SCAPEgoat: Side-channel Analysis Library

Dev Mehta, Trey Marcantino, Mohammad Hashemi, Sam Karkache, Dillibabu Shanmugam,
Patrick Schaumont, Fatemeh Ganji
Worcester Polytechnic Institute
Worcester, USA
{dmmehta2, tmmarcantonio, mhashemi, swkarkache, dshanmugam, pschaumont, fganji}@wpi.edu

Abstract—**Side-channel analysis (SCA) is a growing field in hardware security where adversaries extract secret information from embedded devices by measuring physical observables like power consumption and electromagnetic emanation. SCA is a security assessment method used by governmental labs, standardization bodies, and researchers, where testing is not just limited to standardized cryptographic circuits, but it is expanded to AI accelerators, Post Quantum circuits, systems, etc. Despite its importance, SCA is performed on an ad hoc basis in the sense that its flow is not systematically optimized and unified among labs. As a result, the current solutions do not account for fair comparisons between analyses. Furthermore, neglecting the need for interoperability between datasets and SCA metric computation increases students' barriers to entry. To address this, we introduce SCAPEgoat, a Python-based SCA library[1] with three key modules devoted to defining file format, capturing interfaces, and metric calculation. The custom file framework organizes side-channel traces using JSON for metadata, offering a hierarchical structure similar to HDF5 commonly applied in SCA, but more flexible and human-readable. The metadata can be queried with regular expressions, a feature unavailable in HDF5. Secondly, we incorporate memory-efficient SCA metric computations, which allow using our functions on resource-restricted machines. This is accomplished by partitioning datasets and leveraging statistics-based optimizations on the metrics. In doing so, SCAPEgoat makes the SCA more accessible to newcomers so that they can learn techniques and conduct experiments faster and with the possibility to expand on in the future.**

Index Terms—**Side-channel Analysis, DPA, TVLA, JSON.**

I. Introduction

Side-channel analysis (SCA) is a technique to reveal sensitive information in a device's computing on a secret by measuring observables during computation [1]. In this sense, observables, so-called side-channel leakage, include power consumption, timing, or electromagnetic emanation (EM), which can reveal secret data being processed [2]–[5]. SCA involves collecting, storing, and analyzing traces using various software and hardware tools. SCA can be applied to reverse engineer chips [6], launch attacks against edge devices [7], [8], break the security of smart cards [9], hack cars [10], etc. This is why it is important for nations to study these techniques and develop ways to counter them for national security. To develop effective countermeasures, a comprehensive understanding of the techniques in SCA is essential, where a common methodology helps researchers to develop better countermeasures. This requires datasets and evaluation frameworks, which provide consistent benchmarks for security assessment, comparing metric computations, and improving the effectiveness and efficiency of SCA. As a prime example, the U.S. and Canadian standard FIPS 140-3 [11] includes non-invasive physical security in validating implementations. Besides such initiatives by standardization bodies, independent researchers have taken various steps to support the development of datasets and tools [11]–[14].

However, a challenge with existing datasets is the lack of consistency: they are formatted differently, using various file formats and structures, which complicate knowledge sharing between research groups. The ASCAD dataset [13] is a dataset collected from a software implementation of masked AES, where EM traces are stored using HDF [15] with its specific file structure [13]. Secbench [16], a versatile Python toolkit for side-channel and fault analysis, also employs HDF5-based trace format for managing side-channel data. HDF stores data in a hierarchical structure that does not support metadata, and the structure is not human-readable. The SMAeSH dataset contains power traces collected from a masked FPGA implementation of the AES, stored using Numpy format and specific file structure [14]. The SMAeSH dataset uses a JSON file to categorize the dataset and file list, but it is not consolidated for the whole project. These examples indicate the inconsistency, often stemming from dependencies on particular tools and setups each contributor uses. This highlights the second key area needing systematization, i.e., the tools used to compute the metrics. Challenges with such tools include incomplete and inefficient functionality. Based on the purpose of the tools, they might not have all the steps in SCA covered, or they may use a specific way of computing certain metrics. In an extreme but real scenario, existing tools do not share the same way of reading the trace set/storing the results. This kind of mismatch leads to the need for conversions between dataset formats that are time-consuming and computation-hungry, although they might seem trivial to resolve. In fact, such differences also affect the reproducibility of the results.

Lastly, standards like FIPS 140 aim to keep the evaluation cost to an absolute minimum in terms of time, memory, and computational complexity cf. [17]. However, SCA has evolved from typical cryptographic implementation like AES to other sensitive intellectual properties like neural networks (NNs) [18]–[20]. Usually, the secret for a traditional cryptographic algorithm is the key, whereas for NNs, it is the

[1] The library is available here:https://github.com/vernamlab/SCApeGoat.

