SCAPEgoat: Side-Channel Analysis Library

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in
Electrical & Computer Engineering,
Computer Science

Submitted By:

Samuel Karkache
Trey Marcantonio

Advisors:

Professor Patrick Schaumont
Professor Fatemeh Ganji

April 25, 2024

Abstract

A rapidly growing field in hardware security is side-channel analysis. Through side-channel attacks, adversaries can extract secret information from an embedded device by measuring physical observables from the system such as power usage and electromagnetic emanation. Worcester Polytechnic Institute's Vernam Lab conducts side-channel analysis research to analyze embedded systems and their vulnerability to such attacks. However, there is no centralized API or repository of information for the researchers to use to conduct these experiments. Furthermore, there needs to be more consistency and organization regarding how Vernam lab stores experiment data. Therefore, we designed and implemented a side-channel analysis library in Python named SCAPEgoat. This library has three main modules. The custom file framework module implements an organized efficient interface for storing side-channel analysis data. The framework uses JSON to store metadata and organize the file's structure without sacrificing performance. A particular file can have one-to-many experiments with each experiment being able to have one-to-many datasets. This provides a hierarchical structure for storing experiment data similar to what is implemented in the HDF5 file format commonly used in side-channel analysis research. However, unlike HDF5, the framework uses JSON to hold metadata which is superior because it is human-readable, more flexible, and takes up a negligible amount of space. Furthermore, this metadata is queryable using regular expressions, a feature unavailable in contemporary SCA file storage methods like HDF5. The oscilloscope module implements fast custom capture procedures for ChipWhisperer devices. This saves researchers development time and greatly simplifies the process of configuring the ChipWhisperer device and using it to capture power traces. It is significantly easier to use than the standard ChipWhisperer API and aims to prevent researchers from having to write excess amounts of boilerplate code. There is support for a standard capture procedure that will execute a power trace capture for a specified number of traces. Users can supply a custom list of keys and plaintexts to provide for the encryption or a default key-text generation algorithm for AES. Furthermore, there is a dedicated capture procedure for collecting the fixed and random trace sets for the t-test metric. Finally, the metric solver module implements common side-channel analysis metrics in Python with built-in visualization options and fast performance. These metrics include signal-to-noise ratio, t-test, correlation, score and rank, success rate, and guessing entropy. These metrics also have integration with the custom file framework, allowing users to run metrics directly on stored data. The final result is a comprehensive library for side-channel analysis that can be used at WPI and beyond.

Acknowledgments

Thank you to Amit Virchandbhai Prajapati, Dev Mehta, Dillibabu Shanmugam, and Mohammad Hashemi for helping us with our implementation, giving us existing code to integrate into the library, and allowing us to use equipment in the lab. We would also like to thank our advisors Professor Patrick Schaumont and Professor Fatemeh Ganji for their guidance and advice throughout the project.

Contents

1 Introduction **1**

1.1 Project Motivation . . . 1

1.2 Side-Channel Analysis Background . . . 1

1.2.1 Side-Channel Analysis Experiments . . . 2

1.2.2 Power Analysis . . . 2

1.2.3 Metrics . . . 4

1.2.3.1 Signal-to-Noise Ratio . . . 4

1.2.3.2 T-test TVLA . . . 5

1.2.3.3 Pearson Correlation . . . 5

1.2.3.4 Score and Rank . . . 6

1.2.3.5 Success Rate and Guessing Entropy . . . 6

2 Project Overview **7**

2.1 Custom File Framework Overview . . . 7

2.2 Side-Channel Analysis Metric Solver Overview . . . 8

2.3 Oscilloscope Interface Overview . . . 9

2.4 Online Documentation Overview . . . 9

3 Results **10**

3.1 File Framework Implementation . . . 10

3.1.1 Structure . . . 10

3.1.2 File Creation . . . 13

3.1.3 Adding/Deleting Experiments and Datasets . . . 13

3.1.4 User-Specified Metadata . . . 14

3.1.5 Metadata Querying . . . 15

3.1.6 File Structure Integrity Features . . . 16

3.1.7 Metric Integration . . . 16

3.1.8 File Framework Benchmarking . . . 18

3.2 Metric Solver Results . . . 19

3.2.1 Signal-to-Noise Ratio Implementation and Verification . . . 19

3.2.2 T-test Implementation and Verification . . . 20

3.2.3 Correlation Implementation and Verification . . . 22

3.2.4 Score and Rank Implementation and Verification . . . 23

3.2.5 Success Rate and Guessing Entropy Implementation and Verification . . . 25

3.2.6 Metric Benchmarking . . . 27

3.2.6.1 Signal-to-Noise Ratio Benchmark . . . 27

3.2.6.2 T-Test TVLA Benchmark . . . 27

3.2.6.3 Correlation Benchmark . . . 28

3.2.6.4 Score and Rank Benchmark . . . 28
3.2.6.5 Success Rate and Guessing Entropy Benchmark . . . 29
3.3 ChipWhisperer Oscilloscope Interface Implementation . . . 30
3.3.1 Standard Capture Procedure . . . 31
3.3.2 T-Test TVLA Capture Procedure . . . 33
3.3.3 ChipWhisperer to File Framework Adapter . . . 34
3.4 Existing Code Integration . . . 35
3.4.1 Differential Power Analysis (DPA) . . . 35
3.4.2 Lecroy Scope Interface . . . 36
3.5 Read The Docs Online Documentation . . . 37
4 Conclusion **38**
4.1 Project Outcome . . . 38
4.2 Future Work . . . 38
References **40**

1 Introduction

1.1 Project Motivation

Many years of cryptographic research have focused on the software security of different encryption algorithms. One of the most well-known encryption algorithms, AES-128, has a time complexity of 2^{128} to brute force with improved software attacks only being roughly 3% to 17% faster[1, 2]. However, there exist hardware attacks that could extract this cryptography key in mere hours. A side-channel attack can use the information from device power consumption and electromagnetic emanation to guess a cryptographic key with immense accuracy [3]. Therefore, side-channel analysis is of utmost importance if engineers want to design cryptographic devices that can withstand even the most intense side-channel attacks. Standardization bodies such as the European Commission and NIST recognize the threat of side-channel attacks, which stresses the importance of effective research in the field [4, 5]. One of the research topics at Worcester Polytechnic Institute's Vernam Lab [1] involves side-channel analysis attacks and defense. In attacking encryption algorithms with different side-channel techniques, researchers can get insight into the security of different cryptographic implementations and develop better defenses against such attacks. However, Vernam Lab and Worcester Polytechnic Institute as a whole lack a unified repository of information and internal tools to help with side-channel analysis research and education. The lab has access to the specialized equipment needed to conduct SCA research; however, there is no standardized interface for side-channel data collection, analysis, and data storage for the lab to use to conduct research. Therefore, this necessitates an accessible set of tools that allow researchers and students alike to learn and contribute to the ever-growing field of side-channel analysis.

1.2 Side-Channel Analysis Background

Side-channel attacks exploit relationships between physical observations of a device under test (DUT) and the data that it is processing. Side-channel analysis (SCA) involves evaluating certain characteristics of a cryptographic system and how they relate to the overall security from these hardware attacks [6, 3].

