observation data parallelism ☑
2. Exploitability of data parallelism ☑
3. Leakage Resilience of Fully Masked Implementation ☑

Summary and Conclusions

- **Contribution**: Investigating security impact of micro-architecture on masking schemes
  - **Goal**: explore alternative masking schemes to mitigate micro-architecture impact
  - **How**:
    1. Detect leakage effects on target platform
    2. Analyse exploitability of detected leakage effects
  - **Results**:
    - Efficient exploitation of data parallelism against analysed masking schemes
    - The multiplication algorithm degrades expected security guarantees of Inner-Product masking
- **Conclusions:**
  - Micro-architecture might induce new angle of attacks
  - Masked implementations as an *interconnected systems*
    - Security evaluation needs to consider both *subsystems* and *their interaction*

3. Conclusions and Perspectives

Epilogue of a Three-Year Long Journey

How to mitigate security degradation induced by the micro-architecture?

Two orthogonal approaches

Consider fine-grained micro-architectural details:

- Automate transition-based leakages mitigation

Do not consider micro-architectural details:

- Employ transition-based resilient masking schemes

CONCLUSION:

Micro-architecture hard to handle

Epilogue of a Three-Year Long Journey

How to mitigate security degradation induced by the micro-architecture?

Two orthogonal approaches

Consider fine-grained micro-architectural details:

- Automate transition-based leakages mitigation

Do not consider micro-architectural details:

- Employ transition-based resilient masking schemes

CONCLUSION:

Micro-architecture hard to handle

Unexpected transition-based leakages (i.e., memory-related leakage)

Unexpected exploitable effects (i.e., data parallelism)

Epilogue of a Three-Year Long Journey

How to mitigate security degradation induced by the micro-architecture?

Two orthogonal approaches

Consider fine-grained micro-architectural details:

- Automate transition-based leakages mitigation

Do not consider micro-architectural details:

- Employ transition-based resilient masking schemes

CONCLUSION:

Unexpected transition-based leakages (i.e., memory-related leakage)

Micro-architecture hard to handle

<u>But we can do it</u>

Relying on complete models of the micro-architecture

Unexpected exploitable effects (i.e., data parallelism)

Some Perspectives

- **Inner-Product Masking:**
  - Data parallelism vs optimal codes
  - Avoid data parallelism
- **Masking of order N:**
  - Avoid expensive solutions, e.g., masking of order $N \times 2$
  - Combine leakage effects, i.e., parallelism + transition-based leakage
- **Complex micro-architectures**:
  - More transition-based leakages
  - Increased data parallelism
- **Further micro-architectural effects:**
  - Glitch-based leakages
  - Coupling-based leakages

- **Pairing compiler-based approach:**
  - **Inner-product masking**:
    - Efficient implementation
    - Avoid data parallelism
  - **Hardware-based mitigations**, e.g., [Gao20]:
    - Potentially reduce performance impact
    - Potentially increase mitigation capabilities
  - ***Non-completeness***, e.g., [Gigerl21]
    - Efficiently deal with:
      - Transition-based leakages
      - Glitch-based leakages
      - Data parallelism exploitation

Thank You!

Bibliography

[Balasch14] Balash, J et al. « On the Cost of Lazy Engineering for Masked Software Implementations »

[Gao20] Gao, S., et al. « FENL: an ISE to mitigate analogue micro-architectural leakage. »

[Gigerl21] Gigerl, B., et al. « Secure and Efficient Software Masking on Superscalar Pipelined Processors. »

[Meyer20] Meyer, L.D, et al. . « On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software. »

[Moos17] Moos, T., Moradi, A. « On the Easiness of Turning Higher-Order Leakages into First-Order. »

[Seuschek17] Seuschek, H. « Side-channel leakage aware instruction scheduling. »

[Tsoupidi23] Tsoupidi, R.M, et al « Securing Optimized Code Against Power Side Channels. »

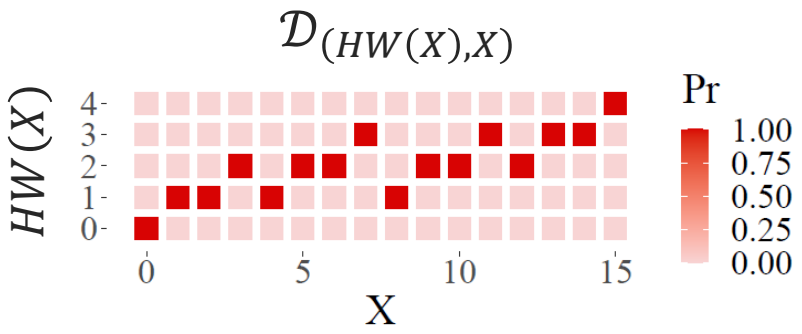
[Wang19] Wang, J., « Mitigating power side channels during compilation. »

Backup!

