Journal of Hardware and Systems Security (2023) 7:72–99
https://doi.org/10.1007/s41635-023-00135-1

Instruction-Level Power Side-Channel Leakage Evaluation of Soft-Core CPUs on Shared FPGAs

Ognjen Glamočanin[1] · Shashwat Shrivastava[1] · Jinwei Yao[1] · Nour Ardo[1] · Mathias Payer[1] · Mirjana Stojilović[1]

Received: 14 January 2023 / Accepted: 26 July 2023 / Published online: 4 October 2023
© The Author(s) 2023

Abstract
Side-channel disassembly attacks recover CPU instructions from power or electromagnetic side-channel traces measured during code execution. These attacks typically rely on physical access, proximity to the victim device, and high sampling rate measuring instruments. In this work, however, we analyze the CPU instruction-level power side-channel leakage in an environment that lacks physical access or expensive measuring equipment. We show that instruction leakage is present even in a multitenant FPGA scenario, where the victim uses a soft-core CPU, and the adversary deploys on-chip voltage-fluctuation sensors. Unlike previous *remote* power side-channel attacks, which either require a considerable number of victim traces or attack large victim circuits such as machine learning accelerators, we take an evaluator's point of view and provide an analysis of the instruction-level power side-channel leakage of a small open-source RISC-V soft processor core. To investigate whether the power side-channel traces leak secrets, we profile the victim device and implement various instruction opcode classifiers based on both classical machine learning algorithms used in disassembly attacks, and novel, deep learning approaches. We explore how parameters such as placement, trace averaging, profiling templates, and different FPGA families (including a cloud-scale FPGA) impact the classification accuracy. Despite the limited leakage of the soft-core CPU victim and a reduced accuracy and sampling rate of on-chip sensors, we show that in a worst-case scenario for the evaluator, i.e., an attacker breaching physical separation, we can identify the opcode of executed instructions with an average accuracy as high as 86.46%. Our analysis shows that determining the executed instruction type is not a classification bottleneck, while leakages between instructions of the same type can be challenging for deep learning models to distinguish. We also show that the instruction-level leakage is significantly reduced in a cloud-scale FPGA scenario with higher soft-core CPU frequencies. Nevertheless, our results show that even small circuits, such as soft-core CPUs, leak potentially exploitable information through on-chip power side channels, and users should deploy mitigation techniques against disassembly attacks to protect their proprietary code and data.

Keywords FPGA · Multitenancy · CPU instruction identification · Power side-channel attack

1 Introduction

Due to the end of Moore's law and the breakdown of Dennard's scaling, datacenters are transitioning from homogeneous and processor-dominated systems towards heterogeneous architectures. As a result, today's datacenters feature not only central processing units (CPUs), but also graphics processing units (GPUs) and special-purpose integrated circuits such as field-programmable gate arrays (FPGAs). FPGAs reached wide deployment in datacenters thanks to their highly parallel architecture, programmability, and energy efficiency [1, 2]. Even though FPGA vendors offer FPGA-based system-on-chips (SoCs) with hardened CPUs [3, 4], cloud providers are exclusively integrating regular FPGAs in their servers because servers are already abundant in high-end server-grade CPUs. Amazon EC2 F1 [1], Azure [2], Baidu [5], and Tencent [6] deploy AMD Virtex or Kintex Ultrascale+ FPGAs, while Alibaba deploys Intel Arria 10 and Agilex FPGAs [7].

In the FPGA-accelerated cloud, highly-parallel tasks are accelerated on FPGAs. At the same time, developers rely on

✉ Ognjen Glamočanin
ognjen.glamocanin@epfl.ch

✉ Mirjana Stojilović
mirjana.stojilovic@epfl.ch

[1] School of Computer and Communication Sciences, EPFL, Route Cantonale, Lausanne 1015, Vaud, Switzerland

host CPUs for general computation, particularly noniterative and user event-dependent control algorithms, which are significantly easier to implement and maintain in software than in hardware. However, the FPGA-CPU communication incurs high latency, especially for short data transfers [8]. If such delays are of no concern, then the control software can be deployed on a cloud CPU instance; yet, only a limited range of FPGA applications—usually data movement ones—can afford the resulting communication latency. Therefore, in the case of latency-critical control algorithms, system designers resort to using soft-core CPUs as real-time co-processors (e.g., Microblaze [9], Nios [10], PicoRV [11]), which allow tight and customizable integration with FPGA accelerators, and short communication latencies.

Recent efforts focus on extending the multitenancy and resource virtualization from CPUs to FPGAs, to improve the efficiency of datacenter resource provisioning. A specific research focus is achieving spatial or temporal multiplexing of FPGA resources [12–14]. The challenges are numerous, such as partitioning the FPGA resources among multiple users, providing communication protocols between the host virtual machine and the accelerators, and ensuring proper physical and logical isolation between the tenants [15].

Unfortunately, FPGA multitenancy introduces security threats that cannot be mediated by physical or logical isolation between tenants. The reason is the shared power delivery network (PDN), which instigates power side-channel attacks, covert communication [16, 17], and denial-of-service and fault attacks [18, 19]. An attacker does not require physical access to the board, as the fine logic and wiring granularity of FPGAs enables malicious users to craft almost arbitrary hardware, which includes implementing on-chip sensors for measuring the shared power supply voltage fluctuations [20]. Several remote power-analysis attacks have already been demonstrated: a simple-power analysis (SPA) attack on RSA exponentiation [21], correlation power analysis (CPA) attacks against AES (requiring a large number of victim power traces) [22–24], and reverse engineering attacks on neural network accelerators (which occupy a significant portion of the FPGA resources) [25–28].

For an FPGA user, secret information is not limited to their bitstream, the cryptographic key, or neural network accelerator parameters and architecture. If their design contains a soft-core CPU, the code being executed can be proprietary or contain secrets. If an attacker, by observing power side-channel traces during CPU code execution, can determine which instructions are being executed, the confidentiality of the code will be compromised. In embedded applications and smart cards, where adversaries have physical access to the target device to measure power and electromagnetic side-channel leakage, attacks that aim at code recovery are termed side-channel disassembly attacks [29, 30]. Unlike statistical-based power analysis attacks such as CPA, side-channel disassembly attacks are profiling attacks and assume the attacker can record a limited number of victim execution traces.

Our work takes an evaluator's point of view: we explore to which extent soft-core CPUs leak instruction-level information through the remote power side channel, in cases when an evaluator (or a potential attacker) has no physical access to the device but can deploy on-chip voltage-drop sensors. Unlike traditional side-channel disassembly attacks—where the CPU runs at frequencies orders of magnitude lower than the sampling rate of the oscilloscope—sensors used in remote power analysis attacks have sampling frequencies in the same operating range as soft-core CPUs. Our work analyzes if and under which conditions soft-core CPU instructions contain power side-channel leakage and incentivizes the use of protection methods in multitenant FPGAs. As our leakage evaluation targets, we choose two RISC-V soft-core CPUs using the 32-bit RISC-V base integer instruction set architecture (RV32I), most suitable for lightweight real-time co-processors [31].

To start, we record the side-channel traces corresponding to the execution of every CPU instruction. Then, to investigate whether the traces leak secrets, we train diverse machine learning (ML) classifiers used in previous work and also explore the use of novel deep learning (DL) classifiers to improve the extraction of the power side-channel leakage. The results reveal that, despite the limited accuracy and sampling rate of on-chip sensors compared to oscilloscopes used in disassembly attacks with physical access, the limited leakage compared to previous remote reverse-engineering attacks, and the limited number of victim trace acquisitions compared to statistical-based attacks, instruction-level leakages still exist: we can determine the executed instructions with average accuracy higher than 80%. These results call for proper mitigations to limit power side-channel leakage of soft-core CPUs in shared FPGAs.