This is the author version of the paper accepted for the presentation at the The IEEE VLSI Test Symposium (VTS) 2025.

weights and the architecture of the NN. The NN circuits are more complex and have more clock cycles, making the individual traces much longer and increasing the number of traces required for attack compared to an AES engine. This leads to larger data files (in order of terabytes), which requires optimization in the metric computations. Most implementations of SCA metric computation and attacks such as TVLA [21], CPA [22], signal-to-noise ratio (SNR), etc. are not optimized for SCA in general. Consequently, users/evaluators with resource-constrained machines cannot efficiently process terabytes of data and evaluate the security as required for certifying vulnerability.

Contributions. Our tool, SCAPEgoat, addresses the issues mentioned above in the entire SCA chain through simple file structures, support for standard capture hardware, and a post-processing software suite. The idea behind the design of SCAPEgoat is to create modules corresponding to (virtually all) steps taken in SCA. The modular nature of SCAPEgoat enables the user to enjoy each module separately or as a complete framework. Moreover, SCAPEgoat introduces memory efficiency, especially regarding RAM usage for large datasets, making our tool ideal for newcomers. In short,

1) We have created a customized, user-friendly file structure using JavaScript Object Notation (JSON), especially for SCA. It enables easy navigation through experiments, storage of metadata, and better organization of datasets concerning an experiment, enabling reproducibility. We also integrate functions to query experiments and datasets based on the metadata.
2) Along with the file format definition, SCAPEgoat offers (usual) capturing interfaces, enabling more efficient analysis. The capturing interfaces currently supported by SCAPEgoat contain high-end Lecroy oscilloscope and Chipwhisperer (CW) capture boards. At one end of the spectrum, high-end devices are used by professional SCA practitioners, while CW boards are affordable to learners and students.
3) To detect the leakage, SCAPEgoat encompasses the test vector leakage assessment (TVLA) methodology that is optimized for lower memory usage and partitioned datasets. This technique also makes live computation of TVLA possible. Regarding the last stage of SCA, i.e., leakage exploitation, a memory-optimized column-wise differential power analysis (DPA) is integrated. Moreover, native support for 2^{nd} order DPA is provided.

II. Implementation

SCAPEgoat is a Python library for SCA. It leverages and is built upon open-source application programming interface (APIs) for capturing traces with a unique touch to the fully custom file structure to enable a smoother learning curve and easier experimentation for researchers. The flow of the SCAPEgoat library is shown in Figure 2. It includes all the steps of an SCA, starting with configuring the capturing equipment and the target board, collecting traces, storing them, and running metrics computation/attacks on them. Now, we will discuss the library implementation and unique features of the file framework and metric computations.

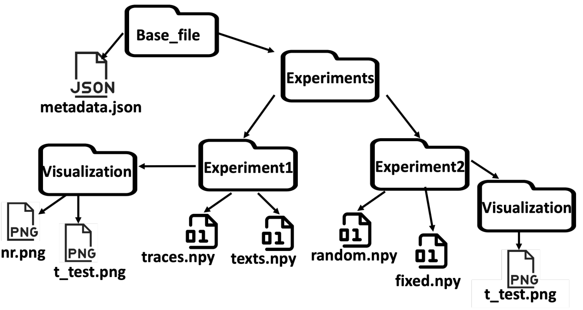


Fig. 1: The folder structure for the SCAPEgoat.

A. Library Implementation

Module 1: File Framework. First, we look at the library's backbone, the file framework. SCAPEgoat uses a custom file framework explicitly designed for SCA. Moreover, long-term projects with multiple experiments require logging experimental variables, which is enabled by metadata stored in a JSON file with the hierarchy. Moreover, we support querying the metadata and functions to store and organize the experiments with metadata in mind. It is noteworthy that thanks to the modular design of SCAPEgoat, if the traces are available in other formats, one can easily transform them to a SCAPEgoat-friendly format for higher efficiency. Afterward, the first steps to understand if there is a leakage (see Module 2). There is, of course, feedback from observing the leakage to Module 1 if more traces need to be collected (one usually starts with a few thousand and increases the size of the trace set if needed). As one still needs to store the traces for further analysis, the file format module is essential to make it more efficient when deciding on an attack. Next, we explain how the file structure makes it easier to organize experiments starting with the heirarchy.

Hierarchical Structure. The structure implemented for the file framework closely follows the structure presented in Figure 1. We chose to model the framework using a hierarchical structure similar to what is implemented using HDF5 [15]. Still, we have used JSON to manage the structure and keep the data in separate Numpy files. Compared to Zarr adopted in [23], JSON serves data storage needs while offering human readability.

