Side-channel security of superscalar CPUs

Federico Zanca

Computer Science and Engineering
Politecnico di Milano

Table of Contents

Table of Contents

Recap: What are Side-Channel Attacks?

Definition

- Side-Channel Attacks (SCAs) exploit unintended physical emanations (e.g., power consumption, electromagnetic radiation, execution time) from a device
- These emanations are correlated with the secret data being processed, allowing an attacker to infer sensitive information (e.g., cryptographic keys)

The Fundamental Question

- How do we accurately *model* these physical emanations to find the leaked information?

Complex CPU Targets

SCA Challenges on Modern CPUs

- SCAs are increasingly targeting complex platforms:
  - SCA targets have evolved from simple micro-controllers and smart-cards to complex single-core and laptop-grade CPUs
  - Now, side-channel attacks are also relevant for superscalar CPUs
- Assessing vulnerability and validating countermeasures becomes significantly more difficult with increasing target complexity

Why Microarchitecture Matters for SCA

Main Idea

- To assess the side-channel vulnerability of software on a CPU, we **must** consider the CPU's microarchitectural features.
- Correct execution only requires the object code to match the CPU at ISA level.
- However, the extent of side-channel leakage also depends on the processor's microarchitecture.

Why Microarchitecture Matters for SCA

Dangers of Ignoring Microarchitecture

- Side-channel countermeasures, if designed without microarchitectural awareness, can be invalidated.
- **Example**: Accidental value combinations due to register reuse, despite careful assembly implementations.

Table of Contents

Uncovering Microarchitecture via CPI Analysis

Problem: Hidden Microarchitecture

- Internal details of CPU microarchitectures (e.g., pipelines, execution units, buffers) are often proprietary and not publicly documented in full detail.
- However, precise knowledge of these details is what allows mounting an effective side-channel attack

Uncovering Microarchitecture via CPI Analysis

Inferring from CPI

- A novel method is proposed: using information leaked by the **Clock cycles Per Instruction (CPI)** achieved on specific instruction sequences.
- **How it works**:
  - Compare CPI of instruction sequences with and without *Read-After-Write (RAW) hazards*.
  - Hazard-free sequences reveal best-case CPU capabilities (e.g., dual-issuing).
  - Hazard-affected sequences show how hazards prevent parallel execution.
- **Key Interpretations:**
  - A CPI of **0.5** indicates **full dual-issuing** (2 instructions per cycle).
  - A sustained CPI of **1** implies a component is **fully pipelined** (can start a new operation every cycle).

Example: ARM Cortex-A7 MPCore

CPU Overview

- A dual-core, in-order CPU with an 8-stage pipeline.
- Described as "partial dual-issue": Not all instruction pairs can be executed simultaneously.

Why Characterize This CPU?

- Public documentation (reference manuals, GCC backend descriptions) provides a logical view (see next slide).
- However, it lacks some microarchitectural details needed for side-channel analysis