We make the following contributions:

- To the best of our knowledge, we present the first analysis of instruction-level leakage of soft-core CPUs in a shared FPGA setting.
- While power side-channel traces recorded by an on-chip FPGA sensor during the execution of one RISC-V soft-core CPU instruction contain limited visually observable leakage, we demonstrate that, in certain conditions, advanced ML techniques can extract sufficient information to identify the opcode of the executed instructions. The maximum average instruction accuracy we achieve on the RV32I instruction set architecture (ISA) is 86.46%.
- Besides evaluating previous side-channel disassembly approaches, we explore new, DL-based instruction classifiers, and experimentally find that they are superior at

extracting leakage compared to common ML techniques deployed in previous work, and should be used for future side-channel security evaluations.

- We perform an extensive experimental analysis that compares how different leakage evaluation scenarios, such as the number and placement of sensors, number of templates, and type of templates, affect the instruction-level leakage. We also demonstrate our results on two soft CPU cores and two different FPGA families. In addition to the leakage analysis of the RISC-Y [32] soft-core CPU running at 80 MHz on the Sakura-X board [33], we show results on a cloud-scale, AMD Alveo U200 datacenter accelerator card, using the compact PicoRV [11] soft-core CPU, running at 320 MHz. With our on-chip sensors running at 320 MHz, the side-channel traces have only four sensor samples per CPU clock cycle on Sakura-X, and only one sensor sample per CPU clock cycle on Alveo U200; significantly lower than in traditional side-channel disassembly attacks.
- We provide a detailed discussion of our experimental results and their impact on soft-core CPU leakage evaluation, which we use to motivate appropriate mitigation techniques.

Our work aims to provide a leakage evaluation methodology for soft-core CPUs in remotely accessible scenarios and to benefit future power side-channel disassembly attacks by providing novel DL power trace classification techniques. Therefore, we make all our FPGA designs, associated software, and ML code openly available for the reproducibility of the experiments and the results in this work [34].

2 Background

Almost a decade ago, Microsoft pioneered the use of FPGAs in cloud computing. Their Catapult project pilot of 1,632 FPGA-enabled datacenter servers demonstrated a dramatic improvement in Bing search latency, launching the era of FPGA-accelerated cloud computing [35]. Other cloud service providers soon followed. Today, Amazon AWS, Azure, Alibaba, Baidu, and Tencent offer their customers remote access to datacenter FPGAs, to develop, test, and deploy their custom hardware accelerators [1, 5–7].

To remote users, FPGAs are typically exposed through a host CPU virtual machine interface and a *shell-role* use model [15]. The shell is deployed by the cloud service providers and shares the FPGA logic with the users. In addition, it implements platform-specific management tasks: PCIe, direct memory access engine, DRAM controller, and debugging interfaces. The FPGA region reserved for each user is called a role, and users deploy their accelerators within their role. The shell-role separation helps faster accelerator deployment and ensures different privilege levels between the cloud service providers and the external users.

In both academia and industry, increased efforts are being made to extend multitenancy and resource virtualization from CPUs to FPGAs, to enable better management and use of available datacenter resources [12, 15, 36–46]. Multitenancy can be achieved through spatial and temporal multiplexing. Temporal multiplexing separates users in time, ensuring that each tenant gets their own, exclusive instance. In spatial multiplexing, FPGA roles are occupied by potentially different tenants, and consequently, the cloud service providers need to ensure security and privacy to all of them [45, 47, 48].

Once the shell and the tenants share the FPGA die, they also share the PDN illustrated in Fig. 1. On the printed circuit board (PCB) level, the PDN starts with the primary voltage regulator. The power is then distributed through several levels of voltage regulators if needed, and the power and ground planes. Inside the FPGA, a PDN resembling a dense mesh supplies power to all FPGA logic and routing resources. On all the levels—board, package, chip—the PDN contains resistive, capacitive, and inductive components, some of them intended and some parasitic, which create a medium for voltage fluctuations in one FPGA role to propagate to another. Gnad et al. were the first to demonstrate that a malicious FPGA tenant can, through excessive logic switching, draw too much current and, consequently, reset the host FPGA [18]. Their findings temporarily put on

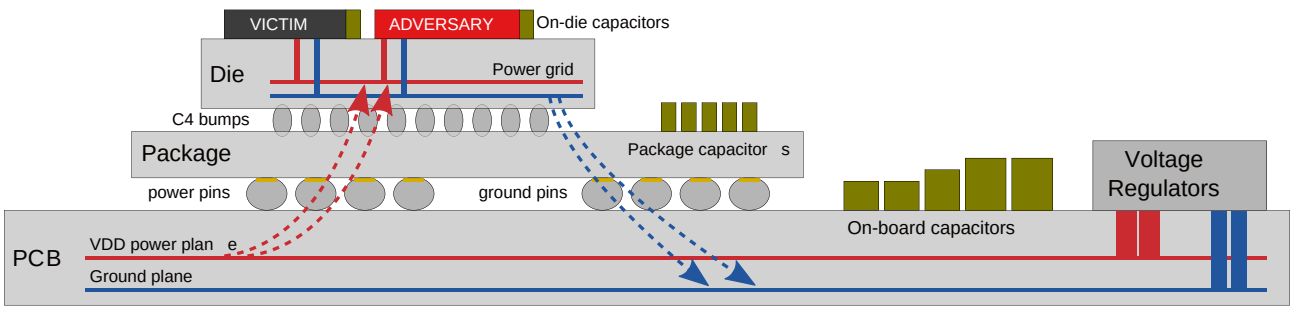


Fig. 1 Power delivery network coupling across the board, package, and the FPGA die

hold the FPGA multitenancy in the cloud and pushed many researchers to investigate new attack surfaces, threat models, and countermeasures [49, 50].

One of such new attack surfaces called *remote power analysis attacks*, was first demonstrated by Zhao and Suh [21] and Schellenberg et al. [22]. Leveraging the fine granularity of reconfigurable logic and routing in FPGAs allowed the designing and implementing of circuits that sense on-chip power supply voltage variations. Unlike traditional power analysis attacks, which require physical access to the victim to measure its power consumption with an oscilloscope [51], using on-chip sensor circuits made these attacks remote, no longer requiring physical access to the device. An example of such a sensing circuit is a simple ring oscillator (RO), composed of an odd number of inverters connected to form a closed chain. Its oscillation frequency depends on the delays of the inverters and routing resources which, in turn, depend on the power supply voltage. Hence, one can also sense the voltage variations by measuring the RO frequency. Another example is a delay-line sensor, also called time-to-digital converter (TDC) [20], which we will discuss in detail in Sect. 4. In a multitenant FPGA setting, an adversary can use such sensors to collect the power side-channel information leaked from a co-located tenant and use it to infer secret information: indeed, on-chip sensors allowed remote attacks on cryptographic circuits, ML accelerators, and other circuits. We summarize the most relevant previous work in Sect. 11.

3 Threat Model

Research on the security of multitenant FPGAs follows a well-established threat model of the fault and side-channel attacks on remote shared FPGAs [19, 21, 23, 28, 49, 52–54]. The primary assumption is that at least two users can remotely deploy their designs on the same FPGA instance simultaneously. For security reasons, these remote users are given control over dedicated partial reconfiguration regions, which are logically and physically isolated; thus, the attacker has no direct access or control over the victim or the victim's deployment. The adversary can deploy voltage fluctuation sensors to record power side-channel traces and send them over the network for remote analysis.

In this work, we assume an evaluator's point of view: we evaluate the security of a victim that uses a soft processor core in their shared FPGA platform, for example, to configure and control the operation of an accelerator. This work analyzes instruction-level leakage to assess if and under which circumstances soft-core CPUs leak instruction information through the power side channel in shared FPGAs, with the goal of motivating the use of countermeasures.

