ARCHER: Architecture-Level Simulator for Side-Channel Analysis in RISC-V Processors

Asmita Adhikary[1][0000−0002−2757−1271], Abraham Basurto-Becerra[1][0000−0002−7164−5952], Lejla Batina[1][0000−0003−0727−3573], Ileana Buhan[1][0000−0001−5494−9164], Durba Chatterjee[1][0000−0001−7665−0876], Senna van Hoek[1][0009−0002−3751−4833], and Eloi Sanfelix Gonzalez

[1]Radboud University, Nijmegen, The Netherlands
`{firstname.lastname}@ru.nl`

Abstract. Side-channel attacks pose a serious risk to cryptographic implementations, particularly in embedded systems. While current methods, such as test vector leakage assessment (TVLA), can identify leakage points, they do not provide insights into their root causes. We propose `ARCHER`, an architecture-level tool designed to perform side-channel analysis and root cause identification for software cryptographic implementations on RISC-V processors. `ARCHER` has two main components: (1) Side-Channel Analysis to identify leakage using TVLA and its variants, and (2) Data Flow Analysis to track intermediate values across instructions, explaining observed leaks. Taking the binary file of the target implementation as input, `ARCHER` generates interactive visualizations and a detailed report highlighting execution statistics, leakage points, and their causes. It is the first architecture-level tool tailored for the RISC-V architecture to guide the implementation of cryptographic algorithms resistant to power side-channel attacks. `ARCHER` is algorithm-agnostic, supports pre-silicon analysis for both high-level and assembly code, and enables efficient root cause identification. We demonstrate `ARCHER`'s effectiveness through case studies on unprotected and protected AES and unprotected Ascon implementations, where it accurately traces the source of side-channel leaks. We report previously undocumented vulnerabilities due to architectural register usage in the ShiftRows operation of the protected AES implementation. For the Ascon implementation, we report leaks both in the substitution layer and in the diffusion layer, thus reflecting its susceptibility to data-dependent side-channel leakage.

Keywords: Side-Channel Analysis · RISC-V · Pre-silicon · Data Flow Analysis · Qiling · AES · Ascon.

1 Introduction

Analyzing assembly language code often poses challenges due to its difficulty in readability and maintenance, and its dependency on specific architectures. Despite these challenges, disassembling an executed binary file to examine low-level code is essential when investigating side-channel leaks in cryptographic software implementations.

Side-channel leaks stem from two sources: 1) improper implementation of the cryptographic scheme, which leads to undesirable interactions of sensitive data within architectural registers, and 2) data interactions from microarchitectural optimizations like pipelining, use of shadow registers, speculative execution, or caching. Two factors complicate the identification of the cause of side-channel leaks. Firstly, the lack of access to cycle-accurate information obscures the precise timing and sequence of events, which we need to pinpoint the instructions that triggered the leak. Secondly, understanding whether the component that caused the side-channel leakage is architectural or microarchitectural is crucial for developing effective remedies. *Best design practices advocate for a top-down approach, which suggests addressing architectural leaks before tackling microarchitectural ones. Differentiating between the two types of leaks requires a detailed analysis to isolate the microarchitectural components effectively.*

Solutions for architectural-level leaks involve reallocating or clearing registers [26,31]. Conversely, fixes for microarchitectural leaks typically target the specific components involved, such as employing fence instructions for speculative executions [25] or adopting constant-time programming to mitigate cache timing attacks [5]. With the advent of open hardware, the appeal of developing cryptographic algorithms for RISC-V architectures has increased [16,24,33], while the tools to support secure cryptographic implementations are few [9,13]. Leakage simulators are one such class of tools aimed at evaluation of side-channel vulnerabilities. While the literature includes several power-based leakage simulators for ARM [8,10,15,22,30,31], tools for RISC-V are geared towards verifying hardware implementations [12,13,29].

Contributions. We propose `ARCHER`, a design tool that focuses on architecture-level leakages for RISC-V cryptographic implementations. `ARCHER` acts as the first step for evaluating the side-channel leakage for any software cryptographic implementation. It assists with the analysis of the binary files using powerful data-flow visualization features. The core contributions of this work are:

1. We propose `ARCHER`, which to the best of our knowledge is the first power side-channel simulator for RISC-V, that isolates architectural side-channel leakage effects, thereby enabling users to focus on the implementation-level vulnerabilities.
2. `ARCHER` can simulate and analyze the *exact* binary file executed by the target device. This avoids the variability of compiler output that could occur otherwise.
3. The integrated *side-channel analysis* module has three leakage models and a built-in leakage assessment module that supports fixed-vs-random and fixed-vs-fixed TVLA tests.
4. The *flow analysis* module aids the data flow visualization and is a valuable tool when determining the root cause of side-channel leaks. We provide the tool, `ARCHER`, the visualizations and reports generated by `ARCHER` in the URL: `https://gitlab.science.ru.nl/cesca/archer`.
5. We demonstrate the working of the tool and the derived insights using unprotected and protected AES and unprotected Ascon as case studies.

Target audience. We develop `ARCHER` for designers/developers who optimize cryptographic implementations and security evaluators who evaluate the impact of a side-channel leak.

2 Related Work

The landscape of cryptographic verification tools is fragmented. At high abstraction levels, tools such as MaskVerif [4], EasyCrypt [6], or Tamarin [7] assist in creating security proofs. Secure compilers like Jasmin [2] can produce low-level assembly; however, the supported architectures are limited. Once an implementation is proven to follow the desired security proof, it must also withstand side-channel attacks. Consequently, the interest in tools to detect, verify, and mitigate side-channel leaks is significant [9]. Papagiannopoulos et al. [27] were among the first to discuss the microarchitectural effects when analyzing side-channel leaks for software implementations. De Meyer et al. [23] and Arora et al. [3] continue their work and discuss additional microarchitecture leakage effects on different target devices.

McCann et al. [22] introduced ELMO, a side-channel leakage simulator which models the power consumption of a software implementation as a linear combination of values and transitions. Shelton et al. [31] improve the leakage model in ELMO by capturing data interactions across multiple cycles. As ELMO models (part of) the microarchitecture of the target device *it can only be used for ARM Cortex-M0* platforms. Abby [8] streamlines the profiling of the target device, which enables the use of machine learning models for capturing (parts of) the microarchitecture, for the *ARM Cortex M0 and M3*. Marshall et al. [21] propose MIRACLE, a generic set of microbenchmarks, which *can detect microarchitecture optimizations* to improve leakage models but does not examine their application in the context of side-channel attacks. In contrast, `ARCHER` *targets the architectural layer for modeling the data dependencies at the architecture level* with a goal to perform side-channel analysis. The closest tool to `ARCHER` is MAMBO [36], which captures architecture-specific *timing* leaks for RISC-V for creating constant-time code. In contrast, `ARCHER` is aimed at addressing *power side-channel leaks*. MAMBO-V uses dynamic binary instrumentation to generate execution traces, which are analyzed in parts to identify leakage. In contrast, execution traces generated in `ARCHER` include a sequence of assembly instructions along with the register contents capturing the processed data.

3 Preliminaries

This section presents the notations followed in this paper and briefly describes the basic concepts required in this work.