[1] Vernam Lab Website: https://vernamlab.org/

1.2.1 Side-Channel Analysis Experiments

Typically, side-channel analysis experiments consist of several components. Firstly, a device under test (DUT) is identified. This device is what runs the encryption algorithm being analyzed. Most encryption algorithms take some data input (usually called the plaintext) and a cryptographic key and map it to an algorithm output (usually called the ciphertext) [3, 7]. A physical observable from the DUT is collected by the attacker/evaluator during the encryption and associated with the cryptographic input and/or output. From the perspective of an evaluator, the encryption key related to each side-channel measurement is also known. In many cases, the amount of data that needs to be collected is massive to exploit the data-dependency and operation-dependency of the side-channel measurements [3]. From the perspective of an attacker, the end goal is recovering the encryption key being used to execute encryption on the DUT. The evaluator usually wants to determine the relative difficulty of extracting the key in terms of time and space complexity.

1.2.2 Power Analysis

The particular focus of this paper, will be power analysis which focuses on observing the power usage of a device and how it relates to the cryptographic information that it is processing. One such side-channel attack that uses power analysis is differential power analysis (DPA). A differential power analysis attack attempts to extract a secret key by collecting a large number of power traces over a fixed timespan [3, 7]. The power consumption traces collected from the DUT are analyzed as a function of input data (usually the plaintext and key). Mangard in [3] outlines five main steps DPA-related attacks rely on.

1. **Intermediate Algorithm Result**: The attacker must define an intermediate result of the algorithm that is running on the DUT. Mangard defines it as a function $f(d, k)$ where d is usually the plaintext or ciphertext of the encryption and k is a subkey guess [3]. An example of an intermediate result is given in chapter 10 of *The Hardware Hacking Handbook* for attacking an AES-128 algorithm[8]. The function can be seen in Equation 1 below.

$$f(d, k) = sbox[d \oplus k] \tag{1}$$

   This intermediate result targets the sbox output of an AES encryption algorithm. The AES sbox can be seen in Figure 4 in [7]. A key guess, k, is XORed with a plaintext input and put into an sbox lookup table. In this scenario, the AES sbox is known beforehand by the attacker.

2. **Power Trace Collection**: The attacker must record power traces from the DUT when data is being

encrypted on the device and associate them with the data provided to the algorithm. Each trace has a plaintext/ciphertext value that corresponds with the power measurement. Assume that there are a total of D data values and each measured power trace has a length of T. The attacker would have a matrix of size $D \times T$ since each data block $d \in D$ has a corresponding power trace $t \in T$ [3].

3. **Hypothetical Values**: Now, for each possible value of the encryption key (typically a partition of a full-length key), hypothetical intermediate values must be created based on the data associated with the power trace collection. As a result, each data value inputted into the DUT will have a corresponding hypothetical algorithm output based on each possible value of the key. Assuming that there are K possible key values, the matrix, V, would be of size $D \times K$ since each $d \in D$ has a corresponding hypothetical algorithm output for each key value $k \in K$ [3].

4. **Intermediate Value, Power Consumption Mapping:** Each hypothetical value in V now must be mapped to a hypothetical power consumption value using power simulation models such as the Hamming-Distance model or the Hamming-Weight model [3, 8, 7]. However, more complex leakage models exist such as the high dimensional leakage models discussed in [9]. This results in matrix H which will represent the mapped hypothetical power consumption.

5. **Compare Hypothetical Power Traces and Collected Power Traces**: Finally, the attacker must compare the hypothetical power usage in matrix H to the actual measured power traces in matrix T. Each column of matrix T represents a power trace associated with some data $d \in D$. Each column in V represents a series of hypothetical power usage values associated with a piece of data $d \in D$. The attacker must compare each of these columns for each possible key value $k \in K$ [3]. There are many different ways that the comparison can be done, one of which is Pearson Correlation. The resulting matrix R is of size $K \times T$ containing the result of the comparison algorithm between the columns of T and H. Each column of R is associated with a key guess $k \in K$. If a given element in $r_{i,j} \in R$ has a maximum value compared to all elements in R, then we can conclude that it is likely that column i corresponds to the correct key guess k_c since it indicates the columns of T and H are the most closely related [3]. It is important to note that all $r_{i,j} \in R$ are approximately equal in magnitude, then not enough traces were collected.

There are many ways that a power analysis attack can be conducted that are different from the method presented above. Furthermore, system evaluators may not even have to run a full attack to draw security conclusions about a given cryptographic device. However, in almost all cases, from both an attacker and evaluator context, a large set of power traces that relate to the input data that generated the trace is required.

1.2.3 Metrics

Side-channel analysis metrics, particularly relating to power analysis, can be used to draw conclusions about the security of a cryptographic system without having to launch an entire attack. Listed below are some of the most comment metrics that relate to the power analysis of cryptographic systems.

1.2.3.1 Signal-to-Noise Ratio

The signal-to-noise ratio of a given signal measurement is defined as the ratio of a signal's data component to the signal's noise component [10]. For side-channel analysis, the SNR of a power trace relates to the ability of an attacker to obtain information from a power trace during a side-channel attack [6, 10]. The effectiveness of power analysis attacks increases for larger SNR values since the signal leakage is more prominent relative to the noise of the signal. If leakage of the signal portion of the power trace is defined as L_d and the noise as L_n, then the formula for SNR would be defined in Equation 2 as follows [6].

$$SNR = \frac{VAR(L_d)}{VAR(L_n)} \tag{2}$$

To implement the SNR metric for side-channel analysis, an attacker must first measure the power traces from the DUT that they want to be analyzed. Next, each power trace must be assigned to a corresponding *label*. A power trace's label is arbitrary and depends on what the attacker is measuring. For instance, the label could be an intermediate value, V, associated with the trace. In this case, the power traces would be split into sets, $l_0, l_1 ... l_V$, such that each set, l_v, contains all power traces where the intermediate value is $V = v$ [6]. Once the power traces are partitioned into their corresponding sets, the statistical mean, $\hat{\mu}_n$, of each set is calculated along with the overall mean of all sets, $\hat{\mu}$. Then, the evaluator calculates the statistical variance of each set and then takes the overall average. Finally, the signal-to-noise ratio is calculated according to Equation 3 below [6].

$$SNR = \frac{VAR(L_d)}{VAR(L_n)} = \frac{\sum_{v=0}^{V} (\hat{\mu_v}^2 - \hat{\mu})^2}{\hat{\sigma}^2} \tag{3}$$

The resulting trace is the value of the signal-to-noise ratio at a given sample. Windows of the resulting trace where the magnitude of the SNR is high may also indicate an area of interest since it implies that there exists a significant amount of leakage at those time samples [10]. Secure embedded implementations can introduce large amounts of noise, increasing the value of $VAR(L_n)$ and thus lowering the SNR spikes. Therefore, designs with enough noise would make the amount of traces that need to be collected too large for a prospective attacker [11].

1.2.3.2 T-test TVLA

The goal of the t-test metric is to assess a device's security by determining its relative venerability using hypothesis testing [12]. A useful TVLA configuration for side channel analysis includes testing a device with a fixed key by sending deterministic and non-deterministic plaintext values [13]. The resulting traces are separated into two different subsets, L_{rand} and L_{fixed} corresponding to if they were recorded from fixed or random plaintext values. Using statistical hypothesis testing where H_0 corresponds to the device being secure and H_1 indicating that the device has security flaws, the following hypothesis test in Equation 5 can let us draw security conclusions about the DUT [6].

$$\hat{\mu}_i = \frac{1}{n_i} \sum_{l \in L_i} l \quad where \quad i \in \{rand, fixed\} \tag{4}$$

$$H_0 : \hat{\mu}_{rand} = \hat{\mu}_{fixed} \quad H_1 : \hat{\mu}_{rand} \neq \hat{\mu}_{fixed} \tag{5}$$

The t-statistic value, t, can then be calculated as shown in equation 7 which corresponds to the equation for student's t-test. [6, 14].

$$\hat{\sigma}_i^2 = \frac{1}{n_i - 1} \sum_{l \in L_i} (l - \hat{\mu}_i)^2 \tag{6}$$

$$t = \frac{\hat{\mu}_{rand} - \hat{\mu}_{fixed}}{\sqrt{\frac{\hat{\sigma}_{rand}^2}{n_{rand}} - \frac{\hat{\sigma}_{fixed}^2}{n_{fixed}}}} \tag{7}$$

H_0 will be rejected if $|t|$ is greater than some threshold th. This threshold is commonly set to $|th| = 4.5$, a value that minimizes the possibility of Type I errors [15]. Therefore, we can conclude that the DUT is leaking information if the t-statistic passes the given threshold [15].