When evaluating the side-channel security of a device, it is a common practice to consider the worst-case estimates (even if not practically achievable by an attacker), as they quantify the limits of the leakage. For example, in the context of cyber-physical devices, white-box power side-channel leakage evaluation methods leverage proprietary architectural information (unavailable to attackers) to build better power models for power analysis attacks [55]. Removing the plastic cover of a chip to record near-field EM emanations is another example of a common practice in leakage evaluations, even though attackers might not always be able to remove the casing. Consequently, our experiments assume and evaluate various scenarios: from worst-case (a breach of physical and logical separation, no additional noise sources, and averaging of traces) to more realistic scenarios, including physical separation, no averaging, and noise from surrounding instructions and the shell.

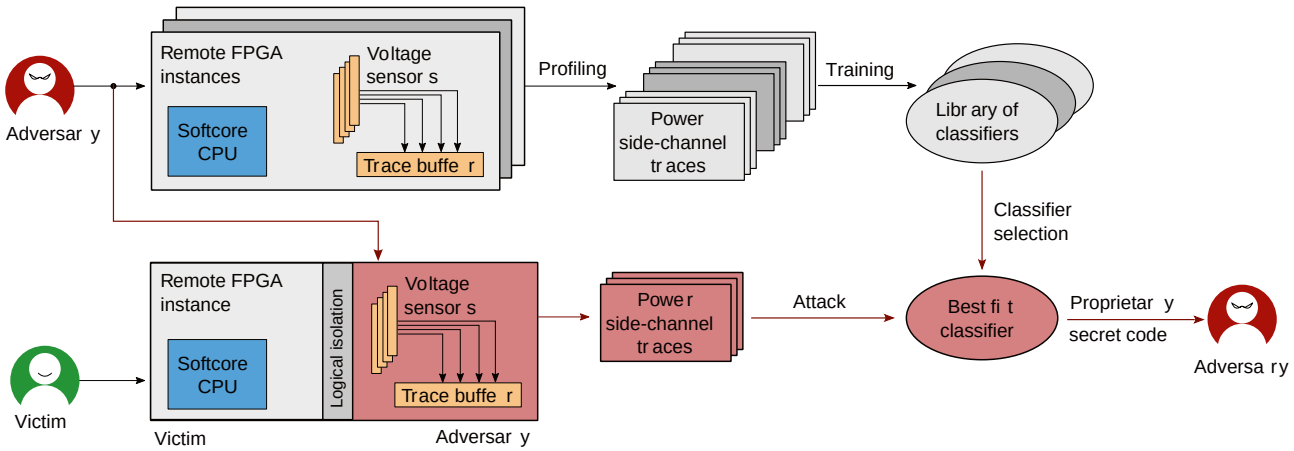


Fig. 2 Threat model. The top half illustrates the profiling phase, which results in a library of side-channel instruction classifiers, for a number of FPGA instances and CPU and sensor placements. The bottom half shows the attack

In reality, a hypothetical attacker mounting a profiling attack on soft processor cores would have to perform a procedure similar to the one shown in Fig. 2. To prepare for the attack, an adversary would start by renting an FPGA instance as its only tenant. On this FPGA instance, the adversary would need to calibrate the voltage fluctuation sensors and use them to profile the execution of the CPU instructions for various operating frequencies and several CPU placements. Then, the attacker could train side-channel instruction classifiers. This step would have to be repeated for many FPGA instances, each uniquely identified (e.g., by fingerprinting cloud FPGAs as suggested by Tian et al. [56]).

To perform an exploit using the library of trained classifiers, the attacker would need to rent a shared FPGA instance. Using fingerprinting to identify the shared FPGA instance, the attacker can focus on the subset of the classifiers in the library trained on that particular FPGA instance. Once side-channel traces are obtained, the adversary would need to identify that the co-located user is using a soft-core CPU (and repeat until a victim with a soft-core CPU is identified), using workload classification techniques [54]. Then the attacker could further prune the subset of trained classifiers using the same workload classification techniques—which can distinguish between different soft-core implementations in shared FPGAs—and run the inference. Finally, in addition to the attack procedure, the attacker would need to train models robust to noise from the shell or any other accelerator the victim might be using alongside their soft processor core.

Our aim is to evaluate how and under which circumstances soft-core CPUs leak instruction information in shared FPGAs, we therefore focus on assessing instruction leakage. We refer to related work for FPGA identification and workload classification.

4 Experimental Setup

The Sakura-X (Sasebo-GIII) board [33] and the Alveo U200 datacenter accelerator card serve as our target evaluation platforms. Sakura-X is an evaluation board designed for power side-channel analysis and, hence, commonly used in both cryptologic research [57, 58] and research on side-channel attacks on shared FPGAs [22, 27, 59]. Sakura-X has one AMD Kintex-7 FPGA and one AMD Spartan-6 FPGA. The former FPGA is the larger of the two, often referred to as *main* or *target* FPGA, as it hosts the adversary and the victim as two logically isolated FPGA tenants. The second FPGA, often referred to as auxiliary or control FPGA, reduces unwanted noise by implementing the communication protocol between the target FPGA and the host machine [33]. For our evaluation, the Sakura-X architecture increases the already low signal-to-noise ratio (SNR) of soft-core CPUs and helps isolate the instruction-level power side-channel leakage. To evaluate the leakages in a more realistic, cloud-scale FPGA scenario, we use the Alveo U200 datacenter accelerator card. This card contains an AMD UltraScale+ XCU200-2FSGD2104E FPGA, and is commonly used in publicly available cloud FPGA instances [2]. Unlike Sakura-X, Alveo U200 contains a single FPGA consisting of three super-logic regions (SLRs). The shell, containing resources necessary for communicating with the DRAM and host CPU, is instantiated in the middle SLR and physically separated from both the attacker and the victim. The placement of the sensor and the victim CPU varies across experiments, however, in most cases, we physically separate the sensors and the victim soft CPU core to conform with the standard shared FPGA threat scenario described in Sect. 3.