The file parent directory is at the top level of SCAPEgoat's JSON hierarchy, represented in the Python API as the `FileParentclass`. All paths of the experiments and datasets are relative to the parent directory. The advantage of this is that the user needs only to remember one path for all of the content stored in the parent directory. As seen in Figure 1, the second level in the hierarchy is experiments, each having its own list of datasets. Each of these has its class with specific functions to update them. For each experiment, there is also a visualization folder where the graphs for the metric computations can be stored. The parent directory also contains the JSON file, which is used to maintain the hierarchical structure and metadata of a project. As shown in Figure 1, the hierarchy is presented for the project with experiments and datasets. This hierarchy is maintained using a nested structure in the JSON file. The JSON file starts with a main structure for the file parent and creates the default fields, like the path of the folder and data created when using the constructor. Next, there are methods in the `Fileparentclass` for creating and managing the experiments. When adding a new experiment, JSON creates a structure nested into the parent structure as an array. Similar structures exist between the datasets and experiments. The visualization folder has no JSON field and is updated only by specific functions. Not only the file structure but also the variable-value pairs in the JSON file have been used to store metadata for the parent and each experiment.

Integration of Metadata. A defining feature of our library is the ability for users to add arbitrary metadata at the file, experiment, or dataset level. As explained before, this feature is enabled using the metadata list in the JSON file. Users often want to associate parameters with the experiments when conducting their research. For example, for users working with the effects of environmental parameters on the experiments, temperature is a prominent piece of information that the authors want to associate with the collected data. The library allows users to specify variable-value pairs such as (temperature,30C) for each dataset. If some parameters are typical for an experiment, the metadata could also be added at the experiment level. We also implement functions to query experiments and datasets based on the metadata. This ability provides ease of sharing the project and reproducibility.

Module 2: Capturing Process. The capturing process has also been integrated for commercially available devices. We support two oscilloscope series: CW [24] and Lecroy [25]. SCAPEgoat supports all CW and Lecroy oscilloscopes (PyVISA). Lecroy is configured using the PyVISA protocol [26]. Theoretically, any other oscilloscope that supports PyVISA could be configured using the function. CW capture boards were selected as they are widely-used starter kits for SCA, while Lecroy oscilloscopes are available in research labs and are also used in the Riscure ecosystem [27]. For target devices, SCAPEgoat supports CW targets natively; nevertheless, capturing can be performed from any target connected to the oscilloscopes and can generate a trigger. We have also developed a comprehensive function, defined as `standard_capture_procedure`, that implements fast capturing of power traces while providing flexibility for the programmer. The standard capturing procedure returns a tuple containing the power traces, the encryption keys, the plaintexts, and the ciphertexts. The performance of the standard capturing procedure varies depending on the CW device, the encryption algorithm being used, and the specification of the computer running the Python script [24]. In addition to the standard capturing function, SCAPEgoat includes a function that collects both fixed and random traces for use in a TVLA metric evaluation. While the standard capturing function returns the captured traces as a tuple, which must be stored as datasets manually, the TVLA function automates the process of storing fixed and random datasets; see Figure 2. This allows the user to do TVLA by running a single line of code.

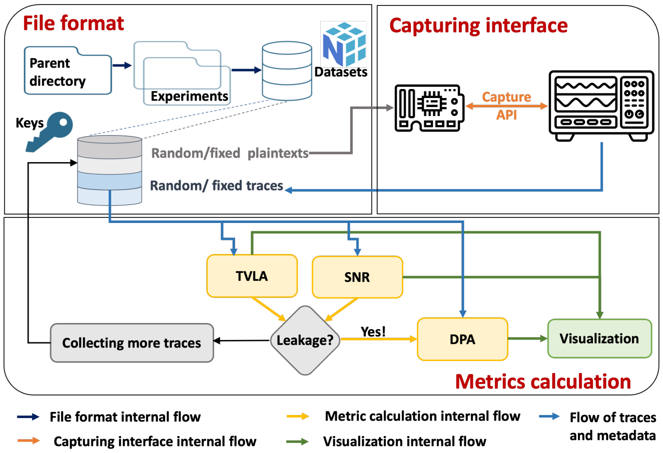


Fig. 2: An example of SCAPEgoat flows. First, the user creates the necessary Paranet directory, experiments, and file hierarchy for datasets. Then, plaintexts (fixed and random) and key datasets are initialized for a required number of traces. To capture the traces using SCAPEgoat, the APIs use the datasets directly from the experiment and save the two trace sets (fixed and random) back under the experiment. The functions in the metric calculation module are part of the experiment class and can now be applied by providing only dataset names as input. If TVLA indicates the leakage, the SNR function is called to examine the exploitation potential. Lastly, DPA can be performed to extract the secret.

