Plan your defense: A comparative analysis of leakage detection methods on RISC-V cores

Konstantina Miteloudi, Asmita Adhikary, Niels van Drueten, Lejla Batina, and Ileana Buhan
Digital Security Group, Radboud University
Nijmegen, The Netherlands
kmiteloudi@cs.ru.nl, asmita.adhikary@ru.nl, niels@vandrueten.nl, lejla@cs.ru.nl, ileana.buhan@ru.nl

***Abstract*—Hardening microprocessors against side-channel attacks is a critical aspect of ensuring their security. A key step in this process is identifying and mitigating "leaky" hardware modules, which inadvertently leak information during the execution of cryptographic algorithms. In this paper, we explore how different leakage detection methods, the Side-channel Vulnerability Factor (SVF) and the Test Vector Leakage Assessment (TVLA), contribute to hardening of microprocessors. We conduct experiments on two RISC-V cores, SHAKTI and Ibex, using two cryptographic algorithms, SHA-3 and AES. Our findings suggest that SVF and TVLA can provide valuable insights into identifying leaky modules. However, the effectiveness of these methods can vary depending on the specific core and cryptographic algorithm in use. We conclude that the choice of leakage detection method should be based not only on computational cost but also on the specific requirements of the system and the nature of the potential threats. Our research contributes to developing more secure microprocessors that are robust against side-channel attacks.**

***Index Terms*—Pre-Silicon Design, Side-Channel Analysis, Leakage Detection Methods, Hardening Microprocessors**

I. Introduction

Microprocessors are at the heart of modern digital systems, from everyday consumer electronics to critical infrastructure. Ensuring their security against various forms of attacks is crucial. One such form of attack, side-channel attacks, exploits information leaked during the execution of cryptographic algorithms, potentially compromising the system's security. As such, a key step in hardening microprocessors against side-channel attacks is identifying and mitigating these "leaky" hardware modules.

Several methods exist for detecting such leakages, each with strengths and weaknesses. Two methods, the Side-channel Vulnerability Factor (SVF) [1] and the Test Vector Leakage Assessment (TVLA) [2], have been commonly used in the field. SVF provides a measure of the potential exploitability of a leak, while TVLA offers a statistical framework for identifying whether a device under test is susceptible to information leakage. However, the effectiveness of these methods can vary depending on the specific hardware and algorithm in use [3], [4]. In this work, we applied these leakage assessment methods on two RISC-V cores: SHAKTI and Ibex. These cores, popular in various applications for their open-source nature, present a range of opportunities and challenges for security.

Our research question is: "In which ways can two common leakage detection methods contribute to hardening of microprocessors against side-channel attacks?". We aim to provide insights into how SVF and TVLA can be used to identify vulnerabilities in different RISC-V cores and how these findings can enforce the best strategies for hardening these cores. Through our experiments with different cryptographic algorithms, i.e. SHA-3 and AES, we seek to offer a comprehensive understanding of the ways these leakage detection methods can contribute to the hardening of RISC-V cores. Our findings will be of interest to security designers and architects, contributing to the development of more secure microprocessors/tools [5].

II. Related work

Several approaches have been developed to identify and eliminate leaks during the pre-silicon phase [6]. These approaches can be roughly divided based on which device layer or development phase they aim to harden [7]. The level of leakage can be detected at top architectural choices [8] or/and can emerge from the micro-architectural behaviour [9].

De Mulder et al. [10] proposed a solution to protect an AES implementation against side channel leakage related to memory accesses on a RISC-V core. Gigerl et al. [11] introduced COCO, a tool that can detect gate-level leakage by simulating execution with Verilator. They annotate the registers and memory that hold secret data and trace their flow through the circuit to find possible sources of leakage. He et al. [12] estimate the power profile of a hardware design using functional simulation at the RTL level. Gao et al. [13] designed and implemented an ISE (Instruction Set Extension) called FENL that localizes and reduces microarchitectural leakage. The ISE acts as a leakage fence that prevents interaction between instructions. A similar approach is taken by Pham et al. [14], which combines a diversified ISE with hardware diversification through a co-processor to achieve leakage mitigation. Bloem et al. [15] extended the concept of hardware-software contracts to power-side channels and formally verified a wide range of instructions for implementing cryptographic algorithms for the RISC-V Ibex core. ACA [16] uses a gate-level model for a target design, typically available after logic synthesis and a side-channel leakage model. Kiaei and Schaumont proposed Root Canal [17], a framework to help a designer with white-box access to the embedded CPU system uncover the origin of a side-channel leak. Root Canal can eliminate side-channel leaks before tape-out. After tape-out, changes to the hardware are no longer possible.