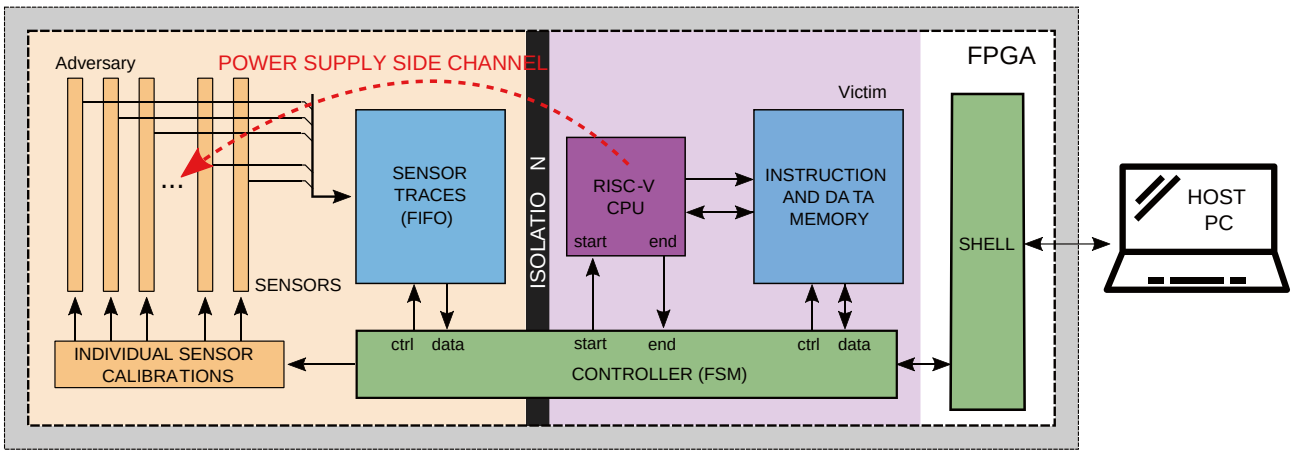


Fig. 3 Overview of the experimental setup

Figure 3 gives an overview of the experimental setup for both boards. The target FPGA design contains the victim and the hypothetical attacker logic and has four main components: a soft-core RISC-V processor, the on-chip voltage-drop sensors, the control finite state machine (FSM), and the shell. As discussed in Sect. 1, the primary purpose of using soft-core CPUs is to implement latency-critical control algorithms, especially ones subject to change over time. Therefore, our study assumes the victim uses small soft-core CPUs, common in embedded bare-metal applications [60]. These soft-core CPUs are usually lightweight, with no advanced microarchitectural features such as cashing or speculative execution. They have a low area overhead and can run at high clock frequencies. Their microarchitectural simplicity allows easy and tight integration with FPGA hardware, facilitating low-latency communication. Integrating larger soft-core CPUs would reduce the operating frequency (e.g., Rocketchip can run on a couple of tens of MHz only [61]), increase the area overhead (reducing the available resources for hardware accelerators), and adversely affect the communication latency (as communication would take place through memory mapped interfaces or an operating system).

For the RISC-V soft-core designs, we chose RISCY and PicoRV32, both openly available [11, 32]. Table 1 summarizes the FPGA resource overhead. As a reference, we also show the resource usage of Rocket Chip [61], a larger, more complex soft-core RISC-V implementation. RISCY, used on the Sakura-X board, implements a classic five-stage pipeline and supports the complete RV32I ISA at the cost of a lower operating clock frequency. On Sakura-X, the maximum operating frequency of the RISCY CPU is 100 MHz; however, our system runs it at 80 MHz, to have an integer number of sensor samples per one CPU clock cycle. PicoRV32, used on the Alveo U200 board, has a multicycle CPU microarchitecture designed to minimize resources and maximize the CPU operating frequency. Our system runs PicoRV32 at the maximum operating clock frequency of 320 MHz. In the following subsections, we describe the voltage-drop sensors and the controller in detail.

4.1 FPGA Voltage-Drop Sensors

Commonly deployed FPGA voltage-drop sensors fall into two groups: TDCs and RO-based sensors [49, 62]. They both produce an output in the function of their circuit delay, which is approximately inversely proportional to the supply voltage. Hence, the change in the sensor logic delays indirectly exposes the core voltage fluctuations, caused by the switching activity and power consumption of the victim [21]. The key criterion when choosing between a TDC and an RO-based sensor is the required sensor sampling rate: RO-based sensors cannot be sampled as frequently as TDC sensors. However, RO-based sensors have a smaller footprint and need not be calibrated, unlike TDCs. Good use cases for RO-based sensors are FPGA undervolting-based attacks [53, 63] and covert communication [64]. For side-channel analysis, given the importance of a high sampling rate, TDCs are the preferred solution [22, 23, 28]; they are able to record voltage fluctuations with sampling periods in the nanosecond range [20].

The baseline design of a TDC was proposed by Zick et al. [20]. It consisted of two principal components, (1) one delay line implemented using fast carry chain logic and (2) latches, connected to the output of every delay element in the delay chain. At the input of the carry chain, a high-frequency clock signal was connected; let us refer to it as *input* clock. Another clock signal, the *sampling* clock, of the same frequency but a slightly different phase, was used to capture the propagation depth of the rising edge of the input clock through the delay chain. The propagation depth reflected the changes of the carry logic delay, which were primarily caused by the power supply variations. More recently, TDC sensors have replaced latches with flip flops and used a digital clock manager (DCM) to control the phase delay between the input and the sample clock. Proper selection of the phase shift and the delay line length is critical for correct sensor calibration, i.e., for ensuring that the rising edge is indeed captured and not missed. Since the calibration is a lengthy process of trial and error, in our attack model it must be automated. Hence, we design and implement a TDC with a tunable phase shift mechanism and, as suggested in previous work, we avoid instantiating a DCM primitive to reduce jitter [65].

Table 1 Resource utilization of the soft-core CPUs

CPU	FPGA	LUT	FF	BRAM36	DSP
RISCY [32]	Kintex-7 XC7K160T-1FBGC	2544	1944	40	0
PicoRV32 [11]	Virtex Ultrascale+ XCU200-FSGD2104-2-E	1442	1473	8	0
Rocket Chip [61]	Virtex Ultrascale+ XCU200-FSGD2104-2-E	25785	12654	12	15

The design of our TDC is inspired by the implementation of Gnad et al. [16]. Its high-level architecture is shown in Fig. 4. The TDC is composed of fine calibration slices, coarse calibration slices, and an observable delay line, which is periodically sampled and its state saved in the *output register*. The input and the sample clocks are the same. To control the phase shift between the input and the sample clock, *fine* and *coarse* calibration slices are inserted on the input clock path. In the fine calibration slice, as shown in Fig. 5a, calibration inputs control the number of carry chain multiplexers on the clock path. The fine calibration slices are then connected to the coarse calibration slices (Fig. 5b), where the calibration inputs control the number of coarser delay elements on the clock path. In our TDC design, unlike in Gnad et al. [16], coarse delay elements are implemented as LUTs followed by latches, to achieve coarser delay increments. The third and last stage is the *observable delay line* (Fig. 5c). This sensor is considered correctly calibrated when the signal propagating through the chain of delay elements reaches approximately the middle of the observable delay line by the moment it gets captured in the output register.

In this work, TDC sensors have a 16-bit observable delay line. Through experimentation, we found that this length is sufficient to capture the supply voltage variations caused by the CPU operation on both FPGA boards. Table 1 lists the FPGA resources used for our TDC implementation on both boards. The sensor clock frequency was set to 320 MHz on both boards, the highest operating frequency that satisfied timing constraints. Consequently, the sensor captures four samples per one clock cycle of the RISCY CPU running at 80 MHz, and one sample per clock cycle of the PicoRV32 CPU running at 320 MHz.

Previous work has shown that the side-channel information captured by voltage-drop sensors varies with both the absolute location of the sensors as well as their relative position to the victim [66]. It is, therefore, to be expected that an attacker may instantiate more than one power side-channel sensor. The exact number is usually limited by the linearly scaling on-chip memory resources and the data transfer word size. For example, to improve the success of their attack, Gravellier et al. [24] deployed eight sensors on an AMD Artix-7 FPGA. In our experimental setup, we instantiate five TDCs on Sakura-X, and 29 TDCs on Alveo U200, the highest number that fits in a communication message exchanged between the FPGA [33] and the host PC. In Sects. 6 and 7, we will show to what extent having multiple sensors affects the attack efficiency.

4.2 Controller

The controller coordinates the experiments by executing and replying to the commands from the host machine through the shell. It is in charge of initializing the CPU instruction memory with the code to be executed, triggering the execution of the code, and saving the corresponding sensor traces to the on-chip memory. Once the CPU code execution is completed, the controller receives a trigger from the CPU, which initiates the transfer of sensor traces to the host machine. In each message sent from the FPGA to the host, the controller inserts five (Sakura-X) or 29 (Alveo U200) simultaneous sensor readings and the 32-bit word of the corresponding CPU instruction. We replace the default read-only instruction memory of both CPUs with a dual-port block RAM, connecting one memory port to the CPU while exposing the other port to the controller. This temporary change permits the controller to write arbitrary code in the CPU instruction memory before triggering its execution and recording the side-channel traces.

