Towards Explainable Side-Channel Leakage: Unveiling the Secrets of Microarchitecture

Ischa Stork[1], Vipul Arora[1], Łukasz Chmielewski[2], Ileana Buhan[3]

[1]Riscure B.V., The Netherlands, emails: {stork,arora}@riscure.com.
[2]Masaryk University, Czech Republic, email: chmiel@fi.muni.cz.
[3]Radboud University, The Netherlands, email: ileana.buhan@ru.nl.

Abstract

We explore the use of microbenchmarks, small assembly code snippets, to detect microarchitectural side-channel leakage in CPU implementations. Specifically, we investigate the effectiveness of microbenchmarks in diagnosing the predisposition to side-channel leaks in two commonly used RISC-V cores: Picorv32 and Ibex. We propose a new framework that involves diagnosing side-channel leaks, identifying leakage points, and constructing leakage profiles to understand the underlying causes. We apply our framework to several realistic case studies that test our framework for explaining side-channel leaks and showcase the subtle interaction of data via order-reducing leaks.

Keywords: Side-Channel Analysis, RiscV, Microarchitecture, Microbenchmarks

1 Introduction

A cryptographic implementation *leaks* information through side channels [1] when its hardware trace depends on input data [2]. Masking, a popular countermeasure to prevent side-channel attacks, is challenging to implement correctly. The *independent leakage assumption* [3] can simplify the analysis of masked cryptographic implementations however, this assumption has important limitations. For example, in a pipelined architecture, *hidden registers*[1] between the different pipeline stages will store the results of the execution of each stage so that the next stage can read them. Sensitive data may interact through hidden registers, creating unexpected leaks [4, 5].

[1]Hidden registers cannot be referenced externally but are implicitly used while processing data, causing unexpected interactions.

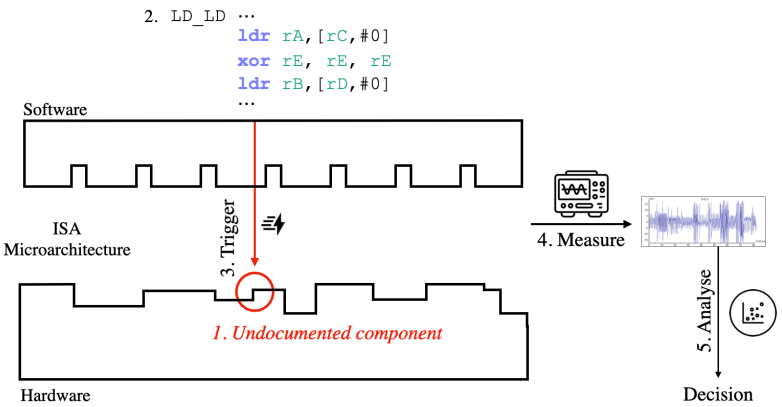

Figure 1: *Microbenchmarks test the presence of undocumented components (1). A code snippet that tests the presence of the component is created (2). The microbenchmark is executed, triggering (3) a specific part of the hardware while we measure side-channel traces (4). We analyze the collected traces (5) for evidence regarding the presence of the undocumented component.*

Leaks that induce a gap between formal security proofs and practical resilience are called *order-reducing* [6]. An important source of order-reducing leaks is the *microarchitecture* [7, 8]. Performance optimizations such as pipelining, out-of-order execution, and speculative execution, necessary for modern computers' performance demands, are part of the microarchitecture. As optimization choices significantly affect the final product's performance, the microarchitecture is considered a trade secret, and its specifications are typically unavailable in the public domain.

There are two approaches to capturing side-channel leakage effects resulting from the microarchitecture. The first is experimental and involves using *leakage simulation*, which generates side-channel measurements from a sequence of instructions using a leakage model. Leakage simulation is device-specific, meaning that traces produced by simulators match the traces obtained from the device using an oscilloscope. The accuracy of the match between the generated and measured traces depends on the quality of the leakage model. However, developing a realistic leakage model that accurately represents the interaction of intermediate values caused by hidden storage elements requires significant effort. Open-source leakage simulators are available for ATMega and ARM architectures [9].

The second approach is formal and uses the d-probing security model [10], where an attacker is assumed to have access to a set of probes that allow observing the device's internal states during operation. Traditionally used to verify masked hardware implementations, recent developments describe new probes, which model microarchitecture effects and allow the verification of masked software implementations. An open-source tool that models an ARM architecture [11] allows a developer to make formal statements about the security of a cryptographic implementation. By choosing different types of probes, the developer can enable the verification of different microarchitecture

effects. While effective, the two approaches above apply to a subset of ARM architectures (ARM Cortex M0/M3). For microcontrollers with a RISC-V architecture, no solution allows the (partial) verification of a cryptographic implementation.

Microbenchmarks [8] are a relatively new method for identifying order-reducing leaks, such as subtle transition leaks. These benchmarks consist of small instructions to detect how operands interact during processing. Microbenchmarks are general because the code can be applied to any device; however, the output is device-specific, as are the hardware components that cause order-reducing leaks. While the authors of [8] focus on documenting microarchitecture effects, they do not delve into exploiting or developing countermeasures based on leakage effects.

Our Contribution. We demonstrate the practical application of microbenchmarks in characterizing side-channel leaks. Specifically, we propose a real-world use for microbenchmarks, showcasing their effectiveness in identifying and understanding potential vulnerabilities that can be exploited. We propose *Diagnose-Identify-Explain*, a framework to explain the origin of order-reducing leaks. *Diagnosis* is the most effort-intensive step, and we use microbenchmarks to examine the source of order-reducing leaks. In the *identify* step, we examine the predisposition to side-channel leaks for two open-source RISC-V cores, consolidating meaningful findings from the literature [6, 8, 12]. In the final step, *explain*, we present a quantitative and qualitative analysis of several realistic case studies that test our framework for explaining side-channel leaks and showcase the subtle interaction of data via order-reducing leaks.

Results reproducibility. During this project, we used open-source cores and algorithms. The dataset collected during this project is large (700GB), which unfortunately limits its open-source release. We will publish all simulation results and the code we created for labeling the traces upon acceptance.

2 Related work

We discuss state-of-the-art work on pre and post-silicon verification as we perform side-channel attacks on cryptographic implementations using traces collected from two RISC-V cores. We use the design information to trace signals and label execution traces (diagnosis and identification).

Pre-silicon hardening of masked circuits. Several approaches have been developed to eliminate leaks during the pre-silicon phase. These approaches can be roughly divided based on which device layer or development phase they aim to harden. De Mulder et al. [13] proposed a solution to protect an AES implementation against side channel leakage related to memory accesses on a RISC-V core. Gigerl et al. [12] introduced COCO, a tool that can detect gate-level leakage by simulating execution with Verilator. They annotate the registers and memory that hold secret data and trace their flow through the circuit to find possible sources of leakage. Arsath et al. [14] developed a framework that analyzes the RTL description of a processor and reports the information leakage in each of the processor modules. He et al. [15] estimate the power profile of a hardware design using functional simulation at the RTL level. Gao et al. [16] designed and implemented an ISE (Instruction Set Extension) called FENL that localizes and reduces microarchitectural leakage. The ISE acts as a leakage fence

that prevents interaction between instructions. A similar approach is taken by Pham et al. [17], which combines a diversified ISE with hardware diversification through a co-processor to achieve leakage mitigation. Bloem et al. [6] extended the concept of hardware-software contracts to power-side channels and formally verified a wide range of instructions for implementing cryptographic algorithms for the RISC-V Ibex core. ACA [18] uses a gate-level model for a target design, typically available after logic synthesis and a side-channel leakage model. Kiaei and Schaumont proposed Root Canal [19], a framework that helps a designer with white-box access to the embedded CPU system uncover the origin of a side-channel leak. Root Canal can eliminate side-channel leaks before tape-out.

Post-silicon verification of side-channel leaks. Papagiannopoulos et al. [7] were among the first to discuss order-reducing leaks which breached the independent leakage assumption. De Meyer et al. [20] continue their work and show the existence of other microarchitecture leakage effects. McCann et al. [21] introduced ELMO, an ARM Cortex M0 leakage simulator. ELMO models power consumption as a linear combination of values and transitions. Shelton et al. [22] improve the leakage model in ELMO by capturing multiple cycles' interactions. They also define small code sequences that can identify effects that happen many cycles apart. Shelton et al. [23] further improve upon previous work by detecting high-order leaks. Arora et al. [24] analyze the influence of the microarchitecture on side-channel leakage and confirm that differences in the microarchitecture cause different leakages. Gao et al. [4] reverse engineer the microarchitectural implementation of a commercial ARM Cortex-M3 microprocessor, and Oleksenko et al. [2] propose a testing framework, Revisor, to detect microarchitectural information leakage in commercial black-box CPUs. Revisor tests compliance with violations of speculative contracts. Marshall et al. [8] propose MIRACLE, a comprehensive and generic set of microbenchmarks, but does not examine their application in the context of side-channel attacks. Grandmaison et al. [25] use the RTL of an Arm Cortex M3 to verify a cryptographic implementation with a microarchitectural model formally. Gaspoz et al. [26] give necessary conditions for masked implementation to be first-order secure in scalar microarchitectures. Buhan et al. [9] provides a comprehensive description of tools that can be used to automate the creation of side-channel resilient implementations.