Notation. We denote an architectural register as r_i, where i is a number from the set $\{1, \ldots, m\}$. We denote with I an assembly instruction executed by the target. For all instructions, the instruction mnemonic (I) is specified first, followed by the destination register (`RD`), the first operand (`OP1`, also known as source register), and the second operand (`OP2`). An *execution trace* contains the sequence of executed instructions, $\{I_1, \ldots, I_N\}$ for a given input, where N represents the total number of executed instructions. For each instruction I_j we store the state of all architectural registers $\{r_1^j, \ldots, r_m^j\}$[1].

Test Vector Leakage Assessment (TVLA) [14] is one of the most popular leakage detection methods based on statistical hypothesis tests. It comes in two flavors: *specific* and *non-specific*. The 'fixed-vs-random' is the most common non-specific test and compares a set of traces acquired with a fixed plaintext with another set of traces acquired with random plaintext. In the case of a specific test, commonly known as 'fixed-vs-fixed', the traces are divided according to a known intermediate value tested for leakage. Welch's two-sample t-test for equality of means is applied for all trace samples in both cases. An absolute difference between two sets larger than the standard threshold of 4.5 is taken as evidence of a leak's presence.

RISC-V is an open-source ISA and follows a `LOAD/STORE` architecture. Due to the `LOAD/STORE` architecture, operations can not be performed directly on memory, and data must be first moved to registers. This implies that any data-dependent activity is visible in the register state.
Next, we briefly describe the algorithms chosen for analysis.

AES-128 is a symmetric-key cryptographic algorithm that transforms a 128-bit plaintext into a 128-bit ciphertext using a 128-bit key [11]. Its execution spans 10 rounds, with each round consisting of AddRoundKey, SubBytes (S-box), ShiftRows, and MixColumns operations, except for the last round, which has only AddRoundKey, SubBytes, and ShiftRows operations.

Byte-Masked AES is a first-order Boolean-masked implementation of AES based on the masking scheme described in [20], which applies masking to plaintext, key schedule, and round functions using six independent masks: m, m', m_1, m_2, m_3, and m_4, which are propagated and updated across AES operations:

1. *AddRoundKey.* The key bytes are XOR-ed with the masked state. Each byte of the state is masked with a common mask m, and this step maintains this masking by XOR-ing the masked state with the (unmasked) key.
2. *SubBytes.* It is the only non-linear operation in AES and is implemented as a masked table lookup. A masked S-box s^m is precomputed as:

$$s^m(x \oplus m) = s(x) \oplus m'$$

   This operation uses a different mask byte m' to XOR with the S-box output, ensuring that the output remains first-order secure under the new mask.
3. *ShiftRows.* This operation rearranges the positions of the bytes in the state. Since each byte is still masked with the same m', the ShiftRows operation does not alter the masking scheme or introduce any mask mismatches.

[1] In our case, the input is a pair (P_j, K_j), where P_j constitutes the plaintext and K_j represents the key. Depending on the specifics of the implementation, it is possible that other inputs, such as nonce, masks, may need to be provided.

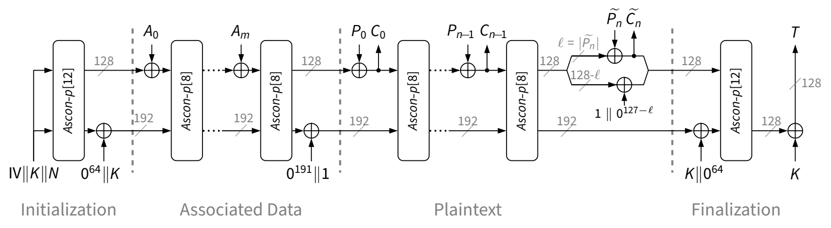


Fig. 1: Schematic representation of Ascon encryption [32]

4. *MixColumns.* To preserve masking security across the linear MixColumns transformation, the four rows of the state matrix are assigned independent masks: m_1, m_2, m_3, and m_4. These masks are derived from m' (the output mask of SubBytes) and applied such that each input byte to MixColumns is masked independently. The transformation outputs are masked with m'_1, m'_2, m'_3, and m'_4, which are reused as the input masks for the next round.
5. *Final Round.* The final AddRoundKey step includes a mask removal process that cancels the active masks and yields the correct unmasked ciphertext.

For AES and Masked AES, we analyze an open-source implementation, wherein enable/disable macros are used to compile the protected/unprotected implementation respectively [2].

Ascon-128 AEAD (Authenticated Encryption with Associated Data) [32], bitsliced by design, processes a 320-bit state comprising a 128-bit key, 128-bit nonce, 64-bit associated data, and 64-bit plaintext to produce an authenticated ciphertext of the same length as the plaintext, along with a 128-bit tag. The algorithm applies a 12-round permutation p^a ($a = 12$) during Initialization and Finalization and a 6-round permutation p^b ($b = 6$) during associated data and plaintext processing as shown in Figure 1. Each round consists of:

1. *Addition of round constant* (p_C): Adds c_i to the 64-bit register x_2 in round i, where the state S stores the initialization vector, the key and the nonce:

$$S = x_0 \| x_1 \| x_2 \| x_3 \| x_4 = IV \| K \| N = IV \| K0 \| K1 \| N0 \| N1$$

2. *Substitution layer* (p_S): A 5-bit S-box ($S(x)$) is applied to each bit-slice of the five state registers $x_0, \ldots, x_4$.
3. *Linear diffusion layer* (p_L): Adds diffusion via a 64-bit linear function $\Sigma_i(x_i)$.

Key points of interest include the first S-box outputs and linear diffusion layer outputs, as this is where the algorithm processes the key and nonce to start Initialization [19]. As shown in [28,35], these intermediates facilitate successful retrieval of key.

[2] `https://github.com/CENSUS/masked-aes-c`

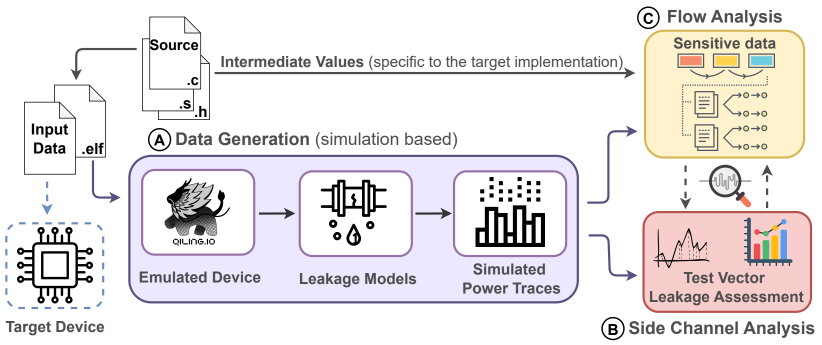


Fig. 2: Overview of side-channel architecture level simulator for RISC-V (ARCHER)

4 ARCHER: Side-Channel Architecture Level Simulator for RISC-V

ARCHER takes a binary file along with the input data (such as plaintext, nonce, keys, and other initialization data) and generates instruction-level interactive visualizations and statistical results pertaining to side-channel leaks. The end-to-end toolflow and component interactions depicted in Fig. 2 are described:

4.1 Data Generation

This module generates simulated power traces for input sets using the Qiling framework[3], executing the binary file with provided data. The number and size of inputs are determined by the cryptographic algorithm, as specified by the user. For leakage evaluation, we adhere to standard data generation guidelines (Section 5.1 of [1]) for creating keys, nonces, plaintexts, associated data and masks. These inputs are used consistently for both simulated and real trace collection. During measurements on the target device, inputs from the two datasets (for TVLA) are interleaved to reduce systematic bias. Execution traces are transformed via leakage models to produce simulated/hypothetical power traces, which are then used for side-channel and data flow analysis. The data generation process for each component is detailed separately in the next section.