Prior to starting the experiments, the controller calibrates every sensor. The calibration is performed iteratively. First, a test code sequence is loaded to the instruction memory, and the number of elements in the sensor's initial delay line is set to zero. The code execution is triggered, and the obtained sensor trace is inspected. If no clock transition is observed or the transition is located too close to the two extremes of the observable delay line, the fine and coarse calibration slices are adjusted. This process is repeated until the sensor is calibrated. The calibration settings are then communicated to the host machine for record keeping.

5 Instruction Classification

Like all hardware circuits, soft-core CPUs leak information through the power side channel. Various ALU operations, memory accesses, and control-flow changes all impact power

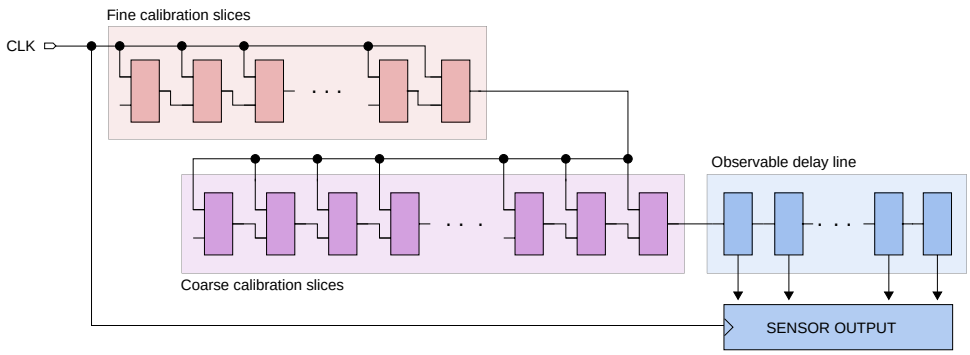
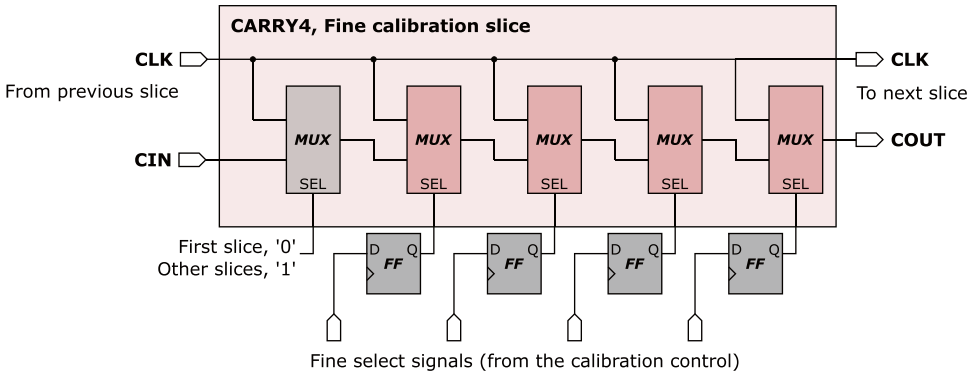
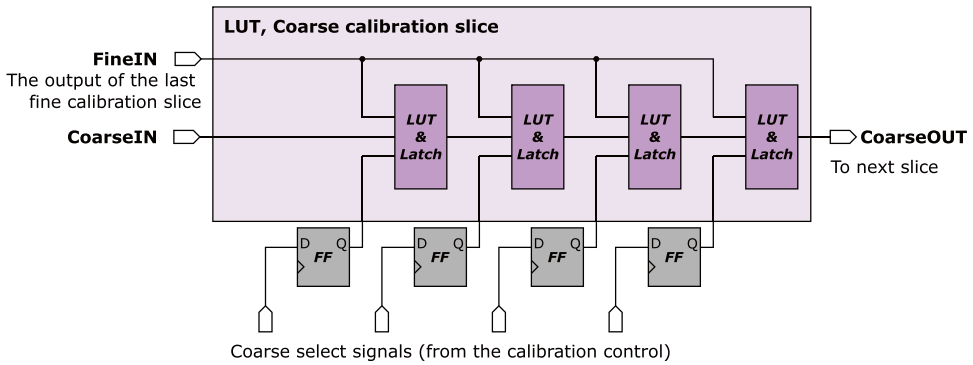


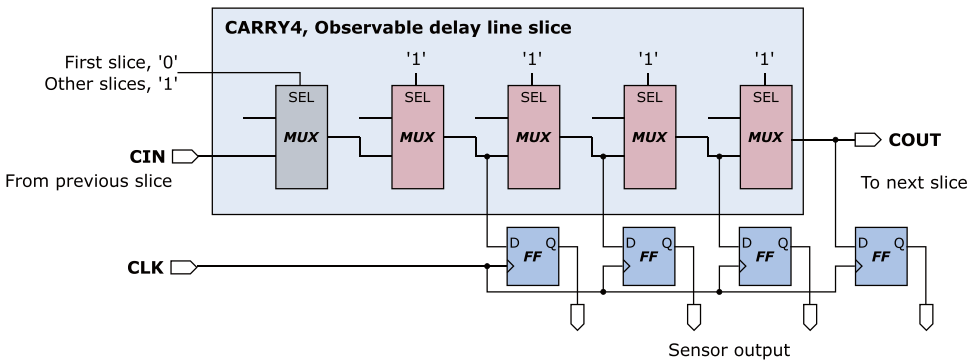
Fig. 4 TDC sensor architecture with a tunable phase shift between the clock that enters the observable delay line and the clock that samples the output (i.e., takes a snapshot of the observable delay line). The exact number of slices in our implementation is in Table 2



(a) Fine calibration slice.



(b) Coarse calibration slice.



(c) Observable delay line slice.

Fig. 5 The implementation of each slice in the TDC sensor in Fig. 4, including the calibration and sensor output registers. For space reasons, CARRY4 chain is shown horizontally; in the FPGA design layout, it spans vertically

consumption differently. In addition, as a combination of fetch, ALU, memory, and program counter operations, instructions also leak information: in the form of unique patterns spread across the time and amplitude domain of the recorded power traces. For example, on the one hand, memory instructions might have high power consumption both in the ALU stage, when the address is computed, and in the later stages of instruction execution, i.e., when the data is read/written to the

Table 2 Coarse calibration, fine calibration, and observable delay line slices per sensor

FPGA	Fine calibration	Coarse calibration	Observable line
Kintex-7	24 slices	8 slices	4 slices
XC7K160T-1FBGC	(= 96 stages)	(= 32 LUTs and Latches)	(= 16 FFs)
Virtex Ultrascale+	12 slices	4 slices	2 slices
XCU200-FSGD2104-2-E	(= 96 stages)	(= 32 LUTs and Latches)	(= 16 FFs)

memory. On the other hand, arithmetic instructions might only have a power consumption peak during the ALU stage.

To analyze the instruction-level power side-channel leakage of soft-core CPUs, we employ an ML-inspired method illustrated in Fig. 6. The key idea behind this approach is that leakage patterns are discovered during ML model training, while the leakage is assessed using the prediction accuracy achieved on templates unseen during training. For this purpose, we first build a large set of template assembly codes for all the target instructions: we generate a set of 10,000 templates for every instruction. Once the templates database is ready, we run the experiments to collect the corresponding power side-channel traces. As leakage evaluators, we reduce the background noise and improve the signal-to-noise ratio by executing each template multiple times and averaging the side-channel traces: 100 times for Sakura-X and 1000 times for Alveo U200. Even though our work represents an instruction-level leakage analysis, averaging is still a commonly used noise reduction approach even in real attack scenarios: for an attack, the victim code is often executed frequently, allowing averaging, while during training, the attacker can execute templates an arbitrary amount of times [67–70]. Finally, to spread out the impact of environmental noise equally across all instruction classes, we record traces in an interleaved fashion: we record a single trace of each class, in a round-robin order, before continuing the acquisition of the next power trace. Subsequently, we prepare the acquired side-channel traces for the training and inference steps. Similar to previous work [67, 71, 72], we partition the final dataset into a training set (for training the instruction classifier) and a test set, for evaluating the instruction classification accuracy and the leakage learned by the models. The following subsections explain the template generation and the training of the side-channel instruction classifiers in greater detail.

