Side-channel disassembly on a system-on-chip: A practical feasibility study

Julien Maillard, Thomas Hiscock, Maxime Lecomte, Christophe Clavier

Side-Channel Disassembly on a System-on-Chip: A Practical Feasibility Study

Julien Maillard[a,b], Thomas Hiscock[a], Maxime Lecomte[a], Christophe Clavier[b]

[a] *Univ. Grenoble Alpes, CEA, LETI, MINATEC Campus, F-38054, Grenoble, France*
[b] *Univ. Limoges, XLIM-MATHIS, Limoges, France*

Abstract

Side-channel based instruction disassembly (SCBD) is a family of side-channel attacks that aims at recovering the code executed by a device from physical measurements. Over past decades researches proved that instruction-level disassembly is feasible on simple controllers. Simultaneously, the computing power and architectural complexity of processors are increasing, even in constrained devices. Performing side-channel attacks on mid or high-end devices is inherently harder because of complex concurrent activities and an important amount of noise. While broad pattern identification, such as cryptographic primitives, has been proved possible, the feasibility of precise SCBD remains an open question on a complex System-on-Chip (SoC).

In this work, we address some of the technical challenges involved in performing SCBD on SoCs. We propose an experimental setup and measurement methodology that enables reliable characterization of instruction-level electromagnetic (EM) leakages. After investigating broad-functional unit activity leakages, we study the feasibility of three instruction-level code reconstruction granularities: functional unit recognition, opcode recognition and bit-level recovery. Under a controlled experimental environment, our results show that broad functional unit activity recognition is achievable as well as opcode-level SCBD. Finally, we show promising results regarding bit-level SCBD practical feasibility by exploiting the prefetching semantics of the CPU.

Keywords: Side-channel, Disassembly, Reverse-engineering, System-on-Chip, ARM

Email addresses: julien.maillard@cea.fr (Julien Maillard), thomas.hiscock@cea.fr (Thomas Hiscock), maxime.lecomte@cea.fr (Maxime Lecomte), christophe.clavier@unilim.fr (Christophe Clavier)

1. Introduction

Side-Channel Attacks (SCA) exploit physical leakages (e.g., power consumption, electromagnetic field, execution time) produced by a device while computing to disclose secret information. *Side-channel Based Disassembly* (SCBD) is a sub-category of side-channel attacks that aims at reconstructing an approximate representation of the assembly code executed by a processor. This reconstruction can either be *fine-grained*, if it recovers instructions or bit encodings, or *coarse-grained* if it recovers higher-level program structures, such as basic blocks or broad functionalities.

SCBD is a serious thread to *Intellectual Property* (IP) of software, especially bootloaders, whose code is usually read protected. However, it can also be used for identifying regions of interest in large side-channel traces, or for synchronizing more specific attacks such as fault injections. Thus, having a better understanding of these attacks is valuable for deploying appropriate countermeasures. Aside from the attacks, SCBD can also be used to implement a non-intrusive control flow monitoring scheme [10] through *Electromagnetic* (EM) or power measurements. This allows detecting abnormal behaviors of a system, such as the execution of malicious codes.

SCBD really shines for the reverse-engineering of bootloaders as it is a non-invasive way to perform black-box dynamic analysis. These programs are stored in an on-chip ROM which is rarely accessible, even through debug ports. A SCBD could also be used as a synchronization tool. Indeed, it would help the adversary to identify *Points of Interest* (PoIs) within the assembly code, such as conditional branch instructions. For instance, targeting memory loads during an AES encryption can help to trigger the *subBytes* routine. Such application can help to provide meaningful side-channel traces realignement (i.e., counter jitter effects), which is necessary for univariate analysis (i.e., per sample). Moreover, in a real-case attack scenario, a SCBD can provide acquisition triggers (e.g., detect the beginning and ending of an AES encryption) for further analysis, such as *fault injection* (FI) attacks or key recovery attacks. Recent works investigated the possibility to perform *Control Flow Integrity* (CFI) checking by modeling side-channel leakage of single instructions so that the device internal state can be verified [10]. Finally, as a leakage modeling tool, a SCBD could help programmers to understand the sources of side-channel leakages, either at a coarse-grained or fine-grained scale. This can facilitate the development of hardware and software based coutermeasures to reduce the information that could be exploited by an attacker. In other terms, SCBD techniques can be used in order to enhance side-channel aware programming.

Recent breakthroughs have pushed the limits of coarse-grained SCBD, enabling detection of cryptographic primitives on low-end *System-on-Chip* (SoC) (ESP8266EX with 80MHz Tensilica CPU) [17], but also OS and malware detection on laptop grade processors such as a x86 Intel NUC board [21]. Fine-grained SCBD, on the contrary, is steered towards code reconstruction for black box reverse engineering or precise monitoring for event identification or malware detection.

Different fine-grained SCBD techniques have been studied in the literature: opcode classification [6, 13, 18], hierarchical classification (i.e., with recognition of opcodes and operands) [15] and bit-level reconstruction [5]. While the recognition of substantial patterns have been proved feasible on complex architectures [17, 21], fine-grained SCBD methods struggle to scale up with CPU complexity. This brings us to the following questions: *is SCBD achievable on a high-end SoC's CPU and what reconstruction accuracies can be achieved?*

1.1. Contributions

In this paper we propose a practical feasibility study of different SCBD paradigms on a Zybo-z7 board. Namely, we investigate coarse-grained, opcode-level and bit-level recovery. After describing related work in section 2 and precising the SCBD attacker model considered in this paper in section 3, we identify the main architectural differences between microcontrollers and modern SoCs. We highlight the impact of these differences on side-channel measurements (see section 4). Then, in section 5, we propose an acquisition methodology for near-field electromagnetic side-channel so as to take the architectural complexity into account: we craft an experimental protocol that mitigates several noise and jitter sources inherent to SoCs. In section 6, we study the feasibility of broad and instruction-level functional unit activity recovery. Afterwards we investigate the leakages of single instructions in section 7 with a hierarchical approach: we show that opcode-level SCBD is feasible. Leakage assessment and classification of single instruction bits is discussed in section 8. Namely, we start by identifying prefetching semantics leakages that are further exploited for a classification task. We show that bit-level reconstruction is much harder than opcode level, but promising results support the bit-level SCBD feasibility. Finally, we conclude in section 9

2. Related Work

2.1. Leakage Analysis

Fine-grained leakage assessment was studied by Gao and Oswald on a Cortex-M3 [8] and by Barenghi and Pelosi on a Cortex-A7 [2]. They performed micro-architectural reverse engineering and developed a leakage assessment methodology in order to quantize the leakage of data manipulated by instruction sequences. Noticeably, they expose pipeline inner semantics (e.g., inter-stage buffers, buses) leakages, possibly with different leakage models, and methodological tools that we will use in this work. Marshall *et al.* designed a series of micro-benchmarks for assessing pipeline leakages of Cortex-M series SoCs [12]. Beyond methodological aspects, they provide insightful findings on data remanence leakage within load / store units.

2.2. SCARE

Side-Channel Analysis for Reverse Engineering (SCARE) is a family of side-channel attacks that targets hidden constants rather than secret data directly.

In that sense, Novak designed an attack that aimed at recovering the content of a lookup table used in A3/A8 algorithm, a GSM authentication algorithm that was implemented in SIM cards [14]. Later, Clavier improved the attack by retrieving the two A3/A8 substitution tables and the secret key [4].

2.3. Bytecode Recognition

Vermoen *et al.* performed the side-channel reconstruction of Java Card bytecode sequences by exploiting a smart card's power consumption [20]. They based their analysis on building templates for several instructions and instruction sequences. Some bytecode mispredictions are commented: these confusions interestingly concerned semantically similar bytecode instructions.