To our knowledge, none of the previous work investigates the impact of the leakage detection methods on identifying leaky modules. Arsath et al. [18] developed a framework, PLAN, that analyzes the RTL description of a processor and reports the information leakage in each of the processor modules. In PLAN, they use a modified version of SVF as a leakage detection method, and they apply this method to a simulated RISC-V core running different cryptographic algorithms. The results of their analysis provide a ranking of the hardware modules based on their contribution to the overall leakage. In our work, we replicate the experimental setup of [18], applying both the SVF and the TVLA methods to compare their effectiveness in identifying leaky modules. This replication is the foundation for further exploration and comparison of these two leakage detection methods.

III. Preliminaries

A. Leakage detection methods

Let $\mathcal{X}(N,d)$ be a set of N traces. A *trace* is a time series with d samples recorded during the processing of an algorithm on a given device (e.g., an encryption operation) for a given input x.

Test Vector Leakage Assessment (TVLA) [2] is the most popular leakage detection method due to its simplicity and relative effectiveness. It comes in two flavours: *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, the traces are divided according to a known intermediate value tested for leakage. Welch's two-sample t-value for equality of means applies to all trace samples in both cases. A difference between two sets larger than a given threshold is evidence of a leak's presence.

Side-channel Vulnerability Factor (SVF) [1] measures side-channel information leakage by recognizing leaked execution patterns. SVF quantifies the similarity between patterns in the observations of the attackers or *side-channel traces* ($\mathcal{S}$, defined in (1), with the actual execution patterns of the victim or *oracle traces* ($\mathcal{O}$, defined in (2).

$$\mathcal{S} = \{(x_i; s^i_1, s^i_2, ...s^i_d), \text{where } 1 \le x_i \le N\} \tag{1}$$

$$\mathcal{O} = \{(x_i; o^i_1, o^i_2, ...o^i_d), \text{where } 1 \le x_i \le N\} \tag{2}$$

where s^i_j (and o^i_j) is sample j in side channel trace (and oracle trace, respectively) corresponding to input x_i.

The original version of the SVF algorithm proposed in [1], which we denote with SVF$_{\text{time}}$, quantifies *patterns in the time-domain* between an oracle and a side-channel trace. Arsath et al. [18] implemented a modified version of SVF$_{\text{input}}$ that is adapted to capture *patterns related to changes in the input data*, the typical cause of side-channel vulnerabilities. After data collection, both algorithms extract patterns in parallel for the oracle and side-channel trace. The difference between the implementation of SVF$_{\text{time}}$ and SVF$_{\text{input}}$ is apparent in the construction of the similarity matrices as shown in equation (3), (4) and (5), (6) respectively.

$$M^{\mathcal{S}}_{\text{time}}(s^i_j, s^i_k) = \begin{cases} D(s^i_j, s^i_k), & \text{if } j < k \\ 0, & \text{if not.} \end{cases} \tag{3}$$

and

$$M^{\mathcal{O}}_{\text{time}}(o^i_j, o^i_k) = \begin{cases} D'(o^i_j, o^i_k), & \text{if } j < k \\ 0, & \text{if not.} \end{cases} \tag{4}$$

When computing SVF$_{\text{time}}$, the first step is to construct $M^{\mathcal{S}}_{\text{time}}$, the similarity matrix for a side-channel trace, using equation (3), where D is a distance (of choice) between samples s^i_j, s^i_k. The next step is to compute the similarity matrix, $M^{\mathcal{O}}_{\text{time}}$, for the oracle trace by computing the distance D' between the samples in the same power trace o_i, o_j, using equation (4). The correlation between $M^{\mathcal{S}}_{\text{time}}$ and $M^{\mathcal{O}}_{\text{time}}$ will give the SVF$_{\text{time}}$ value.