5.1 Instruction Template Generation

For our leakage analysis, we create two templating configurations. In the first, denominated as N, the target instruction is surrounded by NOP instructions. We use this set of templates to analyze the instruction-level leakage without additional noise from the surrounding instructions. In the second configuration, denominated by R, we surround the target instruction with a random instruction before and after. We use the R templating configuration to analyze instruction-level leakage in the presence of other instructions, which represents a more realistic leakage scenario: in practice, the target instruction will be surrounded by a pair of random instructions instead of NOPs.

Algorithm 1 Instruction template generation. N_{NOP} stands for the number of NOPs.

```
for All registers x, except x0 do
    Initialize with a randomly chosen 32-bit value
end for
Prepare to execute the target instruction
for i = 1..N_NOP do
    ADDI x0,x0,0                                  ▷ NOPs
end for
if templating == R then
    insert random instruction
end if
Execute the target instruction                    ▷ Target
if templating == R then
    insert random instruction
end if
for i = 1..N_NOP do
    ADDI x0,x0,0                                  ▷ NOPs
end for
End
```

For both templating configurations, we generate 10,000 templates for every instruction from the RV32I ISA, which are listed in Table 3. The process of template generation is detailed in Algorithm 1. The first step is the initialization of x registers with random values. Then, if needed, we insert additional preparation instructions (e.g., to initialize the contents of a memory location for the load instruction). The central and key part of the template contains the target instruction itself: in the case of N templating, similarly to previous work [71, 72], we surround the target instruction with a few NOPs to separate it from the setup phase, while in the case of R templating, we insert a random instruction before and after, making sure the control flow

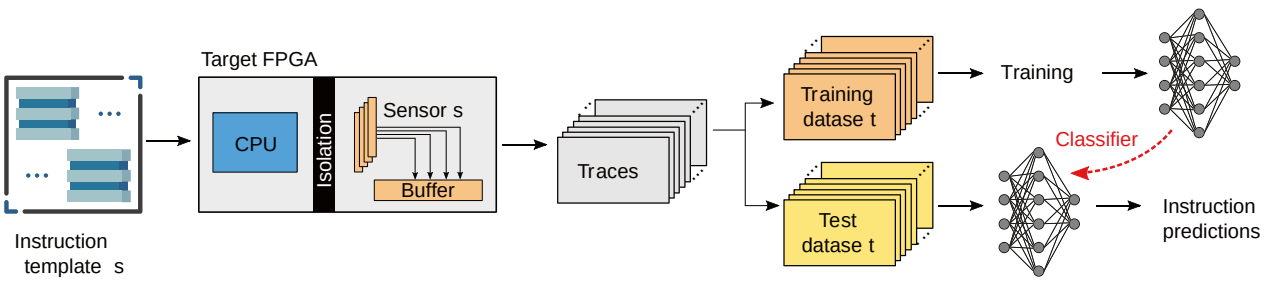


Fig. 6 Side-channel instruction leakage evaluation

Table 3 RV32I base integer instructions for template generation

Category	Instructions
Arithmetic	ADD, ADDI, SUB, LUI, AUIPC
Logical	XOR, XORI, OR, ORI, AND, ANDI
Compare	SLT, SLTI, SLTU, SLTIU
Shifts	SLL, SLLI, SRL, SRLI, SRA, SRAI
Loads	LB, LH, LW, LBU, LHU
Stores	SB, SH, SW
Branches	BEQ, BNE, BLT, BGE, BLTU, BGEU
Jump & Link	JAL, JALR

is not altered. Finally, at the end of the template code, we insert an instruction with an invalid opcode, to trigger a signal to the controller that the code execution is completed (see Fig. 3).

5.2 Instruction Classification Models

Most power side-channel disassemblers in previous work used traditional ML methods and common classification algorithms, e.g., quadratic discriminant analysis (QDA), *k*-nearest neighbors (k-NN), support vector machines (SVM), Gaussian Diffusion Model (GDM) [67, 69, 71–73]. However, the accuracy of these algorithm-driven ML classifiers dramatically depends on the preprocessing for dimensionality reduction and feature extraction. Without suitable preprocessing, the noise in the dataset can significantly affect the classification results. For these reasons, previous research relied on the high sampling rate of the oscilloscope to achieve reasonable accuracy. In this work, considering the limited sampling frequency of the on-chip sensors with respect to the soft-core CPU operating frequency, besides testing how well the ML methods proposed in previous work perform in this scenario, we explore leakage analysis using DL-based classifiers.

First, we treat the side-channel instruction classification as a time-series classification problem, as different instructions have unique patterns spread across the time and amplitude domain. Since we use multiple sensors for classification, we represent the trace of each sensor as a separate input channel. Fig. 7 shows the classification process.

A class of networks naturally suited to processing sequential data is recurrent neural networks (RNNs), specifically long short-term memory (LSTM) models [74]. They have an internal state that can represent context information, and they keep information about past inputs for an amount of time that is not fixed but depends on the weights and the input data. As LSTMs do not perform well when directly extracting features from raw data, they are commonly paired with more complex networks for feature extraction [75, 76], such as convolution neural networks (CNNs). In practice, feature extraction with CNNs can be applied before or after the LSTM model. Moreover, recent work showed that 1D-CNNs consisting of single-dimensional convolutional layers achieved good results in time-series classification [77]. Finally, CNNs structured as residual networks (ResNets) have shown to be very performant in time-series classification, achieving high accuracy across a range of datasets [77]. Therefore, we train and compare the following models: LSTM, a small 1D-CNN, a large 1D-CNN, and the combination of LSTM and 1D-CNN (LSTM followed by 1D-CNN and LSTM preceded by 1D-CNN), a multi-layer perceptron (MLP), and a time-series ResNet [77].

6 Evaluation on Sakura-X

In this section, we provide a detailed instruction-level leakage analysis on Sakura-X. The first step in experimental evaluation is deciding the hypothetical attacker and victim's placement. Given the power delivery network imperfections and knowing that side-channel leakage picked up by the sensors varies with both the absolute and the relative positions of the victim and the attacker [66], we opt to assign the victim to an arbitrary FPGA region and vary the sensor placement.