2.4. Opcode Classification

Opcode level recovery was introduced by Goldack *et al.* [9] which performed a leakage analysis for SCBD on PIC16 devices. In a follow-up work, Eisenbarth *et al.* [6] applied the template approach of Chari et. al [3] for recovering opcodes. In addition, they proposed a new approach to correct errors in opcode sequences using a hidden Markov model built with prior knowledge on opcode distributions. They achieved 70.1% and 50.8% recognition rates on test code and real code respectively, by measuring power on a PIC16 microcontroller running at 1 MHz. Then, Msgna *et al.* built an opcode classifier on power measurements of an ATMEGA163 8-bit microcontroller running at 4 MHz [13]. By combining *Principal Component Analysis* (PCA) for dimensionality reduction and *K-Nearest Neighbors* (kNN) for the classification, they claim a 100% recognition rate on test code for the 39 selected opcodes. On a decapped PIC16F, Strobel *et al.* used *Linear Discriminant Analysis* (LDA) for dimensionality reduction and kNN for opcode classification on EM measurements [18]. By combining side-channel measurements at multiple probe positions, they obtained up to 96.4% success rate for test programs and 87.4% for an AES implementation.

2.5. Hierarchical Reconstruction

Park *et al.* [15] introduced a new approach for SCBD known as "hierarchical classification". The idea is to recover information incrementally, starting from broad instruction categories (load/store, ALU, shifts) and then using more specific classifiers to recover exact opcodes and operands. They pre-process side-channel traces (power measurements) with *Continuous Wavelet Transforms* (CWT) and Kullback-Liebler divergence to select a relevant subset of samples. They feed the resulting data in a PCA for further reducing the dimensions of data. On an ATmega328p AVR microcontroller, they achieved a 99.03% recognition rate on test code (112 different instructions). Also, they adress the covariate shift caused by template portability on other devices.

Similarly, Vafa *et al.* used Kullback-Liebler divergence along with CWT for feature extraction and PCA for dimensionality reduction on a PIC16F microcontroller and ARM Cortex M3 32-bit CPU [19]. Concerning the hierarchical classifier, they observed the best results with Random Forest based Ensemble

Classification. Finally, they reported 99.5% and 93.3% of success rate for test code and real code respectively on the PIC16F, and 98% and 80.2% for the Cortex-M3 core.

Fendri *et al.* used a multi-layer perceptron neural network to perform instruction-level recovery on a 32-bit RISC-V platform running at 100 Mhz [7]. By performing data augmentation and preprocessing the traces with sparse dictionary learning for feature selection and dimensionality reduction, they attest 93.16% instruction recognition accuracy considering EM leakages.

2.6. Bit-Level Reconstruction

Cristiani *et al.* introduced the bit-level SCBD paradigm [5]. They targeted the instruction prefetch buffer, which is shown to leak the *Signed Hamming Distance* (SHD) of the instructions going through it. The bit-level approach has the advantage to be scalable to large instruction sets, and is easier to train than other methods. Using simple LDA-based classifiers, the authors attested a 99.6% recognition rate for single bits, and 95% on the full 14 bits of the target instruction by combining multiple EM probe positions upon a PIC16F microcontroller.

3. Attacker Model

For the rest of this work, we make several assumptions on the attacker model. We consider only supervised SCBD techniques, meaning that an attacker can train a model under a controlled environment before performing the actual attack. This implies that:

- the adversary has a full control over a clone of the target device or the target device itself. Ideally, they should have the ability to execute arbitrary code on the clone device, potentially with root or even kernel-level privileges.
- A complete attack scenario is divided in two steps: a profiling phase (performed on a clone of the device) and an attack phase (on the real target).

While template portability has been proved possible in a SCBD context on small microcontrollers [15], this is left out of the scope of this paper. Namely, we rather perform profiling and attack phases on the same target.

To facilitate our analysis, we assume that the same code can be easily re-executed on the target. This allows collecting several traces of the same programs to reduce the noise in measurements. This assumption is not unrealistic in practice. Typically, bootloaders are executed every-time the device is powered-on: multiple resets can produce measurements of the same program.

4. Challenges of SCBD on Modern SoCs

So far, most fine-grained side-channel disassemblers [5, 6, 13, 18] have been targeting simple and low performance microcontrollers (e.g., Atmel or PIC devices). Side-channel analysis on these devices is eased by several factors: they run at low clock frequencies, their CPU have a simple architecture with few logic gates and large CMOS technology (e.g., > 100 nm). Then, it is important to pinpoint the major differences between microcontrollers and SoCs. Afterwards, we present the main noise sources of a SoC (in terms of side-channel measurement), and some mitigation techniques we identified.

4.1. Simple controller vs. Complex SoC

Unlike microcontrollers, SoCs have a highly sophisticated architecture. We provide in Table 1 a side-by-side comparison of the most significant differences between these systems. From a side-channel analysis perspective, the challenges that appear on SoCs are:

- Contrary to microcontrollers that mostly execute code "close to the metal" (i.e., with small RTOS or no OS at all), modern SoCs are able to run a full fledged operating system such as Linux. From a SCA point of view, this results in eventual **context switches and parallel activity** (even more in multi-core systems) that add noise to the measurements.
- Complex systems run at **higher clock frequency** (close to GHz). As a consequence, high-end oscilloscopes with large bandwidth and sample rate are required in order to perform measurements more precise than a CPU clock cycle.
- Modern SoCs embark many hardware **IPs** (peripherals, hardware accelerators) within a space-optimized package. The measurement chain must be precise enough to observe the leakages. Particularly, it seems more astute to target local EM signal over the chip rather than to target the whole device's power consumption.
- Complex CPUs are designed to optimize the latency of the executed code, they make heavy use of **prediction and parallelisation**. As a consequence, their side-channel activity is often cluttered with a lot of noise (see subsection 4.2).

4.2. Sources of noise in complex SoCs

From a side-channel perspective, observing the side-channel activity of a SoC is inherently noisy because of the high number of in-flight instructions concurrently executed. Then, we shall describe the main elements that trouble side-channel acquisitions and how to mitigate their effect.

Properties	Simple controller	System-on-Chip
Pipeline	Few stages	$>$ 5 stages
Clock frequency	$<$ 300 MHz	$>$ 300 MHz, DVFS
Issue capacity	1	2 or more (superscalar)
Instruction flow	In-order	In-order or Out-of-Order
Branch prediction	Static	Dynamic
Programs	Bare Metal	Operating system
Memory	Flash, ROM	DRAM, caches, MMU, TLB
Instruction set	Restricted	Mult. latencies and encodings

Table 1: Comparison between microcontrollers and SoCs.

4.2.1. DVFS

Frequency scaling is a mechanism that adapts the CPU's frequency proportionally to the current workload. It is often connected with voltage scaling: the conjunction of these two techniques is called *Dynamic Voltage and Frequency Scaling* (DVFS). DVFS introduces variations in the execution time of a program, also called jitter, that desynchronizes side-channel acquisitions.

4.2.2. Multi-Core

Modern SoCs often include more than one processor (2-16) which operate concurrently. From a SCA point of view, observations of a program running on a single core are disrupted with noise due to other cores' activity.

4.2.3. Pipelining and multi-issuing

The key idea of pipelining is to decompose the execution of instructions into several small steps. Moreover, multi-issuing is a mechanism that allows to issue several instructions simultaneously at each clock cycle. Regarding side-channel activity, both pipelining and multi-issuing have the effect of overlapping the leakages of nearby instructions.

4.2.4. Out-of-Order

Out-of-order processors are an elegant solution to the superscalar processor's limits. The key idea is to allow instructions to be dynamically dispatched to available execution units. To be even more efficient, Out-of-Order processors also perform register renaming to break Write-after-Write and Write-after-Read dependencies. This mechanism complicates the targeting of a particular physical register. Moreover, the complexity of modern architectures with all their undocumented details makes the execution order hard to predict in practice. This leads to alignment issues when performing several side-channel measurements.

4.2.5. Branch prediction

Branch prediction is a key part of modern processor architectures, and it needs to be as fast and as accurate as possible as any misprediction, or branch-miss, will pause the CPU and have an important performance impact. When

branch mispredictions occur, the speculatively executed instructions are discarded. Thus, the pipeline needs to restart with the correct branch outcome: this phenomenon introduces a jitter in a SCA context.