1.2.3.3 Pearson Correlation

Another powerful metric that can be used in side-channel analysis, particularly in correlation power analysis attacks, is Pearson's correlation coefficient. The correlation is found between observed power traces, L, and hypothetical power consumption relating to the leakage of some intermediate value $v_1 = f(x_i, k_i)$ [6]. The hypothetical power consumption is generated via a leakage model $g(v_i)$. For example, the hamming weight leakage model counts the number of set bits in the output of the AES sbox. The correlation relating to a

key candidate is given in Equation 8 as established in [16, 6].

$$p_k = \frac{\sum_{i=1}^{n}(l_i - \frac{1}{n}\sum_{i=1}^{n} l_i)(g(f(x_i,k)) - \frac{1}{n}\sum_{i=1}^{n} g(f(x_i,k)))}{\sqrt{\sum_{i=1}^{n}(l_i - \frac{1}{n}\sum_{i=1}^{n} l_i)^2}\sqrt{\sum_{i=1}^{n}(g(f(x_i,k)) - \frac{1}{n}\sum_{i=1}^{n} g(f(x_i,k)))^2}} \tag{8}$$

An attacker can calculate the correlation coefficient for each $k \in K$. The correlation relating to the correct key candidate p_{k_c} will have a larger magnitude than other key candidates assuming that enough traces were collected and the leakage model assumptions were correct [6]. From an evaluator context, the goal would be to minimize the magnitude spikes in p_{k_c} for large trace sets. As previously mentioned, correlation can also be used as a distinguisher for differential power analysis as well.[7, 16]

1.2.3.4 Score and Rank

The score and rank metric is a helpful metric to use both during an attack and in the analysis of a system. This metric relies primarily on two steps, the first being that the full-length cryptographic key is split into multiple subkeys [17]. Typically these partitions are the size of a byte but can be either larger or shorter depending on the particular encryption algorithm and user implementation. This means that for partitions that are the size of a byte, there are 256 key possibilities. Next, a scoring function needs to be specified. This function is arbitrary but needs to return numerical scores such that the higher the score, the more likely a given input key, k_c, actually produced the traces. One example of such a function is Pearson's correlation coefficient. Using the scoring function, for each partition, each possible key is ranked from highest score to lowest score. We will end with a vector with the first element being the key with the highest score [17]. The idea of ranking and scoring the key guesses is that as the number of traces increases, the rank will converge to the point that the actual key will remain in the 1st rank, or very close to it [6]. The goal of the evaluator would be to prevent this convergence for their cryptographic implementation.

1.2.3.5 Success Rate and Guessing Entropy

In the analysis of a system, the Success Rate and Guessing Entropy metrics can be used alongside the Score and Rank metrics to help determine the security of a system [6]. They are specified in Equation 9 and Equation 10 shown below as defined in [6]

$$SR_o^i = \begin{cases} 1 & \text{if } k_c \in [guess_1, guess_2, ..., guess_o] \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$SR_o = \frac{1}{p}\sum_{i=1}^{p} SR_o^i \tag{10}$$

The success rate is calculated over p experiments. If for a given experiment, the correct key, k_c, is within a certain number of ranks, specified by the variable o, the success rate has a value of 1 [6, 18]. Otherwise, the value of the success rate is 0. This computation is done for all p experiments and averaged. A success rate of 1 indicates a successful attack, with the relative success of the attack decreasing as the success rate decreases. While the success rate metric can help to show the overall success of an attack, it does little to show how much work is left to be done unless it is performed multiple times over many orders [6] This is where the guessing entropy metric comes into consideration. Using Equation 11 and Equation 12, the guessing entropy can be calculated and express the average workload left in the attack [6, 19].

$$GE^i = log_2(rank_{k_c}) \tag{11}$$

$$GE = \frac{1}{p}\sum_{i=1}^{p} GE^i \tag{12}$$

A low guessing entropy implies a high success rate and as the guessing entropy decreases the certainty of the key guess increases, to the point where at a GE of 0, the key guess is certain [6, 20].

2 Project Overview

Our Major Qualifying Project (MQP) involves developing a set of side-channel analysis tools that provide useful functionality for researchers of side-channel analysis and students looking to learn about the field. This goal will be realized by implementing three distinct modules packaged together into an open-source Python library that researchers at Vernam Lab and around the world can utilize for their side-channel experiments. Furthermore, there exists code used in Vernam lab that exists in locally saved Python notebooks. Standardizing this existing code will ensure that all researchers use the same implementations while conducting research. Finally, providing robust documentation will make the field of side-channel analysis more accessible for newcomers and allow researchers to adopt our library very easily.

2.1 Custom File Framework Overview

The current way that research labs at WPI store experiment data is inefficient and disorganized. Therefore, we created a custom file framework that provides a standardized structure to store data from side-channel

Figure 1: File Framework Hierarchy

analysis experiments. The file itself will contain one-to-many experiments. Each experiment can have one-to-many numerical datasets that relate to that experiment. This allows the data collected to be stored logically and makes it easier for researchers to reference it later in future experiments. A diagram of this structure can be seen in Figure 1. While collecting side-channel data, oftentimes there are specific conditions that researchers want to associate with an experiment. Therefore, another feature requested by researchers is the ability to associate arbitrary metadata with datasets and experiments. This would entail allowing for arbitrary data to be associated on the Experiment Class level as well as the Dataset Class level. This metadata can be used to quickly gather related experiments and datasets. These features are either unavailable in contemporary SCA data storage methods or are more difficult to utilize effectively. Overall, the main goal of this module is to establish a standard in side-channel data storage. There are numerous reasons as to why this would be beneficial to the field as a whole. Firstly, it would enhance collaboration in a laboratory setting. A particular researcher can send data to a colleague with confidence that the colleague will know how to interface with the file structure and will understand the experiment structure. Furthermore, standardization of side-channel data storage will allow for inter-university collaboration. Researchers at different institutions have different methods of storing experiment data. By establishing a standard for side-channel researchers, each university will be able to understand how to interpret and use data collected from other institutions. Full details regarding the implementation can be found in Section 3.1 of the report.

2.2 Side-Channel Analysis Metric Solver Overview

A centralized repository of useful side-channel analysis metrics is something that would be very useful for researchers of side-channel analysis. Some side-channel analysis libraries exist such as MetriSCA [21]. However, existing libraries like MetriSCA are incomplete, developed in harder-to-use languages such as C or C++, and documented poorly. The implementation of this module would ensure that all researchers are running identical implementations for their experiments along with providing the fastest possible implementation for all users in a Python development environment. The developed metrics are the ones mentioned in Section 1.2.3 which include signal-to-noise ratio, t-test, Pearson correlation, score and rank, success rate, and guessing entropy. Furthermore, to help with the usage of these metrics, we defined the hamming weight and hamming

distance leakage models which can be used to analyze certain AES-128 implementations. Users will have the ability to define custom leakage models to use with the metrics if, for example, they are not analyzing the AES-128 algorithm or want a more robust leakage model. Each metric will also have pre-defined visualization capabilities to save researchers time if they want to visualize the result. The visualization features in the metrics will have integration with the custom file framework to save an image of the visualization in a specific experiment directory. To ensure that the metrics are implemented correctly we will run them on a common dataset and compare what our metrics produce with known results published in academic papers. The authors in [6] provide many figures that can be used for this verification along with the dataset that they used. The implementation of each metric along with benchmarks will be discussed in Section 3.2 of this report.