Figures 8, 9, and 10 zoom in on the FPGA floorplan containing three different placements of the target CPU and the sensors. In the floorplan in Fig. 8, we place the sensors inside the region occupied by the target CPU, in the top-left clock region of the Kintex-7 FPGA (X0Y4). Even though this floorplan does not conform to the standard shared FPGA threat model—where the FPGA regions assigned to

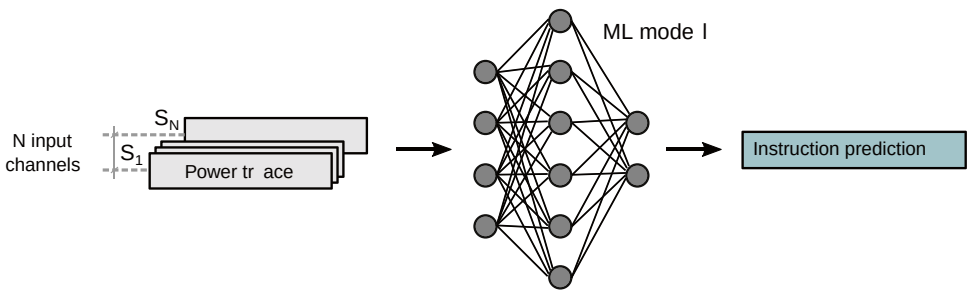


Fig. 7 Classification process. The power trace of each sensor (S_1 to S_N) is used as one of N input channels. The input is then forwarded to the model, and the instruction prediction is collected for accuracy evaluation

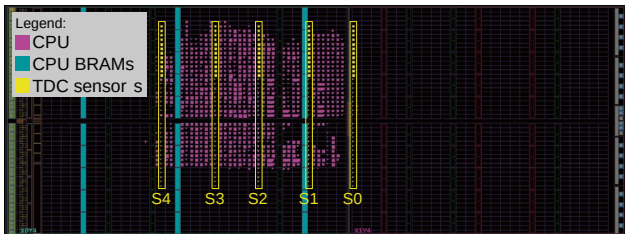


Fig. 8 Sensor delay lines (in yellow) and CPU (in purple) in Exp-IN

the tenants do not overlap—we use it as a worst-case leakage scenario for the evaluator (best-case scenario for the attacker). In the floorplan in Fig. 9, we place our five sensors to the right of the target—in the top-right clock region (X1Y4)—in the space between the CPU and the edge of the FPGA, simulating an attacker that spreads out the available sensors across their entire FPGA region. In the floorplan of Fig. 10, we move the target CPU one clock region down (X0Y3), further away from the sensors. In the remainder of this section, we will refer to the described floorplans as Exp-IN, EXP-OUT1, and Exp-OUT2, respectively.

Using the leakage evaluation setup and following the instruction classification method described in Sects. 4 and 5, we create 10,000 templates per instruction (for both N and R template types) and collect the corresponding power side-channel traces, creating four datasets: Exp-IN-N, Exp-OUT1-N, Exp-OUT1-R, and Exp-OUT2-N. We use the Exp-IN-N dataset to evaluate the worst-case leakage (i.e., with physical separation between the victim and the adversary violated and no noise of surrounding instructions). Exp-OUT1-N and Exp-OUT2-N are collected in addition to Exp-IN-N to evaluate the impact of CPU and sensor placement on the instruction-level leakage and model accuracy. Finally, we use Exp-OUT1-R to evaluate the most realistic scenario, where the templates contain the noise of the surrounding instructions. With these four datasets, we cover the three main goals of our instruction-level leakage evaluation: worst-case for the evaluator (Exp-IN-N), the impact

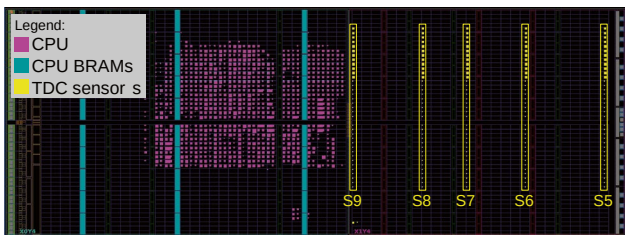


Fig. 9 Sensor delay lines (in yellow) and CPU (in purple) in Exp-OUT1

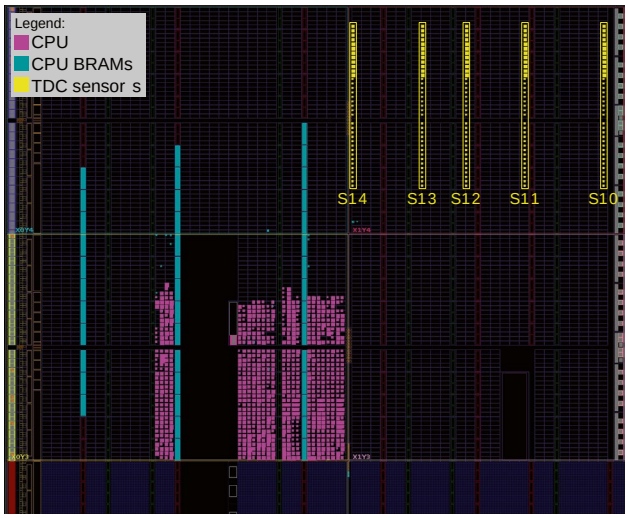


Fig. 10 Sensor delay lines (in yellow) and CPU (in purple) in Exp-OUT2

of CPU and sensor placement on the accuracy (Exp-IN-N, Exp-OUT1-N, Exp-OUT2-N), and a realistic case for the attacker (Exp-OUT1-R).

We set the sensor trace length to samples (i.e., 60 consecutive readings of the TDC output register), to guarantee that all the execution cycles of the instructions in Table 3 are captured. To ensure we are capturing the correct instruction execution, we align the start of all instructions to the same sample in the traces (fourth sample): we center the traces around the correct instruction using the recorded CPU opcode.

In our experimental evaluation, we first visually analyze the recorded power traces for multiple sensor placements. We show that instructions of different types show limited visual leakage patterns, while the instructions of the same type do not display any differences. To determine the limits of the instruction-level leakage, we train a range of DL models on the four datasets and show how the accuracy changes depending on the placement and template type. We also use ML techniques to further evaluate the inter- and intra-type instruction leakage and show that most of the classification confusion comes from two or three instructions with similar leakage. We show that preprocessing techniques and ML approaches used in previous work are outperformed by DL techniques. Finally, we evaluate the limits of the instruction-level leakage by investigating the impact of the number of sensors, averaging, and the dataset size on the accuracy.

6.1 Visual Analysis of Sensor Traces

Before analyzing leakage using the DL-based classification methodology described in Sect. 5.2, we first visually analyze

the recorded sensor traces. In our first experiment, we investigate how sensor placement impacts the waveforms and the leakage in the traces. Figure 11 shows the average trace of all templates of arithmetic and logical instructions across all 15 sensors (five in each of the three floorplans) for the Exp-IN-N, Exp-OUT1-N, and Exp-OUT2-N datasets. We can observe that the sensor placement significantly impacts the shape of the traces, including the peak-to-peak ratio: S10, the furthest from the CPU, has a peak-to-peak ratio of less than one, while S3 has a peak-to-peak ratio of almost six. For some sensors (e.g., S4, S8, S7, S12, and S13), we observe peaks every four samples, perfectly synchronized with the CPU clock. For some other sensors (e.g., S1, S2, and S4), we observe a different pattern: slight dips every 20 sensor samples (around samples 12, 32, and 52), corresponding to five CPU clock cycles, i.e., to the fetch of the next instruction. This experiment already shows the benefit of having multiple sensors for increasing the power side-channel leakage.

In our next experiment, we visually inspect the inter-type instruction leakage, i.e., how different instruction types impact the shape of the recorded side-channel traces.