4.2.6. Cache hierarchy

When a CPU needs to access content at a given address, it first searches through the cache hierarchy, and fetches the main memory only as a last resort. Depending on whether the data is currently in the cache or not, we can observe a *cache hit* or a *cache miss*. Then, from a SCA point of view, data evictions caused by a SoC's inherent concurrency provoke inconsistencies and thus introduce a jitter.

4.3. Noise and Jitter Mitigations

In order to accurately analyze side-channel leakages, the environment between two distinct acquisitions should be as similar as possible. Instead of trying to disable some of the SoC features responsible for noise and jitter, we propose mitigations that limit their impact. Indeed, these mitigations are consistent with the SCBD's profiling phase, where the target device (or a clone) is supposed to be fully controlled by the attacker. In Table 2, we summarize the noise mitigation strategies we used to reduce the noise sources presented in subsection 4.2.

Table 2: Noise sources on SoC and mitigations.

Noise source	**Solution**
DVFS	Run a CPU-stressing workload to reach the maximum frequency.
OS Scheduling	Schedule the task on a specific core.
Trigger Jitter	Use raw memory access to toggle GPIOs.
Caches and branch prediction	Use branch-free code warmup.
Address inconsistency	Use a fixed buffer containing the target code.
Pipeline state	Surround target code with *NOP* instructions.
Environment variations (temperature, humidity)	Execute target code snippets in a random order.
Superscalar and Out-of-Order properties	Introduce carefully crafted data dependency between instructions.

5. Measurement Methodology

We now describe the measurement framework that we use for all experiments described in the rest of this work. Regarding the attacker model (section 3), these experiments take place at the profiling stage of the attack, where the adversary has full control over a clone of the target.

5.1. Device Under Test

We selected the Zybo XC7Z020-1CLG400C board as our *device under test* (DUT). It is a ready-to-use SoC built around the Xilinx Zynq™-7000 architecture and incorporates a dual-core Cortex-A9 CPU running up to 667 MHz. The Cortex-A9 CPU is a high-performance and low-power processor that belongs to the ARMv7-A family (32-bit) [11]. We chose this DUT because (i) the Zybo-z7 is an all-in-one board, with various components such as a FPGA: this type of target is similar to real-world devices; and (ii) the Cortex-A9 CPU contains several optimizations such as out-of-order execution, dynamic branch prediction, dual-issuing of instructions and a deep pipeline. We stress that the FPGA is not used in our experiments.

5.2. Side-channel Acquisition Setup

We acquire the near field EM emanations of the undecapped DUT through a 100 μm EM Langer H-field ICR HH 100-27 probe connected to a Tektronix 6 series oscilloscope with a 2.5 GHz bandwidth (to avoid being disrupted by high frequency RF signal) through a +45/50 dB low noise amplifier. The probe is attached to a 3-axis motorized bench. We use a sampling rate of 12.5 GS/s, and an *Analogic to Digital Converter* (ADC) precision of 12 bits. The acquisition setup is represented on Figure 1.

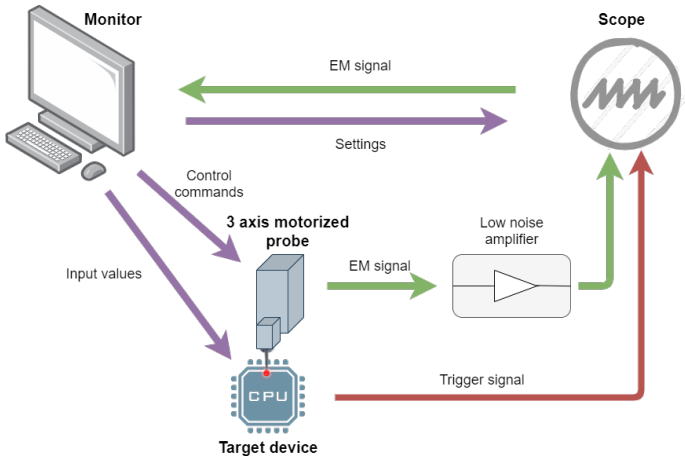


Figure 1: Measurement setup.

5.3. Software Architecture

The DUT runs a standard ARM Hard-Float Debian 10 distribution. The software architecture used to perform acquisitions is designed under a client / server model that is depicted in Figure 2. The DUT runs a TCP server in userspace, which accepts connections from a control computer. In order to trigger acquisitions, we use a GPIO of the board raised up and down. Using GPIO drivers

provided by the Linux kernel (i.e., `/dev/gpio*`) would introduce a high amount of jitter within measurements, because of context switches. To mitigate this issue, we toggle GPIOs with raw physical memory accesses to the GPIO peripheral address[1].

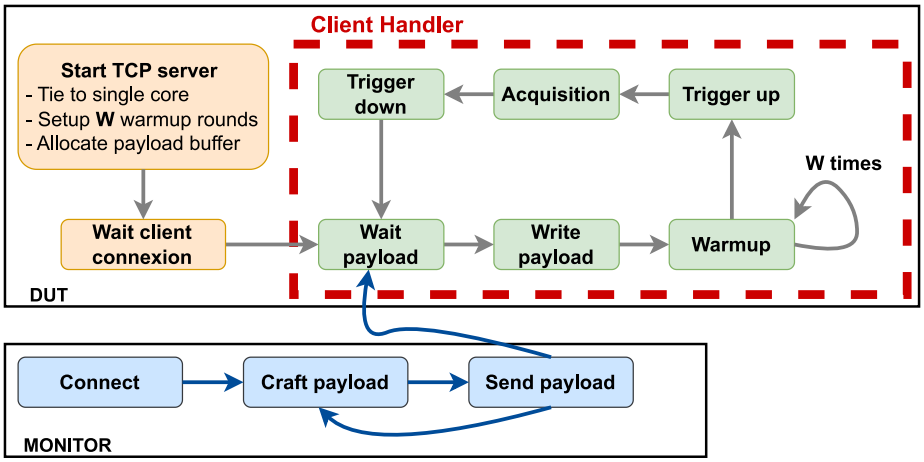


Figure 2: Overview of the software running on the DUT.

The TCP server initializes the trigger context and waits for a client to connect before launching the *daemon* that executes assembly code payloads sent by them. The payloads are written inside a page-aligned buffer marked as executable. We rely on the *mmap* function for this step. This allows to have an executable region in memory that is writeable, executable and at a fixed address. Then a function pointer is used to execute the buffer's content. To ensure deterministic execution and measurements, we tie the TCP server on a single core of the DUT.

Before acquiring traces for a payload, a "warmup phase" is mandatory to reduce the impact of several noise sources (see Table 2). This can be done by executing the payload several times within a branch-free loop (see algorithm 1)[2]. We stress that the code must be as similar as possible between the warmup rounds and the acquisition round. During our experiments, 100 warmup rounds proved to be sufficient to obtain stable execution times.

5.4. Characterization programs

In our model, the client (i.e., the laboratory computer) is responsible for crafting characterization programs (payloads) and sending them to the DUT. Characterization programs take the form of assembly functions, which contain

[1] On Linux, it can be achieved by obtaining a virtual memory address (using `mmap`) on a fragment of the `/dev/mem` file, which gives raw physical memory access.

[2] Avoiding conditional branches allows to reduce the jitter due to branch prediction mechanisms.

Algorithm 1: Generic warmup function.

Input: x, r
$cond \leftarrow false$;
for i *from* r *down to* 1 **do**
    `// cond must be a branchless expression`
    `// to avoid branch misses.`
    $cond \leftarrow i == 1$;
    trigger_set(cond);
    process(x);
    trigger_unset();
end

(i) a function prologue, (ii) a sequence of `NOP` instructions, (iii) the target code, (iv) a sequence of `NOP` instructions and (v) a function epilogue.