3 Preliminaries

RISC-V is an open-source ISA established on the principles of RISC design. RISC-V is a `LOAD/STORE` architecture, which means that there are specific instructions to interface with memory, making it impossible to directly access memory when, for example, executing an `add` instruction. As a result, the design of RISC-V is highly modular: a CPU designer can pick one of the various base ISAs to implement, depending on requirements and hardware support. Optional extensions can be added to the CPU design for auxiliary functionality. For example, the **M**ultiplication extension provides integer multiplication and division through the `MUL` and `DIV` instructions, respectively. Similarly, extensions exist that enable compressed instructions (C), bit operations (B), and vector operations (V). The open nature of the architecture has led to many

contributions, resulting in a comprehensive platform that includes several publicly available RISC-V core implementations and a well-maintained toolchain[2].

In this work, we evaluate two open-source designs regarding side-channel leakage. All programs are compiled and linked using the aforementioned toolchain. We use the RV32I base of RISC-V, which has 32 fast storage registers. Each register can hold 32 bits of information. Following the RISC-V calling conventions, eight of these registers (referred to as `a0 - a7`) can be used to pass arguments to functions, where `a0` and `a1` are also used to pass back return values. `x0` is a special register hardwired to zero and is prevalent in many pseudo-instructions. The binary encoding of an operation is referred to as its *instruction format*. RISC-V has six core instruction formats, which determine how instruction processes data. Each format accommodates certain types of instructions: register-register ALU instructions `(R)`, ALU and load immediate instructions `(I)`, store, comparison, and branch instructions `(S/B)`, and finally, jump and link instructions `(U/J)`.

Anatomy of a microbenchmark Microbenchmarks use side-channel information to determine undocumented features of the microarchitecture, as shown in Figure 1. The first step is to make a hypothesis about the microarchitecture implementation. For example, consecutive memory accesses interact via a hidden register. This effect may occur one or many cycles apart as memory access instructions may be separated by non-memory related instructions (e.g., arithmetic instructions). Our working hypothesis for this example is that there is *a hidden state in the memory access path* for `ldr` instructions.

```
1  ldr rA, [rC, #0]
2  xor rE, rE, rE
3  ldr rB, [rD, #0]
```

Listing 1: `LD_LD`: *triggers hamming distance leakage of the values loaded from memory locations rC and rD.*

Next, we create (or use an existing) microbenchmark to test the existence of a hidden register. Listing 1 loads two intermediate values in the memory from the addresses specified in registers `rC` and `rD`. Since `rC` and `rD` refer to different registers, we do not expect interaction between the values loaded from the two registers. However, the loaded values will interact if a state (e.g., register) exists in the memory path. We repeat the execution of the microbenchmark, varying the values loaded from the two registers while measuring the power consumption. If there is an interaction, we will observe evidence of this interaction when analyzing the power traces. For example, we can look at the correlation between the values of these two registers and the measured power traces. Marshall et al. [8] created a set of microbenchmarks that target six microarchitecture optimization features related to an embedded CPU's pipeline and memory subsystems. An appealing feature of the proposed microbenchmarks is that these are generic, meaning that can they can be run on different architectures. We list the existing microbenchmarks in Table 1.

Key rank estimate is a commonly used metric in SCA for assessing the performance of an attack. It is performed in a known key scenario and returns the rank of the

[2] https://github.com/riscv-collab/riscv-gnu-toolchain

correct key candidate in the sorted score vector of all key candidates. The key rank estimate is related to the success rate curve [27], which shows the evolution of the correct key candidate as more traces are added. There are two differences compared to the success rate: first, key ranking is performed on a fixed set of traces, whereas the success rate is performed on a variable set of traces to capture the evolution of the correct key candidate; second, key ranking can be performed for all samples in the trace, whereas the success rate is typically shown for one sample. The result of the key rank estimate is affected by the number of traces used for analysis. If leaks are present, the correct key rank converges towards the first position as more traces are added.

4 Step 1: Diagnosis of predisposition to side-channel leaks

All experiments are run on two RISC-V cores: Picorv32[3] and Ibex[4]. We chose them because they are relatively small, uncomplicated, well-documented, and support the same RISC-V ABI. Additionally, both cores have already played a significant role in academic research, meaning any (interesting) findings can be related to and corroborated. In the following two sections, we discuss the high-level design for both cores and the integration into their respective SoCs.

Picorv32 is a multicycle RISC-V core that implements the `RV32(I/E)` bases and supports the **M**ultiplication/division and **C**ompressed instructions extensions. The core is size-optimized but is still configurable by setting certain Verilog parameters that alter the core's behavior. The core is synthesized with the barrel shifter and a dual-port register bank to shorten the computation time and, thereby, the number of samples in the power traces. The core is integrated into an SoC containing the bare minimum peripherals: 1KB of RAM, a UART controller to handle communication with the workstation, and a GPIO controller used to toggle the trigger. This simple setup is chosen because it minimizes peripheral noise and is convenient for simulation.

Ibex is a two-stage pipelined RISC-V core written in SystemVerilog. Similarly to Picorv32, this core implements the RV32I and RV32E base and has multiplication and compressed instruction extensions. The core supports several parameters that can be used to configure the core. However, we did not enable any options as they complicate the microarchitecture. The core is integrated into a simple SoC with all essential components, such as SRAM (16KB), a UART controller, and a GPIO controller. The SoC also has a timer and debug support.

Measurement setup. All experiments are carried out on a *Arty A7-35T* [5] development board powered by a Xilinx Artix-7 FPGA. This board is convenient to work with, as it comes with an FTDI USB - UART bridge that can be used for serial communication. It also serves as a controller for the included USB-JTAG circuitry, allowing the FPGA to be programmed over USB. For all experiments, the sample rate is set to 500 MHz. *Arty* also contains circuitry to measure the power consumption of the FPGA's core. However, this circuitry is incompatible with the available acquisition

[3] https://github.com/YosysHQ/picorv32
[4] https://github.com/lowRISC/ibex
[5] https://digilent.com/reference/programmable-logic/arty-a7/start

equipment and tooling, so we made a new power cut to measure the FPGA's power consumption. In addition to this modification, several decoupling capacitors along the VCCINT power rail are removed to improve the quality of the power signal.

Microbenchmarks results. We apply the microbenchmarks to both cores using the above-mentioned measurement setup. Although Picorv32 has been profiled in [8], we repeat the experiments for three reasons. First, this core is a baseline to verify our setup. Second, the explanation for some of the leaks observed in the original article is brief, and we aim for a more in-depth description. Finally, configuration differences between our core and the version in [8] could cause different leakage effects, and we are interested in learning to what extent the microbenchmarks are reusable. Table 1 summarizes the results obtained for these experiments. Similarly to [8], we use correlation to demonstrate the existence of leakage.

We observe a marked difference in the signal-to-noise ratio between the trace sets collected for the two cores. This means that for the Picorv32, we used a set of 250k traces for each experiment, while for the Ibex core, we observed a clear correlation at 50k traces. To diagnose the side-channel leakage of this core, we performed a total of 26 benchmarks, as shown in Table 1, consisting of 76 experiments. An *experiment* is a small variation in the parameters of the microbenchmark code. For example, for the `BUS-WIDTH-LD-BYTE`, we experiment with 8 offsets. We list all microbenchmarks (code and side channel results) that show leaks in Appendix C.

4.1 Discussion of the results for the Picorv32

Picorv32 core is part of the MIRACLE experimental setup [8]. Picorv32 was the only device (among the fourteen tested in [8]) that did not show positive results for the *hidden state* effect. Picorv32 suffers from leaks caused by a mismatch between *bus width* and data size. Inspecting the Picorv32 source code, it becomes clear that the observed leakage is indeed expected. When performing a memory look-ahead, the least significant two bits are set to zero, leading to word-size memory accesses. This can be seen in Listing 2, which defines the memory look-ahead address signal. Additionally, the address on the memory bus is dual-purpose; there is no strict separation between read and write addresses. This means that a store operation also inadvertently leads to a read operation. This explains why store operations also show leakage.

```
1   assign mem_la_addr = (mem_do_prefetch || mem_do_rinst) ? {next_pc[31:2] +
        mem_la_firstword_xfer, 2'b00} : {reg_op1[31:2], 2'b00};
```

Listing 2: *The look-ahead address signal,* reg_op1 *is the address to be read from in memory.*

No interaction between sequentially loaded values was observed (`SEQ-LD`). On the contrary, sequential store operations show clear leakage (`SEQ-ST`). We believe this leakage is caused by the multiplexer when selecting the source register.

Positive leaks for the *register overwrite* microbenchmark might seem counterintuitive given the absence of a pipeline. However, in the absence of a pipeline, the instruction operands have to be temporarily saved, and Picorv32 has two internal registers `reg_op1` and `reg_op2` for this purpose. These are not pipeline registers, but leakage still occurs when values in these registers are overwritten.