4.2 Side-Channel Analysis (SCA)

This component is responsible for the identification of side-channel leakages in an implementation. ARCHER uses TVLA as a statistical test to identify leaks. The steps in this process are described as follows:

1. *Generate input data.* This module generates cryptographic inputs (e.g., plaintext, key, associated data, nonce) based on user-specified parameters, such as the number of input bytes and total inputs.

[3] `https://qiling.io/`

2. *Generate execution traces.* The `.elf` file is executed with the generated inputs, and execution traces are saved in a `.csv` file, capturing all executed instructions and the register states after each instruction.
3. *Create simulation traces.* For each instruction recorded in the `.csv` trace file, we apply a transformation function called *leakage model*, which takes in the values of all the registers and produces an estimate for the data-dependent power consumption incurred by the target instruction (also referred to as hypothetical power trace). `ARCHER` supports three commonly used leakage models in side-channel analysis, namely Identity (ID), Hamming Weight (HW) (for identifying value-based leaks), and Hamming Distance (HD) (for identifying transition-based leaks). We describe the three models as follows:
   - ID: computes the summation of the contents of all the registers (Eq. 1). For an execution trace of length N, the resultant trace also contains N values.
   - HW: Computes the summation of HW of all the register contents after each instruction resulting in a trace of length N (Eq. 2).
   - HD: Calculates the sum of HD between the contents of all registers for each consecutive pair of instructions, resulting in a trace of length $N-1$. For computing the j^{th} value in the simulated power trace, we take the HD between the register content at the end of j and $j+1$ instructions (Eq. 3).

$$L_{ID}(I_j) = \sum_{i=1}^{m} r_i^j, \qquad j = 1, \ldots, N \tag{1}$$

$$L_{HW}(I_j) = \sum_{i=1}^{m} HW(r_i^j), \qquad j = 1, \ldots, N \tag{2}$$

$$L_{HD}(I_j, I_{j+1}) = \sum_{i=1}^{m} HD(r_i^j, r_i^{j+1}), \qquad j = 1, \ldots, N-1 \tag{3}$$

4. *TVLA Analysis.* `ARCHER` uses TVLA for detecting leaks in two modes:
   Fixed-vs-Random: This mode compares traces generated with fixed and random plaintexts for the same key, providing quick leakage detection.
   Fixed-vs-Fixed: This mode compares traces where only one or more plaintext bytes vary, enabling detailed root-cause analysis.

`ARCHER` provides us with two kinds of TVLA graphs:

1. *Classic TVLA plot*: A plot with the sequence of instructions on the x-axis and the *t-score* on the y-axis, featuring red lines at 4.5 and -4.5 to indicate the threshold for side-channel leakage. Sample points that exceed these thresholds suggest the presence of side-channel information leakage (Fig. 4).
2. *Interactive plot*: A plot displaying the sequence of instructions along the x-axis and the different registers on the y-axis, where leaky instructions are highlighted in red and non-leaky instructions are shown in grey. This visualization is appended with intermediate values obtained from *data flow analysis* to identify the root cause of the leakage (Fig. 5 and Fig. 8). This interactive visualization is browser-integrated. The HTML file can be viewed and operated standalone on any browser. It includes interactive features such as zoom and hover-text to aid the developer.

4.3 Flow Analysis

This component tracks sensitive data bytes in different registers across various instructions. The module takes as input the simulated power traces and intermediate values (such as the output of the substitution layer, a combination of plaintext and keys). Optionally, TVLA results can be added as input to generate interactive, annotated visualizations as depicted in Fig. 5 and Fig. 8. These figures provide information about i) the distribution of leaky instructions across different registers, ii) the content of registers after every instruction execution, iii) redundant entries of bytes in different registers, and the remanence of bytes across several instructions (which may potentially lead to leakage), iv) the usage pattern of registers for each algorithm execution. These features enable designers to identify the root cause of leakage. To aid designers in identifying and explaining the source of leakage, this component generates i) interactive visualizations incorporating the TVLA leakage along with the intermediate bytes and ii) a detailed report on the execution, register usage, and side-channel leaks. The steps involved in this component are described as follows:

1. *Generate intermediate values (to track).* For cryptographic implementations, the key bytes and sensitive intermediate data (e.g., S-box outputs, round results, etc.) typically form the focus of SCA. `ARCHER` extracts these intermediate values through a separate execution of the cryptographic algorithm. Since only the data values are relevant, this process remains platform-independent.
2. *Generate execution traces.* This step generates detailed execution traces with additional information, such as instruction mnemonics, operators, and machine code, to pinpoint the location of intermediates.
3. *Generate interactive visualizations.* In this step, the tool places markers in the interactive plots to highlight the presence of intermediate/sensitive bytes in the architectural registers through the execution. To locate markers, multiple execution traces are generated with different values for the sensitive data. The correct marker positions are found by intersecting the respective markers across the different traces.
4. *Visualize markers.* The intermediate bytes are plotted on top of the TVLA plots using different marker symbols as depicted in Fig. 5. The y coordinate of each marker is determined from the destination register holding the value.
5. *Generate report.* A document is generated based on the execution traces and TVLA results. It highlights information such as frequently executed leaking instructions and their locations in the source code and details the distribution of instruction types contributing to leakage.

5 `ARCHER` Implementation Details

This section describes the implementation details of `ARCHER`, followed by its workflow using the example of an unprotected AES implementation compiled for a RISC-V core, PicoRV32[4]. The tool is implemented in Python3.

[4] `https://github.com/YosysHQ/picorv32`

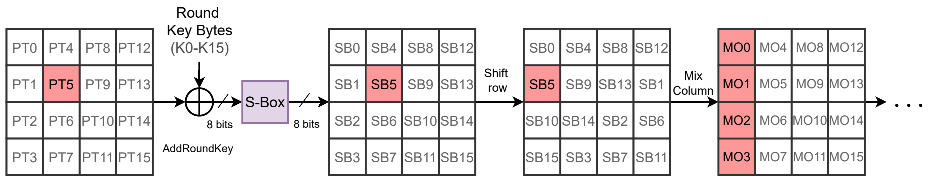


Fig. 3: Sequence of AES operations highlighting the bytes impacted on modifying 6th byte of plaintext

Target Device Setup. `ARCHER` takes the binary file of the target cryptographic implementation as input. To obtain the compiled binary, i.e., the `.elf` file, we cross-compile the AES implementation with the RISC-V GNU Compiler Toolchain (version 2023.11.20). The C source code is cross-compiled for the RV32I architecture using the −`Os` optimization level, as it reduces the size of the executable and is a popular choice for embedded systems. The leakage results reported are specific to the compiler version and the optimization level.