$$M^{\mathcal{S}}_{\text{input}}(s^i_t, s^j_t) = \begin{cases} \bar{D}(s^i_t, s^j_t), & \text{if } i < j \\ 0, & \text{if not.} \end{cases} \tag{5}$$

and

$$M^{\mathcal{O}}_{\text{input}}(o^i_t, o^j_t) = \begin{cases} \tilde{D}(o^i_t; o^j_t), & \text{if } i < j \\ 0, & \text{if not.} \end{cases} \tag{6}$$

For computing SVF$_{\text{time}}$, one side-channel trace is sufficient however for computing SVF$_{\text{input}}$, multiple side-channel traces are required (to capture changes in the input). The procedure for computing SVF$_{\text{input}}$ is very similar to SVF$_{\text{time}}$. The difference is in the choice of samples for computing the similarity matrix is illustrated in equations (5) and (6). We first construct $M^{\mathcal{S}}_{\text{input}}$, the similarity matrix for the side-channel traces by computing the distance $\tilde{D}$ between the sample s_t corresponding to different input values x_i, x_j. In the same way, we calculate $M^{\mathcal{O}}_{\text{input}}$, the similarity matrix for the oracle traces, by computing the distance $\bar{D}$ between the sample o_t and the input x_i, x_j. All similarity matrices are triangular, as the main diagonal, which contains only zero values, is removed, and distance measures are commutative. In this work we implement SVF$_{\text{input}}$.

IV. Experimental setup

Simulation setup. In our setup, we use two different 32-bit RISC-V cores, SHAKTI-C and Ibex, and two different algorithms, AES and SHA-3. SHAKTI-C [19] is a 5-stage pipeline in-order processor, while Ibex [20] is a 2-stage in-order processor. From each core, we selected specific modules to examine. We targeted the ones responsible for processing data and instructions and we excluded those that do other work, such as error checking. Specifically, for SHAKTI-C, we examine:

1) `RF` (Register File): implements integer and floating point registers.
2) `CSR` (Control and Status Register): handles special RISC-V instructions to raise interrupts on the processor.
3) `ALU` (Arithmetic Logic Unit): part of the execute stage and performs the arithmetic and logic operations of the processor.
4) `FPU` (Floating Point Unit): part of the ALU and handles instruction operations with floating points.
5) `Dcache`: stores data values in a cache and is part of the memory hierarchy closer to the `ALU`.
6) `MBOX`: implements the multiplication and division instructions.

7) BPU(Branch Prediction Unit): part of the fetch stage and decides the next program counter.
8) ITLB (Instruction Translation-Look aside Buffer): keeps track of instructions addresses recently used to avoid second access to memory.
9) DTLB (Data Translation Look aside Buffer): keeps track of data addresses recently used to avoid second access to memory.

For Ibex, we examine modules with the same functionality:

1) RF(Register File): implements integer register files.
2) CSR (Control and Status Register): handles special RISC-V instructions to raise interrupts on the processor.
3) ALU (Arithmetic Logic Unit): part of the Instruction Decode and Execute (ID/EX) stage and performs the integer computational instructions and the comparison operations.
4) MULT/DIV (Muliplier/Divider Block): under Arithmetic Logic Unit (ALU), a state machine, that performs multiplication and division.
5) PF-BUF (Prefetch Buffer): part of the Instruction Fetch (IF) stage, fetches instructions from the memory for optimal performance.
6) LSU (Load-Store Unit): interfaces with the main memory to deal with memory accesses via load and store operations.

We run simulations of every algorithm with Verilator v4.210 simulator for 256 different inputs, randomly generated. For each simulation run, we take one Value Change Dump (VCD) file. VCD files show the value of every signal, of every module of the RISC-V core for every timestamp of an implementation. We process the vcd files, as well as analyze them, using Python. We parse every file, and for each module, signals are concatenated for each timestamp, creating a composite signal. This signal represents the collective behaviour of all signals and the module's state at that specific point in time. Then, all the concatenated signals for all the selected timestamps, are processed differently, depending on the methodology selected.

SVF computation. For SVF computation, the concatenated signals are retained in their original form. These signals serve as a complex representation of the module's state. Each module contains N rows of signal values per timestamp s_t. Computing SVF requires the generation of oracle traces, which contain the intermediate values of the cryptographic algorithm during its execution. They are the expected values that the algorithm will produce at each time step given a particular input. We run simulations with the gcc compiler on a Linux system, using the same 256 inputs that we used in the simulations on RISC-V. We record different intermediate values in order to examine how the choice of the oracle trace affects the SVF. For the oracle set, we use the Hamming distance metric to compute $M^{\mathcal{O}}_{\text{input}}$, as described in 5 and 6. The oracle set will contain one sample for each input, so the size of $\mathcal{O}$ is N.

