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

Asmita Adhikary
Radboud University
Nijmegen, The Netherlands
asmita.adhikary@ru.nl

Abraham J. Basurto Becerra
Radboud University
Nijmegen, The Netherlands
abraham.basurto@ru.nl

Lejla Batina
Radboud University
Nijmegen, The Netherlands
lejla@cs.ru.nl

Ileana Buhan
Radboud University
Nijmegen, The Netherlands
ileana.buhan@ru.nl

Durba Chatterjee
Radboud University
Nijmegen, The Netherlands
durba.chatterjee@ru.nl

Senna Van Hoek
Radboud University
Nijmegen, The Netherlands
senna.vanhoek@ru.nl

Eloi Sanfelix Gonzalez
Binary Gecko
Nijmegen, The Netherlands
eloi@limited-entropy.com

***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 AES and ASCON implementations, where it accurately traces the source of side-channel leaks.

***Index Terms*—Side-Channel Analysis, RISC-V, Pre-silicon, Data Flow Analysis, Qiling**

I. 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 [1], [2]. Conversely, fixes for microarchitectural leaks typically target the specific components involved, such as employing fence instructions for speculative executions or adopting constant-time programming to mitigate cache timing attacks [3]. With the advent of open hardware, the appeal of developing cryptographic algorithms for RISC-V architectures has increased [4], [5], [6], while the tools to support secure implementations are few [7]. Existing power leakage simulator available for RISC-V are geared towards verifying hardware implementations [8], [9], [10].

Related Work. The landscape of cryptographic verification tools is fragmented. At high abstraction levels, tools such as Tamarin [11], EasyCrypt [12], or MaskVerif [13] assist in creating security proofs. Secure compilers like Jasmin [14] 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 [7]. Papagiannopoulos et al. [15] were among the first to discuss the microarchitectural effects when analyzing side-channel leaks for software implementations. De Meyer et al. [16] and Arora et al. [17] continue their work and discuss additional microarchitecture leakage effects on different target devices.

McCann et al. [18] 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. [19] 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 [20] 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* 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 [22], 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*.

Our Contributions. We propose `ARCHER`, a design tool that focuses on architectural-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) `ARCHER` 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. *We plan to make the tool open-source upon publication.*
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 visualizations and reports generated by `ARCHER` in the URL: https://anonymous.4open.science/r/ARCHER-D744/.
5) We demonstrate the working of the tool and the derived insights using AES and 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.

II. Preliminaries

Notation. We denote an architectural register as r_i, where i is a number from the set $\{1, \dots, m\}$. 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, $\{I_1, \dots, I_N\}$ for a given input, where N represents the total number of executed instructions. For each instruction

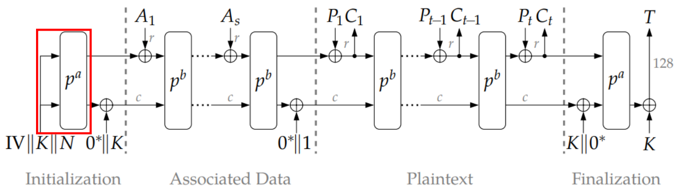

Figure 1: Schematic representation of ASCON encryption [24]

I_j we store the state of all architectural registers $\{r_1^j, \dots, r_m^j\}$[1]. We briefly describe the concepts required in this work.

Test Vector Leakage Assessment (TVLA) [23] 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 [25]. 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.

ASCON-128 AEAD (authenticated encryption with associated data) [24], 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 SPN-based 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
$$\begin{aligned} S = x_0|||x_1|||x_2|||x_3|||x_4 &= IV|||K|||N \\ &= IV|||K0|||K1|||N0|||N1. \end{aligned}$$
2) *Substitution layer* (p_S): A 5-bit S-Box ($S(x)$) is applied to each bit-slice of the five state registers $x_0, \dots, x_4$.
3) *Linear diffusion layer* (p_L): Adds diffusion via a 64-bit linear function $\Sigma_i(x_i)$.

[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 algorithm implementation, it is possible that other inputs, such as nonce, masks, may need to be provided.

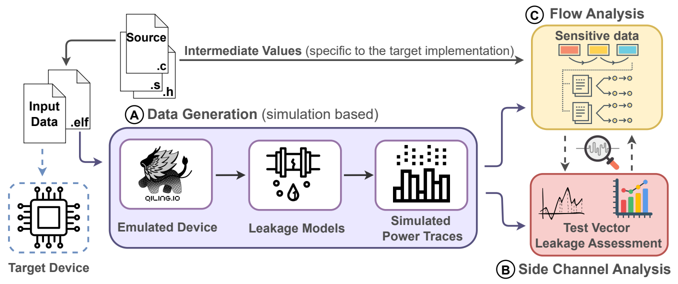

Figure 2: Overview of side-channel architecture level simulator for RISC-V (ARCHER)

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 [26]. As shown in [27], [28], these intermediates facilitate successful retrieval of key.