ARM Cortex-A7 Pipeline Logical View

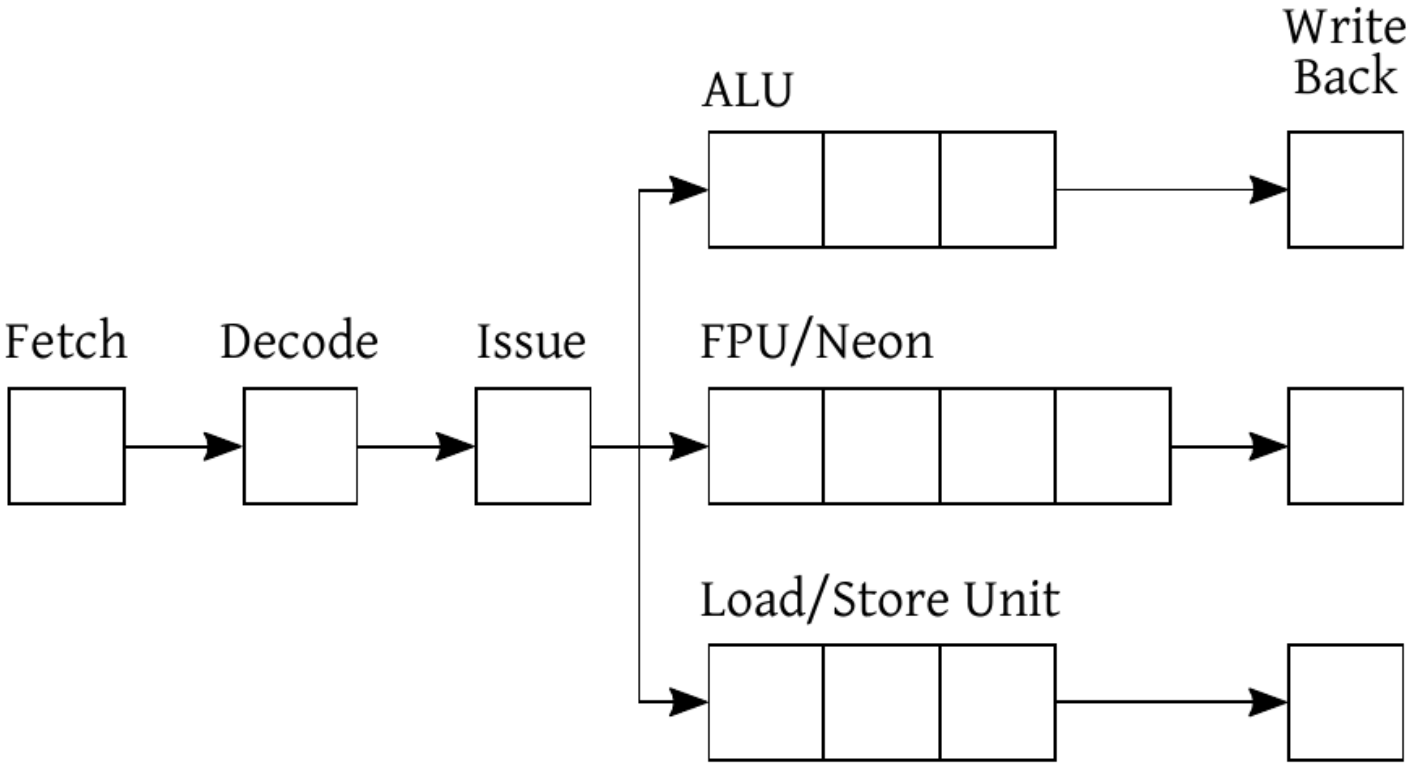

Figure: ARM Cortex-A7 MPcore pipeline logical view

Undercovering Microarchitectures (Cortex-A7 Example)

Inferences from CPI Analysis

- **ALUs:** Two ALUs are present, but they are not identical. Only one is equipped with a barrel shifter and multiplication unit.
- **Pipelined Units:**
  - The **Load Store Unit (LSU)** is **fully pipelined** (sustained CPI of 1 for load/store sequences).
  - The **multiplier** within the ALU is also **fully pipelined** (CPI of 1 for multiplication sequences).
- **Data Bus Structure:**
  - Three data buses connect the Register File (RF) to the Execution (EX) stage.
  - Two buses connect the EX stage back to the RF, implying the RF has two write-ports and three read-ports.
- **Unexpected NOP Behavior:** Counter-intuitively, `nop` instructions are **not dual-issued** by the Cortex-A7.

Deduction: Cortex-A7 Pipeline Structure

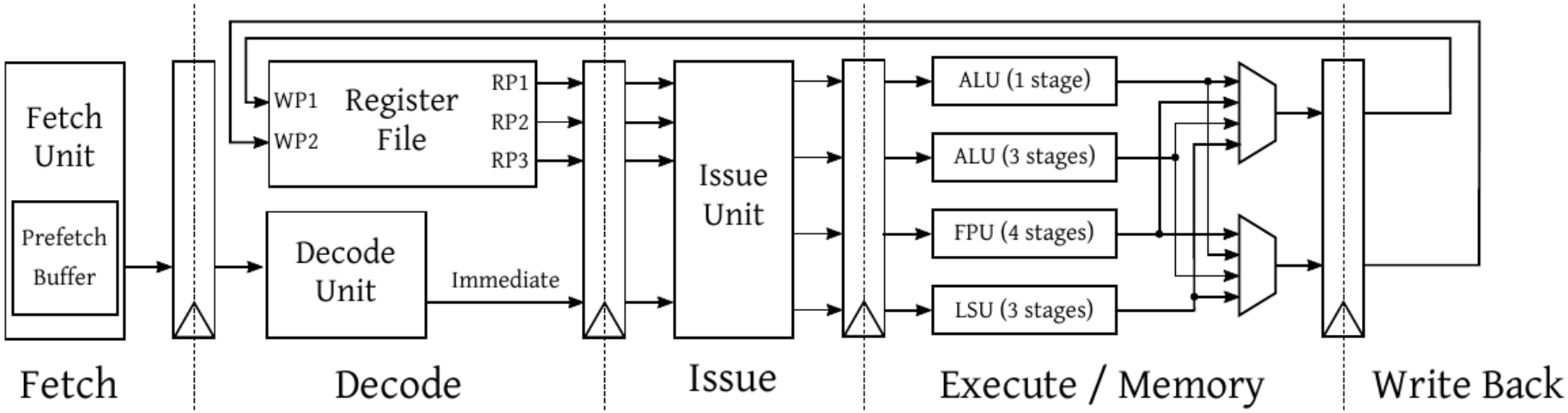

Figure: Alleged ARM Cortex-A7 pipeline structure according to CPI analysis deductions