The core shows positive leakage for the control-flow microbenchmarks. In the `BRANCH-POST` and `JUMP-POST`, there is an immediate, unconditional branch or jump

Type	Microbenchmark		Picorv32	Ibex
Memory	Hidden state	LD-LD	✗	✗
		LD-ST	✗	✗
		ST-LD	✗	✗
		ST-ST1	✗	✗
		ST-ST2	✗	✗
		ST-ST3	✗	✗
	Bus-width	LD-BYTE	✓(0-3)	✗
		LD-HALF	✓(2-3)	✗
		ST-BYTE	✓(0-3)	✗
		ST-HALF	✓(2-3)	✗
	Sequential access	SEQ-LD	✗	✗
		SEQ-ST	✓	✓
Pipeline	Register overwrites	EOR-EOR	✓	✓
		EOR-ADD	✓	✓
		EOR-LSL	✓	✓
		EOR-ROR	not supported	not supported
		EOR-LDR	✓	✓
		EOR-STR	✓	✓
		NOP-EOR	not performed	not performed
	Speculative execution	BRANCH-FWD	✗	✓
		BRANCH-BWD	✗	✓
		JUMP-FWD	✗	✓
		JUMP-BWD	✗	✓
		LOOP-0	✗	✓
	Control-flow	BRANCH-PRE	✓	✓
		BRANCH-POST	✓	✓
		JUMP-PRE	✓	✓
		JUMP-POST	✓	✓

Table 1: *Overview of the diagnosis. The ✗ indicates no leakage; the ✓indicates leakage; the ✓+ numbers indicate that the experiment succeeded for the specified parameter.*

to the second XOR, thus skipping the XOR between operands A and B entirely. Except one experiment, our results align with those reported in [8]. Despite the jump instruction [8], report correlation between operands A and B for `BRANCH-POST` and `JUMP-POST` microbenchmarks, we do not see this leakage in our experiments.

4.2 Discussion of the results for the Ibex core

According to the documentation, most instructions in the Ibex core are executed independently of their input operand [28], which should offer a certain degree of protection against side-channel attacks. The Ibex core has a 2-stage pipeline and, surprisingly, does not exhibit leaks caused by the pipeline registers. Bloem et al. [6] also note the

lack of transitional leakage effects in memory stores separated by non-memory instructions. Gigerl et al. [12] mention that a hidden state in the LSU exists and is a source of leakage. However, this hidden state only handles word accesses on non-word-aligned addresses—no microbenchmarks tests for this leakage pattern.

We observe transitive leakage for sequential store instructions and assume that this leakage effect is caused by switching multiplexers. Gigerl et al. [12], report that the read addresses from the register file are not glitch-free since they come from combinatorial logic. We see leakage due to operand interaction, but only in specific instruction combinations and operands. For example, `EOR_EOR` and `EOR_ADD` leak the second instruction operand but not the first operand. The `EOR_SRLI` benchmark transitional leakage is observed. It is unclear what the cause behind this unusual operand interaction is.

Branch instructions can introduce operand interaction between an instruction that is never executed (i.e., an instruction that comes after a branch is taken) and the instruction that sequentially follows the branch. The control-flow experiments show varying interactions. `BRANCH_PRE` and `JUMP_PRE` are consistent and do not show any operand interaction, but the correlation is seen with the results of the executed bitwise additions. For `JUMP_POST`, only bitwise addition for the instruction after the jump shows leakage. Considering a jump is executed unconditionally, the CPU will not load operands for an instruction that is not executed. `BRANCH_POST` is the only microbenchmark that shows operand interaction. In this benchmark, the assumption is made that the branch will be taken. The operands of the instruction at the branch address are loaded. When the condition is processed, the operands are already in the pipeline registers and can interact with the subsequent instruction.

5 Step 2: Identifying exploitable leaks

Exploitable leaks are sample points in a measured trace set where the correct key is ranked highest when performing an attack. When constructing our experimental dataset, we identify such exploitable leaks for various implementations of AES, the most well-established symmetric cryptographic primitive now.

We present three case studies. Two of them consider the execution of an unprotected implementation running on the two cores we profiled; here, we chose a small and portable implementation —Tiny AES in C[6]. The third case study targets a Boolean masked AES implementation[7] based on Tiny AES, documented in [29](page 228-231). Since this implementation admittedly contains no protections against glitching, we discuss two interesting leakage effects on Ibex. For the first two use cases, we perform a first-order univariate Differential Power Analysis (DPA) [30] targeting the first round SubBytes operation assuming that it leaks the Hamming weight model[8]. Every SubBytes operation consists of multiple S-box substitutions, each using a different part of the state and key.

[6] https://github.com/kokke/tiny-AES-c
[7] https://github.com/CENSUS/masked-aes-c
[8] To be precise, we use a variant of DPA, called Correlation Power Analysis [31], but for simplicity, we use the customary DPA term. Initially, we also used Test Vector Leakage Assessment (TVLA) [32] to see whether the implementation leaks; however, we decided to employ DPA for the sake of precision since TVLA is known to result in false positives.

For convenience, we define SubBytes SB^K as the K^{th} iteration of the first SubBytes operation. To reliably detect exploitable leakage, we divide the samples of the traces into small fragments and attack these fragments separately. This way, we can detect which samples leak. In all our tests, the fragments are sequential and do not overlap. Depending on the experiment, they contain different samples, such that the total number of fragments is 400; due to different core frequencies, the fragment size was 17 for Ibex and 25 for Picorv32.

The correct key was recovered with 500k traces for Picorv32 for some trace fragments (note that different key bytes leak at different points in time). On the other hand, recovering the key on the Ibex required only 100k traces. We attribute the difference in the number of traces required to perform a successful attack to the Picorv32 core being area-optimized.

In our third case study, AES is protected with Boolean masking. We target the re-masking step between ShiftRows and MixColumns in the first round. The old mask is removed and replaced with a new one during the re-masking. The assembly code for this operation is shown in Listing 7. We leave evaluation using higher-order and multivariate analysis as future work.

5.1 Results of the first-order DPA attack

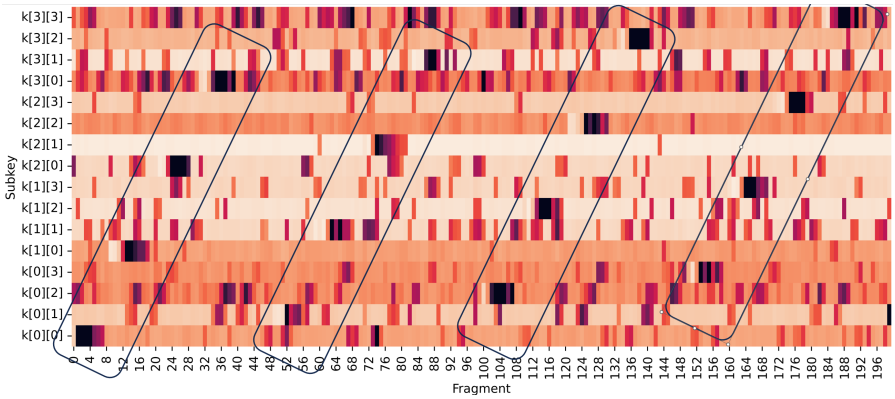

Figure 2: *Key ranking plot of a first-order attack on the first SubBytes operation on Ibex core for the unmasked AES implementation. Dark color shows high leakage.*

Figure 2 shows the attack results on the unprotected implementation on Ibex core; for each attacked fragment, we show the rank of the correct subkey using a logarithmic heatmap. Black indicates recovery of the correct subkey and white indicates the opposite: the correct key is ranked low (so there is no leak). Note that a staircase-like leakage pattern emerges, which makes sense since the key is fitted into a two-dimensional state, after which different key parts are used sequentially.

For Picorv32, the plot is very similar. Namely, a similar leakage is found, as shown in Figure 3.

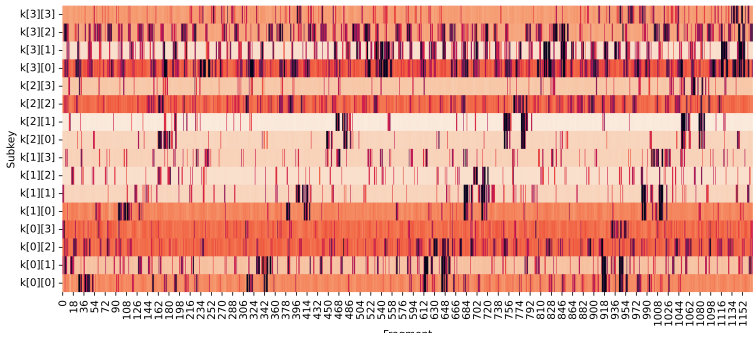

Figure 3: *Key ranking plot of a first-order attack on the first SubBytes operation on Picorv32 core for the unmasked AES implementation. Dark color shows high leakage.*

5.2 Results of the first-order DPA attack

For both cores, we noticed several key bytes leak at the exact moment, even if they are not accessed simultaneously. We investigate such aspects in the remainder of this paper. Upon closer inspection of the last four iterations of SubBytes, we observe interesting leakage patterns.

During the last iteration of SubBytes SB^{15}, when the last subkey (`k[3][3]`) is used for a Sbox lookup, leakage also occurs for subkeys `k[3][2]`, `k[3][1]` and `k[3][0]`. This suggests that not only the expected subkey `k[3][3]` is used, but also the other subkeys are somehow involved

We also observed slightly less pronounced additional leakage patterns for the Ibex core. We verified that these extra leakage points could be used in a practical attack to recover the key. Similarly to the previous experiment, we succeeded with 500k and 75k traces for Picorv32 and Ibex, respectively.

Subsequently, we ran the first-order attack on the masked implementation targeting the re-masking step after the first ShiftRows operations; we did not notice first-order leakage during other operations.

Similarly to the unprotected implementations, we found a staircase-like leakage pattern, though leakage was only observed for the first two rows of the AES state. Due to space constraints, we do not present the heatmap plot for this case in this version of the paper.

5.3 Identifying the source of the exploitable leakage

To understand which instructions are causing leaks, we match the samples in a power trace with the instruction corresponding to the respective sample. For this purpose, we simulate the cores and the cryptographic computations running on these cores and generate an execution trace. This execution trace can then be aligned with the power trace to obtain instruction-labeled power traces. During the measurement campaign, the core tracing was disabled to not influence the power traces. We enabled core tracing when generating the execution trace.