The function prologue is responsible for setting up the stack for the function and saving registers' values if needed. The function epilogue does the opposite operations: it restores registers and the stack. Prologue and epilogue may also used to switch back and forth Thumb execution mode and set register values to random to reduce unexpected biases. The `NOP` sequences surrounding the target code allow to entirely flush the pipeline state and provide a recognizable visual pattern within the traces.

We show on Figure 3 a sample EM side-channel trace obtained for a characterisation program. We clearly distinguish the different steps mentioned before (prologue, `NOP`s, target instructions, epilogue). The center area ("Target Instruction Sequence") is where we perform the leakage analyses.

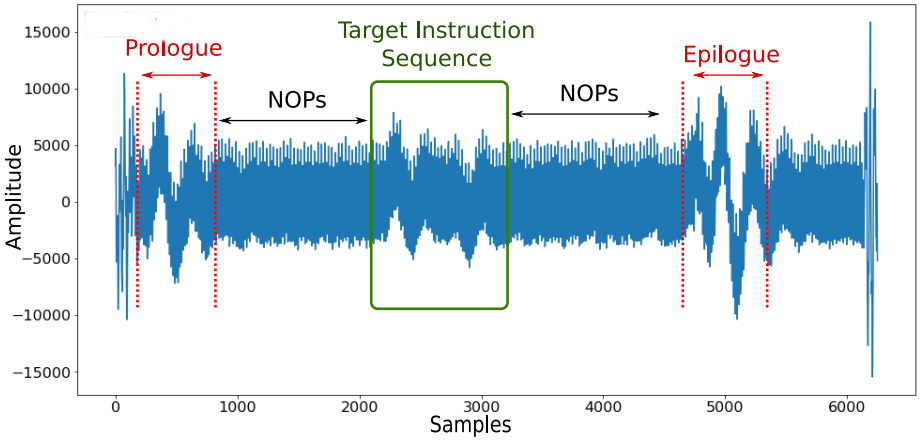


Figure 3: example mean trace of a payload execution: here the target code is a sequence of `EOR` instructions linked by *read-after-write* dependencies.

6. Coarse-grained analysis

This section is dedicated to the analysis of the leakages produced by the activity of functionnal units (e.g., *Arithmetical and Logical Unit* (ALU), *Load/Store Unit* or *Barrel Shifters*). After exposing the crafting of our target code in subsection 6.1, subsection 6.2, and subsection 6.3, we turn to broad functional unit activity classification (subsection 6.4).

6.1. Structure of Instruction Sequences

To perform classification of functional units activity, we craft several instruction sequences $\mathcal{D} = (D_i)_{1 \leq i \leq n}$ such that (i) every instruction in a particular $D \in \mathcal{D}$ goes through the same functional unit (e.g., ALU, MUL, Load/Store unit), (ii) Read-after-Write dependencies force in-order and single-issue execution, (iii) execution times of all $D \in \mathcal{D}$ must be equal, so that the classification step would not exploit leakage resulting from statistical differences between target instructions and surrounding `NOP`s and (iv) all D patterns should be chainable without loosing the RAW dependencies properties (see Figure 4).

```
EOR r0, r5          LDR r0, [r0, r1]
EOR r1, r0          LDR r0, [r0, r1]
EOR r5, r1          LDR r0, [r0, r1]
```

Figure 4: Chainable dependency patterns D examples.

6.2. Handling Memory Loads

Crafting sequences with memory loads needs a special attention to guarantee in-order and constant time execution. The TCP server running on the platform allocates a memory buffer of m addresses that we call the *sled buffer*. Let $\vec{sb} = (sb[1], ..., sb[m])$ be such a buffer (as a sequence of `void*`, equivalently 32-bit words) and let $\&sb[i]$ denote the address of $sb[i]$. We define $\vec{sb}$ as follows:

$$\begin{cases} sb[i] \leftarrow \&sb[i+1] & \text{if } 1 \leq i \leq m-1 \\ sb[m] \leftarrow \&sb[1] & \text{otherwise} \end{cases}$$

Then, the address of the sled buffer (equivalently $\&sb[1]$) is given as parameter of the characterization program, such as the register `r0` contains this value at the beginning of the execution of the payload. Suppose `r1` contains the value 0, then we can introduce RAW dependency with the code snippet illustrated in Figure 4.

6.3. Spatial Positioning

As we target local EM leakages in our experiments, the probe position on top of the CPU along the x, y and z axis has an important influence. Hence, the best positions have to be found by measuring a leakage metric of a target variable over a $n \times m$ grid set of different positioning. We consider that the closer to the CPU the probe is on the z axis, the better the results will be. As the signal amplitude can vary drastically from one position to another, an automatic calibration of the oscilloscope's voltage range focus is performed after moving to a new position.

6.4. Classification of Broad Functional Unit Activity

In this subsection we aim at distinguishing broad functional unit activity. Namely we design 4 sets of D patterns that respectively hold (i) ALU operations, (ii) barrel shifters related instructions, (iii) multiplication operations, (iv) memory loads. We depict these 4 classes as C_{ALU}, C_{BARREL}, C_{MULT} and C_{LOAD}.

In this experiment, we explore a 20×20 spatial grid covering a small region of interest over the chip and we gather 6000 traces (each one being the average of 100 repetitions of a D pattern execution) per position. We split the dataset into 90% for training and 10% for validation. We choose the LDA as our classifier, relying on the implementation provided by the *Scikit-Learn* framework [16]. The validation accuracy of the LDA is used as a metric for evaluating the spatially the leakage of the two Cortex-A9 cores. The results of functional unit classification are shown on Figure 5.

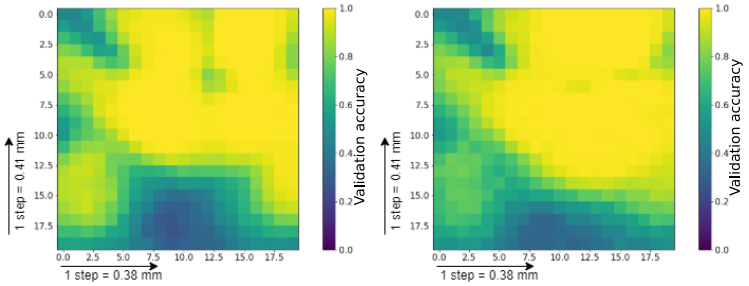


Figure 5: Validation accuracy of the LDA for functional unit prediction over our cartography grid for first (left) and second (right) core of the Cortex-A9.

We observe that the classifier's validation accuracy is spread out differently along the spatial dimension according to the target core, and it reaches 100% success rate on the best positions.

This experiment shows that the coarse-grained recognition of broad functional unit activity can be done accurately by exploiting local EM leakages. Indeed, some probe positions allow to reach 100% accuracy with our experimental setup. Note that such a high classification accuracy is depends on several factors. First, C_{ALU}, C_{BARREL}, C_{MULT} and C_{LOAD} concern D patterns with

several instructions. Hence, as an example, we expect several contiguous `LDR` instructions to produce a leakage significantly different of a sequence of `MUL` instructions, mostly because they are manipulated through distinct hardware components during the execution stage of the pipeline. Also note that there exist a link between the functional units manipulated and the *Cycles Per Instruction* (CPI): this characteristic can be exploited by the classifier. Then, jitter and noise mitigation techniques described in Table 2 improve the alignment of the measurements: this allows the sample-wise linear combination performed by the LDA to be more accurate.

7. Instruction-level Analysis

The coarse-grained recognition of broad functional unit activity allows to investigate finer grained leakages. Especially, the goal of this section is to assess the feasibility of opcode-level disassembly by investigating single-instruction leakages on the Cortex-A9 CPU. To do so, we opt for a hierarchical approach: we start by characterizing the leakages of instructions going through distinct functional units in subsection 7.1 and subsection 7.2 before performing a classification subsection 7.3. Then, we study leakages of arithmetical intructions in subsection 7.4. Finally, we discuss classification results in subsection 7.5.