Analysis of the Leakage Model Distributions

Observation

- $\mathcal{D}_{(HW(X),X)} \neq \mathcal{D}_{(SHW(X_0,X_1),X)}$

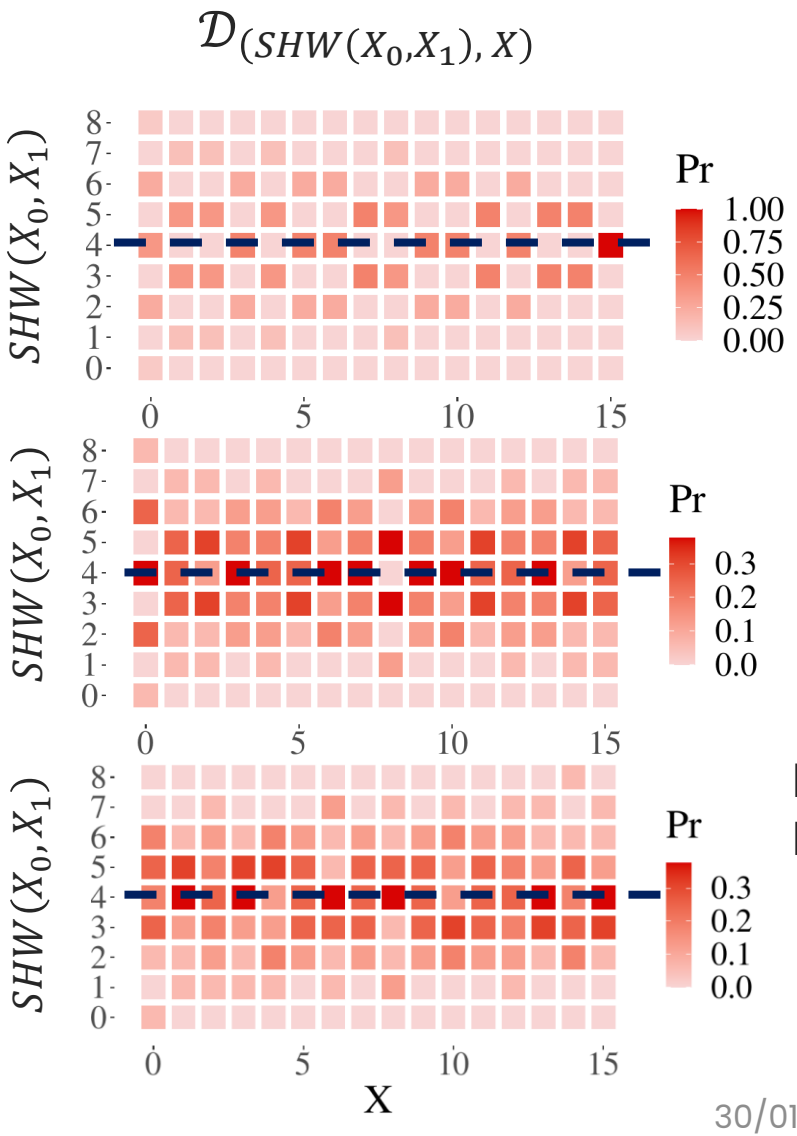


Consequence

- Sub-exploiting the available information

ALSO

$$mean\big(SHW(X_0,X_1)\big) = mean\big(HW(X_0) + HW(X_1)\big) = constant$$

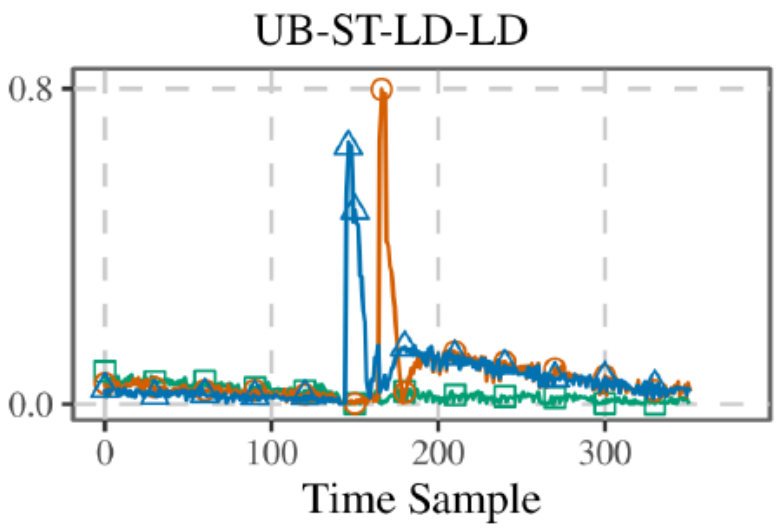


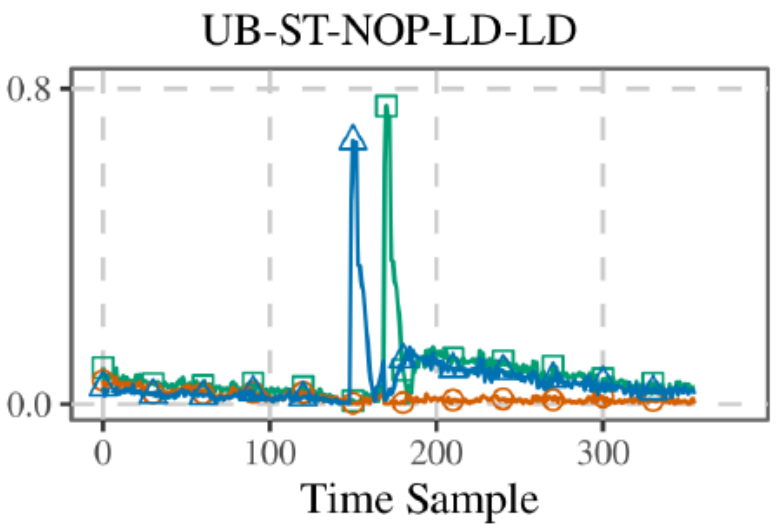
Boolean Masking

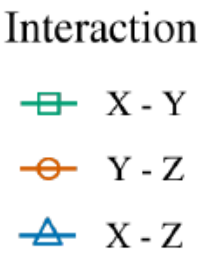
Arithmetic Masking

Inner-Product Masking

Security Evaluation—Root Cause Analysis







<UB-ST-LD-LD>:

```
write Z
R0 = read X
nop
nop
nop
nop
R1 = read Y
```

<UB-ST-NOP-LD-LD>:

```
write Z
nop
R0 = read X
nop
nop
nop
nop
R1 = read Y
```