Simulation. The Picorv32 core can generate execution trace by setting the Verilog parameter `ENABLE_TRACE`=1. The generated trace is not cycle-accurate, i.e., it is impossible to determine which samples belong to which instructions precisely. To overcome this, we adjusted the trace generation mechanism to include cycle information using the clock cycle counter already integrated into the core. We use Icarus Verilog [9] for this simulation. To ensure that the simulation matches FPGA execution, the same SoC that is flashed on the FPGA is instantiated by a testbench. This testbench supplies the required input/output signals, like the clock signal, and the UART Tx/Rx lines.

We conveniently match the simulation's clock frequency to the supplied clock on the FPGA. We use a similar approach for the Ibex core. However, we used Verilator[10] since support for this simulation tool is well-documented. Conveniently, Ibex a has a separate module called `ibex_tracing` which can generate a clock-cycle accurate execution trace. Matching instruction labels to power trace samples are more cumbersome in Ibex since the core has a two-stage pipeline. This makes it more difficult to pinpoint leakage to a specific instruction. This is less problematic for microbenchmarks, as the instructions of interest are isolated (e.g., by surrounding them with `nop` instructions).
Attaching instructions to traces. The firmware running on the simulated cores is slightly adjusted: hard-coded input values are replaced by the UART communication that supplies input for the experiments. This should not impact the resulting execution trace since only the operation during power acquisition has to be accurately simulated, and the UART communication is not part of the acquisition. For both cores, we compute the number of samples per cycle. The Picorv32 and the Ibex core are running at 100 MHz and 50 MHz, respectively, and the sample rate is consistently 500 MHz, meaning that for Picorv32, there are ten samples per cycle, whereas for Ibex, there are five samples per cycle.

6 Step 3: Explaining exploitable side-channel leakage

We identified several interesting leakage patterns in the previous section, including the instructions causing these leaks. In this section, we aim to explain the root cause of the identified side-channel leaks using the diagnosis of the leakage profile for each core. To have an accurate view of the state of registers and executed assembly instructions for the cryptographic implementations, we disassembled the .elf files using Ghidra[11].

6.1 CASE STUDY: unprotected AES on Picorv32

Picorv32 is a multi-cycle CPU; most leakage occurs during the last cycles of an instruction, when data is fetched and instructions are executed. Figure 4 shows the correlation between the power traces and several sub-keys during the SB^{13}. Four clear leak patterns can be identified (A, B, C, and D), which we attempt to relate to the earlier constructed leakage profile.

[9] http://iverilog.icarus.com/
[10] https://verilator.org/guide/latest/
[11] https://github.com/NationalSecurityAgency/ghidra

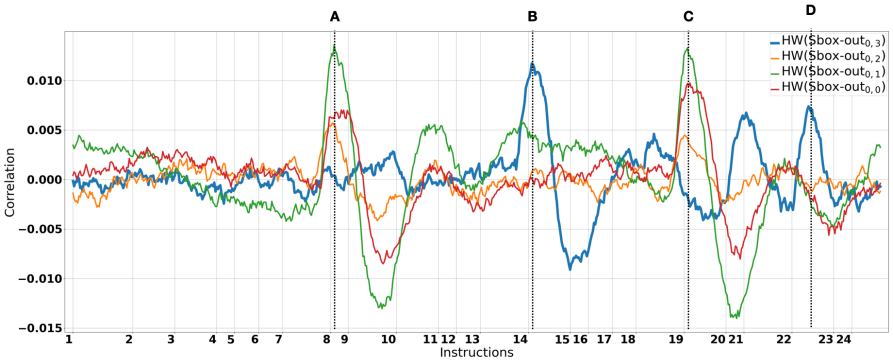

Figure 4: *Leaks observed during SB^{13} for the Picorv32 core. The x-axis corresponds to the line numbers in the Listing 3.*

Load AES state. Instruction `lbu a5,0(a5)` (line 7, Listing 3) loads one state byte S_{ij} from memory; this byte is later used to retrieve a value from the Sbox. While `lbu` instruction specifies that only a single byte should be loaded, correlation can be observed with other state bytes. This happens because of the bus width leakage effect that was part of the leakage profile. The Sbox output values of [0][0], [0][1], [0][2] are computed in earlier iterations of SubBytes and stored in the state in memory. When retrieving state byte [0][3], these bytes are transported on the bus, resulting in leakage of earlier computed Sbox output values.

```
1  lbu a4,-18(s0) ; Load iterator j
2  lbu a5,-17(s0) ; Load iterator i
3  lw a3,-36(s0) ; Load state S
4  slli a4,a4,0x2
5  add a4,a3,a4
6  add a5,a4,a5 ; S_ij
7  lbu a5,0(a5) ; Load S_ij from memory
8  mv a2,a5 ; Store S_ij in register a2
9  lbu a3,-18(s0) ; Load iterator j
10 lbu a5,-17(s0) ; Load iterator i
11 li a4, 0 ; 0 is the base address of RAM
12 add a4,a4,a2 ; Sbox[S_ij]
13 lbu a4,0(a4) ; Load Sbox[S_ij] from memory
14 lw a2,-36(s0) ; Load state S
15 slli a3,a3,0x2
16 add a3,a2,a3
17 add a5,a3,a5
18 sb a4,0(a5) ; S_ij = Sbox[S_ij]
19 lbu a5,-18(s0) ; Load iterator j
20 addi a5,a5,1 ; j = j + 1
21 sb a5,-18(s0) ; Store iterator j
22 lbu a4,-18(s0) ; Load iterator j
23 li a5,3 ; a5 = 3
24 bgeu a5,a4,100a14 ; j>=3?
```

Listing 3: *One iteration of AES SubBytes (for both cores).*

Load Sbox byte. Instruction `lbu a4, 0(a4)` (line 13, Listing 3) loads a value from Sbox, i.e., $Sbox[S_{ij}]$. A correlation peak for this value is expected since load and store operations are known to cause leaks as they directly put a value on the bus. The bus-width micro-benchmark implicitly tests for this kind of leakage. An experiment loads an earlier stored value from memory and uses the correct memory address to load from. Given this result, the bus-width microbenchmarks anticipated this leakage, but not because of a mismatch between the size of retrieved data and the bus width.
Store Sbox byte. Instruction `sb a4, 0(a5)` (line 18, Listing 3) stores the retrieved Sbox output value at the right location in the state. Like loading, the operand we correlate with is directly put on the bus; therefore, we anticipate leaks. The bus-width store micro-benchmark is not applicable as this benchmark attempts to find leakage in situations where a target intermediate is overwritten by zero. We attribute this leak to sensitive data being put on the data bus.
Load loop iterator. Instruction `lbu a4, -18(s0)` (line 22, Listing 3) loads loop iterator j after it was incremented. Such a load would not be expected to leak. However, register `a4` previously hosted the Sbox-out value, which is now overwritten. In the 13th iteration, the value of j equals 1, meaning all bits of the Sbox values are overwritten with zeroes except the least significant bit. MIRACLE does not incorporate any microbenchmarks that can trigger this effect. The hidden-state microbenchmarks come closest but focus on overwriting effects in memory rather than in registers.

6.2 CASE STUDY: unprotected AES on Ibex core

Figure 5 shows the correlation of the Sbox output values during the SB^{13} for the Ibex core. We do not plot other sub-keys since no clear leakage was observed in the key ranking. Instead, we investigate other intermediate values and plot the Hamming distance between plaintext and Sbox-in.

Additionally, the interaction between all intermediates is plotted. Again four leakage points are seen (Figure 5, A,B,C and D). Interestingly, these leaky instructions correspond to the exact instructions as Picorv32. Below, these points of leakage are discussed in more detail. Ibex has a 2-stage pipeline; therefore, some leakage from instruction may occur during the preceding instruction.
Move loaded state. Instruction `mv a2,a5` (line 8, Listing 3) moves state byte S_{ij} from register $a5$ to register $a2$. Leakage is seen with the loaded (Sbox-in) value and plaintext. The explanation for this leakage can not be found in the leakage profile, as the `mv` instruction was not part of any of the microbenchmarks. However, the instruction directly writes the leaked value to register a2, which explains the leak.
Load Sbox byte. Instruction `lbu a4, 0(a4)` (line 7, Listing 3) loads a value from Sbox; leakage here is not expected as such leakage was not implicitly seen during the bus-width microbenchmarks and was, therefore, also not part of the leakage profile.
Store Sbox byte. Instruction `sb a4, 0(a5)` (line 13, Listing 3) stores the retrieved Sbox output value at the right location in the state. During this operation, the correlated value is directly put on the bus leakage is anticipated. As more thoroughly described in the Picorv32 paragraph, no micro-benchmark specifically tests for this kind of leakage. Furthermore, note that for Sbox-out, there appear to be two correlation peaks at this location, which are "merged" together. Looking at the simulation, an

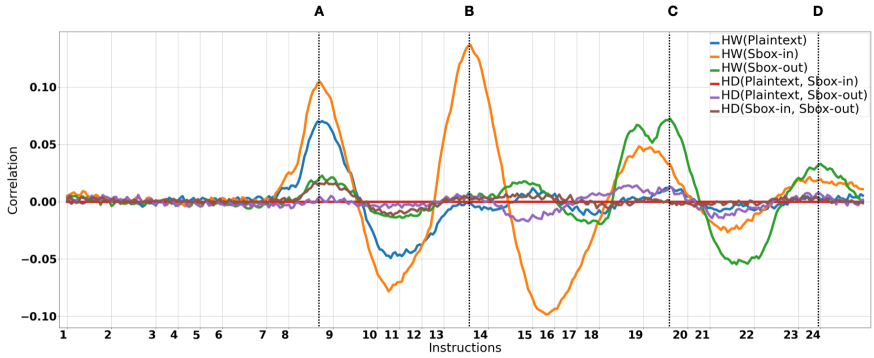

Figure 5: *Leaks during SB^{13} for the Ibex core. The x-axis corresponds to the line-numbers in the Listing 3.*

interesting effect is seen: just before the value in register `a4` is stored at the memory location of `a5`, the value of `a4` is loaded from the register. This is because after Ibex decodes an instruction, it interprets parts of the instruction as registers and immediately reads from them, even when this is not needed to execute the instruction. These unnecessary reads from the register file were also noted by [12]—no microbenchmark tests for this specific effect.