2.3 Oscilloscope Interface Overview

One of the most important parts of power analysis experiments is the collection of trace data. The oscilloscope module creates an interface for easy power trace collection using pre-defined capture procedures. Many researchers in Vernam Lab and research labs across the world use ChipWhisperer devices to capture side-channel analysis data. This module will use the open-source ChipWhisperer API [22] to quickly configure the device and capture power traces. There will be further integration with the custom file framework module to allow researchers to quickly configure a ChipWhisperer target board, capture traces using the built-in oscilloscope (if present), and save them to our custom framework. The lab also has a set of Lecroy digital oscilloscopes that they use to capture trace data. Fortunately, there already exists a Python notebook that the laboratory uses to interface with Lecroy scopes. This can be easily integrated with our library and documented so that it can be used more effectively. Implementation details of the capture procedures and supported oscilloscopes with be discussed in Section 3.3 and Section 3.4.2 of this report.

2.4 Online Documentation Overview

As previously alluded to, all of the functionality that we develop and integrate into this library will have robust documentation that outlines its usage. The rationale behind this is that it allows students who do not know much about side-channel analysis to be able to learn more about the field. Furthermore, it enables researchers to better utilize the tools that we are providing them. The documentation will be hosted online so that it can be accessible from anywhere and that an individual can read it without having to download the library itself. Each class and function in each module will have documentation that outlines what it

does, the function's parameters, and what it returns. This documentation will also be present as docsrings in the source code itself. The Sphinx framework [23] provides an easy way to compile markdown into HTML and CSS which makes the creation of documentation efficient.

3 Results

The library was developed in Python and hosted on a GitHub repository[2] as a part of the Vernam Lab GitHub organization. In this section, we will talk about the implementation of each aspect of the library and how it can be used.

3.1 File Framework Implementation

We implemented a custom file framework for side-channel analysis. Users can interface with this framework using a Python API that we developed. Using the Python function calls, users can store and organize their side-channel experiments and data easily. We will discuss the structure of the framework, some of the included features, and show how a user can use it in a research environment. We describe how we utilize JSON [24] for metadata storage and NumPy [25] binaries for numerical storage. Furthermore, we benchmark the file framework to show that the design does not sacrifice performance. The source code can be found in the FileFormat.py file in the GitHub repository.

3.1.1 Structure

The structure that was implemented for the file framework module closely follows the structure presented in Section 2.1 and visualized in Figure 1. We chose to model the framework using a hierarchical structure similar to what the authors in [26] implemented using HDF5. At the top level is the file parent directory which is represented in the Python API as the `FileParent` class. This is the base directory of the file framework. All paths of the experiments and datasets are pathed relative to the parent directory. The path of the base directory is specified during file creation when the user calls the `FileParent` constructor. The advantage to this is that the programmer needs to only remember one path for all of the content stored in the parent directory. Inside the parent directory is another directory named "experiments". This directory contains all experiments associated with the current file. The parent directory also contains the metadata file. We chose to store metadata using JavaScript Object Notation (JSON) [24]. The JSON

[2] https://github.com/vernamlab/SCApeGoat

Figure 2: Example JSON Structure

file, named `metadataholder.json`, is what enforces the file framework structure and holds critical information related to the experiments and datasets. The JSON tracks the name of the file, the user-specified metadata, and a list of all experiments in a file. Each experiment in the list contains its path (relative to the parent directory), name, metadata, and a list of datasets. Finally, each dataset specified in the list of an experiment contains the dataset name, path (relative to the experiment), and metadata. The JSON file is created and populated upon the creation of the file, experiment, or dataset by the user. Figure 2 shows an example JSON file with two experiments. In this example, the file's name is `"anotherFile"` as specified by the `"fileName"` field. The next field, the `"metadata"` key, stores arbitrary key-value pairs that can be specified by the user. By default, a `"dateCreated"` field is added when the file is first created. This particular file was created on April 3, 2024. The user can add additional fields if they desire by modifying the metadata. This feature will be discussed in length in Section 3.1.4. The third field in Figure 2 is the `"path"` field. This contains the absolute path to the parent directory. This file is located at `"C:\\Users\\samka\\PycharmProjects\\SCLA_API_MQP\\AnotherFile"`. The next field in Figure 2 is

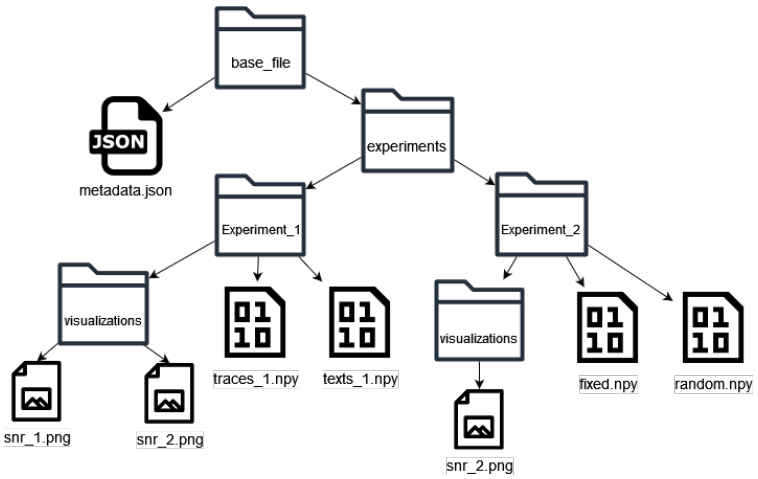

Figure 3: Example File Hierarchy

the `"experiments"` field which holds a list of all experiments in the file. In our Python library, experiments are represented using the `Experiment` class. Each experiment contains additional fields in the JSON structure. The `"path"` field for an experiment contains the path to the experiment directory relative to the absolute path to the parent file. For the first experiment in Figure 2, the path field has a value of `"\\Experiments\\experiment1"` which is a relative path to the base directory path specified above. The experiment also has a name field. We can see the names of the two experiments in Figure 2 are `experiment1` and `experiment2`. Similar to the metadata field for the entire file, each experiment also has a metadata field that is user-specified. There is a list of datasets for each experiment with their own fields as well. Datasets are represented in Python using the `Dataset` class. We can see that there is a dataset in `experiment1` named `"random"`. Its path is specified as `"\\random.npy"` which is relative to the path of the experiment directory. The dataset has a metadata field just like the parent directory and the experiment. Finally, as seen in Figure 2, there are multiple instances of a field named `"index"`. This is used internally by our API to create this structure and has no semantic meaning to the user. Experiments have visualization directories where the visual results of experiments can be stored. These directories are not represented in the JSON structure since their contents do not need to be nested. Figure 3 depicts a file hierarchy diagram for an example file with two experiments with two datasets each and a few visualizations. Again, this structure is strictly enforced using the `metadataholder.json` file during file creation. The visualization folders are contained at the experiment level. The example in Figure 3 shows `PNG` files but it supports any file type.