The next step is to calculate a similarity matrix for the oracle trace and a similarity matrix for the side channel trace as described in III-A. This step is necessary for correlation because the two traces contain different information and cannot be compared directly. We get two lists of Hamming distance values, and for each timestamp, we compute the Pearson correlation value between the oracle list and the side-channel list. This value shows whether there is a linear correlation between the two lists in our implementation. The SVF value of a module and an oracle is the maximum of all Pearson correlation values. A module's final SVF value is the oracles' maximum SVF value. Arsath et al. [18] use 4 categories to show how much a module leaks: (1) 0.0 - 0.1: No leakage, (2) 0.1 - 0.3: Mild leakage, (3) 0.3 - 0.6: Medium leakage, (4) 0.6 - 1.0: Severe leakage.

TVLA computation. Our TVLA computation is based on the nonspecific fixed versus random test as specified in [2]. Since we work on simulated executions, we need a hypothetical power consumption model. This model is implemented with the Hamming weight (HW), and every timestamp of the concatenated signals takes an HW value. To calculate the t-value per timestamp, we use from the SciPy Python library the ttest_ind function that calculates the t-value for the means of two independent samples of values. The non-specific fixed versus random test executes the fixed set multiple times to eliminate noise during a run. In our case, the runs do not contain noise because the run is simulated, and we know exactly all the signal values at any given time. Once all data for TVLA have been collected, we compute a t-value per cycle. This t-value is calculated from the fixed set of size one and the random set of size $N = 128$ for SHAKTI and $N = 256$ for Ibex.

A. Target cryptographic implementations

We chose unprotected implementations without any countermeasures, as our goal is to examine how leaks impact the different hardware modules.

AES: We use the Tiny-AES[1] implementation, written in C. It provides options for 128-bit, 192-bit, or 256-bit key sizes and options for ECB, CTR, and CBC modes. We use a key size of 128 bits in the ECB mode, and we encrypt one block of data.

SHA-3: We use the tiny_sha3[2] implementation, written in C. SHA-3 is a sponge function with the KECCAK-f[1600] as permutation function. We provide 832 bits of data as input, so we do not need padding. The output of SHA-3 is 384 bits.

V. Experimental results

To determine whether the choice of the leakage detection function influences the decision about the leakiness of a module, we use the experimental setup described in IV. We select nine and six modules for the SHAKTI core and the Ibex core, respectively, that target processing instructions and data. We determine how "leaky" a module is by recording the maximum SVF value.

In addition, we run TVLA in fixed versus random mode. As this is a nonspecific test, we do not explicitly target intermediate variables. When comparing the results of TVLA with the results of SVF, we expect that TVLA will reveal more leaky points since our SVF procedure does not target all the intermediates.

AES. For experiments, we chose the typical candidate intermediate variables as the target: the first byte of the S-box output, $\texttt{sbox\_out}_1$, fifth byte of the S-box output,

[1]https://github.com/kokke/tiny-AES-c
[2]https://github.com/mjosaarinen/tiny_sha3

$\texttt{sbox\_out}_5$, and the first byte of the S-box input, $p \oplus k$ ($\texttt{sbox\_in}_1$). For the round output, we have oracles for the full round output (mc_out) and for the first byte of the round output ($\texttt{mc\_out}_1$).

SHA3. SHA-3 is the other implementation we analyze. From the first round of the Keccak permutation function, we target χ as the only nonlinear operation. We define three oracles based on the χ step: bc, a SHA-3 implementation-specific operation $bc[i] = st[j + i]$ where $i = 0$ and $j = 0$, not operation in $x \leftarrow x \oplus (\neg y \& z)$ where $i = 0$ and $j = 0$ and xor operation in $x \leftarrow x \oplus (\neg y \& z)$ where $i = 0$ and $j = 0$.