Module 3: Metric Computation. The file framework integrates the computation of the leakage detection metrics, namely TVLA and SNR; see Figure 2. Leakage exploitation is also possible by conducting DPA with Pearson correlation as its distinguisher. These are selected as representative of fundamental tests for leakage detection and leakage exploitation [17], [28]. Specifically, DPA is chosen due to its unmatched power in extracting sensitive information from subtle variations in power consumption.

Rather than manually parsing and organizing the data, once the traces, keys, and plaintexts are stored in the datasets, the integration allows the user to only pass the dataset-specific variables to perform the full computation of the metrics, i.e., detect the leakage and launch DPA with the correlation distinguisher. In addition, the user has the option to save the metric result as a dataset and save a plot of the metric visualization to the experiment's visualization folder. The library also has stand-alone implementations for these three metrics and other metrics like score and rank. These provide the user with a wider range of metrics. In the next section, we examine the optimizations made to the metrics, particularly the file structure.

B. Memory-efficient Computation of Metrics

The memory efficiency becomes more critical when considering implementations with more clock cycles, i.e., longer traces. Unique challenges have to be dealt with when analyzing such unconventional schemes rather than usual encryption engines, e.g., AES. Recently, AI accelerators and post-quantum-cryptographic implementations have been targeted by SCA [18], [20], [29]. The trace set collected from these targets can be huge, in the range of 100s of GBs. To address this, SCAPEgoat's memory-efficient metrics computation relies primarily on performing the relevant operations on partitions of the trace set depending on the memory size and then accumulating the results for each set. This allows the user to perform analysis on computationally weaker devices with less RAM available to maintain efficiency and even potentially allows for simultaneous calculations as traces are being collected.

Running TVLA. We used running statistics to implement the metrics to achieve memory efficiency. Running statistics is a statistical method that can implement ongoing statistical computations, i.e., the statistic can be updated with each new value. This means one does not need to compute the complete statistics again with each new value. This has multiple benefits in terms of memory and time-saving. This technique can be applied to the TVLA, defined as follows:

$$t = (\mu_1 - \mu_2)/\sqrt{(s_1^2/n_1^2) + (s_2^2/n_2^2)},$$

where μ_1 and μ_2 are the means, s_1 and s_2 are the standard deviations, and n_1 and n_2 are the total number of the captured traces for the first and second population, respectively. In the context of SCA, these two populations correspond to two trace sets collected by feeding fixed/random plaintexts (see [28] for details on TVLA). To calculate the mean and standard deviation, the matrix composed of all traces must be loaded in memory ; hence, this becomes a bottleneck as the dataset size increases. We solve this by using the running mean and standard deviation.

$$\begin{aligned} \mu_{[n+1]} &= \mu_{[n]} + (t_{[n+1]} - \mu_{[n]})/n + 1 \qquad (1) \\ var_{[n+1]} &= var_{[n]} + (t_{[n+1]} - \mu_{[n]}).(t_{[n+1]} - \mu_{[n+1]}) \\ std_{[n+1]} &= \sqrt{var_{[n+1]}/n}, \end{aligned}$$

where μ is the mean, t is the trace, var is the variance, and std is the standard deviation. These equations indicate that the mean and standard deviation depends on the previous mean and standard deviation calculated for n traces and the new trace $t_{[n+1]}$. This allows a trace-by-trace computation for the TVLA test instead of matrix-wise. An evaluator can leverage this only to compute enough traces to get a desired t-value and stop the trace capture automatically.

This method, in coordination with the file framework's partitioning of datasets, allows for memory-efficient computation of the TVLA even after storing the datasets. This is because the running statistics could be calculated for the first part, and then the traces could be flushed out, and only the intermediate variables needed to be kept in the memory. This makes it possible to run larger datasets on smaller memory budgets, allowing users to employ low-powered machines. Moreover, this approach also helps to share the datasets as uploading and downloading can happen in parts, which enables the user to decide how many traces are needed for the analysis.

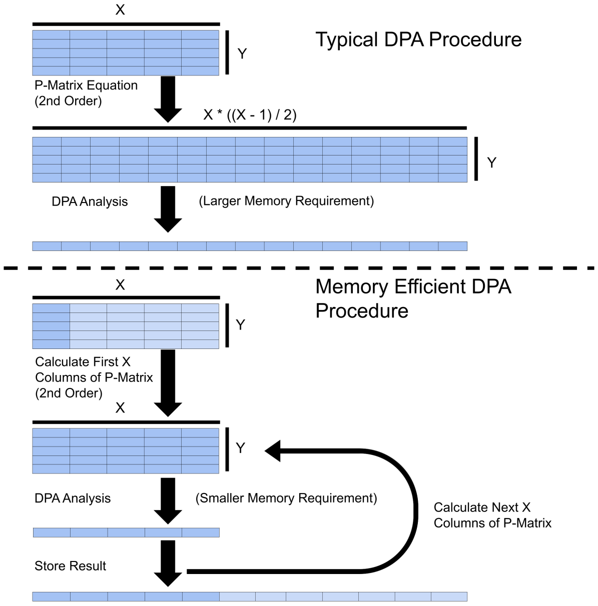


Fig. 3: Improvement in the DPA process, where the computation is done per column to reduce memory usage.