Listing 1: Creating a FileParent Object

```
name = "VernamLab"
path = "C:\\users\\username\\documents\\"
file = FileFormatParent(name, path, existing=False)
```

3.1.2 File Creation

The file framework was implemented in Python by using an object-oriented class structure in the `FileFormat.py` file in our repository. At the top level, the `FileParent` class acts as the primary reference for all information contained in the file. The constructor of this class is used to create the file for the first time or to get a reference to an existing file. The constructor for the `FileParent` class takes the name of the file, an absolute path, and an optional parameter indicating whether or not the file already exists. By default, the `existing` parameter is set to false. Let's assume that a user wants to create a file named `VernamLab` on their C drive in their documents directory. To make this file the user would indicate the file name and an absolute path to their documents directory as shown in Listing 1. Running this code will create the parent file in the directory specified. This code also creates the JSON file along with an empty experiments directory. If a file by this name and located in the same directory already exists, it will still be created except that the name will be changed slightly. If a file named `VernamLab` is already in the downloads directory, the program will rename the new file to `VernamLab-1`. If the names `VernamLab` and `VernamLab-1` already exist in this directory the program will create the file with the name `VernamLab-2` and so on. This behavior mimics how Windows deals with files with the same name in the same directory. The idea behind this feature is to prevent researchers from having to capture or parse data only to have it not saved because the directory they want to create already exists.

3.1.3 Adding/Deleting Experiments and Datasets

Creating experiments and datasets can be done once the programmer obtains a reference to the `FileParent` object by either creating a new file or referencing an existing one. Listing 2 shows how to create an experiment, named `experiment1` and add two datasets to it named `fixed` and `random`. We can assume that the `fixed_t` and `random_t` variables are NumPy arrays containing numerical power trace data from some experiments. The code in Listing 2 creates the `experiment1` directory in the file system and an entry in `metadataholder.json`. Similarly, the code saves the data in `fixed_t` and `rand_t` into NumPy binaries, as specified in the NumPy documentation [25], and adds dataset entries to `experiment1` in the JSON file. The

Listing 2: Creating Experiments and Datasets

```
experiment1 = file.add_experiment(name="experiment1")
fixed_dataset = experiment1.add_dataset("fixed", fixed_t, datatype='float32')
rand_dataset = experiment1.add_dataset("random", rand_t, datatype='float32')
```

Listing 3: Deleting Files Experiments and Datasets

```
file.delete_experiment(name="experiment1")
experiment1.delete_dataset(name="fixed")
experiment1.delete_dataset(name="random")
file.delete_file()
```

corresponding `Experiment` object and `Dataset` objects are returned by the functions upon creation. Similar to the behavior of file creation, if a user tries to create an experiment or dataset with the same name as another experiment or dataset, the framework will append numbers to the end of the dataset/experiment name and path to prevent naming collisions. If at any point the user wants a reference to any experiment, they can call the `FileParent` method called `get_experiment(name)` to get the `Experiment` object that they want by its name. Similarly, using an `Experiment` object we can call `get_dataset(name)` to get a reference to a given dataset in the experiment. The user can read the data of any dataset by getting a reference to the `Dataset` object and using the `read_all` method. Users can also delete experiments, datasets, or even the file as a whole. Listing 3 shows how the user can delete the parent file, experiments, and datasets defined in the previous two listings. The system asks the user for confirmation before deleting any file, experiment, or dataset to prevent users from accidentally deleting large amounts of data and experiments.

3.1.4 User-Specified Metadata

A defining feature of our library is the ability for users to add arbitrary metadata at the file, experiment, or dataset level. This metadata is saved in the JSON file. Researchers often want to associate parameters when conducting their research. For example, the authors in [27] explored the effects of heating circuits to induce leakage. An obvious piece of metadata that the authors [27] would want to associate with collected data is temperature. Our framework implementation would allow for such associations to be made at the file, experiment, and dataset level. Let's assume that we have the file structure defined in Listing 2 where we have an experiment named `experiment1` and two datasets named `fixed` and `random`. Let us also assume that we are interested in the temperature of the environment in which the experiment ran. The user has two options. Firstly, the user can specify the temperature metadata at the dataset level

Listing 4: Metadata Querying with Regular Expressions

```
exp = r'[1-7]?[0-9]C'
expriments = file.query_experiments_with_metadata(key="temp", value=exp, regex=True)
```

using `fixed_dataset.update_metadata(key="temp",value="30C")`. The user can then add this to the other dataset, named `random`, as well. However, assume that many datasets have the same value for their temperature. It would be extremely tedious to have to add the same metadata many times. Alternatively, the user can associate the temperature at the experiment level by calling the metadata updating function on the experiment object as follows `experiment1.update_metadata(key="temp", value="30C")`. This would indicate that all datasets contained in `experiment1` have a temperature of 30 degrees Celsius. The metadata can also be inputted manually using a text editor by putting the key-value pairs in the metadata field indicated in Figure 2. Since JSON is human readable, it is superior to the HDF5 implementation presented in [26] since it does not necessarily require a functioning development environment to view.

3.1.5 Metadata Querying

In addition to being able to specify metadata to experiments and datasets, our framework allows users to query experiments and datasets based on that metadata. This is useful when a user wants to quickly locate experiments or datasets that match some key-value pair. This is a lot faster than having to manually look through the JSON file. For example, if a user wants to find all datasets in some experiment with a temperature value of `"70C"`, they can call: `experiment.query_datasets_with_metadata(key="temp", value="70C")`. The implementation also supports the star operator which would return all datasets that have the key `"temp"` and any value by supplying `"*"` as the value. An equivalent implementation can be used for experiments called `query_experiments_with_metadata` which is called on a `FileParent` instance. For more flexibility, the querying supports using regular expressions. An optional parameter called `"regex"` can be set to `True` to use a regular expression as the value. Listing 4 shows how a user can get all experiments with temperatures less than `"80C"` using regular expressions. The regular expression, defined as `exp`, matches all strings that are a number followed by the character `'C'`. The `[1-7]?` part of the regular expression represents the first digit of the number. Since it needs to be less than 80 the only valid digits are 1 to 7. The question mark indicates that this character is optional since the number can be less than ten which will have no digit in that location. The next part of the regular expression, `[0-9]`, enforces that there is a second digit which can be any number from 0 to 9. Finally, the `C` character is required at the end of the string. If the value of the metadata matches the regular expression then it is returned in the list. At a high level, regular expressions

are extremely powerful in string matching which can be used to implement custom metadata filters.

3.1.6 File Structure Integrity Features

To enforce the file structure and protect against changes that may be made using the user's operating system file explorer, the framework has certain features to make sure the file is still usable. As explained in Section 3.1.2, users create the base directory of the file by providing an absolute path to where they want it to be stored. This path is stored in the JSON file and used to access all of the file's experiments and datasets. In practice, a file's directory may change. As such, if a file is moved from its original location, the framework will detect this and change the absolute path in the JSON as long as the user provides the new absolute path of the file. For example, assume that a user moved the file from `"C:\\Users\\username\\Downloads"` to `"C:\\Users\\username\\Desktop"` using their file explorer. When the user attempts to get a reference to the existing file now located at a different path, they use the `FileParent` constructor with the new path and the `existing` field set to `True` as follows: `FileParent(name="a_file", path="C:\\Users\\username\\Desktop", existing=True)`. The `FileParent` constructor will realize that the JSON is accessible from the new path supplied by the user but is different than the path stored in the JSON file. Since the JSON file was able to be read, this implies that the file has changed directories and the path needs to be updated to the new one supplied by the user. Therefore, the framework will update the path. Another structure integrity feature available relates to the manual deletion of experiments and datasets. If a dataset binary or an experiment directory is deleted via the file system, this will be detected in the `FileParent` constructor when getting a reference to an existing file. Suppose an experiment or dataset is listed in the JSON file but the directory itself does not exist in the file structure. In that case, the entries will be removed in the JSON automatically. This implicitly allows the user to delete experiments and datasets using the file system rather than having to run a Python script.