7.1. Functional-unit leakage assessment

This analysis can be seen as analogous to the first step of a hierarchical approach, as presented in [15]. The main goal is to provide answers to the following questions:

- what is the best spatial region to characterize single-instruction leakages over our DUT?
- can we identify temporal instants that correspond to the pipeline path of an instruction?

Note that this experiment is tied to only one core of the CPU. We craft a set of five different instruction sequences $\mathcal{D}$, each one containing a prologue, a target instruction I preceded and succeeded by 200 `NOP`s, and a function epilogue. The `r1` register is set to a random value and `r2` is fixed before each execution. For each payload, the target instruction I is chosen from a set $\mathcal{I}$ defined as:

$$\mathcal{I} = \left\{ \begin{array}{l} \texttt{NOP} \\ \texttt{ADD r3, r2, r1} \\ \texttt{MUL r3, r2, r1} \\ \texttt{LDR r0, [r0, \#0]} \\ \texttt{LSL r3, r2, r1} \end{array} \right\} \tag{1}$$

For each instruction I from $\mathcal{I}$, the payload is executed 5000 times per position on a 10×20 cartography grid. For each position, we split the traces between 5 classes, namely C_{NOP}, C_{ADD}, C_{MUL}, C_{LDR} and C_{LSL}. Then, we

compute *Signal-to-Noise Ratio* (SNR) for this set of classes as a leakage assessment metric.

The SNR is an univariate leakage metric that allows a comparison between the level of the useful signal and the level of noise in some observations. The higher the SNR is, the most likely the information will be recoverable. The SNR is defined as:

$$\frac{Var(signal)}{Var(noise)}$$

Namely, if we denote Y the random variable representing the class of an observation X, the SNR can be expressed as follows:

$$SNR(X, Y) = \frac{Var(E(X \mid Y))}{E(Var(X \mid Y))} \tag{2}$$

The numerator, shows that the "useful signal" is $E(X \mid Y)$ (the average per class). The denominator is an estimation of the variance of the noise (the variance inside each class). As the SNR is a univariate metric, it needs to be applied on each samples of the traces individually.

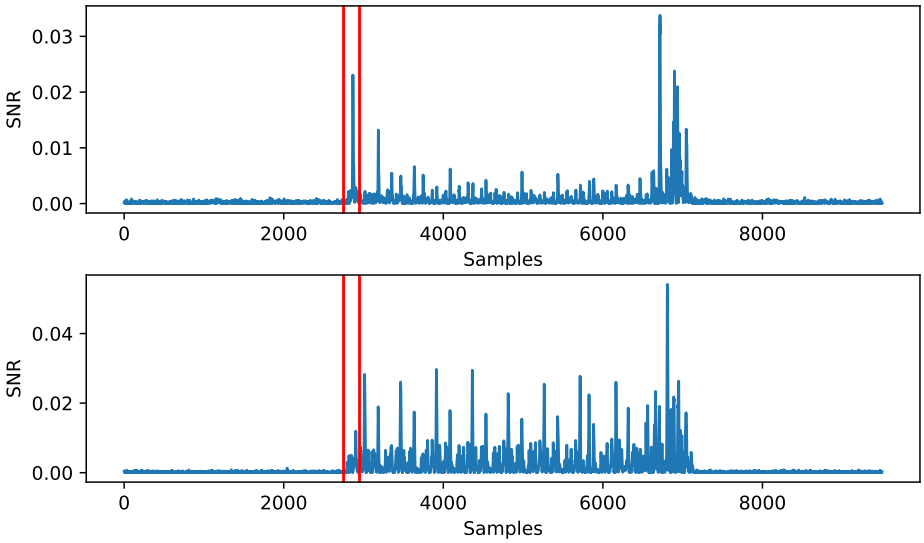


Figure 6: SNR output for the best cartography position with `LDR` instructions enabled (left) and disabled (right), the RoI is bounded by the two red lines.

We can observe in Figure 6 that (i) the SNR starts to rise approximately at the middle of the payload execution (the epilogue occurs around sample 7000), which is the instant I is expected to be executed, (ii) the SNR plot exposes several high peaks until the end of the payload execution. The latter observation can be explained by the difference between the CPI of the different instructions. Indeed this can introduce a slight jitter that desynchronizes the traces and induce biased univariate statistical divergences. Consequently, we define a *Region*

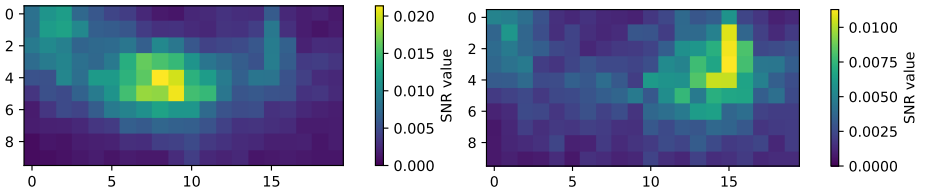


Figure 7: SNR cartography with LDR instructions enabled (left) and disabled (right).

of Interest (RoI) that corresponds to the beginning of the SNR rising. The SNR peak in the RoI is significantly higher when the LDR instruction is enabled (see Figure 6). This can be explained by the activation of several hardware components when a memory load is performed (e.g., data buses, load/store unit, caches etc.).

Hence, the cartography in Figure 7 output the maximum SNR value upon the RoI. The cartography highlights different spatial regions of leakages, depending on the presence of memory loads within the payloads. From Figure 6 and Figure 7, we can deduce that memory loads produce a significantly higher leakage than the other benchmarked instructions: the suitable spatial positions to characterize assembly instructions is depending on the hardware components involved in the processing of the latter instructions.

7.2. Pairwise Comparisons

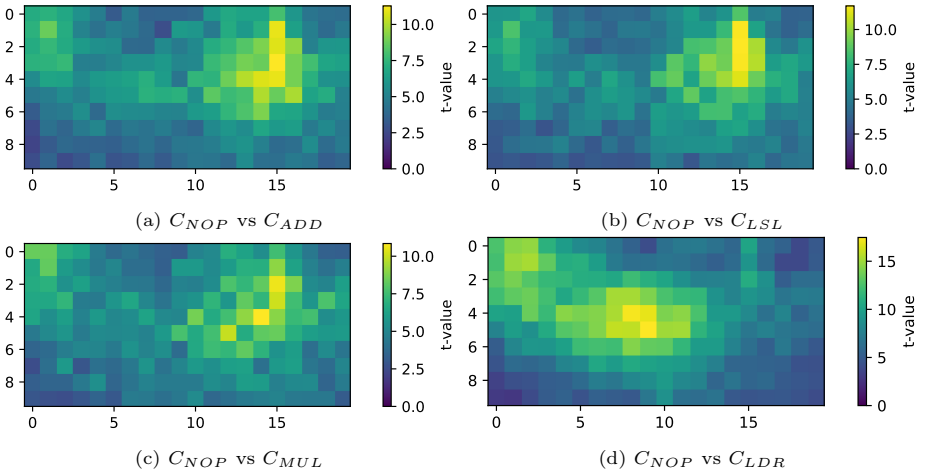
In order to further characterize fine-grained leakages, it seems important to find a probe position where all the instructions from $\mathcal{I}$ leak significantly. Also, we expect this position to expose similar leakages for all $I \in \mathcal{I}$, at least at some temporal instants. Such a finding could lead to both a probe position and time instant that are suitable to investigate early pipeline stages' leakages. Hence an interesting question at this point is:

- in which positions do the different instructions distinguish the most from a NOP instruction?

To answer this question we perform a Welch's t-test between traces from C_{NOP} and each of the other target instructions related traces on every along upon the cartography grid.

We see on Figure 8 that (i) several positions present a high t-value on the RoI for each instruction. (ii) a zone near the top right corner of the grid contains the highest t-values for ADD, LSL and MUL instructions, (ii) the most significant LDR operation spatial leakage zone is separated from the other instruction's ones. At this point we can identify a position that exposes high leakage values for all payloads (i.e., the position at $(3, 15)$ on the grid).