Simulation Setup. The simulation component of `ARCHER` takes the `.elf` file and the number of traces to be generated as input parameters. It retrieves the memory address of the essential elements of the compiled binary, such as the addresses of *key*, *plaintext*, *masks*, depending on the target implementation. After extraction of the addresses, `ARCHER` leverages the Qiling emulation framework [34,18] to run the same binary as would be flashed to a target board. Qiling uses Unicorn for CPU emulation, single-stepping through each instruction while Unicorn handles execution and reports results back to Qiling for state tracking. To support more architectures, Qiling includes its modifications to Unicorn. A callback is configured to be executed for every emulated instruction (code hook). The code hook collects the register states and disassembles the instruction using Capstone. These traces, consisting of assembly instructions and register states, are stored in `.csv` files. `ARCHER` using the Qiling engine, can capture the architectural register content at every instruction. The leakage models simulate the data-dependent power consumption at the architectural level. No microarchitectural feature (pipeline, caching, etc.) is supported.

Data Generation for Side-channel Analysis. In the example, we compute the *fixed-vs-fixed* TVLA wherein we want to highlight leakage caused by a one-byte change in the plaintext. For this, two datasets are generated, differing only in the 6th byte of the plaintext (referred to as `PT5` in Fig. 3). The first dataset consists of a single fixed plaintext and key, while the second includes 500 inputs where all plaintext bytes remain the same except for byte 6, which is randomly varied. These inputs are provided to Qiling along with the AES `.elf` file, which generates one execution trace for the fixed input and 500 traces for the varied inputs. We choose HD model for our analysis.

TVLA Analysis. Next, we compute the TVLA for the two sets of power traces and plot the t-score for each instruction. Fig. 4 depicts the classic TVLA graph that indicates leakage throughout the algorithm's execution. The interactive vi-

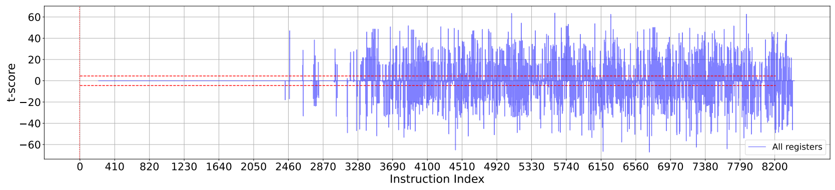


Fig. 4: t-score of unprotected AES obtained from HD simulated traces

sualization plots the TVLA leaks spread across different registers. Fig. 5 depicts a zoomed portion illustrating the leakage points in the first round of AES. The horizontal dotted lines correspond to different registers denoted on the left end. The vertical lines represent the executed instructions. The x coordinate of a vertical bar denotes the instruction index (in the execution trace), and the y position denotes the destination register of the executed instruction. The leaky instructions are highlighted in red. To explain the source of the leaky points, we proceed to the flow analysis module.

Flow Analysis. This component involves generating intermediates for tracking and overlapping them with TVLA results. The performed steps are:

Generation of intermediate values. For AES, we track individual bytes of the plaintext (denoted by `PT0 - PT15`), key (denoted by `K0 - K15`), S-box output (denoted by `SB0 - SB15`), MixColumn output (denoted by `MC0 - MC15`), round keys (denoted by `RK0 - RK15`). These are obtained by executing the C implementation independently. Since we are tracking individual bytes, we might encounter some byte values in unexpected instruction indices. We refer to these unexpected byte appearances as *ghost* values that can be attributed to other byte operations. To remove such *ghost* values, we preprocess the intermediate byte locations before generating the visualizations.

Pre-processing. We work with three distinct inputs (randomly generated plaintext and key). We compute the instruction sequence of all the intermediate bytes for these inputs. An instruction sequence of a byte contains a mapping of the registers where the byte value occurs, along with the list of instruction indices when it appears in the particular register. We then compute the intersection of instruction sequences for each tracked byte across the inputs. The indices that appear in the intersection represent the legitimate points where the intermediate bytes are actually present. This step filters out illegitimate instruction indices where the intermediate byte might appear due to unrelated or independent computations. The final visualizations are generated using the filtered indices.

Though we demonstrate ARCHER on an RV32I architecture, it can be readily extended to 64-bit implementations, by: ① compiling for a 64-bit target architecture, and ② updating the Qiling script to emulate the resulting 64-bit binary. The leakage models will require adaptation to accommodate 64-bit registers. The root cause analysis of leakage is based on the relationship between internal data and observed leakage, independent of the underlying word size.

Visualizations. We utilize the `plotly` library in Python3 to generate interactive visualizations. A snapshot of the visualization for the first round of AES execution is shown in Fig. 5. The red lines indicate the leaky instructions identified by TVLA under the HD leakage model. The legend provides details of the various markers used to represent different intermediate values, with each byte shown in a distinct color and each intermediate output denoted by a unique marker, as seen in the legend. Hovering over a vertical line reveals the executed instruction, program counter (PC), and instruction index, while hovering over the colored markers displays the corresponding byte. For enhanced clarity, we annotate the plot with labels indicating the outputs of various operations. While the TVLA results (Fig. 4) identify the leaky instructions, these visualizations allow us to trace the leakage back to specific intermediate bytes during execution. Key insights derived from these visualizations are discussed in the subsequent section.

6 Insights from ARCHER

We evaluate ARCHER on AES (unprotected and protected) and an unprotected Ascon. The total instruction count for unprotected and masked AES, and unprotected Ascon is 8433, 14784, and 8629, respectively, demonstrating ARCHER's scalability. Each simulated trace size for unprotected AES, masked AES and unprotected Ascon is 4.6kB, 59.13kB and 7.1kB respectively. Simulated trace generation takes 2-3 hours per implementation, and capturing real masked AES measurements takes 34 minutes 45 seconds. We present our insights as follows:

6.1 Case Study: Unprotected AES

We examine an unprotected implementation of AES-128 [11], using the TVLA results obtained in Fig. 4. To understand the source of leakage, we first identify the bytes impacted by randomizing `PT5` and track them via the visualizations. We restrict our analysis to the first round as the difference is propagated to all bytes in subsequent rounds. Fig. 3 depicts the bytes impacted on randomizing `PT5` during the first round of AES. From Fig. 5, we obtain the following insights:

– The registers used by each AES operation are localized for this implementation. For instance, S-box outputs are always stored in `a2` (labeled C), and plaintext bytes always appear in register `t3` (labeled B). Thus, the red lines atop the horizontal line `a2` indicate leakage in the S-box operation.
– *Leakage in C*: From the intermediates, we gather that instructions in the range 2583 to 2624 (labeled C) correspond to S-box operation. TVLA reports leaks at three instruction indices, 2631, 2632, and 2636 (marked in red). The cause of these leakages can be understood by observing the contents of `a2` and `a4` registers at these instructions, illustrated as follows. Here, & is used to denote addresses.

```
2631 add a2,s5,a2     # a2=&(S-box mapping); s5=&state[5]
2632 lbu a2, 0(a2)    # a2=SB5
...
2636 lbu a2, 0(a4)    # a2=state[9]; a4=&state[9]
```