Column-wise DPA. DPA [3] is a side-channel attack that exploits the power consumption of cryptographic devices during operations to extract sensitive information, such as secret keys. By observing and correlating power traces collected during encryption or decryption processes, DPA reveals patterns that correspond to internal data processing. This attack can leverage statistical distinguishers, such as correlation, to compare the observed power consumption with ones corresponding to different key guesses. The comparison ultimately identifies the correct key when a strong correlation is found.

DPA can be classified into different orders based on the complexity of the analysis [30]. First-order DPA, as the simplest form, focuses on observable variations in power consumption that correspond to individual bits or operations. This makes it relatively straightforward to perform when no countermeasures are in place. Second-order DPA is more complex and involves analyzing the interaction between multiple points in the power trace. In second-order DPA [30], attackers combine power consumption values at different times to amplify the correlation between power traces and internal operations. Second-order DPA focuses on scenarios where simple first-order attacks are insufficient due to countermeasures that resist more straightforward first-order attacks. For example, when considering Hamming weight as the leakage model, Sakic et al. [31] implemented second-order DPA as:

$$\begin{aligned} C(k) = \frac{1}{n}\sum_{i=1}^{n}(P_i^{(2)}(t_1,t_2) - \bar{P}^{(2)}(t_1,t_2))\cdot \qquad (2) \\ \cdot\,(HW(v_i(k,p)) - H\bar{W}(v(k,p)) \end{aligned}$$

Here, $P_i^{(2)}(t_1,t_2)$ is the product of power consumption at two-time points, t_1 and t_2, in the i^{th} trace. $\bar{P}^{(2)}(t_1,t_2)$ is

TABLE I: Memory complexity of TVLA while capturing the traces and after collecting traces (standard).

	Standard Methodology		Running Methodology	
	Memory Usage	Time [s]	Memory Usage	Time [s]
Single file	2.4GB	987.92	960 KB	986.66s
Partitions	47382MB	1225	15893MB	1440

the average of $P_i^{(2)}(t_1, t_2)$ across all traces. p represents the plaintext or input data, and k denotes the correct key/secret utilized in the algorithm. HW indicates the Hamming weight, v is the intermediate value where v_i is the intermediate value for the i^{th} trace. $H\bar{W}$ is the average hamming weight of the intermediate value v across all traces. $C(\cdot)$ is the correlation coefficient and n is the total number of traces.

In second-order DPA, the matrix composed of $P^{(2)}$ entries can often be too large to store in a typical computer's RAM, as the number of columns is quadratic in the number of samples per trace. To address this overhead, SCAPEgoat introduces column-wise DPA processing. Instead of calculating the entire $P^{(2)}$ before computing on the matrix, SCAPEgoat calculates segments of the $P^{(2)}$, performs calculations on the subset, and then repeats this process many times. A column-wise approach is allowed because the results of DPA are independent of one another column-wise. The number of samples per trace depends on the window of interest, where the evaluator expects the leakage according to leakage detection metrics, e.g., TVLA or SNR. The evaluator can select the window, allowing for a flexible approach tailored to the user's requirements.

III. Results on Metric Optimization

An example of use-cases: side-channel protected neural networks. The dataset used for benchmarking our metric computation modules is a recent one [32]. Besides being collected from an interesting target, i.e., first-order masked neural networks, this dataset is selected because the traces are long (10,400 time samples per trace and 2M traces), making scalability assessment feasible. The size of this dataset is 153GB, which we stored in 20 partitions of 7.63GB each. The analyses presented in this section were performed on a single core of an Intel E5-2695 v4 CPU using 64GB of RAM.