Figure 9 illustrates the outputs of the different Welch's t-tests at the latter best position. We can observe that there are three patterns where the leakages of the different instructions from $\mathcal{I}$ behave analogously (i.e., between samples 2760 and 2830). This can indicate that the instructions receive a similar treatment



(a) C_{NOP} vs C_{ADD} (b) C_{NOP} vs C_{LSL}

(c) C_{NOP} vs C_{MUL} (d) C_{NOP} vs C_{LDR}

Figure 8: Pairwise Welch's t-test cartography where C_{ADD}, C_{LSL}, C_{MUL}, C_{LDR} payloads are compared to C_{NOP}.

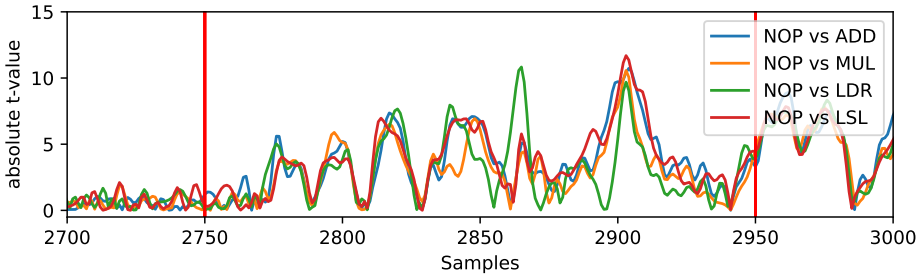


Figure 9: Outputs of the Welch's t-test upon the best selected position for the ADD, LSL, MUL, LDR payloads, the two red lines represent the RoI.

at this step, and thus potentially indicate an early pipeline stage. Then the LDR instruction leaks more than the others for a one cycle duration approximately (sample 2870). From now on, single position experiments will be conducted with the same probe position used here.

7.3. Classification

We build a classifier that aims at discriminating the instructions from $\mathcal{I}$. The goal of this model is to discriminate the use of different functional at a single-instruction level. Hence, for each class C_{NOP}, C_{ADD}, C_{LSL}, C_{MUL}, C_{LDR}, we acquire 500 thousand traces at the best probe position. For each class, a per batch average is computed with varying number of traces: the average traces constitute our dataset. A LDA is then applied upon the normalized RoIs of the traces, with a 90%/10% training/validation split. The classifier reaches up to 96% accuracy in average with a batch averaging size of 4900. We notice in Figure 10 that a higher per batch averaging induces a higher accuracy.

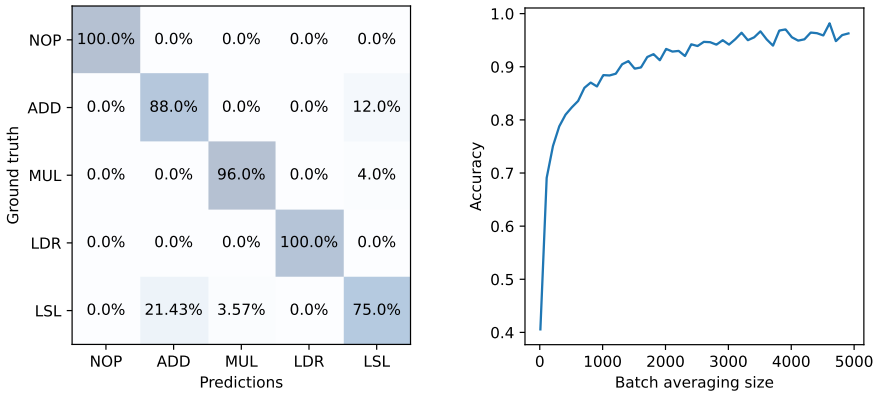


Figure 10: Confusion matrix of the LDA classification between instructions that use different functional units, with an averaging size of 2000 (left), sensitivity of the accuracy compared to the averaging size (right).

This experiment shows that, under rigorous conditions, single-instructions targeting different functional units can be discriminated with a high-accuracy (i.e., superior to 90%). Note that several classes of instructions have been evicted from our analysis, essentially to preserve the stability of our experiments, and are left as further work. For example, branching instructions would impact the control flow of our target code and, thus, necessitate special investigation.

7.4. ALU instructions leakage assessment

Here we want to answer the following question: *is this possible to classify different instructions that go through the same functional unit?*

We select the ALU as our target functional unit. Hereafter we depict the $\mathcal{I}$ chosen for this experiment:

$$\mathcal{I} = \left\{ \begin{array}{l} \texttt{ADD r2, r1, \#imm} \\ \texttt{SUB r2, r1, \#imm} \\ \texttt{EOR r2, r1, \#imm} \\ \texttt{MOV r1, \#imm} \\ \texttt{MVN r1, \#imm} \end{array} \right\} \tag{3}$$

For each acquisition, we generate a target code D that contains a single instruction $I \in \mathcal{I}$. The `r1` register, as well as the immediate value $imm \in \{0, ..., 256\}$ are chosen randomly for each execution. Both randomized values and single functional unit targeting operated by $\mathcal{I}$ allow to limit the leakages in late pipeline stages. Moreover, we expect a distinguishable leakage in the decode stages of the pipeline. We add that the register renaming phase of the Out-of-Order semantics really helps to limit the leakages of the physical registers: the choice of the latter for a specific register label (e.g., `r1` or `r2`) is hardly predictable.

From $\mathcal{I}$ we derive 5 classes C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV}, C_{MVN}. We acquire a total of 12.5 million traces in total for this step We first perform a leakage assessment phase with a SNR computation on a set of 12.5 million traces, equally distributed between the different classes.

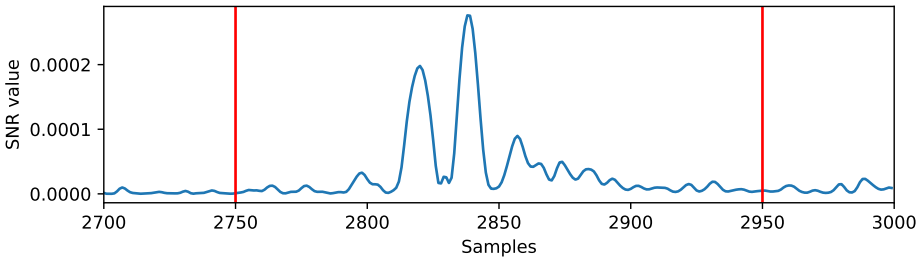


Figure 11: SNR output for classes C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV}, C_{MVN}, the previously identified RoI is bounded with red lines.

The SNR output is depicted in Figure 11: we remark that it exposes two major peaks. Interestingly, the separation between these two peaks is of one CPU cycle. Moreover, it is stated in [1] that the Cortex-A9 incorporates a two stages decoding routine. These observations, jointly with the properties of our characterization code, are strong arguments for assessing that the observed leakage stems from the decoding mechanics of the CPU.

7.5. Classification

We aim at classifying the traces between the 5 classes C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV}, C_{MVN}. As for previous experiments, a per trace batch averaging is performed with a varying number of batch size. We depict the LDA classification result in Figure 12.

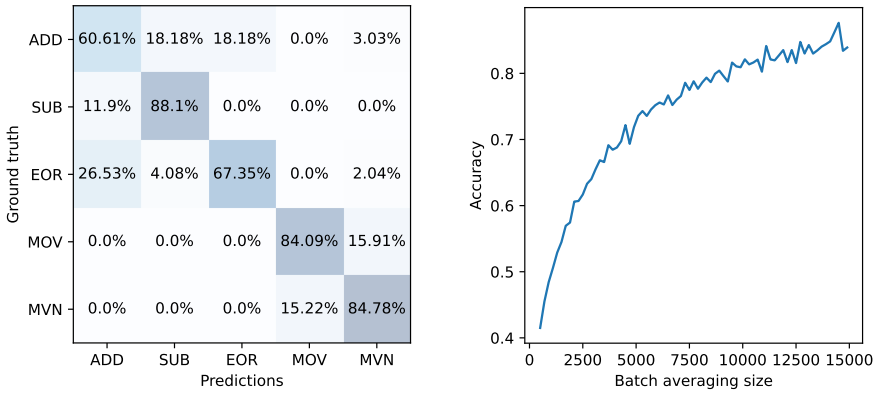


Figure 12: Confusion matrix of the LDA classification between ALU operations, with an averaging size of 6000 (left), sensitivity of the accuracy compared to the averaging size (right) for same functional unit instruction classification.