III. 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 as follows:

A. Data Generation

This module generates simulated power traces for input sets using the Qiling framework [29], executing the binary file with provided data. The number and size of inputs are determined by the cryptographic algorithm, as specified by the user. 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.

B. Side-Channel Analysis (SCA)

This component is responsible for the identification of side-channel leakages in an implementation. ARCHER uses Test Vector Leakage Assessment (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.
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 Hamming Weight (HW), Hamming Distance (HD), and Identity (ID). HD leakage model assumes the adversary can observe any pairwise combination of the intermediate values. HW leakage model assumes the adversary can observe the HW of a cryptographic algorithm's set of intermediate values.
4) *TVLA Analysis.* ARCHER uses TVLA for detecting leaks in two modes:
   - *Fixed-vs-Random*: This configuration 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, as depicted in 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, as illustrated in Fig. 5 and Fig. 6.

C. 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. 6. 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, or other critical data) typically form the focus of side-channel analysis. 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.

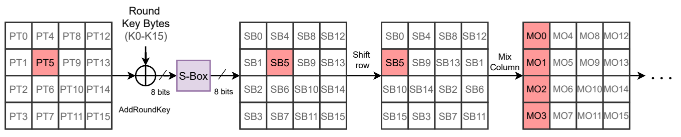

Figure 3: Schematic representation of AES operations highlighting the bytes impacted on modifying 6^{th} byte of plaintext

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.

IV. ARCHER Implementation Details

This section describes the implementation details of ARCHER, followed by its functioning using the example of an open-source AES implementation [30] compiled for a RISC-V core, PicoRV32 [31]. The tool is implemented in Python3.

Target Device Setup. To obtain the compiled binary, i.e., the `.elf` file for the target device, we cross-compile the AES implementation on a machine with the RISC-V GNU Compiler Toolchain (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.

Simulation Setup. The simulation setup takes the AES `.elf` file and the number of traces to be generated as input parameters. It retrieves the address of the essential elements of the compiled binary, such as the addresses of the *key* and the *plaintext* for AES. As parameters vary between schemes and implementation, some configuration may be needed to adapt the simulation code and define how parameters are provided. ARCHER leverages the Qiling emulation framework to run the same binary as would be flashed to a target board. 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 disassembled instructions and register states are stored in `.csv` files.

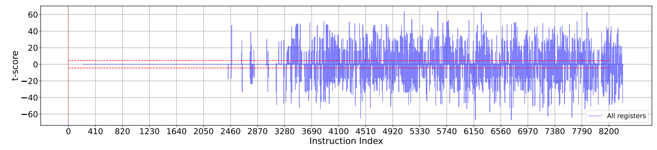

Figure 4: t-score for simulated traces using HD model

Data Generation for Side-channel Analysis. During simulation, we use the same set of plaintexts and keys used in the target device. In the running example, we compute the *fixed-vs-fixed* TVLA to highlight leakage caused by a one-byte change. Two datasets are generated, differing only in the 6^{th} 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 processed by Qiling using the `.elf` file, generating one execution trace for the fixed input and 500 traces for the varied inputs. Each trace captures the sequence of 8434 instructions. For this example, let us choose HD as the leakage model.

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 visualization 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 involves generating intermediates for tracking and overlapping it 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) for this step. 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