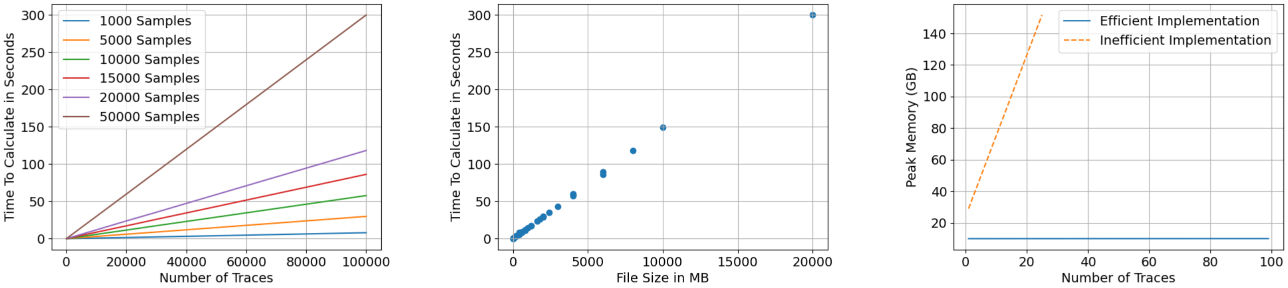
First, we discuss the results of the metric computation for our TVLA algorithm. As shown in Figure 4(a), the algorithm's speed is determined by the number of traces and samples per trace. However, as reflected by the linearity in Figure 4(b), the change in time to calculate appears proportional to the file size. This implies that this algorithm can scale up with the length of traces; having a large number of time samples does not increase the time complexity exponentially. The same holds for analyzing a large number of traces.

However, the primary goal for optimizing the TVLA is to reduce memory usage. Table I lists the peak memory usage of two implementations; one utilizes live metric solving without saving the trace data to a file (running), and the other collects trace data and then performs the TVLA test (standard). The memory and time for this experiment were computed using the memit and time it magic commands, respectively, with 30 runs for each method. As seen in Table I, when collecting 10,000 traces at 60,000 samples per trace (float 32), the times to solve are similar, while the second methodology exhibits dramatically lower RAM usage. In this test, four traces are collected parallel to the computation of TVLA, and the required memory to store them is solely 960KB. For the partitioned dataset, the standard method takes 3 times more memory than the running method, as noted in Table I. The peak memory usage for the running method is about 16GB. This usage corresponds to 2 parts of the dataset –random and fixed– that have been loaded into memory and some overhead for intermediate values and Python variables. Note that this is the peak memory usage in Windows, and we do not account for the committed/paged memory usage at the time. As for the time complexity, there is a difference in the methods compared to the single file results. This difference is attributed to the overhead of managing partitions in both methods. For the running method, the computation is done per trace, requiring more computations than the matrix-based computation in the standard method. This is why the timing differs for the methods when dealing with larger datasets.

Figure 4(c) depicts the memory improvement achieved by the column-wise second-order DPA approach. As can be seen, the inefficient method can only compute DPA up to 36 traces as it has reached the computer's memory limit. Meanwhile, with the memory-efficient version, column-wise computation allows us to overcome that limitation. Also, as seen in our results, memory usage is constant, so there is no limit to the number of traces. Thus, the optimization we provided enables the computation of complex metrics on resource-constrained machines. To validate the correctness of column-wise DPA, we experimented on 500K traces to reproduce the results in [20]. We observed identical results as in that paper when employing our column-wise DPA. This confirms that our optimization techniques do not adversely impact the DPA's precision.

IV. SCAPEgoat vs. Existing Libraries

This section compares our library with other open-source libraries. SCAlib was developed by Cassiers et al. [33], where the capturing process is implemented in Python. On the other hand, the metrics are implemented in Rust language to make them more efficient with support for processing the data in chunks. CW has a library that uses its wide range of hardware to capture traces and compute metrics [24]. SCARR is developed by Bosland et al. as a library to compute metrics faster with a particular focus on high-performance machines [23]. They have achieved this using high-performance computing like GPU acceleration and multi-core computation. One of their main ideas is faster computations on compressed datasets to save storage space. SCARED is a framework developed by eShard, a for-profit organisation [34]. It is a basic framework mainly used for AES and DES-based security evaluation. It has pre-processing capabilities for different file formats but does not support capturing capabilities. LASCAR is another tool made by a for-profit company for their side channel project [35]. They have a complete pipeline from captur-



(a)

(b)

(c)

Fig. 4: (a) The time complexity of TVLA for various sample sizes for different numbers of traces; (b) the relation of time and file size when performing TVLA; (c) memory complexity for efficient vs. conventional second-order DPA.

TABLE II: The table illustrates the differences between the various libraries available for side-channel analysis.