As for Figure 10, we observe in Figure 12 that increasing the averaging batch size is beneficial regarding the LDA's accuracy. The classifier reaches around 85% accuracy in average for highest batch averaging sizes. This experiment shows that, under a highly controlled execution context, opcode discrimination from a set of instructions $\mathcal{I}$ manipulated through an ALU is doable by exploiting the decoding semantics of a high-end SoC's CPU. Together with the experiments performed in subsection 7.3, we show that the opcode reconstruction based SCBD is doable with high accuracy on a Out-of-Order SoC's CPU under a controlled execution context.

8. Bit-level approach

This section is dedicated to the investigation of single instruction bits leakages within the pipeline stages displayed in the RoI identified in section 7.

When focusing on single instruction bits, we must categorize two types of leakages: (i) direct leakages that are related to the actual manipulation of bit values and (ii) indirect leakages, where an instruction bit influences the behaviour of the CPU. While the first category is more likely to happen during prefetch or fetching semantics (i.e., when the actual bits that encode an instruction go through inter-stages buffers and buses), the second is more likely to be observable during late pipeline stages. This part is dedicated to investigate direct leakages of the instruction bits.

A bit-level approach comes with bitwise leakage models considerations. Indeed, for an instruction, three main leakage models can be considered at a bit granularity:

- the Hamming Weight model (HW), which corresponds to the actual bit value,
- the Hamming Distance model (HD), that exploits the transition of bit values.
- the Signed Hamming Distance model (SHD) that also exploits transitions of bit values while making a distinction between $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions.

Note that the target code structure we opted for is designed not to be restrictive regarding the leakage models. Indeed, as I is surrounded by `NOP`s, considering a HW leakage model can also lead to the identification of HD and SHD leakages between the target instruction and a constant `NOP`. Moreover, if distance based leakages were to be present while a target instruction I is processed, such distances could happen with several instructions, that might be separated from I by distinct clock cycles. This would highly impact the complexity of this analysis, and is thus left out of the scope of this work.

8.1. Bit-Level Approach Against Out-of-Order CPUs

As mentioned in subsection 4.2, Out-Of-Order CPUs perform register renaming to break some data dependences in order to speed-up execution time. Register renaming is a technique that consists in mapping virtual registers (i.e., registers tagged in the instruction encoding, such as `r1` or `r2`) to physical registers, which are the actual hardware resources. As this mapping process is almost unpredictable in practice, it seems hardly possible to exploit the leakage of physical registers in order to guess the virtual register tagged in the original assembly instruction. Then, one can rely on the actual bits that encode the virtual registers. This implies targeting early execution semantics (i.e., fetching or prefetching), in which bit-level SCBD have been shown to be accurate [5].

8.2. Prefetching semantics leakage assessment

Discovering a prefetching routine leakage could allow to exploit single-bit leakages with few biases [5]. To highlight such leakages, we design an experiment which goal is twofold. Firstly, we want to show that the previous RoI corresponds to the target instruction I being processed within the CPU's pipeline. Secondly, as the prefetch of an instruction occurs several cycle before the latter goes through the pipeline, we expect a leakage of a prefetch routine to be anterior to the pipeline's one. For this experiment, the choice of the payload set $\mathcal{D}$, and more precisely the set of target instructions $\mathcal{I}$ is to be especially cared of. Namely we must ensure that:

- Every instruction $I \in \mathcal{I}$ have the same CPI, so that the observed leakage is not resulting from a "shift" in the traces (i.e., jitter), and all leakages are as much aligned as possible.
- Every instruction $I \in \mathcal{I}$ incorporate a random component (e.g., the value contained in a register), to limit statistical biases in late pipeline stages.
- All instructions in $\mathcal{I}$ have a variable operand (e.g., an immediate value) that is located at the same position regarding the ISA's encoding. Indeed, this would allow to target the same buffer location regardless the chosen instruction from $\mathcal{I}$.

Hereafter we depict the $\mathcal{I}$ chosen for this experiment:

$$\mathcal{I} = \left\{ \begin{array}{l} \texttt{ADD r2, r1, \#0} \\ \texttt{ADD r2, r1, \#255} \\ \texttt{SUB r2, r1, \#0} \\ \texttt{SUB r2, r1, \#255} \\ \texttt{EOR r2, r1, \#0} \\ \texttt{EOR r2, r1, \#255} \\ \texttt{MOV r1, \#0} \\ \texttt{MOV r1, \#255} \\ \texttt{MVN r1, \#0} \\ \texttt{MVN r1, \#255} \end{array} \right\} \tag{4}$$

Note that the `r1` register is randomly initialized before each execution, and `r2` is fixed. As previous experiments performed in this section, every I is surrounded by a fixed number of `NOP`s.

We acquire 500 thousand traces for each $I \in \mathcal{I}$, leading to a total of 5 million traces. Then we divide the traces into several classes:

- C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV} and C_{MVN} containing traces that correspond to the precised opcode, regardless the immediate values,
- C_0 and C_{255} that contain traces that correspond to the precised immediate value, regardless the opcode.

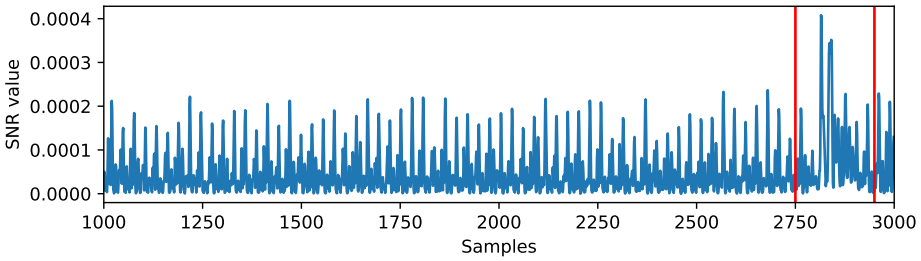


Figure 13: Output of the SNR computed on C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV} and C_{MVN}, the two red lines represent the RoI.

The SNR computed on classes C_{ADD}, C_{SUB}, C_{EOR}, C_{MOV} and C_{MVN} is depicted in Figure 13. Interestingly, one can remark that the highest SNR peaks are located in the same RoI as for Figure 9. This means that this zone indicates the locations where, independently from the operands, instructions of same execution time leak the most. Hence this RoI seems to encompass leakages of stages (or inter-stages buffers) at least from the decoding part of the pipeline to the writeback. Note that within each class, the immediate values 0 and 255 are equally represented. Hence Figure 9 should not be directly influenced by these values.

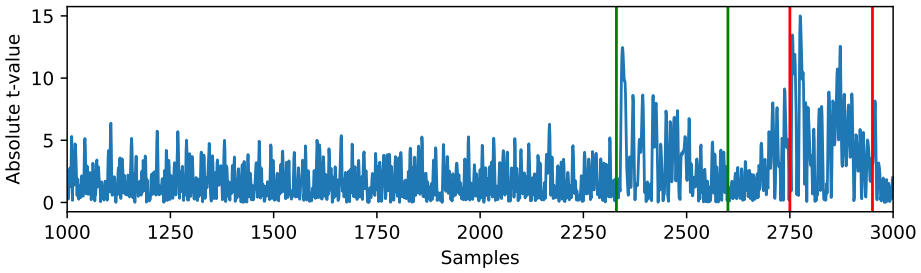


Figure 14: Outputs of the Welch's t-test between C_0 and C_{255}, the two red lines represent the RoI, the two green lines surround the prefetching routine.