3.1.7 Metric Integration

The file framework has integration with the metrics defined in the metric solver module to allow researchers to quickly run metrics on data stored in the file format. Rather than having to parse and organize the data manually, users can simply specify the names of the datasets containing the data needed to run a metric. The framework will then load that data and run the metric automatically, saving users development time. Three metrics from the list specified in Section 2.2 were chosen to have integration with the file framework. The integrated metrics are signal-to-noise ratio, t-test, and Pearson correlation. These metrics were chosen for integration because they required minimal user setup and can be computed using only a few datasets.

Listing 5: Metric Integration Example

```
file = FileParent("ExampleFile", "C:\\Users\\samka\\SCLA_API_MQP\\", existing=True)
exp = file.get_experiment("T_test_Experiment")
t = exp.calculate_t_test("fixed", "rand", visualize=True, save_data=True, save_graph=True)
```

Metrics like score and rank as well as success rate and guessing entropy require too much setup to comfortably integrate with the file format. Note that there do exist stand-alone metrics that can be used without the file framework discussed in Section 3.2. At a high level, metrics integrated with the file framework require the user to supply the names of the datasets they want to supply to the metric. In addition, there are also options for the user to save the result of the metric as a dataset as well as save a photo of the metric visualization to the experiment's visualization folder. Let us assume that we want to run the t-test metric on some data that we have collected and saved in an experiment in our file framework. Listing 5 shows how we can accomplish this using an existing file. Assume that we have a fixed trace set saved at a dataset named `"fixed"` and a random dataset saved at a dataset named `"rand"`. Both of these datasets reside in an experiment named `"T_test_Experiment"`. In the call to `calculate_t_test` the programmer specified the `visualize`, `save_data`, and `save_graph` parameters as true. The `visualize` parameter simply displays a preset graph of the metric result using the `matplotlib` Python library. The `save_data` parameter specifies whether or not to save the metric result. Since the programmer set this to true, the result of the metric will be saved as a dataset in the current experiment. Finally, the `save_graph` parameter asks whether or not the user wants to save the default `matplotlib` graph to that experiments visualization folder. In Listing 5, the graph will be saved in the visualization directory of the `T_test_Experiment` experiment. As mentioned above, there is integration for the SNR and correlation metrics as well by calling `calculate_snr` and `calculate_correlation`. Predictably, they require different datasets to run the metric. The correlation integrated metric just needs the names of the observed leakage dataset and the modeled leakage dataset. In other words, it assumes that the predicted leakage has already been calculated and saved as a dataset in the file framework. Alternatively, the signal-to-noise integrated metric requires a bit more information. Since the non-integrated version of SNR requires a Python dictionary, the integration requires a bit of pre-processing. The integrated metric will organize the labels for use as long as you provide a callback function that specifies how the labels are calculated as well as the additional datasets needed to calculate these labels. Assuming that the labels the unmasked AES sbox output, we would also need to specify the names of the datasets containing the keys and plaintexts as well. The implementation of `calculate_snr` supports an arbitrary number of dataset names to be specified to support however the user wants to calculate the labels. More information on SNR labels and the SNR implementation can be found in Section 3.2.1.

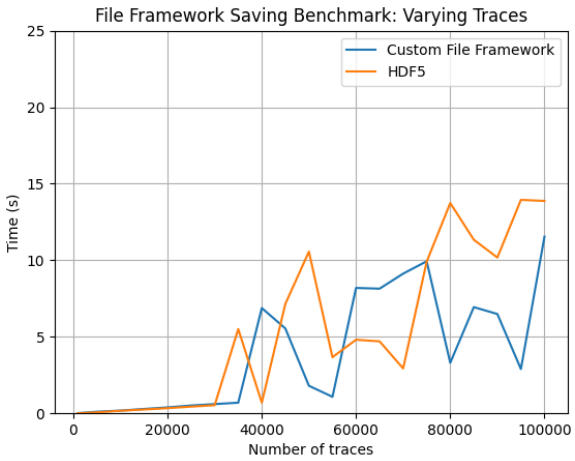

Figure 4: Data Saving Benchmark

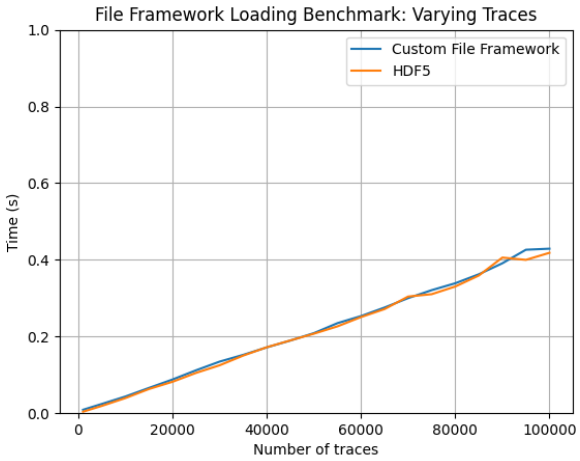

Figure 5: Data Loading Benchmark

3.1.8 File Framework Benchmarking

To get a general idea about the performance of our implementation, we conducted a benchmark to see how long it takes to save and load datasets of varying sizes. We can safely assume that the number of experiments and datasets in the file does not affect the performance of the implementation. This is because experiments and datasets are stored in Python dictionaries (more commonly known as hash maps). Getting a reference to a specific `Dataset` or `Experiment` object has a time complexity of $O(1)$ because the framework is simply indexing into the dictionary. Therefore, it would be trivial to conduct a benchmark with varying numbers of experiments and datasets. Instead, we will consider the size of the dataset itself (i.e. the number of traces). We will assume that the base file and experiment have already been created. In the first benchmark, we will the measure time it takes to save data of varying sizes to a dataset. We will also assume that the data to be added has already been collected and is in a NumPy array. Figure 4 shows the time it takes to save a trace dataset (each sample is a 32-bit float) as a function of the number of traces compared to using the HDF5 file specification in [28]. We can see the performance of both file frameworks was very similar with our implementation being slightly faster. It is important to note that many other factors can affect the results seen in Figure 4 including the datatype of the stored traces and the specifications of the computer that ran the experiment. An additional benchmark was conducted by loading the data that was saved in the previous benchmark. This benchmark can be seen in Figure 5. As we can see, there is no discernible difference between the parsing speed of the two file frameworks.

3.2 Metric Solver Results

We implemented a suite of side-channel analysis metrics in Python. The implemented metrics are listed in Section 2.2 along with pre-defined leakage models as well. In this section, we will discuss the implementation details of all metrics, prove that they are functionally correct, and benchmark their performance. Our metrics were implemented using the theoretical basis discussed in Section 1.2.3. To verify the metric's functionality and prove that they are correct we ran our metrics using the datasets in [6] and [26] and compared them with figures in those papers. If our metric results are visually identical to the referenced figures in [6] and [26] then we can conclude that our metrics are functionally correct. All metrics include built-in visualization presets using the Python library `matplotlib` [29]. This means that in addition to the parameters needed to run the metric, each implementation will have two additional parameters related to visualization. The `visualize` parameter is a boolean which when set to true will display the metric visualization. Furthermore, the `visualization_path` parameter allows the user to specify a path to a location on their computer where the visualization can be saved as a `PNG` file. Note that visualize must be set to true to save the plot. The metric implementation source code is available in the Metrics.py file on GitHub with the leakage models being available in the LeakageModels.py file.