Load loop iterator. `lbu a4, -18(s0)` (line 18, Listing 3) Similar to Picorv32, the register that holds the Sbox-out value is overwritten, which results in an overwrite leakage effect. The resulting correlation peak is rather low compared to the other leakage points, but considering the absolute correlation is so high, it is still significant. This effect is not part of MIRACLE and, therefore, not the profile.

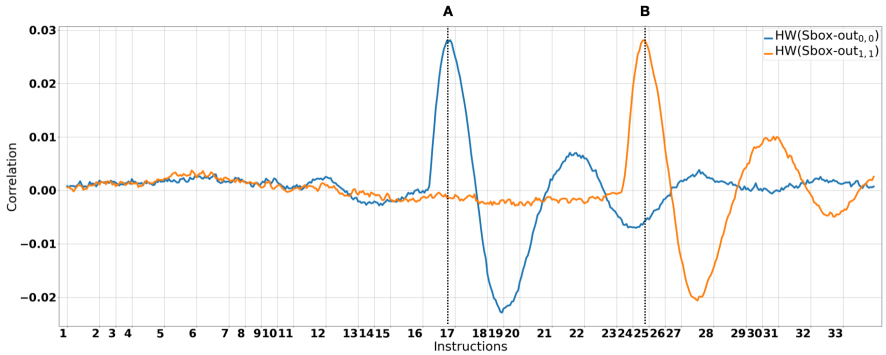

Figure 6: *Leaks during the first iteration of the re-masking step for the Ibex core. The x-axis corresponds to the line numbers in Listing 7.*

6.3 CASE STUDY: masked AES on Ibex core

Figure 6 shows the correlation with the Sbox-out values corresponding to `k[0][0]` and `k[1][1]` when the first two AES state bytes are re-masked. Since the re-masking step involves a loop going over all of the AES state columns, it is sufficient to focus on the first iteration of the loop. The leakage during the other iterations repeats what is seen during the first iteration. Figure 6 shows two high correlation peaks (patterns A and B). One peak (pattern A) of leakage corresponds to when the re-masked value is written to memory. The second peak (pattern B) of leakage corresponds to the masked, and mask values are loaded from memory. These correlation peaks are discussed in detail below.

```
lw a4,-36(s0)
lw a5,-20(s0)
slli a5,a5,0x2
add a5,a4,a5
lbu a4,0(a5)
lbu a3,-37(s0)
lbu a5,-41(s0)
xor a5,a3,a5
andi a5,a5,255
xor a5,a4,a5
andi a4,a5,255
lw a3,-36(s0)
lw a5,-20(s0)
slli a5,a5,0x2
add a5,a3,a5
sb a4,0(a5)
```

```
lw a4,-36(s0) ; Load state S
lw a5,-20(s0) ; Load iterator i
slli a5,a5,0x2
add a5,a4,a5 ; Compute S_i location
lbu a4,0(a5) ; Load state byte S[0][i]
lbu a3,-38(s0) ; Load new mask
lbu a5,-42(s0) ; Load old mask
xor a5,a3,a5 ; old mask XOR new mask
andi a5,a5,255
xor a5,a4,a5 ; S[0][i] XOR (old mask XOR new mask)
andi a4,a5,255
lw a3,-36(s0) ; Load state S
lw a5,-20(s0) ; Load iterator i
slli a5,a5,0x2
add a5,a3,a5 ; Compute S[0][i] location
sb a4,0(a5) ; Store remasked S[0][i]
```

Figure 7: *Re-masking operation of a state byte in the first row (left) and in the second row (right). The comments give a semantic context for the assembly instructions.*

(R) add a_5,a_3,a_5	0000 0000 1111 0110 1000 0111 1011 0011
(S) sb a_4,0(a_5)	0000 0000 1110 0111 1000 0000 0010 0011
(I) lw a_4,-36(s_0)	1111 1101 1100 0100 0010 0111 0000 0011

Figure 8: *Unintended register read of `lw a4,-36(s0)` reads bits `11100` as an operand. The color code shows instruction encoding according to the instruction type (Section 3).*

Store re-masked byte. Instruction `sb a4,0(a5)` stores the remasked AES state byte in memory. Ibex performs unintended register loads, as noted in [12]. In this instance, the instruction decoder from Ibex decodes the subsequent instruction `lw a4,-36(s0)` where it interprets the -36 immediate value not only as a constant but also as an encoding for the 28th register. Therefore, an unintended read is performed from the 28th register, also known as `t3`. Figure 8 shows how the RISC-V instruction decoder/encoder[12] translate the instructions to machine code. Register 28 temporarily

[12] https://luplab.gitlab.io/rvcodecjs/

Example	Core	Type of Leak	Listing	Figure	Notes
CS1	Picorv32	Load AES state	3(line 7)	4 (A)	Bus width effect (`LD-BYTE`)
CS1	Picorv32	Load Sbox byte	3(line 13)	4 (B)	Bus width effect (`LD-BYTE`)
CS1	Picorv32	Store Sbox byte	3(line 18)	4 (C)	Data bus leakage
CS1	Picorv32	Load loop iterator	3(line 22)	4 (D)	Register overwrite
CS2	Ibex	Move loaded state	3(line 8)	5 (A)	Register overwrite
CS2	Ibex	Load Sbox byte	3(line 13)	5 (B)	Register overwrite
CS2	Ibex	Store Sbox byte	3(line 18)	5 (C)	Unintended register read
CS2	Ibex	Load loop iterator	3(line 22)	5 (D)	Register overwrite
CS3	Ibex	Store remasked byte	7(lines 16-17)	6 (A)	Unintended register read
CS3	Ibex	Load old mask byte	7(lines 22-23)	6 (B)	Multiplexer switching

Table 2: *Overview of the explained leak examples.*

stores the new mask in the prelude of the re-masking function, which is not shown here. Therefore the new mask is unnecessarily loaded from the register file where it interacts with the newly masked Sbox-out value, loaded from the register file during the `sb` instruction.

As the new masks are row specific, this effect is only seen when re-masking values in the first row of the AES state. The first-row mask is also loaded for the other rows, but since this mask is not used to mask the state bytes in these rows, leakage for these state bytes is not observed.

Load old mask. Instruction `lbu a5, -42(s0)` loads the old mask. The second leakage effect only occurs when re-masking a byte in the second row of the AES state. Correlation with Sbox-out values is seen before the old and new masks are XORed. Since the Sbox-out is not yet re-masked, leakage happens because of an interaction between the masked Sbox-out value and the old mask.

Similar to the previous correlation peak, there is an instruction with an immediate value interpreted as a register encoding, resulting in a redundant read from the register file. The value -42 is interpreted as the 22nd register, and the value in this register is loaded. However, this register does not contain the masked Sbox-out value or the old mask. We explain this leak due to *switching wires in the multiplexer tree of the register file* effect described in [12]. When two consecutive read operations occur, the multiplexer wires switch values, leading to transitional leakage. In our case, a load is performed from the 22nd register (`s6`) followed by a read from `a5` (15th register). The least significant bit of the read address switches from 0 to 1, which causes changes in multiplexer wires. The multiplexer connected to registers `a4` and `a5` switches from the value in `a4` (the masked Sbox-out) to `a5` (the old mask), which results in transitional leakage of these values and thus leakage of the Sbox-out.

7 Conclusions

As a result of the complexity of devices and their undocumented features, detecting the source of a side-channel leak is not a trivial exercise. This paper introduces a three-step methodology for systematically explaining side-channel leaks using microbenchmarks.

Reusability of side-channel profile. The first step, diagnosis of predisposition for a side-channel leak, is effort intensive. However, as the microbenchmarks results for the Picorv32 core matched that of [8] we conclude that microbenchmarks are reusable and such profiling needs to be done only once for each device. A diagnosis can be an

invaluable guide to a designer who must implement a masked cryptographic algorithm. The advantage of microbenchmarks is that they can be run on *any* embedded CPU, regardless of the available knowledge about its design.

Efficacy in identifying leakage. The experiments performed in this research, see Table 2 show that microbenchmarks partially explain seven out of ten leaks seen during our examples. However, not all leaks could be identified by relating them to the leakage profile. More subtle effects, like the unintended redundant register file read in Ibex, were not explained by existing microbenchmarks. The diagnosed leakage profiles included effects related to control-flow and speculative execution, which are interesting but only when the core has a deep pipeline or the assembly code has a lot of branch statements. Cryptographic implementations consist mainly of arithmetic instructions; branch instructions are often avoided as much as possible since they leak trivial side-channel information. Ultimately we conclude that in terms of efficacy, improvements justify the usage of microbenchmarks.