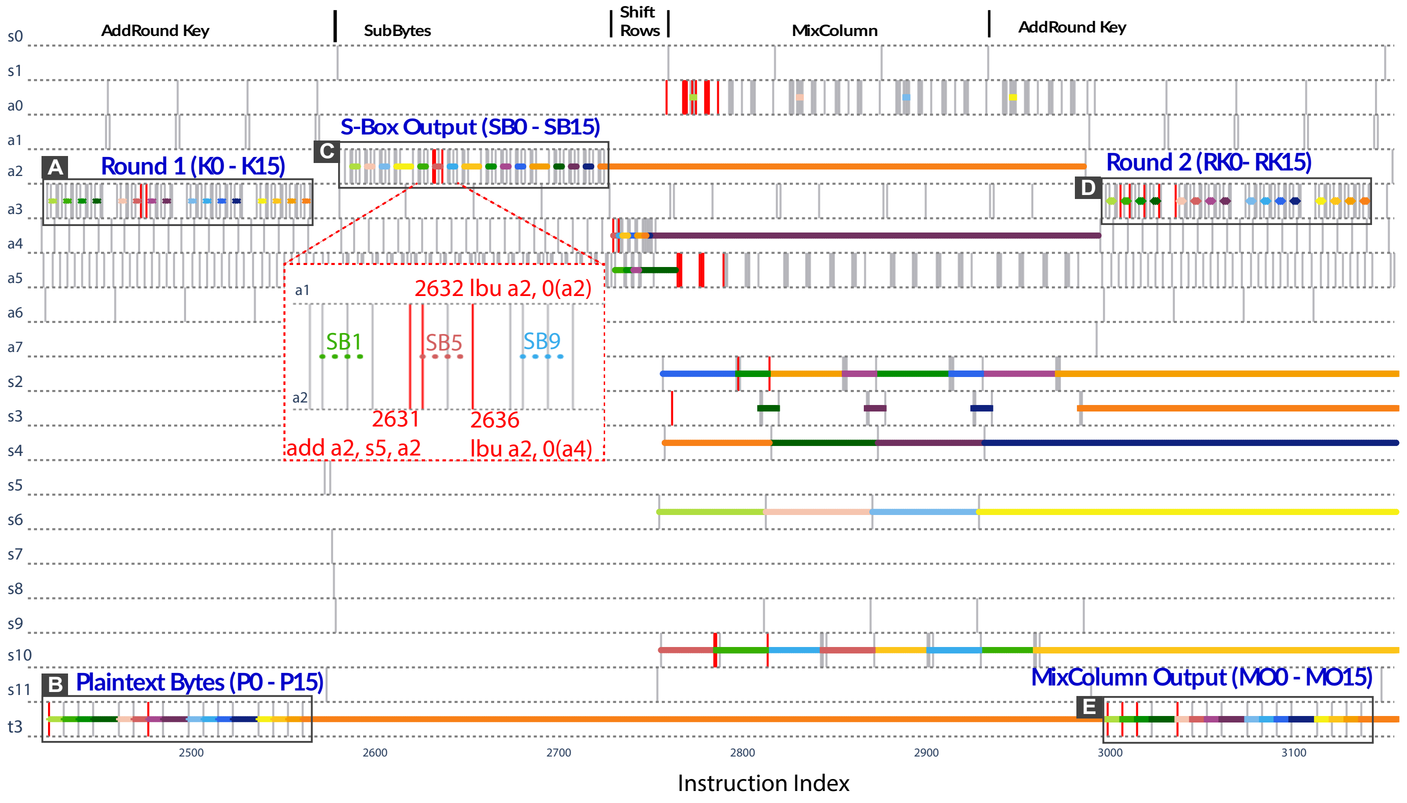

Figure 5: Visualization of interim AES bytes over TVLA results

illegitimate instruction indices where the intermediate byte might appear due to unrelated or independent computations. The final visualizations are generated using the filtered indices. 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.

V. Insights from ARCHER

We apply ARCHER to two unprotected cryptographic implementations of AES and ASCON with different mathematical structures. While AES is still the most used symmetric algorithm in real-world applications, ASCON is a recent standard for authenticated encryption and a permutation-based cryptosystem. We demonstrate the tool first on two unprotected cryptographic implementations, as these implementations will leak side-channel information, making them an attractive choice for a tool aimed at root-cause analysis.

A. Case Study: Unprotected AES

We examine first an unprotected implementation of AES-128 [25]. We continue with the TVLA results obtained in Fig. 4. To understand the source of leakage, we first identify the bytes impacted on 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 this visualization depicted in 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` (box labeled C), and plaintext bytes always appear in register `t3` (box labeled B). Thus, the red lines atop the horizontal line `a2` indicate leakage in the S-Box operation.
- *Leakage in Box C*: From the intermediates, we gather that instructions in the range 2583 to 2624 (Box 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. In the following code listings, & is used to denote addresses.

```
2631 add a2,s5,a2     #a2=&SB; s5=&state[5]
2632 lbu a2, 0(a2)    #a2=SB5
...
```