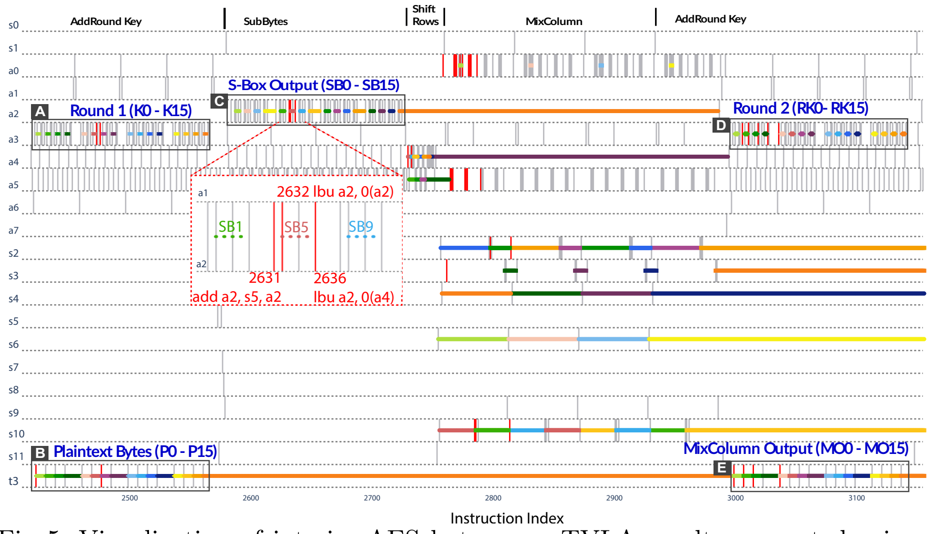


Fig. 5: Visualization of interim AES bytes over TVLA results computed using HD simulated traces

At instruction 2631, `s5` stores the pointer to the S-box mapping in memory, and `a2` stores the 6th byte of input to the S-box. The `add` operation computes the pointer to the resultant S-box output (&`state`[5]), and the value is loaded in `a2` in instruction 2632. In all three instructions, `a2` stores a value that is impacted by a change in `PT5`, thus explaining the TVLA leaks.

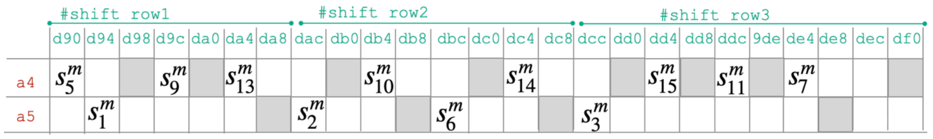
– *Leakage in B,E:* The leaky instructions in B correspond to instruction `0x9bc lbu t3, 0(a4)` that load `PT5` in `t3`. `a4` stores the pointer to the state array, iterates over each byte, and stores the content to `t3`. In E, we observe leaks for the same instruction. The reason for leaks is the loading of MixColumn outputs (`MO0 - MO4`), precisely the bytes impacted by a change in `PT5` (Fig. 3).

– *Leakage in D:* The leaks correspond to instructions `0x9cc xor a3,a3,t3` and `0x9b8 add a3, a6, a5`, that constitute the AddRoundKey operation (Round 2). The reason for leaks is MO bytes, which are impacted by change in `PT5`.

```
3005 add a3,a6,a5     # a6=&state; a5=1 (index)
3010 xor a3,a3,t3     # t3=MO1; a3=RK1
3018 xor a3,a3,t3     # t3=MO2; a3=RK2
3026 xor a3,a3,t3     # t3=MO3; a3=RK3
```

6.2 Case Study: Masked AES

In this case study, we analyze a byte masked implementation of AES, compiled on a PicoRV32 core with `-Os` optimization level. In particular, we focus on the `ShiftRows` function of the implementation as described in 3. As a result of the `ShiftRows` operation, the first state row ($s_0^m, s_4^m, s_8^m, s_{12}^m$) remains



	#shift row1							#shift row2								#shift row3									
	d90	d94	d98	d9c	da0	da4	da8	dac	db0	db4	db8	dbc	dc0	dc4	dc8	dcc	dd0	dd4	dd8	ddc	9de	de4	de8	dec	df0
a4	s_5^m		▨	s_9^m	▨	s_{13}^m			▨	s_{10}^m			▨	s_{14}^m			▨	s_{15}^m	▨	s_{11}^m	▨	s_7^m			▨
a5		s_1^m					▨	s_2^m			▨	s_6^m			▨	s_3^m							▨		

Fig. 6: Visual representation of compiler-generated assembly for the ShiftRows operation in Masked AES at level `-Os`, where s_i^m denotes the i^{th} byte of S-box output. At each PC index, we show the content of the destination registers (`a4`, `a5`). Blank boxes indicate the value is retained till the next load instruction. Grey boxes represent a store operations. Corresponding assembly snippets are in Appendix A.

unchanged. Each byte of the state of the second row $(s_1^m, s_5^m, s_9^m, s_{13}^m)$ is left-shifted by one position to produce $(s_5^m, s_9^m, s_{13}^m, s_1^m)$. Similarly, the state bytes of the third $(s_2^m, s_6^m, s_{10}^m, s_{14}^m)$ row are left-shifted by two yielding $(s_{10}^m, s_{14}^m, s_2^m, s_6^m)$. Finally, the state bytes on the fourth row $(s_3^m, s_7^m, s_{11}^m, s_{15}^m)$ are shifted by three resulting in the byte order $(s_{15}^m, s_3^m, s_7^m, s_{11}^m)$. Figure 6 illustrates the register usage pattern involved in the execution of the generated assembly code. This operation is implemented via a series of load-store operations where the value to be shifted is first loaded into a register. Then, a store operation overwrites the state byte corresponding to the desired shift. The rows in the figure correspond to the registers used in this operation, and the columns correspond to the program counters (PCs).

As shown in Figure 6, only two registers `a4` and `a5` are used. Register `a4` stores temporary variables, while `a5` loads the byte value being shifted. The values s_5^m, s_9^m, and s_{13}^m share the same mask[5]. When loading byte s_9^m in the register containing s_5^m, the effect, shown in eq 4, removes the mask.

$$s_i^m \oplus s_j^m = (s_i \oplus m') \oplus (s_j \oplus m') = s_i \oplus s_j \tag{4}$$

As a result, the code will leak $HD(s_5, s_9)$ and $HD(s_9, s_{13})$. The same effect can be observed when shifting the third row of the state, where the code will leak $HD(s_{10}, s_{14})$, $HD(s_2, s_6)$ and when shifting the fourth row of the state, $HD(s_{15}, s_{11})$, $HD(s_{11}, s_7)$. The code will also leak $HD(s_{13}, s_{10})$, $HD(s_{14}, s_{15})$, $HD(s_1, s_2)$, $HD(s_6, s_3)$. With these leaks, an adversary can recover 12 of the 16 bytes of the total AES key[6].