3.2.1 Signal-to-Noise Ratio Implementation and Verification

As we know, the signal-to-noise ratio metric requires that we organize the collected power traces into multiple label sets corresponding to algorithm intermediate values [6]. Therefore there are two portions of our SNR implementation. The first function we developed, `organize_snr_labels`, creates a Python dictionary that accomplishes this task. Each key in the Python dictionary is an intermediate result with the value being a set of traces associated with it. The label organization function requires a NumPy array representing all of the collected power traces. The second parameter is a callback function that specifies how the intermediate values will be calculated. This function can be defined by the user to fit the algorithm they are evaluating. The only requirement is that it returns a NumPy array of all possible labels. We pre-defined one such function that can be used with our SNR metric. The `unmasked_sbox_output_intermediate` function uses the output of the unmasked AES sbox to calculate labels. Notice that this callback function will require the keys and plaintexts associated with each trace to calculate the intermediate values. As such, the `organize_snr_labels` function supports an arbitrary number of additional arguments that are needed for the callback function. The final output of `organize_snr_labels` will be the Python dictionary expected by the input of the SNR metric. The signal-to-noise ratio metric, defined as `signal_to_noise_ratio`, takes the label dictionary as an input

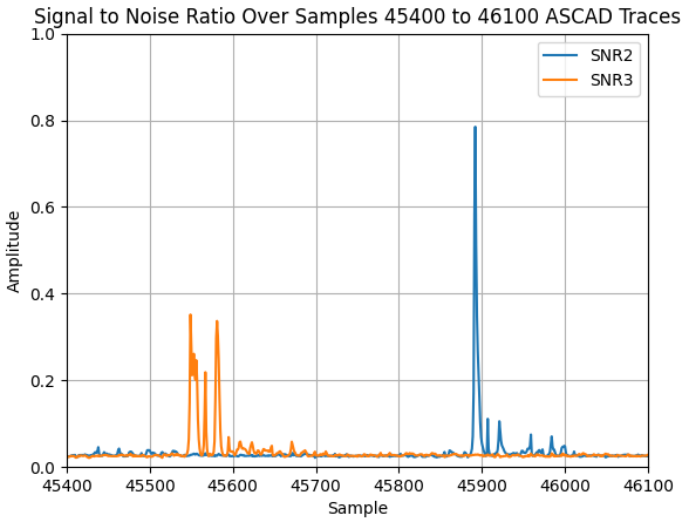

Figure 6: SNR Verification

argument. The implementation itself follows the operations presented in Equation 3 using NumPy operations such as `np.mean` and `np.var` to compute the metric result. All NumPy arrays used are pre-allocated before conducting the calculation itself to ensure the best possible performance. The result of the SNR metric is then returned after finishing the computation. To verify that all functions that we developed associated with the SNR metric worked correctly, we used the ASCAD dataset, specifically the first 10,000 traces in `ATMega8515_raw_traces.h5`, provided by the authors in [26] to test our implementation. We can then cross-reference the results of our metric using this dataset with Figure 3 in [26] which shows SNR results based on different intermediate value labels. We will focus on the left side of Figure 3 in [26] which uses the masked sbox output (SNR2) and common sbox output mask (SNR3) to calculate the labels [26]. Figure 6, shown above, is the result of our metric using the dataset provided in [26]. This matches the left half of Figure 3 in [26], verifying our signal-to-noise ratio implementation. The code used to generate this plot can be found in the `snr_verification` function in `MetricVerification.py` in the repository.

3.2.2 T-test Implementation and Verification

Our t-test implementation is fairly standard and requires very little setup. The metric, defined as `t_test_tvla`, simply requires fixed and random trace sets as NumPy arrays. The implementation itself was adapted from code that was already being used in the laboratory. The t-statistic is calculated iteratively with the value of `t_max` being added at each stage. This is significantly faster than attempting to calculate the t-statistic over the entire trace set all at once. Since this implementation was already being used for laboratory experiments

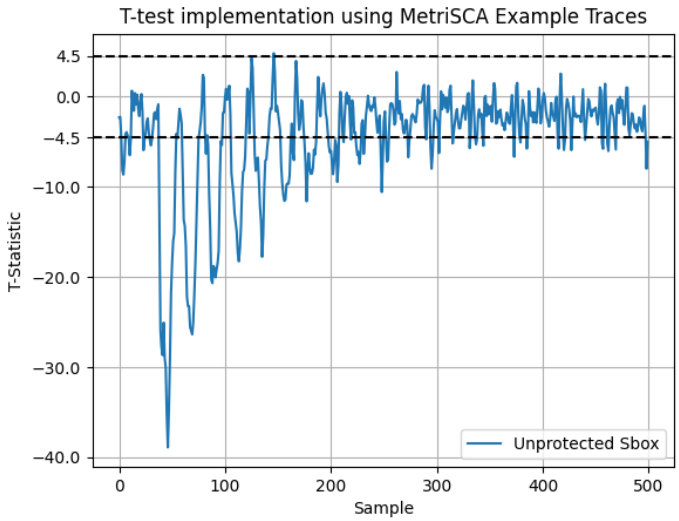

Figure 7: T-Statistic Verification

and was very efficient there was very little reason to change the implementation drastically. However, we discovered a few issues with the original implementation that we fixed once we integrated it with our library. Firstly, some of the first division operations computed in the iterative calculation have the possibility of throwing a `ZeroDivisionError`. Therefore, we put the computation in a `try-except` block and manually set the t-statistic to zero to prevent the metric from halting execution. This does not impact the overall metric and only adds a negligible amount of edge effects in the first few samples. Secondly, traditionally in TVLA, the mean of the random dataset is subtracted from the mean of the fixed dataset in the numerator as seen in Equation 7. The implementation we were given had this flipped which reverses the magnitude of the resulting t-statistic. While in practice this does not impact the meaning of the results since the threshold for security vulnerabilities is $|th| = 4.5$, we wanted to keep the implementation as similar as possible to the equations presented in [6] and decided to make the change. To verify that the metric implementation that we were given, along with all of the changes we made functioned correctly, we used the dataset provided in [6] to test our implementation. There are two parts to verifying this metric, ensuring that the t-statistic is correct and ensuring that the t-max value is correct. Figure 8(a) and Figure 8(b) in [6] will be used as a reference to confirm the result of our implementation. Firstly, Figure 7 above shows the result of the t-statistic on unprotected AES using the dataset in [6]. We can see that this is visually identical to the t-statistic of the unprotected sbox shown in Figure 8(a) in [6] by examining the peaks in the magnitude of the result. Similarly, in Figure 8 we can see the value of t-max given by our metric. The value of t-max grows as the number of traces increases before stopping just under a magnitude of 40. This is identical to the t-max of the unprotected sbox shown in Figure 8(b) in [6]. The only noticeable difference is the value

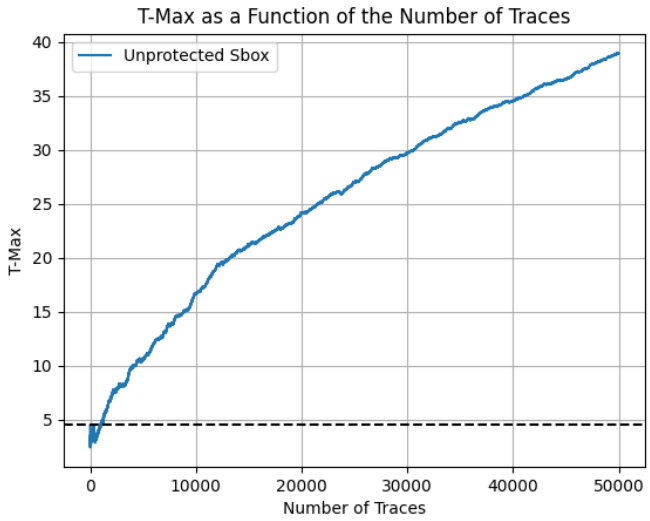

Figure 8: T-Max Verification

of t-max in the first few samples. Looking at Figure 8(b) in [6], the authors seemed to have chosen to omit those values in their graph of t-max. Regardless, whether these edge effects are included or not does not impact the overall trend that is expressed when plotting t-max vs. the number of traces.