A. Case study: the SHAKTI Core

Figure 1 shows the leakage in ALU from the SHAKTI core, for both AES and SHA-3. The first plot shows the evolution in time of the SVF value, during the first round of AES. We represent the different target intermediates with different unique symbols. Horizontal dotted lines are drawn to indicate leakage thresholds. The yellow dotted line at 0.3 shows the minimum threshold for what we consider to be medium leakage. The red dotted line at 0.6 shows the minimum threshold for what we consider to be a severe leak according to [18]. The oracles are highlighted with yellow where the SVF value is ≥ 0.3 and red when SVF ≥ 0.6. The second plot shows the combined SVF with the t-value for the same module, i.e., ALU. In TVLA, we see leaks in almost all cycles and for most of the cycles, it shows leaks unrelated to the SVF oracles. For example, during the execution of the SubBytes operation (cycle 23.000 - 25.000), the SVF only finds leaks when the first or fifth S-box is computed. TVLA looks at all S-box operations and shows more leaks in cycles 23.000 - 25.000. This observation also holds for the cycles after SubBytes. TVLA indicates leaks in cycles where SVF does not show leakage. There are two possible explanations for this behaviour. The first is that TVLA shows false positives. The second is that the Hamming distance between the power traces and the oracles, that were used to calculate the SVF values, does not capture all the relations between the samples. The third plot shows the evolution in time of the SVF value, during the first round of the first execution of KECCAK-f, the SHA-3 permutation, for ALU. The fourth plot shows the combined SVF with the t-value for the same module and implementation, i.e., ALU and SHA3. We observe that SHA-3 is extremely leaky, according to TVLA. While SHA-3 executes the KECCAK-f function multiple times, and one execution of KECCAK-f takes multiple rounds, SVF will only find leaks in the intermediate values we target. TVLA finds multiple leaks during the whole execution of SHA3.

B. Case study: the Ibex Core

Figure 2 shows the leakage in ALU from the Ibex core for both AES and SHA-3. The first plot shows the evolution in time of the SVF value, during the first round of AES. Similar to SHAKTI, the second plot shows combined the SVF with the t-value. Also, the third plot shows the evolution in time of the SVF value during the first execution of SHA3 and the fourth plot shows the combined SVF with the t-value for the same module, ALU. As we observed in SHAKTI, for ALU, TVLA shows leakage in almost all cycles, while SVF shows only at some. We also observe that both methods find the same pattern of leaky cycles. This might indicate that the leakage is not caused only by one instruction but by a sequence of instructions as they progress over time.

Figure 3 shows the leakage in the Register File from Ibex core for both AES and SHA-3. The second plot shows the SVF combined with the t-value for the same module. The Register File and ALU identify the same sequence of leaky operations. Again, TVLA shows more leaky points than SVF. The third plot shows the leakage value for SHA3, and the fourth plot, the combined SVF-TVLA. We observe that SVF shows severe leakage in the second half of this execution timing window. Specifically, the oracle not shows the same leaky points as TVLA, while in the first half of the execution, it identifies only a couple of leaky instructions. We also observe that oracle xor identifies leaky instructions, while oracle bc does not.

Figure 4 shows the combined SVF-TVLA on module MBOX from SHAKTI and Mult/Div from Ibex. Both modules are responsible for multiplication division. The first two plots show AES and SHA-3, respectively, for MBOX, while the last two show AES and SHA3, respectively, for Mult/Div. If we compare these plots with the plots from ALU, we observe a similar pattern of leakage. This is unexpected since we do not have any multiplications or divisions in our code. Our hypothesis is that there are common connections between the ALU and multiplier modules inside the datapath.

C. Performance results

All experiments, for both the SHAKTI and the Ibex core, were done on an AMD Ryzen THREADRIPPER 3990X 4.3GHz CPU with 128 cores and 256GB RAM. On this PC, the run-time of the two methods is significantly different. For the SHAKTI core, the SVF computation of all the modules for the AES case study took around 30k cycles, which lasted approximately eight hours. The same experiment for the SHA-3 case study took around one day of computation. On the other hand, TVLA computation lasted about an hour for all modules for all case studies. For the Ibex core, the SVF computation of the algorithms took about twice the time compared to the SHAKTI core. The TVLA computation lasted less than an hour for all the modules of AES and SHA3. We can easily observe the contrast in run-time efficiency between the SVF and TVLA, which is crucial when choosing the appropriate method.

VI. Conclusions and Future work

In this study, we compare the performance of two different leakage detection methods in detecting leaky modules on microprocessors. We have conducted experiments on two different RISC-V cores, SHAKTI and Ibex, using two cryptographic algorithms, AES and SHA3.