Side-Channel and Root Cause Analysis. We perform a correlation to illustrate the HD leaks predicted from architectural register usage (Figure 6). Figure 7 depicts the correlation results performed on simulated power traces of masked AES implementation. We have annotated the results to indicate the duration of each AES operation. This experiment aims to testify to the side-channel leaks in the ShiftRows operation of masked AES, a protected implementation. Herein, we correlate the instruction-granular power values with the HD of 6th

[5] As all S-box output boxes share the same mask

[6] The remaining bytes correspond to the first ShiftRow operation s_0, s_4, s_8, s_{12}.

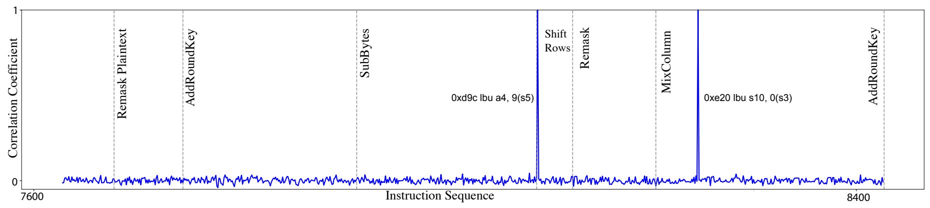


Fig. 7: Correlation results with $HD(s_5^m, s_9^m)$ for the masked AES implementations over 5000 traces where plaintext byte 9 is randomly selected, and all other bytes are fixed. We see a clear leak in the ShiftRow operation.

byte and 10^{th} byte of the SubBytes output. From Figure 6 and Figure 7, we observe a correlation peak of 1 is observed during the ShiftRows operation at instruction `lbu a4, 9(a5)`, which loads SubBytes 10^{th} byte into register `a4`. Since the 6^{th} byte remains in the same register from a prior operation, it leads to an overwrite. It is to be noted that despite considering the content of all registers in the simulated power value, the correlation peak reveals the effect of this overwrite. This is due to the reuse of the same architectural register used in the ShiftRow operation as depicted in Figure 6. We refer the reader to Appendix A for the assembly instructions involved.

6.3 Case Study: Unprotected Ascon

We illustrate the functionalities of `ARCHER` using an open-source unprotected Ascon implementation[7] compiled on the Ibex core[8] with $-$`Os` optimization level. `ARCHER` retrieves addresses of *key*, *plaintext*, *nonce*, and *associated data.*

Side-Channel Analysis. We perform a fixed-vs-fixed TVLA by fixing all the bytes of the nonce except the 0^{th} byte, while the key, associated data, and plaintext are kept fixed. Fig. 8 illustrates the visualization depicting TVLA results, highlighting the leaky instructions in red if they leak as per the HD model. The instructions marked in grey does not leak side-channel information.

Data Flow Analysis. For Ascon, we track bytes of the key (denoted by `K0-K4`), nonce (denoted by `N0-N4`), S-box outputs (denoted by `SB0-SB9`), round outputs (denoted by `RO0-RO9`), associated data (denoted by `A0-A1`), and plaintext (denoted by `M0-M1`). We get the following insights from Fig. 8 showing the first four intermediates, of Round 1 of Initialization:

- The key bytes, `K3`, `K0` and `K1` (at A), are loaded into `s5`, `s6` and `s7` registers where they remain until completion, while the nonce, `N1` and `N0` (at B), loaded at `s10` and `s11` remain until the completion of p^a of Initialization. The rest of the intermediates are spread across 21 out of the 32 RISC-V registers. Unlike the first S-box output, which is present within the range of Round 1

[7] `https://github.com/ascon/ascon-c`
[8] `https://github.com/lowRISC/ibex-demo-system`

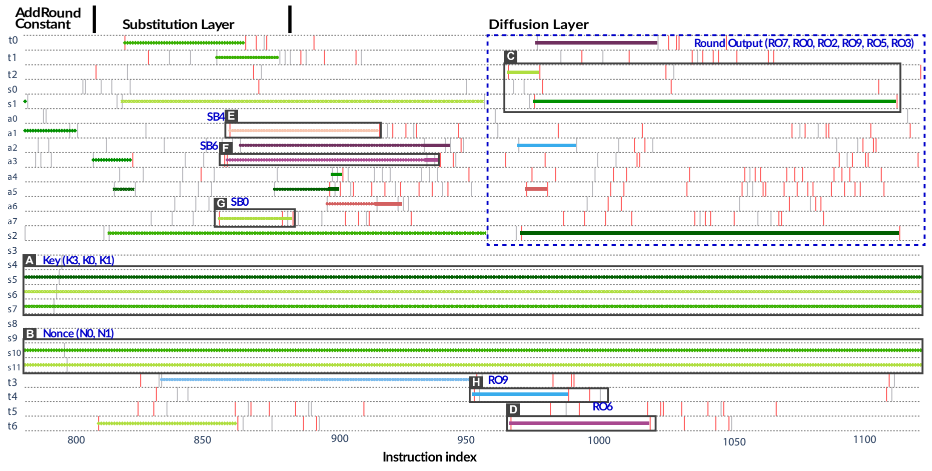


Fig. 8: Visualization of interim bytes of Ascon over TVLA results for HD model

of p^a (instruction index 809 to 905), the round output extends into Round 2 (starting from index 906) for most of its bytes.

– *Leakage at C, D, E, F, and G :* The intermediates RO0 and RO2 (at C), RO3, RO5, RO6 (at D) and SB4 (at E) exhibit leakage for the HD model, when they are either loaded into registers or are overwritten by other values in the registers. In the case of SB6 (at F) and SB0 (at G) in registers `a3` and `a7`, the S-box values get overwritten by RO6 and RO0. For RO0, we see leakage because the value stored in `t2` changes:

```
869 slli t2,a7,4       # t2=Shifted SB0 -> R0 ; a7=SB0
...
966 lw t2,0x4(a0)      # t2=R0; a0=0x18(sp), sp=&Ascon_state
967 lw t6,0x1c(a0)     # t2=&N0; t6=R06
```

– *Leakage at H:* The intermediate RO9 (at H) leaks for the HD model. In register `t4`, RO9 is plotted, coinciding with leaks at L: `xor t4,a4,t4`. Just after the completion of RO9, at N: `not t4,s0`, the HD leaks. The leakage at L and N can be attributed to the change of value in `t4` at $i^{th}+1$ instruction with respect to i^{th} instruction. The disassembled code from Capstone shows that instruction at indices i and $i+1$ updates the state S.x[4], i.e., the output of the linear diffusion layer applied on the nonce, N1, for L. Similarly, at N, using execution trace, we can identify that these instructions correspond to the S-box operation on the IV, K0, K1 and N0. Illustrating N:

```
953 xor t4,a4,t4   #t4=R09; a4=intermediate results
...
989 not t4,s0      # t4=R09
990 and t3,t3,s1   # t4=not(s0); t4=R0(S.x[4]=N1)
```

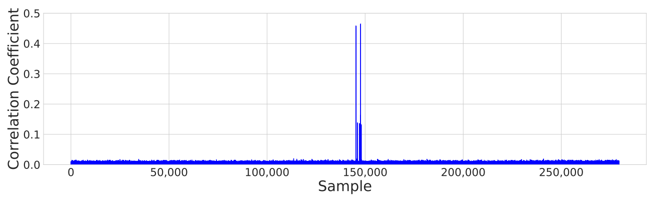


Fig. 9: Correlation results for masked AES using over 10000 power traces.