The Welch's t-test between C_0 and C_{255} is illustrated in Figure 14. Several observations can be made from this figure: (i) the RoI still exposes leakages,

this can indicate that inter-stages buffers that carry the immediate values of instructions are leaking, (ii) the beginning of the RoI (around sample 2750) is much more leaky than in Figure 13, this can highlight leakages in the fetch stages of the pipeline, and (iii) we observe a region (delimited by green lines), several cycles anterior to the pipeline leakages, that displays immediate values dependant leaks.

Based on these two experiments, we can conclude that we were able to design an accurate methodology to detect prefetch leakages on the Cortex-A9 architecture. This prefetch leakage could originate from the instruction cache, the prefetch unit, interconnection buses or all these components together.

8.3. Classification

In this experiment, we make use of the set of instructions $\mathcal{I}$ depicted in subsection 7.4. Hence we gather 12.5 million traces of payload execution, with an equal proportion for each immediate value.

As the immediate value is drawn from $\{0, ..., 255\}$, it is represented with 8 consecutive bits within the encoding of all $I \in \mathcal{I}$. We consequently create 8 classifiers, one for each bit, that aims at recovering the value of this bit. Regarding the RoI used for classification, we select the samples between the beginning of the prefetch zone (sample 2730 in Figure 14) until the beginning of the pipeline zone (sample 2800 in Figure 14), before the decoding semantics location identified in Figure 11. Once again, we use an in-class per batch averaging before providing the data to the LDA models. The validation accuracy of the 8 LDA classifiers is illustrated in Table 3.

Bit index	7	6	5	4	3	2	1	0
Accuracy	92.1%	64.5%	65.9%	50.5%	57.9%	53.3%	57.5%	50.0%

Table 3: Classification validation accuracies of the immediate value's bits, from most significant bit (bit 7) to least significant bit (bit 0), with a batch averaging size of 6000.

We observe that one bit is accurately classified (i.e., 92.1% for bit 7), and two other bits are recognized with an accuracy superior to 60%. As 0 and 1 values for each bit are equally distributed for a uniform distribution of immediate values from the interval $\{0, ..., 256\}$, the expected accuracies for a guess at random on a bit value is 50%.

Consequently, this experiment shows that, considering a prefetching and fetching temporal region and under a strictly controlled context, we are able to accurately classify one bit of an immediate value, and produce classifications that are better than random guesses on most of the other bits. These results support the feasibility of a bit-level SCBD on a high-end SoC's Out-of-Order CPU. Moreover, one could consider increasing the number on measurements, opting for a more complex learning model or combining information at several probe positions to enhance the accuracy.

9. Conclusion and Further Work

In this paper, we highlight the differences between microcontrollers and high-end SoCs through the lens of a side-channel analyst. This allows us to derive methodological insights that provides an appropriate side-channel acquisition context for SCBD. With such measures, we were able to reliably (i.e., with 100% accuracy) distinguish broad side-channel activities between different functional units without the need of advanced deep learning techniques. The low quantity of measurements (i.e., less than a million) for coarse-grained functional unit activity detection is a good indicator concerning to the feasibility of such recovery in a realistic scenario. At a finer scale (i.e., single instruction level), we show that single instructions going through different functional units also leak, and we were able to identify the temporal region where an instruction is processed through the pipeline. Interestingly, we also show that memory interactions have a major contribution to the leakage on the Cortex-A9 CPU. Classification results show the feasibility of the first level of a hierarchical classifier, that aims at recovering the "category of an instruction" based on the functional units involved in its computation. Next we show that ALU related instructions can be discriminated with good confidence by exploiting deconding semantics of the pipeline. These two results support the feasibility of opcode level SCBD on a high-end SoC.

Afterwise, we identify prefetching related leakage, and exploit the latter to craft a bit-level classifier that is able to classify single bit values of immediate operands, with variable accuracies depending on the bit index. This result shows that bit-level SCBD could be practical on a high-end SoC's CPU, but the effort to perform such a technique is much superior to previous methods.

Most of the analyses presented in this paper are performed at a single EM probe position, and with simple machine learning techniques. Future researches can be headed with more advanced models, such as neural networks, and potentially exploiting leakages at multiple probe positions. As a final word, we recall that all the experiments described in this paper are performed under a strictly controlled execution environment. This means that extrapolating this work onto a more realistic scenario would necessarily add supplementary difficulty from an attacker's perspective. Consequently, further work could then tend to alleviate the attacker model presented in this work.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 883156 (EXFILES).

References

[1] *7-cpu*.

[2] Alessandro Barenghi and Gerardo Pelosi, *Side-channel security of superscalar cpus: evaluating the impact of micro-architectural features*, 2018 55th acm/esda/ieee design automation conference (dac), 2018, pp. 1–6.

[3] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi, *Template attacks*, International workshop on cryptographic hardware and embedded systems, 2002.

[4] Christophe Clavier, *Side channel analysis for reverse engineering (SCARE), an improved attack against a secret A3/A8 GSM algorithm* (2004).

[5] Valence Cristiani, Maxime Lecomte, and Thomas Hiscock, *A bit-level approach to side-channel based disassembling*, Proceedings of the smart card research and advanced application conference, 2019.

[6] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel, *Building a side channel based disassembler*, Transactions on computational science, 2010.

[7] Hedi Fendri, Marco Macchetti, Jérôme Perrine, and Mirjana Stojilovic, *A deep-learning approach to side-channel based cpu disassembly at design time* (2022).

[8] Si Gao, Elisabeth Oswald, and Dan Page, *Reverse engineering the micro-architectural leakage features of a commercial processor*, Cryptology ePrint Archive (2021).

[9] Martin Goldack and Ing Christof Paar, *Side-channel based reverse engineering for microcontrollers*, Master's thesis, Ruhr-Universität Bochum, Germany (2008).

[10] Hong Liu and Eugene Vasserman, *Side-channel programming for software integrity checking*, EAI Endorsed Transactions on Security and Safety **8** (2021), no. 28, e2.

[11] ARM LTD, *Cortex-a9 technical reference manual*, Revision: r4p1 (2012).

[12] Ben Marshall, Dan Page, and James Webb, *Miracle: Micro-architectural leakage evaluation*, Cryptology ePrint Archive (2021).

[13] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes, *Precise instruction-level side channel profiling of embedded processors*, International conference on information security practice and experience, 2014.

[14] Roman Novak, *Side-channel based reverse engineering of secret algorithms*, Proceedings of the electrotechnical and computer science conference, 2003.

[15] Jungmin Park, Xiaolin Xu, Yier Jin, Domenic Forte, and Mark Tehranipoor, *Power-based side-channel instruction-level disassembler*, Design automation conference, 2018.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research **12** (2011), 2825–2830.

[17] Pieter Robyns, Mariano Di Martino, Dennis Giese, Wim Lamotte, Peter Quax, and Guevara Noubir, *Practical operation extraction from electromagnetic leakage for side-channel analysis and reverse engineering*, Conference on security and privacy in wireless and mobile networks, 2020.

[18] Daehyun Strobel, Florian Bache, David Oswald, Falk Schellenberg, and Christof Paar, *SCANDALee: a side-channel-based disassembler using local electromagnetic emanations*, Design, automation & test conference, 2015.

[19] Shahram Vafa, Massoud Masoumi, and Amir Amini, *An efficient profiling attack to real codes of pic16f690 and arm cortex-m3*, IEEE Access **8** (2020), 222520–222532.

[20] Dennis Vermoen, Marc Witteman, and Georgi N Gaydadjiev, *Reverse engineering java card applets using power analysis*, International workshop on information security theory and practices, 2007.

[21] Miller L Wilt, Megan M Baker, and Stergios J Papadakis, *Toward an rf side-channel reverse engineering tool*, Physical assurance and inspection of electronics (paine), 2020.