Our results are presented in Table I and Table II for SHAKTI and Ibex core, respectively, for all modules that have been examined. We did not observe leaks in the BPU and ITLB module on SHAKTI or the Prefetch buffer on Ibex, with both algorithms. Our findings suggest that both SVF and TVLA can provide valuable insights into identifying leaky modules

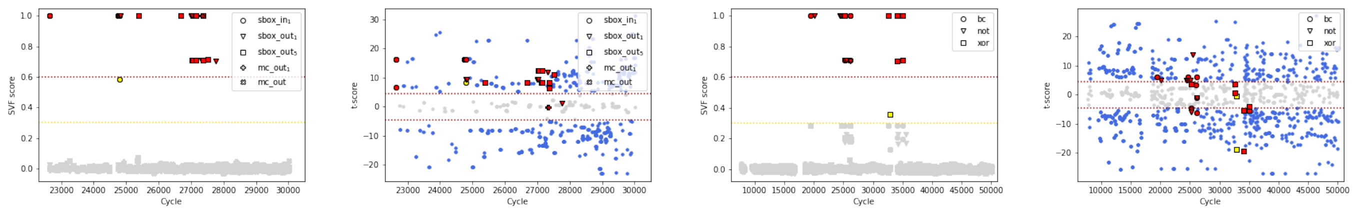

Fig. 1. SVF AES, combined SVF-TVLA AES, SVF SHA-3, combined SVF-TVLA SHA-3. For ALU module on SHAKTI core (left to right).

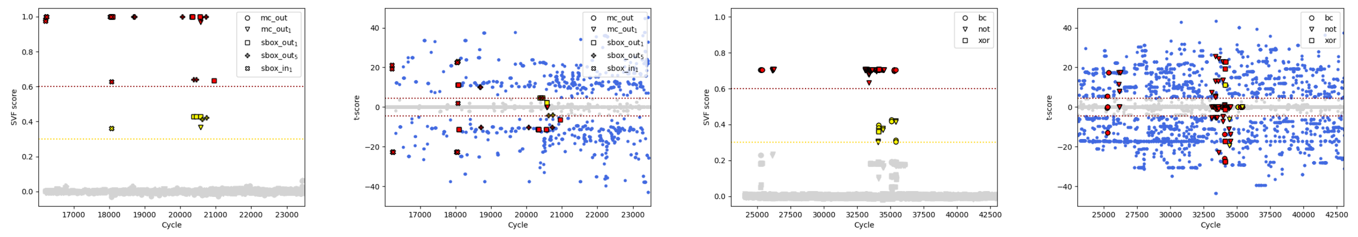

Fig. 2. SVF AES, combined SVF-TVLA AES, SVF SHA-3, combined SVF-TVLA SHA-3. For ALU module on Ibex core (left to right).

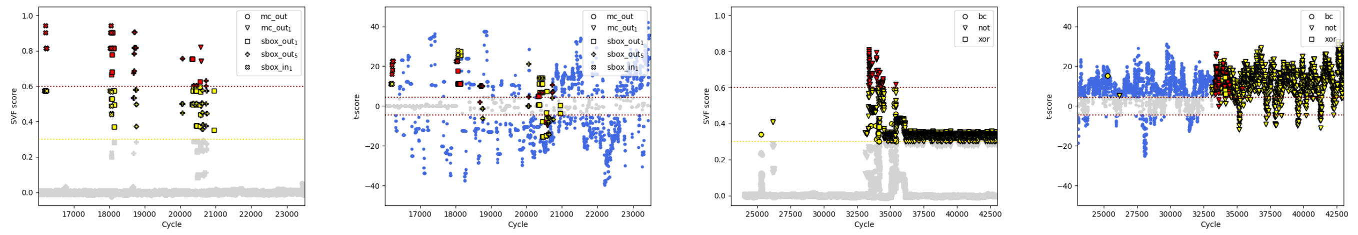

Fig. 3. SVF AES, combined SVF-TVLA AES, SVF SHA-3, combined SVF-TVLA SHA-3. For Register File on Ibex core(left to right).