7 Experimental Validation

To validate the findings in 6.2, we analyze measured power traces collected from an ASIC implementation of the PicoRV32 core. The experiments presented in this section focus on highlighting the side-channel leakage on power traces collected from a real target. The details of our target/hardware setup are as follows:

7.1 Hardware Setup

PicoRV32 is a CPU core that implements the RISC-V RV32IMC Instruction Set. We use the ASIC implementation of the PICO chip on the Saidoyoki board [17] as our target. For power trace collection, a ChipWhisperer-Husky[9] is used, set up on a machine equipped with an Intel Core i7-6700 CPU running Ubuntu 22.04 operating system. The ChipWhisperer-Husky uses synchronous sampling and generates a 5 MHz clock signal for the target while the capture ADC clock frequency is set to 20 MHz. To provide the key, plaintext, and mask to the target device and to receive the corresponding ciphertext, the SimpleSerial[10] v1.1 communication protocol is used via the UART bus.
Test vectors. We built 10,000 test vectors consisting of a fixed 128-bit key, a 128-bit plaintext with 6^{th} and 10^{th} byte randomized, and a random 48-bit mask. The key and plaintext are generated according to the guidelines for random dataset (in Section 5.1 of [1]). The mask is obtained using urandom entropy pool, its 48 bits translate to m_1, m_2, m_3, m_4, m, and m', each 1 byte in length.
Power traces. For the masked implementation, 10,000 power traces—one per test vector—of the entire encryption operation are captured.

7.2 Experimental Validation of ShiftRow Leaks

To validate the leakage in `ShiftRows` operation of masked AES described in 6.2, we correlate the real power value with the Hamming distance between the 6^{th} and 10^{th} byte of the first round S-box output. It is noteworthy that in this experiment, we do not use any information related to the masks.

Figure 9 depicts the correlation plot where we observe two clear peaks with a correlation coefficient around 0.45. This clearly demonstrates that, despite

[9] `https://rtfm.newae.com/Capture/ChipWhisperer-Husky/`
[10] `https://chipwhisperer.readthedocs.io/en/latest/simpleserial.html`

masking being theoretically secure, implementation factors such as register usage can introduce vulnerabilities like byte overwrites. These overwrites effectively cancel the masking, rendering it ineffective in practice. This result is particularly significant given the presence of noise and constant power components (P_{noise} and P_{const}) in the real power traces, (as described in the power consumption model in [20]), highlighting that the architectural component of power leakage remains detectable even in real-world traces.

8 Conclusion and Future Directions

This paper presents ARCHER, an architecture-level simulator for analyzing side-channel vulnerabilities in RISC-V processors. ARCHER integrates binary, side-channel, and data-flow analysis, empowering developers to identify and understand leakage sources at the architecture level. Operating on binary cryptographic implementations independent of specific target hardware, it serves as a foundational tool for early-stage side-channel evaluation. We present case studies using unprotected and protected AES and unprotected Ascon, highlighting the capability of ARCHER to uncover the root causes of side-channel leaks. ARCHER identifies the root cause of a previously undocumented vulnerability in the ShiftRows operation of masked AES, caused by repeated register usage to store sensitive intermediate variables. This leakage is detected by ARCHER and linked to specific instructions and registers, validating the tool's diagnostic utility. As future work, we plan to incorporate advanced side-channel assessment methods, such as mutual information-based techniques, and extend ARCHER to analyze a broader range of implementations. Additionally, automating the identification of leakage causes from visualizations, by detecting patterns in data modifications, is another key research direction.

Acknowledgments. This work was (in part) supported by Dutch Research Council (NWO) through the PROACT project (NWA.1215.18.014), TTW PREDATOR project 19782, TTW BASES project 20858 and the CiCS project of the research programme Gravitation under the grant 024.006.037.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

A Appendix: Architectural register leaks of ShiftRow operation of Masked AES

```
0xd8c bne  a5,a3,LAB_00000d54
# shift row 1 of state
0xd90 lbu  a4,0x5(s5) #load in a4, state(5)
0xd94 lbu  a5,0x1(s5) #load in a5, state(1)
0xd98 sb   a4,0x1(s5) #store the content of a4 to state(1)
0xd9c lbu  a4,0x9(s5) #load in a4, state(9)
```

```
0xda0 sb   a4,0x5(s5) #store the content of a4 to state(5)
0xda4 lbu  a4,0xd(s5) #load in a4, state(13)
0xda8 sb   a5,0xd(s5) #store the content of a5 to state(13)
# shift row 2 of state
0xdac lbu  a5,0x2(s5) #load in a5, state(2)
0xdb0 sb   a4,0x9(s5) #store the content of a4 to state(9)
# finish row 1
0xdb4 lbu  a4,0xa(s5) #load in a4, state(10)
0xdb8 sb   a5,0xa(s5) #store the content of a5 to state(10)
0xdbc lbu  a5,0x6(s5) #load in a5, state(6)
0xdc0 sb   a4,0x2(s5) #store the content of a4 to state(2)
0xdc4 lbu  a4,0xe(s5) #load in a4, state(14)
0xdc8 sb   a5,0xe(s5) #store the content of a5 to state(14)
# shift row 3 of state
0xdcc lbu  a5,0x3(s5) #load in a5, state(3)
0xdd0 sb   a4,0x6(s5) #store the content of a4 to state(6)
# finish row 2
0xdd4 lbu  a4,0xf(s5) #load in a4, state(15)
0xdd8 sb   a4,0x3(s5) #store the content of a4 to state(3)
0xddc lbu  a4,0xb(s5) #load in a5, state(11)
0x9de sb   a4,0xf(s5) #store the content of a4 to state(15)
0xde4 lbu  a4,0x7(s5) #load in a5, state(7)
0xde8 sb   a5,0x7(s5) #store the content of a5 to state(7)
0xdec li   a5,0xa     # load value 0xa to a5
0xdf0 sb   a4,0xb(s5) #store the content of a4 to state(11)
```

Listing 1.1: Assembly snippet of Shift-Row operation in Masked AES compiled with optimization level `-Os`.