	Metadata Handling	Integrated Metric Solving	Integrated File Format	Integrated Trace Capturing	Easily Expandable	DPA
SCAPEgoat	Yes	Yes	Yes	Yes	Yes	$1^{st}/2^{nd}$ order
SCALib	No	Yes	No	No	Yes	None
CW	No	Somewhat	Yes	Yes	Somewhat	1^{st} order
SCARR	No	Yes	Yes	Yes	Yes	1^{st} order
SCARED	No	Yes	Yes	No	No	1^{st} order
LASCAR	No	Yes	Yes	Yes	Somewhat	1^{st} order
Sedpack/SCAAML	Yes	Somewhat	Yes	No	No	None

ing to computing the TVLA test. It also supports machine learning-based analysis in SCA. Lastly, Sedpack/SCAAML are Google-made side channel libraries mainly focused on machine learning-based attacks in the domain [36]. Sedpack is an upgrade to the SCAAML with added custom metadata [37].

SCAPEgoat offers unique advantages compared to other libraries, as seen in Table II. One of the main ones is the metadata-based functionality. Out of all the libraries, only the Sedpack file format offers the ability to store and read metadata. In contrast, SCAPEgoat additionally allows sorting through experiments based on metadata parameters. SCAPEgoat also has direct support for trace capturing, which metric-based frameworks like SCARR do not support. Finally, SCAPEgoat is easily expandable because of function handles, and its open-source nature allows for direct source code modification. SCAPEgoat also includes integrated metric solving, which is also available in frameworks like SCARR and SCARED and, to some extent, in CW [24]. SCAPEgoat supports first- and second-order DPA to exploit the leakage, optimized for efficiency on typical machines.

V. Conclusion and Future Work

SCAPEgoat is a library that aims to make SCA techniques more accessible to newcomers, such as students, and more efficient for security evaluators. It also helps reproduce the results by introducing a better file structure. Besides, the custom metadata field per experiment and dataset helps create a verbose description for the whole project in one place. Furthermore, SCAPEgoat improves the memory complexity of metric computation (TVLA and DPA) and support live and partitioned TVLA. Lastly, SCAPEgoat natively supports memory-efficient first- and second-order DPA. SCAPEgoat is an ever-evolving and improving framework. Here, we highlight some improvements to be considered in the future.

Integration of SCA Hardware. Based on applications, oscilloscopes and target devices can be upgraded/added to the library. In this regard, APIs for other oscilloscope that supports PyVISA could be considered.

Generic Metrics. Chipwhisperer [24] and Riscure [27] already have a robust metric set. However, the examples and the main engines they are geared towards are mainly AES and other standard cryptographic engines. While this provides ease for testing conventional cryptographic implementations, changing to other designs is not straightforward for the user. In addition to the metrics introduced in this paper, we plan to add more metrics with optimizations that can verify the security of designs.

GPU acceleration. Regarding computation time complexity, reading and writing trace data to and from the file cause a bottleneck on metric and attack calculations. GPU acceleration excels in situations where a large amount of computation is needed in parallel, like matrix mathematics, and struggles when there are large I/O requirements in sequential algorithms. We should stress that reduced attack cost is one of the primary metrics for the severity of an attack [17]. If an SCA metric can be analyzed efficiently using a CPU with its limited memory, then the attack cost is much reduced. This is why SCAPEgoat focuses on the implementation that does not rely on GPUs. Nonetheless, supporting GPU acceleration can be thought of as a future direction.

Acknowledgment

This work has been supported partially by NSF under award numbers 2138420 and DMS-1337943.

References

[1] J. Van Woudenberg and C. O'Flynn, *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*. No Starch Press, 2021.

[2] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual international cryptology conference*, pp. 388–397, Springer, 1999.

[4] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems—CHES 2001*, pp. 251–261, Springer, 2001.

[5] J. J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Smart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001 Cannes, France*.

[6] S. S. Ensan, K. Nagarajan, M. N. I. Khan, and S. Ghosh, "Scare: Side channel attack on in-memory computing for reverse engineering," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 12, pp. 2040–2051, 2021.

[7] P. Bhade, J. Paturel, O. Sentieys, and S. Sinha, "Lightweight hardware-based cache side-channel attack detection for edge devices (edge-cascade)," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 4, pp. 1–27, 2024.

[8] S. Dey, A. K. Singh, and K. McDonald-Maier, "Thermalattacknet: Are cnns making it easy to perform temperature side-channel attack in mobile edge devices?," *Future Internet*, vol. 13, no. 6, p. 146, 2021.

[9] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*, vol. 31. Springer Science & Business Media, 2008.

[10] H. Wang, S. Salehi, H. Sayadi, A. Sasan, T. Mohsenin, P. S. Manoj, S. Rafatirad, and H. Homayoun, "Evaluation of machine learning-based detection against side-channel attacks on autonomous vehicle," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 1–4, IEEE, 2021.

[11] NIST, "Security requirements for cryptographic modules." Federal Information Processing Standards Publication FIPS 140-3 https://doi.org/10.6028/NIST.FIPS.140-3, 2019. Accessed: 2024-12-1.