Accuracy of identified leakage. The main challenge in creating effective microbenchmarks is establishing that the code interacts only with the intended part of the microarchitecture. If the relationship between the undocumented microarchitectural component and the code of the microbenchmarks is inaccurate, the results of the diagnosis phase cannot be relied upon. For example, in Picorv32, leakage was observed for several pipeline benchmarks. Much of this leakage is believed to be caused by values colliding in registers that hold the instruction operands and is not, for example, related to control flow or speculative execution. Naturally, isolating the microarchitecture with software-based testing is difficult, and it could be argued that this is just a general disadvantage of this technique. Although microbenchmarks cannot (yet) explain all microarchitectural features, they are a fundamental step toward explainable side-channel analysis.

We conclude that existing microbenchmarks are a good start to help explain the root cause of side-channel leakage but are incomplete. Our framework *Diagnose-Identify-Explain*is general and can be applied to any architecture.

Appendix A Unprotected AES Implementation

Figure A1 shows the execution trace (with cycle information) of the execution of the unprotected AES implementation of the IBEX core.

Appendix B Masked AES Implementation

The protected AES implementation we used for our experiments is the boolean masking described in [29]. For completeness, we include the overview of the masking scheme and the execution trace on the IBEX core scheme (with cycle information), shown in Figure B2. The scheme uses six independent masks (one byte each). The masks, denoted with m and m', are the input and output for the masked SubBytes operation. The remaining four masks, m_1, m_2,m_3, and m_4, are the MixColumn Operation's input masks.

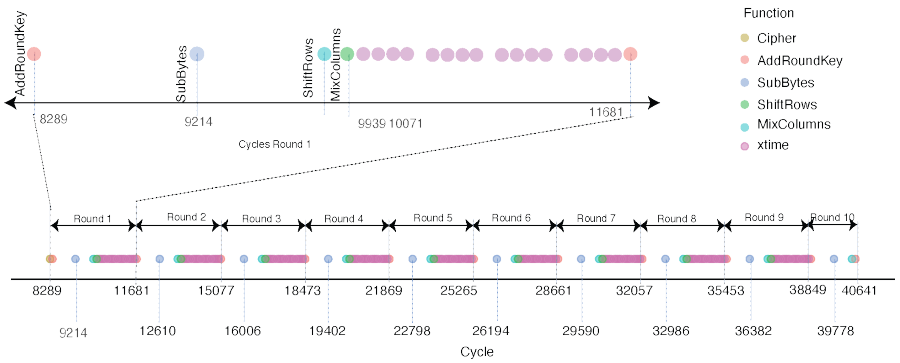

Figure A1: *Execution trace for the unprotected AES implementation on the IBEX core.*

Initialize masking scheme

At the beginning of an AES encryption, two precomputations take place:

1. The masked Sbox table S_m is precomputed, such that

$$S_m(x \oplus m) = S(x) \oplus m'.$$

2. Compute output mask for MixColumn operation by applying the MixColumn operation to m_1, m_2,m_3, and m_4. The result of this operation is denoted with m'_1, m'_2,m'_3, and m'_4.

Masking the key schedule

The `remask` function is called ten times to mask the round keys. Each round key is masked such that the masks change from m'_1,m'_2,m'_3,m'_4 to m. The last round key is an exception and is only masked with m'.

Masking the plaintext

At the beginning of the first round, the `remask` function is called to mask the plaintext with m'_1, m'_2,m'_3, and m'_4.

Masking the round transformation

- **AddRoundKey**: the round key is masked such that the masks change from $(m'_1, m'_2, m'_3, m'_4) \rightarrow m$
- **SubBytes:** Is the only non-linear operation of AES in SubBytes and is implemented as a table look-up. The Sbox table lookup operation is protected with a mask and precomputed when the algorithm is initialized. S_m is performed. The mask has been changed to m'.
- The **ShiftRows** moves the bytes of the state to different positions. Therefore, this operation does not influence the masks.
- **MaskedMixColumn** mixes different bytes from each column. Each row is masked with a different mask. The protected implementation changes the masks

$m_i \rightarrow m'_i$ for $i \in 1, 2, 3, 4$. The `remask` function is called on the input to MixColumn, and it changes the mask from m_1, m_2,m_3, and m_4 to m'.

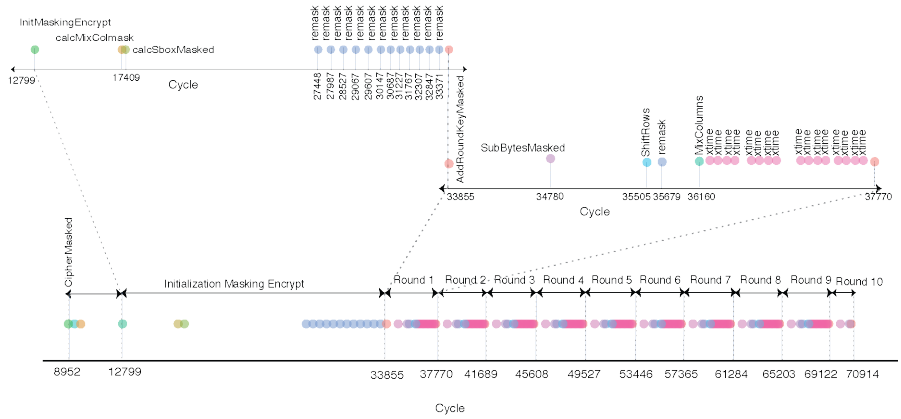

Figure B2: *Execution trace for the protected AES implementation on the IBEX core. The masked execution of AES starts at cycle 8952, the masking scheme is initialized at cycle 12799, and the first round starts at cycle 33855. The whole encryption is completed at cycle 70914.*

Appendix C Microbechmarks results

Here, we show the figures for all the microbenchmarks that show leaks

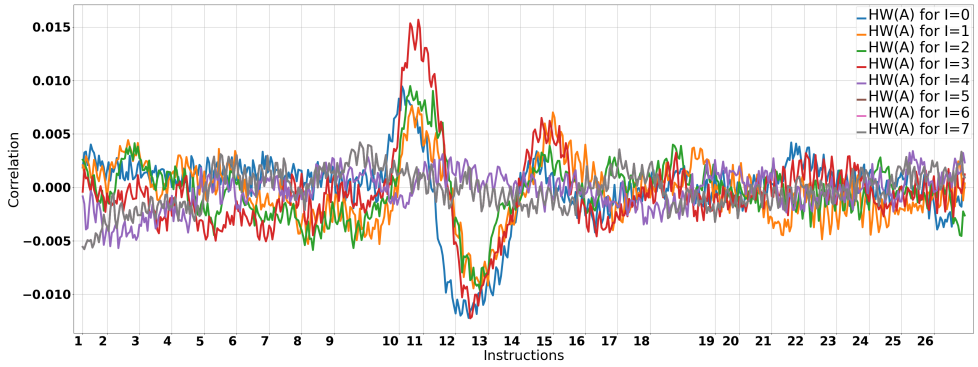

Figure C3: `LD_BYTE` *results for Picorv32, the instructions corresponding to the line numbers can be found in 4.*

```
1     lbu a3,0(a0)
2-9   nop
10    lbu a2,0(a1) # a2 = A only for I=3
11-18 nop
19    lbu a3,0(a0)
20-26 nop
```

Listing 4: `LD_BYTE` *experiment code listing, the instruction highlighted with yellow is the target instruction of this experiment.*

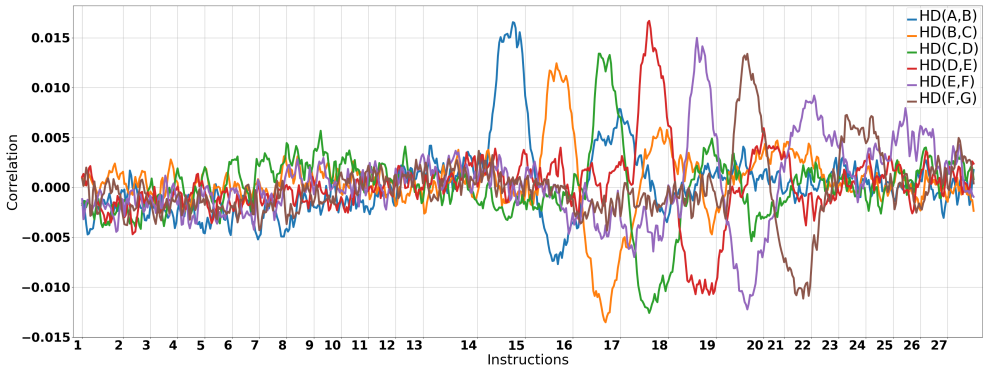

Figure C4: `SEQ_ST` *results for Picorv32, the instructions corresponding to the line numbers can be found in 5.*

```
1     jal ra,10059c
2     xor t0,t0,t0
3     xor t1,t1,t1
4     xor t2,t2,t2
5     xor t3,t3,t3
6-13  nop
14    sb a1,0(a0) # a1 = A
15    sb a2,1(a0) # a2 = B
16    sb a3,2(a0) # a3 = C
17    sb a4,3(a0) # a4 = D
18    sb a5,4(a0) # a5 = E
19    sb a6,5(a0) # a6 = F
20    sb a7,6(a0) # a7 = G
20-27 nop
```

Listing 5: `SEQ_ST` *experiment code listing, the instructions highlighted with yellow are the targeted instructions of this experiment.*