References

1. Test Vector Leakage Assessment (TVLA) Derived Test Requirements (DTR) with AES (Aug 2015), `https://www.rambus.com/test-vector-leakage-assessment-tvla-derived-test-requirements-dtr-with-aes/`
2. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.: Jasmin: High-assurance and high-speed cryptography. In: ACM CCS 2017. pp. 1807–1823. ACM (2017). `https://doi.org/10.1145/3133956.3134078`
3. Arora, V., Buhan, I., Perin, G., Picek, S.: A tale of two boards: On the influence of microarchitecture on side-channel leakage. In: CARDIS 2021. LNCS, vol. 13173, pp. 80–96. Springer (2021). `https://doi.org/10.1007/978-3-030-97348-3_5`
4. Barthe, G., Belaïd, S., Fouque, P., Grégoire, B.: maskverif: a formal tool for analyzing software and hardware masked implementations. IACR Cryptol. ePrint Arch. p. 562 (2018), `https://eprint.iacr.org/2018/562`
5. Barthe, G., Böhme, M., Cauligi, S., Chuengsatiansup, C., Genkin, D., Guarnieri, M., Romero, D.M., Schwabe, P., Wu, D., Yarom, Y.: Testing side-channel security of cryptographic implementations against future microarchitectures. In: ACM CCS 2024. pp. 1076–1090. ACM (2024). `https://doi.org/10.1145/3658644.3670319`

6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easycrypt: A tutorial. In: FOSAD 2012/2013 Tutorial Lectures. LNCS, vol. 8604, pp. 146–166. Springer (2013). https://doi.org/10.1007/978-3-319-10082-1_6
7. Basin, D.A., Cremers, C., Dreier, J., Sasse, R.: Tamarin: Verification of large-scale, real-world, cryptographic protocols. IEEE S&P **20**(3), 24–32 (2022). https://doi.org/10.1109/MSEC.2022.3154689
8. Bazangani, O., Iooss, A., Buhan, I., Batina, L.: ABBY: automating leakage modelling for side-channel analysis. In: ACM ASIA CCS 2024. ACM (2024). https://doi.org/10.1145/3634737.3637665
9. Buhan, I., Batina, L., Yarom, Y., Schaumont, P.: Sok: Design tools for side-channel-aware implementations. In: ASIA CCS. pp. 756–770 (2022). https://doi.org/10.1145/3488932.3517415
10. Corre, Y.L., Großschädl, J., Dinu, D.: Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In: COSADE 2018. LNCS, vol. 10815, pp. 82–98. Springer (2018). https://doi.org/10.1007/978-3-319-89641-0_5
11. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002). https://doi.org/10.1007/978-3-662-04722-4
12. F, M.A.K., Ganesan, V., Bodduna, R., Rebeiro, C.: PARAM: A microprocessor hardened for power side-channel attack resistance. In: 2020 IEEE HOST 2020. pp. 23–34. IEEE (2020). https://doi.org/10.1109/HOST45689.2020.9300263
13. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-design and co-verification of masked software implementations on cpus. In: USENIX Security 2021. pp. 1469–1468. USENIX Association (2021)
14. Goodwill, G., Jun, J., P.Rohatgi: A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop (2018)
15. de Grandmaison, A., Heydemann, K., Meunier, Q.L.: ARMISTICE: microarchitectural leakage modeling for masked software formal verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(11), 3733–3744 (2022). https://doi.org/10.1109/TCAD.2022.3197507
16. Karl, P., Schupp, J., Fritzmann, T., Sigl, G.: Post-quantum signatures on RISC-V with hardware acceleration. ACM TECS **23**(2), 30:1–30:23 (2024). https://doi.org/10.1145/3579092
17. Kiaei, P., Liu, Z., Eren, R.K., Yao, Y., Schaumont, P.: Saidoyoki: Evaluating side-channel leakage in pre- and post-silicon setting. IACR Cryptol. ePrint Arch. p. 1235 (2021), https://eprint.iacr.org/2021/1235
18. Liu, Q., Zhang, C., Ma, L., Jiang, M., Zhou, Y., Wu, L., Shen, W., Luo, X., Liu, Y., Ren, K.: Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In: ASE 2021. pp. 792–804. IEEE (2021). https://doi.org/10.1109/ASE51524.2021.9678653
19. Liu, Z., Schaumont, P.: Root-cause analysis of the side channel leakage from ascon implementations (2023)
20. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
21. Marshall, B., Page, D., Webb, J.: MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. IACR TCHES **2022**(1), 175–220 (2022). https://doi.org/10.46586/TCHES.V2022.I1.175-220

22. McCann, D., Oswald, E., Whitnall, C.: Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In: USENIX Security 2017. pp. 199–216. USENIX Association (2017)
23. Meyer, L.D., Mulder, E.D., Tunstall, M.: On the effect of the (micro)architecture on the development of side-channel resistant software. IACR Cryptol. ePrint Arch. p. 1297 (2020), https://eprint.iacr.org/2020/1297
24. Miteloudi, K., Bos, J.W., Bronchain, O., Fay, B., Renes, J.: PQ.V.ALU.E: post-quantum RISC-V custom ALU extensions on dilithium and kyber. In: CARDIS 2023. LNCS, vol. 14530, pp. 190–209. Springer (2023). https://doi.org/10.1007/978-3-031-54409-5_10
25. Olmos, S.A., Barthe, G., Blatter, L., Grégoire, B., Laporte, V.: Preservation of speculative constant-time by compilation. Proc. ACM Program. Lang. **9**(POPL), 1293–1325 (2025). https://doi.org/10.1145/3704880
26. Olmos, S.A., Barthe, G., Gonzalez, R., Grégoire, B., Laporte, V., Léchenet, J., Oliveira, T., Schwabe, P.: High-assurance zeroization. IACR TCHES **2024**(1), 375–397 (2024). https://doi.org/10.46586/TCHES.V2024.I1.375-397
27. Papagiannopoulos, K., Veshchikov, N.: Mind the gap: Towards secure 1st-order masking in software. In: COSADE 2017. LNCS, vol. 10348, pp. 282–297. Springer (2017). https://doi.org/10.1007/978-3-319-64647-3_17
28. Samwel, N., Daemen, J.: DPA on hardware implementations of ascon and keyak. In: Proceedings of the Computing Frontiers Conference. pp. 415–424. ACM (2017)
29. Sehatbakhsh, N., Yilmaz, B.B., Zajic, A.G., Prvulovic, M.: Emsim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals. In: IEEE HPCA 2020. pp. 71–85. IEEE (2020). https://doi.org/10.1109/HPCA47549.2020.00016
30. Shelton, M.A., Chmielewski, L., Samwel, N., Wagner, M., Batina, L., Yarom, Y.: Rosita++: Automatic higher-order leakage elimination from cryptographic code. In: ACM CCS 2021. pp. 685–699. ACM (2021). https://doi.org/10.1145/3460120.3485380
31. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In: NDSS 2021. The Internet Society (2021)
32. Sönmez Turan, M., McKay, K., Chang, D., Kang, J., Kelsey, J.: Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions. Tech. rep., NIST (2024)
33. Stoffelen, K.: Efficient cryptography on the RISC-V architecture. In: Progress in Cryptology - LATINCRYPT 2019. LNCS, vol. 11774, pp. 323–340. Springer (2019). https://doi.org/10.1007/978-3-030-30530-7_16
34. Vouvoutsis, V., Casino, F., Patsakis, C.: On the effectiveness of binary emulation in malware classification. J. Inf. Secur. Appl. **68**, 103258 (2022). https://doi.org/10.1016/J.JISA.2022.103258
35. Weissbart, L., Picek, S.: Lightweight but not easy: Side-channel analysis of the ascon authenticated cipher on a 32-bit microcontroller. IACR Cryptol. ePrint Arch. p. 1598 (2023), https://eprint.iacr.org/2023/1598
36. Wichelmann, J., Peredy, C., Sieck, F., Pätschke, A., Eisenbarth, T.: MAMBO-V: dynamic side-channel leakage analysis on RISC-V. In: DIMVA 2023. LNCS, vol. 13959, pp. 3–23. Springer (2023). https://doi.org/10.1007/978-3-031-35504-2_1