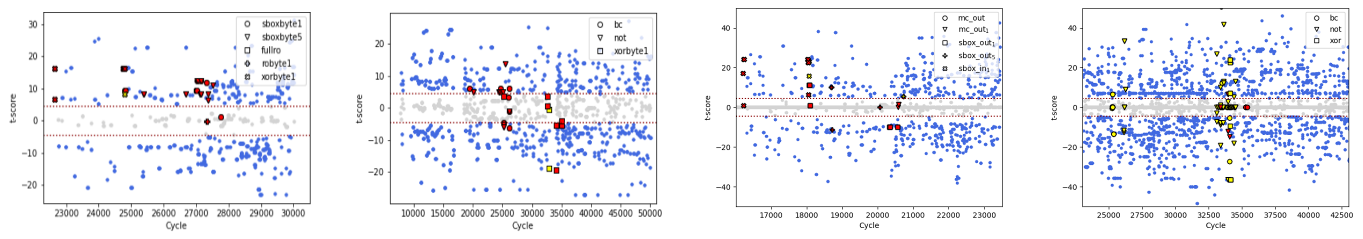

Fig. 4. SVF-TVLA for: AES MBOX(SHAKTI), SHA-3 MBOX(SHAKTI), AES Mult/Div(Ibex) and SHA-3 Mult/Div(Ibex) (left to right).

TABLE I
RESULTS FOR AES AND SHA-3 ON SHAKTI CORE

	AES						SHA-3			
Type	max(SVF)					TVLA	max(SVF)			TVLA
	sbox_in_1	sbox_out_1	sbox_out_5	mc_out_1	mc_out		bc	not	xor	
Dcache	1	1	1	1	0.05	✓	0.99	1	1	✓
RF	1	1	1	1	0.05	✓	1	1	1	✓
CSR	0.98	0.98	0.98	0.98	0.05	✓	0.97	0.97	.97	✓
ALU	1	1	1	1	0.05	✓	1	1	1	✓
FPU	1	1	1	1	0.05	✓	1	1	1	✓
MBox	1	1	1	1	0.05	✓	1	1	1	✓
BPU	0	0	0	0	0	-	0	0	0	-
ITLB	0	0	0	0	0	-	0	0	0	-
DTLB	0.87	0.87	0.87	0.87	0.06	✓	0.3	0.3	0.31	✓

TABLE II
RESULTS FOR AES AND SHA-3 ON IBEX CORE

	AES						SHA-3			
Type	max(SVF)					TVLA	max(SVF)			TVLA
	$\texttt{sbox\_}in_1$	$\texttt{sbox\_}out_1$	$\texttt{sbox\_}out_5$	$\texttt{mc\_}out_1$	$\texttt{mc\_}out$		bc	not	xor	
RF	0.94	0.9	0.9	0.82	0.05	✓	0.59	0.81	0.58	✓
CSR	0.97	0.97	0.97	0.95	0.05	✓	0.69	0.70	0.70	✓
ALU	1	0.99	1	0.99	0.06	✓	0.70	0.70	0.70	✓
MULT/DIV	0.99	0.99	1	0.84	0.04	-	0.63	0.63	0.64	✓
PF-BUF	0	0	0	0	0	-	0	0	0	-
LSU	1	0.99	1	0.98	0.06	✓	0.96	0.96	0.96	✓

in these cores. The effectiveness of these methods can vary depending on the specific core and cryptographic algorithm in use. The use of these methods must take into account the level of granularity of the analysis for the side channel leakage. For more coarse-grained analysis, TVLA is recommended as the computation effort and time is smaller compared to SVF. On the other hand, if fine-grained analysis is needed for the examination of micro-architecture behaviour, SVF is the better candidate. Regarding the timeline of the leaks, our results indicate that TVLA and SVF show leaks in different cycles. We have not yet verified which metric indicates the "correct" leak, and we leave this for future work.

Another point that needs to be addressed is the selection of oracles. As we noticed, not all oracles show leakage. For example, we may choose an oracle that will show no leakage, and we may erroneously think that the module is not leaky. In order to select the proper oracles, one must know very well the cryptographic algorithm being run on the target platform. On the other hand, the generality of TVLA allows more applications in different formats.

These findings underscore the importance of using multiple leakage detection methods in hardening microprocessors. Moving forward, we plan to investigate further the differences between the SVF and TVLA methods and their implications for the hardening of microprocessors. We also aim to explore other leakage detection methods and the impact of different implementations of cryptographic algorithms on the leakage profile of microprocessors. Ultimately, our goal is to contribute to developing more secure microprocessors that are robust against side-channel attacks.