Dual-Issue Capabilities: Instruction Pairs

Table: Instruction pairs executed in dual-issue by the Cortex-A7 MPCore CPU.

	mov	**ALU**	**ALU w/imm**	**mul**	**shifts**	**branch**	**ld/st**
mov	✓	✓	✓	✗	✓	✓	✗
ALU	✓	✗	✓	✗	✗	✓	✗
ALU w/ imm	✓	✓	✓	✗	✓	✓	✓
branch	✓	✓	✓	✓	✓	✗	✓
ld/st	✓	✗	✓	✗	✗	✓	✗
mul	✗	✗	✗	✗	✗	✓	✗
shifts	✗	✗	✓	✗	✗	✓	✗

Table of Contents

Characterizing Microarchitecture Leakage

The Foundation of Leakage

- Gates driving large capacitive loads are primary sources of side-channel leakage.
- Power consumption in such scenarios is well-modeled by the **Hamming distance** of two values asserted on their outputs in subsequent clock cycles

Detecting Leakage

- **Pearson's correlation coefficient** is used to statistically compare measured power consumption with the predicted leakage model
- A correlation statistically distinguishable from zero (with ¿99.5% confidence) in the correct clock cycle indicates a leakage

Leakage Measurement - Cortex-A7

Data Collection

- Seven micro-benchmarks: small (2-4) instruction sequences designed to trigger specific component activities
- Ran with randomly generated values at each execution, triggered by GPIO
- Cache effects eliminated by measuring executions after the first one and inserting 100 `nops`
- Data acquired: **100,000 power traces** per benchmark
- Each trace was an average of 16 individual executions with the same input, reducing noise
- Register File (RF) leakage was evaluated separately by pre-charging destination registers

Observed Leakage: Registers

Register File (RF)

- No statistically significant leakage observed from RF read-ports.
- Attributed to a short capacitive load, as Issue Stage (IS) buffers drive execution units

Observed Leakage: IS/EX

Issue/Execution (IS/EX) Buffers

- Outputs show significant leakage
- **Modeled by Hamming Distance:** between values of a source operand of an older and a younger single-issued instruction
  - Leakage prominent when operands share the same bus (same source operand position)
- **Hamming Weight Leakage:** Also observed when `movs` are interleaved with `nops` (due to `nop` implementation with zero-valued operands)
- Dual-issued arithmetic instructions show no measurable leakage among source operands (no shared resources before computation)

Observed Leakage: ALUs

ALU and Shift Buffer

- **ALU:** Leakage dependent on the **Hamming weight** of the instruction result
  - Inferred due to ALUs asserting results on previously zero-precharged signals
- **Shift Buffer:** A small leakage proportional to the **Hamming weight** of the shifted value is present

Observed Leakage: EX/WB Buffers

Execution/Write-Back (EX/WB) Buffers

- Leakage mirrors IS/EX buffers.
- **Modeled by Hamming Distance:** between the *results* of subsequent single-issued instructions.
  - Occurs regardless of destination register sharing or data-flow relationship
- **Hamming Weight Leakage:** Also present for instruction results.
  - Attributed to `nop` instructions resetting the WB bus to zero

Observed Leakage: MDR

Memory Data Register (MDR) and Align Buffer

- Potential leakage source during load/store instruction sequences
- **Modeled by Hamming Distance:** between two subsequently loaded or stored values
- Suggests a separate buffer in the Load Store Unit (LSU) for sub-word realignment
- Presence of this **align buffer** and its leakage was confirmed experimentally

Superscalar Leakage Modeling

Leakage from Algorithmically Independent Instructions

- Observed information leakage that combines values from potentially independent instructions, driven by four causes:
  1. Instruction scheduling order
  2. Position of source operands
  3. Single or dual-issuing of instructions
  4. Potential data remanence in LSU buffers

Superscalar Leakage Modeling

Subtle Code Changes can be Harmful

- Even apparently harmless changes to assembly code (e.g., swapping source operands of a commutative operation like XOR) can lead to side-channel leakage
- This is due to altered pipeline resource sharing
- Such changes are difficult to detect by tools focusing only on instruction semantics or manual audits

Superscalar Leakage: Vulnerabilities

Impact of Dual-Issuing

- Worsens effects of instruction scheduling and operand position.
- Leakage can stem from combinations of source operands of *non-consecutive* instructions if an intermediate instruction is dual-issued
- Highlights the crucial, often-neglected importance of instruction scheduling for preventing side-channel leakage
- Dual-issuing could also potentially **enhance security** by enabling parallel computation of two shares in masking schemes

Superscalar Leakage: Vulnerabilities

Data Remanence and NOPs

- **Data Remanence:** In MDR and LSU buffers, old data can accidentally combine with current computation results, creating harmful leakage.
- **NOP Operations:** While semantically neutral, `nop` instructions can introduce new leakage modes (e.g., by resetting buses to zero). They are not *security neutral*