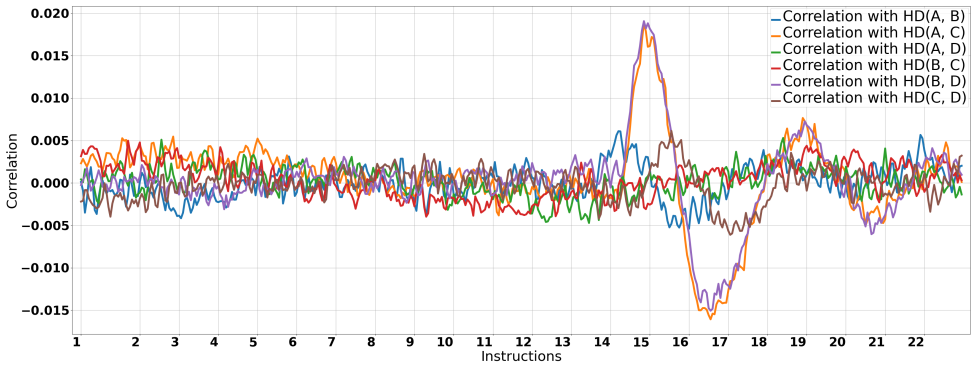

Figure C5: `XOR_XOR` *results for Picorv32, the instructions corresponding to the line numbers can be found in 6.*

```
1     jal ra,100214
2     xor t0,t0,t0
3     xor t1,t1,t1
4     xor t2,t2,t2
5     xor t3,t3,t3
6-13  nop
14    xor a0,a0,a1 # a0 = A, a1 = B
15    xor a2,a2,a3 # a2 = C, a3 = D
16-23 nop
```

Listing 6: `XOR_XOR` *experiment code listing, the instruction highlighted with yellow is the target instruction of this experiment.*

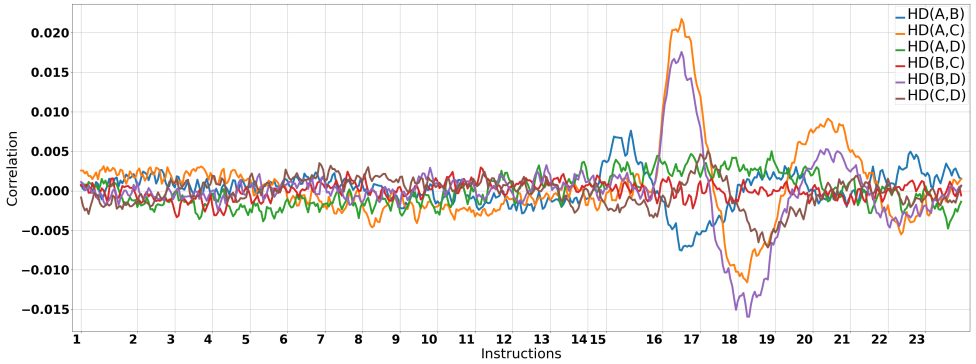

Figure C6: `JUMP_PRE` *results for Picorv32, the instructions corresponding to the line numbers can be found in 7.*

```
1     jal ra,1002f8
2     xor t0,t0,t0
3     xor t1,t1,t1
4     xor t2,t2,t2
5     xor t3,t3,t3
6-13  nop
14    xor a0,a0,a1 # a0 = A, a1 = B
15    j 10026c
16    xor a2,a2,a3 # a2 = C, a3 = D
10    nop
```

Listing 7: `JUMP_PRE` *experiment code listing, the instructions highlighted with yellow are the targeted instructions of this experiment.*

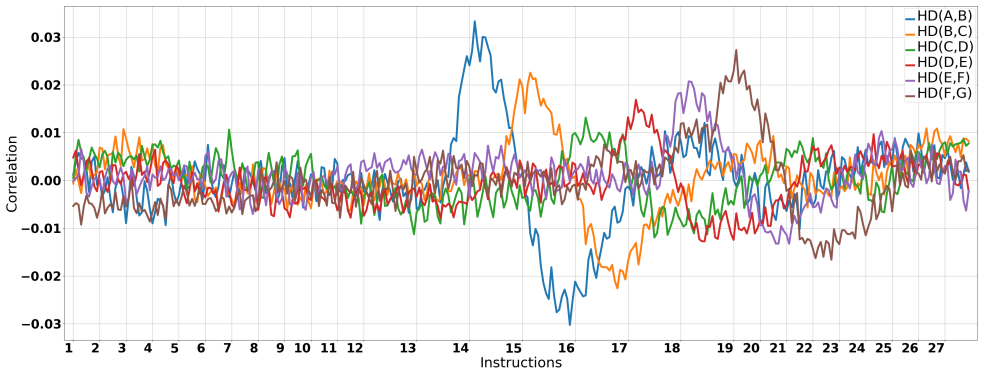

Figure C7: `SEQ_ST` *results for Ibex, the instructions corresponding to the line numbers can be found in 8.*

```
1     jal ra,10059c
2     xor t0,t0,t0
3     xor t1,t1,t1
4     xor t2,t2,t2
5     xor t3,t3,t3
6-13  nop
14    sb a1,0(a0) # a1 = A
15    sb a2,1(a0) # a2 = B
16    sb a3,2(a0) # a3 = C
17    sb a4,3(a0) # a4 = D
18    sb a5,4(a0) # a5 = E
19    sb a6,5(a0) # a6 = F
20    sb a7,6(a0) # a7 = G
20-27 nop
```

Listing 8: `SEQ_ST` *experiment code listing, the instructions highlighted with yellow are the targeted instructions of this experiment.*

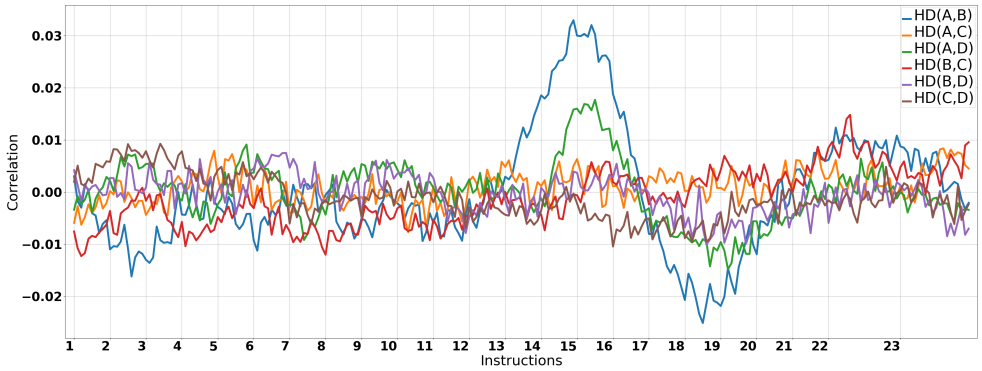

Figure C8: `XOR_SRLI` *results for Ibex, the instructions corresponding to the line numbers can be found in 10.*

```
1     jal ra,100214
2     xor t0,t0,t0
3     xor t1,t1,t1
4     xor t2,t2,t2
5     xor t3,t3,t3
6-13  nop
14    xor a0,a0,a1 # a0 = A, a1 = B
15    srli a2,a3,8 # a2 = C, a3 = D
16-23 nop
```

Listing 9: `XOR_SRLI` *experiment code listing, the instructions highlighted with yellow are the target instructions of this experiment.*

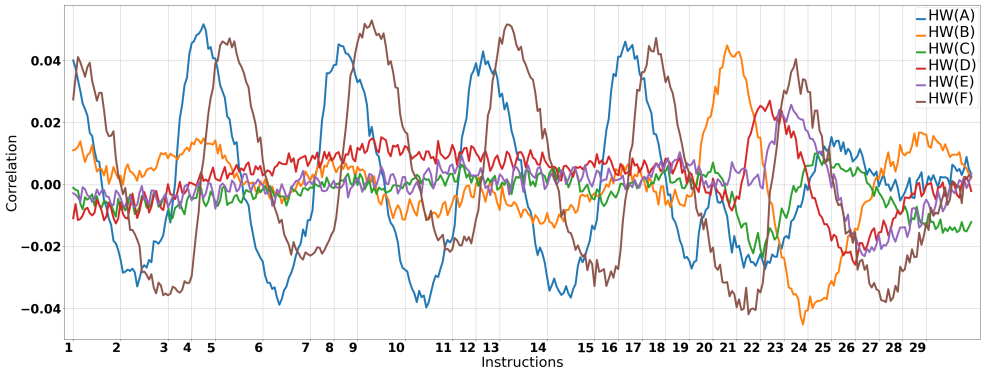

Figure C9: *`LOOP_0` results (only the last five iterations are shown) for Ibex, the instructions corresponding to the line numbers can be found in* **??**.

```
1  li a7,0
2  addi a6,a6,-1
3  bnez a6,100804 # 100804: xor a0, a0, a1
4  xor a7,a7,a5
5  li a7,0
6  addi a6,a6,-1
7  bnez a6,100804 # 100804: xor a0, a0, a1
8  xor a7,a7,a5
9  li a7,0
10 addi a6,a6,-1
11 bnez a6,100804 # 100804: xor a0, a0, a1
12 xor a7,a7,a5
13 li a7,0
14 addi a6,a6,-1
15 bnez a6,100804 # 100804: xor a0, a0, a1
16 xor a7,a7,a5
17 li a7,0
18 addi a6,a6,-1
19 bnez a6,100804 # 100804: xor a0, a0, a1
20 xor a0,a0,a1
21 xor a2,a2,a3
22 xor a4,a4,a5
23-30 nop
```