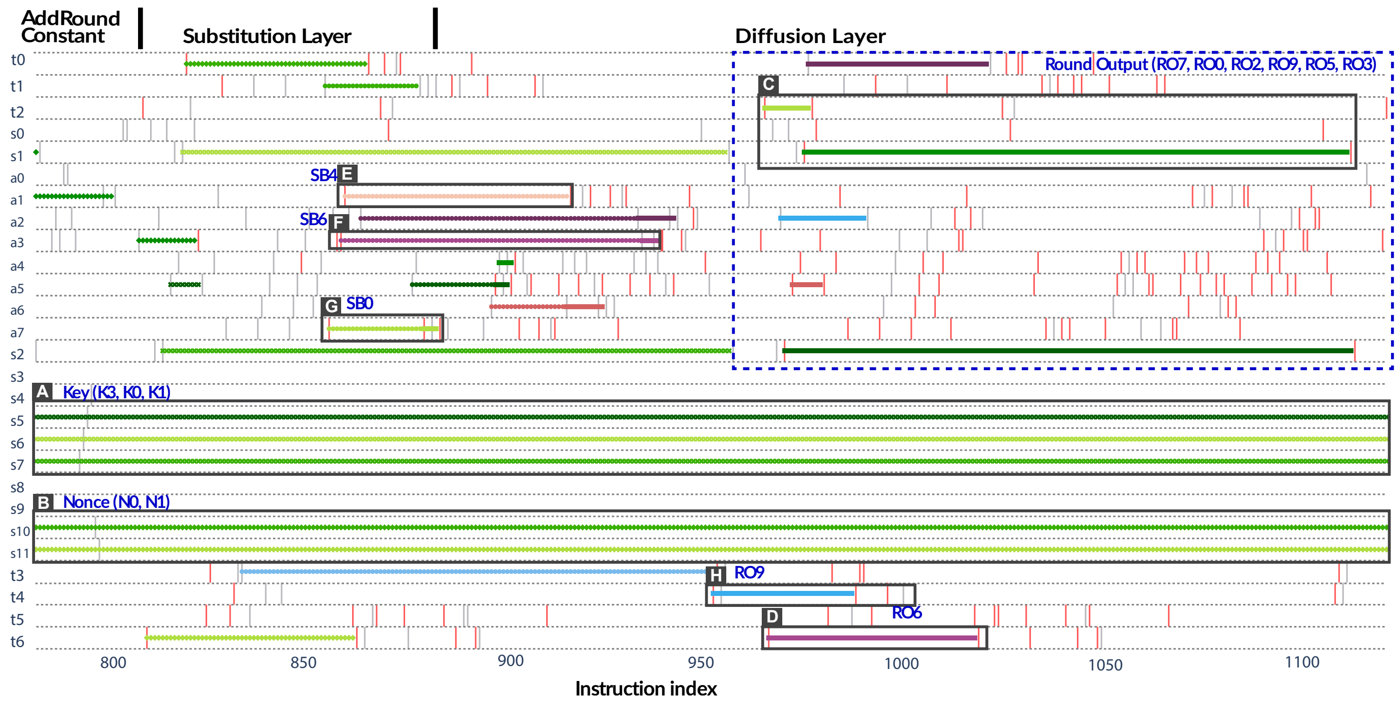

Figure 6: Visualization of interim bytes of ASCON over TVLA results for HD model

```
2636 lbu a2, 0(a4)    #a2=state[9]; a4=&state[9]
```

Here, `s5` stores the pointer to the SBox mapping in memory, and `a2` stores the input to the SBox. The `add` operation computes the pointer to the resultant SBox output. In all three instructions, `a2` stores a value that is impacted by a change in `PT5`, thus explaining the TVLA leaks.

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

B. Case Study: Unprotected ASCON

We illustrate the functionalities of ARCHER using an open-source unprotected ASCON implementation [32] compiled on the Ibex core [33] with −Os optimization level. We follow a similar approach to generate the `.csv` files for ASCON. For ASCON, ARCHER retrieves the addresses of the *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. For ASCON, each execution trace comprises 8629 instructions. Fig. 6 illustrates the interactive visualization depicting TVLA results, highlighting the leaky instructions in red if they leak as per the HD leakage model.

Data Flow Analysis. For ASCON, we track key bytes (K0-K4), nonce (N0-N4), S-Box outputs (SB0-SB9), round outputs (RO0-RO9), associated data (A0-A1), and plaintext (M0-M1). For this use case, we focus on Round 1 of Initialization. The first four intermediates are shown illustrated as markers in Fig. 6. We get the following insights from the interactive visualization:

- 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 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:

```
965 c.lw a3,0x24(a0) #t2=RO for S.x[0]=IV; a0=0x18(sp
    ), sp=&ASCON_state
966 lw t2,0x4(a0) #t2=&N0; a0=0x18(sp),
sp=&ASCON_state
```

- *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 w.r.t. 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:

```
989 not t4,s0 #t4=initial results of S-Box
990 and t3,t3,s1 #t4=not(s0); t4=RO(S.x[4]=N1)
```

VI. Conclusion and Future Directions

This paper presents ARCHER, an architecture-level simulator for systematically analyzing side-channel vulnerabilities in RISC-V processors. ARCHER integrates side-channel and flow analysis, enabling developers to identify and understand leakage sources at the architecture level. It operates on binary cryptographic implementations independent of target hardware. Using AES and ASCON as case studies, we demonstrate ARCHER's ability to uncover the root causes of side-channel leaks. As future work, we plan to incorporate advanced side-channel assessment methods, such as mutual information-based techniques, and extend ARCHER to analyze protected implementations. Additionally, automating leakage cause identification from visualizations by detecting patterns in data modifications is another key research direction.