REFERENCES

[1] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *39th International Symposium on Computer Architecture (ISCA*, 2012, pp. 106–117.

[2] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi *et al.*, "A testing methodology for side-channel resistance validation," in *NIST non-invasive attack testing workshop*, vol. 7, 2011, pp. 115–136.

[3] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel*. ACM, 2013, p. 2.

[4] V. Arora, I. Buhan, G. Perin, and S. Picek, "A tale of two boards: On the influence of microarchitecture on side-channel leakage," in *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS*, ser. LNCS. Springer, 2021, pp. 80–96.

[5] P. SLPSK, P. K. Vairam, C. Rebeiro, and V. Kamakoti, "Karna: A gate-sizing based security aware EDA flow for improved power side-channel attack protection," in *Proceedings of the International Conference on Computer-Aided Design*, 2019, pp. 1–8.

[6] A. V. Lakshmy, C. Rebeiro, and S. Bhunia, "FORTIFY: analytical pre-silicon side-channel characterization of digital designs," in *27th Asia and South Pacific Design Automation Conference, ASP-DAC*. IEEE, 2022, pp. 660–665.

[7] I. Buhan, L. Batina, Y. Yarom, and P. Schaumont, "Sok: Design tools for side-channel-aware implementations," in *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan*. ACM, 2022, pp. 756–770.

[8] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. B. Taylor, and R. Kastner, "Hiding intermittent information leakage with architectural support for blinking," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA, Los Angeles, CA, USA*, 2018, pp. 638–649.

[9] B. Marshall, D. Page, and J. Webb, "MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pp. 175–220, 2022.

[10] E. D. Mulder, S. Gummalla, and M. Hutter, "Protecting RISC-V against side-channel attacks," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC, Las Vegas, NV, USA*, 2019, p. 45.

[11] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco: Co-design and co-verification of masked software implementations on cpus," in *30th USENIX Security Symposium, USENIX*, 2021, pp. 1469–1468.

[12] M. T. He, J. Park, A. Nahiyan, A. Vassilev, Y. Jin, and M. M. Tehranipoor, "RTL-PSC: automated power side-channel leakage assessment at register-transfer level," in *37th IEEE VLSI Test Symposium, VTS Monterey, CA, USA*, 2019, pp. 1–6.

[13] S. Gao, J. Großschädl, B. Marshall, D. Page, T. H. Pham, and F. Regazzoni, "An instruction set extension to support software-based masking," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pp. 283–325, 2021.

[14] T. H. Pham, B. Marshall, A. Fell, S. Lam, and D. Page, "XDIVINSA: extended diversifying instruction agent to mitigate power side-channel leakage," in *32nd IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*, 2021, pp. 179–186.

[15] R. Bloem, B. Gigerl, M. Gourjon, V. Hadzic, S. Mangard, and R. Primas, "Power contracts: Provably complete power leakage models for processors," in *Conference on Computer and Communications Security, CCS , Los Angeles, CA, USA*. ACM, 2022, pp. 381–395.

[16] Y. Yao, T. Kathuria, B. Ege, and P. Schaumont, "Architecture correlation analysis (ACA): identifying the source of side-channel leakage at gate-level," in *International Symposium on Hardware Oriented Security and Trust, HOST, San Jose, CA, USA*. IEEE, 2020, pp. 188–196.

[17] P. Kiaei and P. Schaumont, "Soc root canal! root cause analysis of power side-channel leakage in system-on-chip designs," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pp. 751–773, 2022.

[18] M. A. K. F, V. Ganesan, R. Bodduna, and C. Rebeiro, "PARAM: A microprocessor hardened for power side-channel attack resistance," in *International Symposium on Hardware Oriented Security and Trust, HOST, San Jose, CA, USA*. IEEE, 2020, pp. 23–34.

[19] SHAKTI, "Family of processors," https://shakti.org.in/processors.html, 2021, online; accessed 21-01-2022.

[20] lowRISC, "Lowrisc/ibex-demo-system: A demo system for ibex including debug support and some peripherals," accessed 15-10-2023. [Online]. Available: https://github.com/lowRISC/ibex-demo-system