[12] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, S. D. Leigh, M. Levenson, M. Vangel, N. A. Heckert, and D. L. Banks, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[13] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ascad database," *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020.

[14] G. Cassiers and C. Momin, "The smaesh dataset," *Cryptology ePrint Archive*, 2024.

[15] H. Group, "Hd5." [Online]https://www.hdfgroup.org/solutions/hdf5/ [Accessed: Sept.26, 2024], 2002.

[16] CEA-Leti Cybersecurity Teams, "The secbench framework." [Online] https://github.com/CEA-Leti/secbench [Accessed: Mar.2, 2025], 2025.

[17] M. Azouaoui, D. Bellizia, I. Buhan, N. Debande, S. Duval, C. Giraud, É. Jaulmes, F. Koeune, E. Oswald, F.-X. Standaert, *et al.*, "A systematic appraisal of side channel evaluation strategies," in *Security Standardisation Research: 6th International Conference, SSR 2020, London, UK, November 30–December 1, 2020, Proceedings 6*, pp. 46–66, Springer, 2020.

[18] A. Dubey, R. Cammarota, and A. Aysu, "Maskednet: The first hardware inference engine aiming power side-channel protection," in *2020 IEEE Intrl. Symp. on Hardware Oriented Security and Trust (HOST)*, pp. 197–208, IEEE, 2020.

[19] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symp. (USENIX Security 19)*, pp. 515–532, 2019.

[20] D. M. Mehta, M. Hashemi, D. S. Koblah, D. Forte, and F. Ganji, "Bake it till you make it: Heat-induced power leakage from masked neural networks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 4, pp. 569–609, 2024.

[21] G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, *et al.*, "Test vector leakage assessment (tvla) methodology in practice," in *International Cryptographic Module Conference*, vol. 1001, p. 13, sn, 2013.

[22] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Intrl. workshop on cryptographic hardware and embedded systems*, pp. 16–29, Springer, 2004.

[23] J. Bosland, S. Ene, P. Baumgartner, and V. Immler, "High-performance design patterns and file formats for side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 769–794, 2024.

[24] C. O'flynn and Z. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *Constructive Side-Channel Analysis and Secure Design: 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers 5*, pp. 243–260, Springer, 2014.

[25] Teledyne, "Lecroy oscilloscopes." [Online]https://www.teledynelecroy.com/oscilloscope/ [Accessed: Dec.2, 2024], 2024.

[26] H. E. Grecco, M. C. Dartiailh, G. Thalhammer-Thurner, T. Bronger, and F. Bauer, "Pyvisa: the python instrumentation package," *Journal of Open Source Software*, vol. 8, no. 84, p. 5304, 2023.

[27] Riscure, "Riscure." [Online]https://www.riscure.com/ [Accessed: Nov.8, 2024], 2024.

[28] K. Papagiannopoulos, O. Glamočanin, M. Azouaoui, D. Ros, F. Regazzoni, and M. Stojilović, "The side-channel metrics cheat sheet," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–38, 2023.

[29] B. Gigerl, F. Mendel, M. Schläffer, and R. Primas, "Efficient second-order masked software implementations of ascon in theory and practice," *Cryptology ePrint Archive*, 2024.

[30] E. Oswald, S. Mangard, C. Herbst, and S. Tillich, "Practical second-order dpa attacks for masked smart card implementations of block ciphers," in *Cryptographers' Track at the RSA Conference*, pp. 192–207, Springer, 2006.

[31] E. Sakic, "Second-order dpa matlab code." https://github.com/ermin-sakic/second-order-dpa/blob/master/second_order.m, 2024. Accessed: 2024-10-23.

[32] D. M. Mehta, M. Hashemi, D. S. Koblah, D. Forte, and F. Ganji, "Bake it till you make it: Heat-induced power side-channel analysis against masked neural networks." [Online]https://github.com/vernamlab/Bake-it-till-you-make-it [Accessed: Oct.22, 2024], 2024.

[33] G. Cassiers and O. Bronchain, "Scalib: a side-channel analysis library," *Journal of Open Source Software*, vol. 8, no. 86, p. 5196, 2023.

[34] eShard, "Scared v2022." [Online]https://github.com/eshard/scared [Accessed: Sept.24, 2024], 2022.

[35] L. Donjon, "Lascar v2023." [Online]https://github.com/Ledger-Donjon/lascar/tree/master [Accessed: Sept.24, 2024], 2023.

[36] E. Bursztein, L. Invernizzi, K. Král, and J.-M. Picod, "SCAAML: Side Channel Attacks Assisted with Machine Learning," 2019.

[37] Google, "Sedpack v2024." [Online]https://github.com/google/sedpack/tree/main [Accessed: Sept.24, 2024], 2024.