REFERENCES

[1] Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. High-assurance zeroization. *IACR TCHES*, 2024:375–397.

[2] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*, 2021.

[3] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. Testing side-channel security of cryptographic implementations against future microarchitectures. *CoRR*, abs/2402.00641, 2024.

[4] Ko Stoffelen. Efficient cryptography on the risc-v architecture. In *LATINCRYPT 2019*, page 323–340. Springer-Verlag, 2019.

[5] Konstantina Miteloudi, Joppe W. Bos, Olivier Bronchain, Björn Fay, and Joost Renes. PQ.V.ALU.E: post-quantum RISC-V custom ALU extensions on dilithium and kyber. In *CARDIS*, volume 14530 of *Lecture Notes in Computer Science*, pages 190–209. Springer, 2023.

[6] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on risc-v with hardware acceleration. *ACM TECS*, 23(2), 2024.

[7] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. Sok: Design tools for side-channel-aware implementations. In *ASIA CCS*, pages 756–770, 2022.

[8] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka G. Zajic, and Milos Prvulovic. EMSim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals. In *HPCA*, pages 71–85, 2020.

[9] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}. In *USENIX Security*, pages 1469–1468, 2021.

[10] Muhammad Arsath K F, Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. Param: A microprocessor hardened for power side-channel attack resistance. In *HOST*, pages 23–34, 2020.

[11] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of large-scale, real-world, cryptographic protocols. *IEEE S& P*, 20(3):24–32, 2022.

[12] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[13] Gilles Barthe, Sonia Belaïd, Pierre-Alain Fouque, and Benjamin Grégoire. maskverif: a formal tool for analyzing software and hardware masked implementations. *IACR Cryptol. ePrint Arch.*, page 562, 2018.

[14] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *CCS*, pages 1–17, 2017.

[15] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *COSADE 2017*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

[16] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. Cryptology ePrint Archive, Report 2020/1297, 2020. https://eprint.iacr.org/2020/1297.

[17] Vipul Arora, Ileana Buhan, Guilherme Perin, and Stjepan Picek. A tale of two boards: On the influence of microarchitecture on side-channel leakage. In *CARDIS 2021*, volume 13173 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2021.

[18] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *USENIX*, pages 199–216. USENIX Association, 2017.

[19] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*, 2021.

[20] Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. ABBY: automating leakage modelling for side-channel analysis, 2024.

[21] Ben Marshall, Dan Page, and James Webb. MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR TCHES*, 2022(1):175–220, 2022.

[22] Jan Wichelmann, Christopher Peredy, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. MAMBO-V: dynamic side-channel leakage analysis on RISC-V. In Daniel Gruss, Federico Maggi, Mathias Fischer, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*, volume 13959 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2023.

[23] G. Goodwill, J.J.B. Jun, and P.Rohatgi. A testing methodology for side channel resistance validation. *NIST non-invasive attack testing workshop*, 2018.

[24] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *JoC*, 34(3), 2021.

[25] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Information Security and Cryptography. Springer, 2002.

[26] Zhenyu Liu and Patrick Schaumont. Root-cause analysis of the side channel leakage from ascon implementations. 2023.

[27] Niels Samwel and Joan Daemen. DPA on hardware implementations of ascon and keyak. In *Proceedings of the Computing Frontiers Conference*, pages 415–424. ACM, 2017.

[28] Léo Weissbart and Stjepan Picek. Lightweight but not easy: Side-channel analysis of the ascon authenticated cipher on a 32-bit microcontroller. *IACR Cryptol. ePrint Arch.*, page 1598, 2023.

[29] Qiling framework. https://qiling.io/, 2023-08-04. Accessed: 2024-04-17.

[30] CENSUS. Masked aes.

[31] YosysHQ. Yosyshq/picorv32: Picorv32 - a size-optimized risc-v cpu.

[32] Reference, highly optimized, masked c and asm implementations of ascon.

[33] lowRISC. Lowrisc/ibex-demo-system: A demo system for ibex including debug support and some peripherals. accessed 15-10-2023.