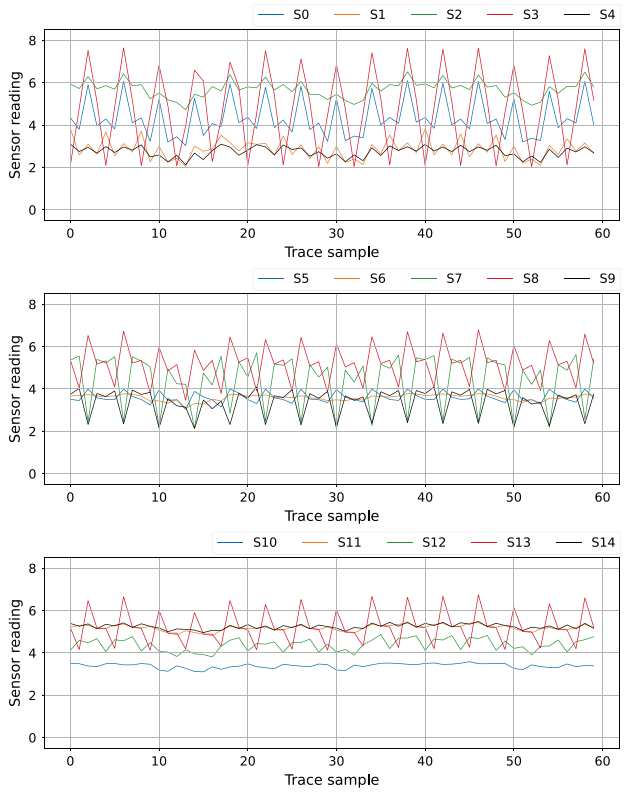


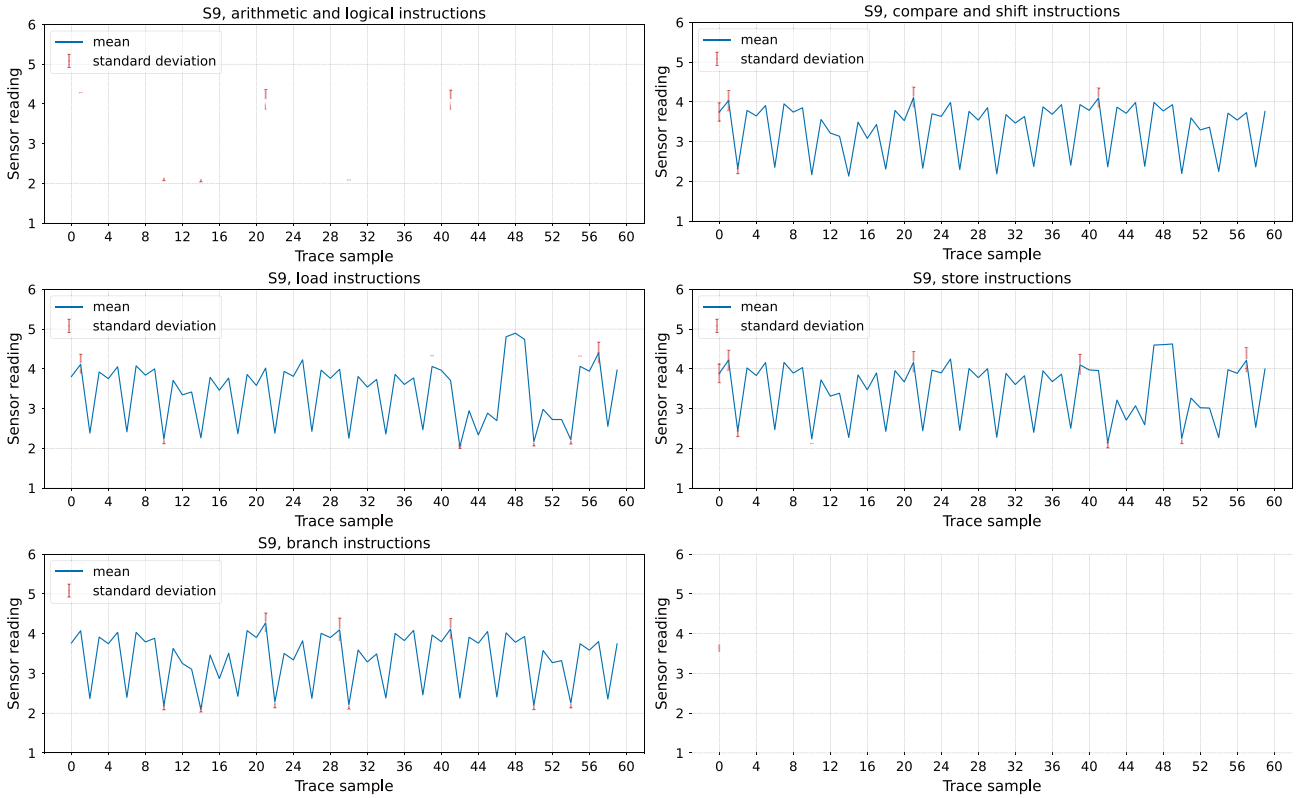
Fig. 11 Average sensor traces for the arithmetic and logical instructions in Table 3

Figure 12 shows the average traces of sensor S9 (Exp-OUT1-N) for the six instruction groups in Table 3. We chose sensor S9, as the plots in Fig. 12 were most visually distinguishable for this particular sensor and it represents the worst-case scenario for an evaluator. The peak in sample 48 makes the load and store instructions clearly distinguishable from other groups. Branches and jumps also contain a distinguishable peak centered around sample 28, surrounded by dips on both sides. This experiment shows that, after significant averaging, some distinct visual traits can be attributed to specific instruction groups. However, not all instruction groups can be identified visually. For example, just like loads and stores, jumps and branches have very similar power consumption traces, and it is difficult to tell the exact instruction type from visual analysis alone.

As the final visual experiment, we compare the side-channel traces of several instructions of the same instruction type. Figure 13 shows average sensor S3 traces for eight instructions. We chose S3 because it is in the heart of the soft-core CPU (Fig. 8), and it shows, when averaged, the biggest visual differences between instructions of the same type. On the left, we overlap the average traces for OR, AND, ORI, and ANDI. The differences, located between samples 10 and 20, are difficult to notice even with averaging across all templates, as all four instructions use the same datapath. On the right, we overlap the average traces of four branch instructions: BEQ, BNE, BLT, and BGE. We practically see no difference between these instructions and cannot distinguish them visually. Therefore, even though the visual classification of instructions is possible for some victims (e.g., sizeable ML-based accelerators [28]), soft-core CPUs require more advanced methods for instruction-level leakage analysis.

6.2 Deep Learning-Based Instruction Leakage Evaluation

After showing that visual analysis is insufficient to identify CPU instructions executing on remote FPGAs, we deploy advanced DL techniques. We obtain our four datasets by collecting all the sensor traces for each instruction in Table 3, as described in Sect. 5.2. Each data point, corresponding to one instruction template, is represented as a matrix with five rows, where each row, i.e., the input channel, is the trace of one of the five sensors. Using the newly created dataset, we first train our deep learning models from Sect. 5.2 using 10-fold validation and compare the resulting accuracy. Then, we compare the results of our DL models with the classical ML methods previously proposed for side-channel disassembly attacks, and we evaluate if frequency-based preprocessing methods,



shown promising in previous work [72], have any impact on the extracted leakage. Furthermore, we evaluate how the number of sensors used in the attack impacts the final accuracy. Finally, we evaluate how the amount of averaging or a smaller dataset size can impact the leakage, i.e., the best model accuracy.

To train our deep learning models, we set the number of epochs and the batch size to 100 and 64, respectively. We use the Adam optimizer with an initial learning rate of 0.0001 and the loss to monitor and adjust the learning rate. Table 4 summarizes the model details. To facilitate reproducibility, we choose deep learning models with standardized parameters and openly available implementations [77, 78].

Table 5 lists the average test accuracy obtained with the four datasets, with the highest accuracy in bold. We can observe that overall, ResNet and 1D-CNN2, the two most complex DL models, achieve the highest accuracy for all datasets. Results in Table 5 also show that models without convolutional layers do not manage to extract leakage well and result in low classification accuracy. Moreover, the accuracy drops as the sensors are placed further away from the target CPU. For example, for the best model (ResNet), Exp-OUT1-N has a 16.07% lower accuracy than Exp-IN-N, and Exp-OUT2-N has a 22.5% lower accuracy than Exp-OUT1-N. Therefore, an evaluator testing local CPU leakage with sensors placed inside the CPU will achieve