Listing 10: `XOR_SRLI` *experiment code listing, the instructions highlighted with yellow are the target instructions of this experiment.*

```
1  jal ra,100624
2  xor t0,t0,t0
3  xor t1,t1,t1
4  xor t2,t2,t2
5  xor t3,t3,t3
6-13 nop
```

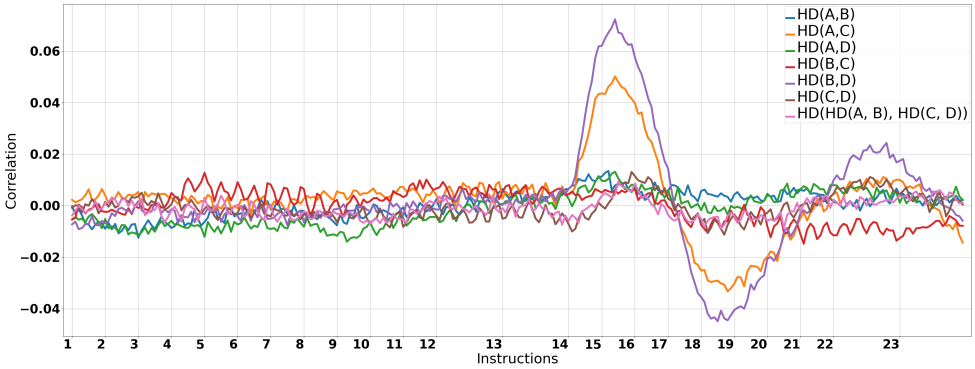

Figure C10: `BRANCH_POST` *micro-benchmark results for Ibex the instructions corresponding to the line numbers can be found in 11.*

```
14    beqz zero,1005f4 # 1005f4: xor a0, a1, a1
15    xor a2,a2,a3 # a2 = C, a3 = D
16-23 nop
```

Listing 11: `BRANCH_POST` *experiment code listing, the instructions highlighted with yellow are the target instructions of this experiment.*

References

[1] Kocher PC, Jaffe J, Jun B. Differential Power Analysis. In: Wiener MJ, editor. Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. vol. 1666 of Lecture Notes in Computer Science. Springer; 1999. p. 388–397. Available from: https://doi.org/10.1007/3-540-48405-1_25.

[2] Oleksenko O, Fetzer C, Köpf B, Silberstein M. Revizor: testing black-box CPUs against speculation contracts. In: Falsafi B, Ferdman M, Lu S, Wenisch TF, editors. ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022. ACM; 2022. p. 226–239.

[3] Renauld M, Standaert F, Veyrat-Charvillon N, Kamel D, Flandre D. A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In: Paterson KG, editor. Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. vol. 6632 of Lecture Notes in Computer Science. Springer; 2011. p. 109–128. Available from: https://doi.org/10.1007/978-3-642-20465-4_8.

[4] Gao S, Oswald E, Page D. Towards Micro-architectural Leakage Simulators: Reverse Engineering Micro-architectural Leakage Features Is Practical. In: Dunkelman O, Dziembowski S, editors. Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III. vol. 13277 of Lecture Notes in Computer Science. Springer; 2022. p. 284–311.

[5] Akkar M, Giraud C. An Implementation of DES and AES, Secure against Some Attacks. In: Koç ÇK, Naccache D, Paar C, editors. Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings. vol. 2162 of Lecture Notes in Computer Science. Springer; 2001. p. 309–318. Available from: https://doi.org/10.1007/3-540-44709-1_26.

[6] Bloem R, Gigerl B, Gourjon M, Hadzic V, Mangard S, Primas R. Power Contracts: Provably Complete Power Leakage Models for Processors. In: Yin H, Stavrou A, Cremers C, Shi E, editors. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. ACM; 2022. p. 381–395. Available from: https://doi.org/10.1145/3548606.3560600.

[7] Papagiannopoulos K, Veshchikov N. Mind the Gap: Towards Secure 1st-Order Masking in Software. In: Guilley S, editor. Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers. vol. 10348 of Lecture Notes in Computer Science. Springer; 2017. p. 282–297. Available from: https://doi.org/10.1007/978-3-319-64647-3_17.

[8] Marshall B, Page D, Webb J. MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices. IACR Trans Cryptogr Hardw Embed Syst. 2022;2022(1):175–220. https://doi.org/10.46586/TCHES.V2022.I1.175-220.

[9] Buhan I, Batina L, Yarom Y, Schaumont P. SoK: Design Tools for Side-Channel-Aware Implementations. In: Suga Y, Sakurai K, Ding X, Sako K, editors. ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022. ACM; 2022. p. 756–770. Available from: https://doi.org/10.1145/3488932.3517415.

[10] Ishai Y, Sahai A, Wagner DA. Private Circuits: Securing Hardware against Probing Attacks. In: Boneh D, editor. Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. vol. 2729 of Lecture Notes in Computer Science. Springer; 2003. p. 463–481.

[11] Zeitschner J, Müller N, Moradi A. PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries. IACR Trans Cryptogr Hardw Embed Syst. 2023;2023(3):391–421. https://doi.org/10.46586/TCHES.V2023.I3.391-421.

[12] Gigerl B, Hadzic V, Primas R, Mangard S, Bloem R. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In: Bailey MD, Greenstadt R, editors. 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. USENIX Association; 2021. p. 1469–1468. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[13] De Mulder E, Gummalla S, Hutter M. Protecting RISC-V against side-channel attacks. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). IEEE; 2019. p. 1–4.

[14] KF MA, Ganesan V, Bodduna R, Rebeiro C. PARAM: A microprocessor hardened for power side-channel attack resistance. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE; 2020. p. 23–34.

[15] He M, Park J, Nahiyan A, Vassilev A, Jin Y, Tehranipoor M. RTL-PSC: Automated Power Side-Channel Leakage Assessment at Register-Transfer Level. In: 2019 IEEE 37th VLSI Test Symposium (VTS); 2019. p. 1–6.

[16] Gao S, Großschädl J, Marshall B, Page D, Pham TH, Regazzoni F. An Instruction Set Extension to Support Software-Based Masking. IACR Trans Cryptogr Hardw Embed Syst. 2021;2021(4):283–325. https://doi.org/10.46586/TCHES.V2021.I4.283-325.

[17] Pham TH, Marshall B, Fell A, Lam SK, Page D.: XDIVINSA: eXtended DIVersifying INStruction Agent to Mitigate Power Side-Channel Leakage. Cryptology ePrint Archive, Report 2021/1053.

[18] Yao Y, Kathuria T, Ege B, Schaumont P. Architecture Correlation Analysis (ACA): Identifying the Source of Side-channel Leakage at Gate-level. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020. IEEE; 2020. p. 188–196.

[19] Kiaei P, Schaumont P. SoC Root Canal! Root Cause Analysis of Power Side-Channel Leakage in System-on-Chip Designs. IACR Trans Cryptogr Hardw Embed Syst. 2022;2022(4):751–773. https://doi.org/10.46586/TCHES.V2022.I4.751-773.

[20] De Meyer L, De Mulder E, Tunstall M.: On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software. https://eprint.iacr.org/2020/1297. Cryptology ePrint Archive, Report 2020/1297.

[21] McCann D, Oswald E, Whitnall C. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In: Kirda E, Ristenpart T, editors. 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. USENIX Association; 2017. p. 199–216. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann.

[22] Shelton MA, Samwel N, Batina L, Regazzoni F, Wagner M, Yarom Y. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society; 2021. Available from: https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/.

[23] Shelton MA, Chmielewski L, Samwel N, Wagner M, Batina L, Yarom Y. Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In: Kim Y, Kim J, Vigna G, Shi E, editors. CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. ACM; 2021. p. 685–699. Available from: https://doi.org/10.1145/3460120.3485380.

[24] Arora V, Buhan I, Perin G, Picek S. A tale of two boards: On the influence of microarchitecture on side-channel leakage. In: Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers. vol. 13173 of Lecture Notes in Computer Science. Springer; 2021. p. 80–96.

[25] de Grandmaison A, Heydemann K, Meunier QL. ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2022;41:3733–3744.

[26] Gaspoz J, Dhooghe S. Threshold Implementations in Software: Microarchitectural Leakages in Algorithms. IACR Trans Cryptogr Hardw Embed Syst. 2023;2023(2):155–179. https://doi.org/10.46586/TCHES.V2023.I2.155-179.

[27] Standaert F, Malkin T, Yung M. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In: Joux A, editor. Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings. vol. 5479 of Lecture Notes in Computer Science. Springer; 2009. p. 443–461.

[28] Chadwick G.: Ibex Reference Guide - Security Features.

[29] Mangard S, Oswald E, Popp T. Power analysis attacks - revealing the secrets of smart cards. Springer; 2007.

[30] Kocher PC, Jaffe J, Jun B. Differential Power Analysis. In: Wiener M, editor. Advances in Cryptology – CRYPTO '99. vol. 1666 of LNCS. Springer; 1999. p. 388–397. http://www.cryptography.com/public/pdf/DPA.pdf.

[31] Brier E, Clavier C, Olivier F. Correlation Power Analysis with a Leakage Model. In: Joye M, Quisquater JJ, editors. Cryptographic Hardware and Embedded Systems – CHES 2004. vol. 3156 of LNCS. Springer; 2004. p. 16–29.

[32] Goodwill G, Jun B, Jaffe J, Rohatgi P.: A testing methodology for side-channel resistance validation, NIAT.