3.2.3 Correlation Implementation and Verification

The correlation metric was implemented from the ground up and requires two parameters, observed leakage and modeled leakage. The observed leakage is simply a set of traces. The modeled leakage, however, needs to be created using a leakage model. To help programmers get started with using the correlation metric, we defined the two most common leakage models for AES: the hamming weight leakage model and the hamming distance leakage model. Both leakage models are defined in a file named `LeakageModels.py` in our repository. Both leakage model functions defined in the file return a NumPy array that represents the modeled leakage and can be plugged directly into the correlation metric itself. The correlation metric, defined as `pearson_correlation`, is implemented using NumPy operations according to Equation 8. Since multiple parts of Equation 8 require summation over large values of n, the arrays that hold the results of the sum are pre-allocated as an array of zeros. In terms of verification for the correlation metric, the authors in [26] and [6] did not provide a figure that can be referenced. However, there remain two things that we can do to ensure that the implementation is correct. Firstly, using the dataset in [6], we can run the correlation metric on some of the traces using the hamming distance leakage model. We can then see which key guess has the highest peak in correlation and compare it to the actual key. The dataset in [6] has traces collected

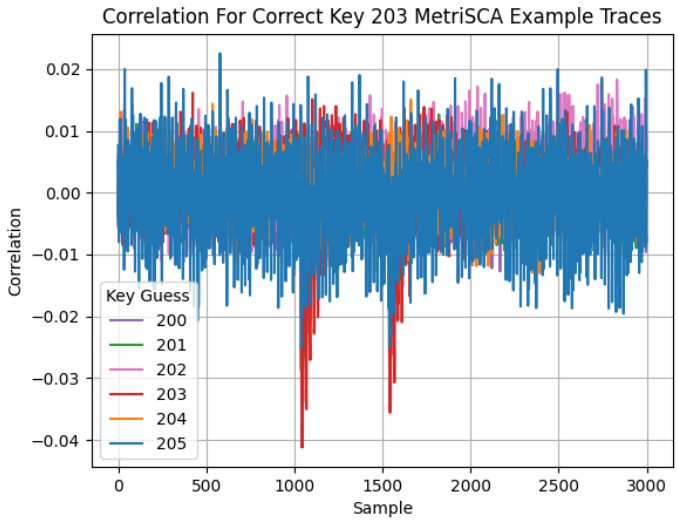

Figure 9: Correlation Metric on MetriSCA Traces

from an unprotected sbox implementation of AES. Therefore it should be relatively easy to use correlation to determine the actual key. Figure 9 shows the results of this experiment. Since the dataset provides the encryption keys used for each trace, we know that the highest magnitude for correlation should be with modeled leakage generated using key candidate $k = 203$. In Figure 9 above we can see that $k = 203$ has a clear spike in correlation magnitude. However, this may not be enough to definitively say if the correlation metric works. Fortunately, the authors in [6] use correlation to score key candidates when discussing the score and rank metric. Therefore, by verifying the functionality of the score and rank method in section 3.2.4, we can implicitly verify the correlation metric.

3.2.4 Score and Rank Implementation and Verification

The score and rank metric provides a way for researchers to define an arbitrary scoring function and use it to rank a set of key candidates. The focal point of the implementation is that the score and rank function, defined as `score_and_rank`, allows the user to supply a callback function that defines how a given key in a set of key candidates is scored. The function requires that the user supply the key candidate set, the target byte, and a set of power traces. However, how these parameters are used depends entirely on the implementation of the aforementioned scoring function. An arbitrary amount of additional arguments can be supplied if the callback function implementation requires additional data. User-defined scoring functions must be in the form `score_fcn(traces, key_guess, target_byte, ...)` and return a numerical value. The scoring function does not need to use all the required arguments, but they need to be included as

Listing 6: Score and Rank Metric

```
# all possible sub-keys
key_candidates = range(256)
# we need texts and a leakage model for the correlation scoring function
*args = (texts, leakage_model_hamming_distace)
# score with correlation, target byte 0
res = score_and_rank(key_candidates, 0, traces, score_with_correlation, *args)
```

indicated above. We pre-defined a scoring function for use with this metric. The `score_with_correlation` function uses the maximum magnitude in a correlation trace to score a key guess. The higher the value, the higher the key guess will be ranked. The correlation scoring function requires the user to supply a set of traces, a key guess, and a target byte as required. Furthermore, the correlation scoring function requires the set of plaintexts associated with the power traces and a callback to a leakage model function. The function generates the modeled leakage and uses the trace set as the observed leakage. The maximum value of the correlation trace is returned and subsequently used to score the key guess. Listing 6 shows how the score and rank method would be used in practice. This example assumes that we are targeting an 8-bit subkey for AES. All 256 possibilities are ranked using correlation. Since the `score_and_rank` function does not have dedicated parameters for supplying plaintexts and a leakage model callback, they are packed into an additional argument variable. The result, stored in the variable `res` above, is an NumPy array of tuples. The tuple contains the key and its score. The value of `res[0][0]` would be the highest-ranked key guess with `res[0][1]` being the score of the highest-ranked key guess. For verification, Figure 2(a) in [6] shows the score of the correct key as a function of the number of traces. Figure 10 below shows the result using our implementation of score and rank. Similarly, Figure 2(b) in [6] shows the score of the correct key as a function of the number of traces. Our implementation's result is shown in Figure 11. Upon close inspection, we can see slight discrepancies between the figures we present and the figures in [6]. However, these slight differences are minimal and most likely stem from differences in how the experiment was run. To collect the data necessary for the plot, we had to run the score and rank metric using varying numbers of power traces. The authors in [6] do not disclose the exact values they used for the number of traces. Therefore, we had to make an educated guess to choose the number of traces to test the metric on. Despite these differences, we can be certain that they produce the same plot by examining the general features of the graph. For example, consider the spike in rank in Figure 10 at around 5000 traces. This is also present in Figure 2(a) in [6]. That feature is too similar to be coincidental. However, what the differences indicate in this example is that we probably have more x-axis data points than the figures in [6]. We can also look at specific points on

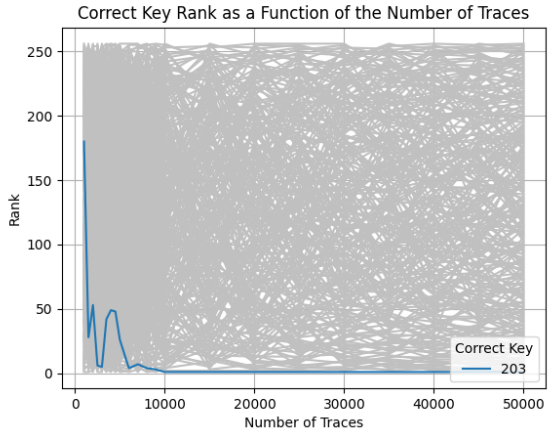

Figure 10: Rank Verification

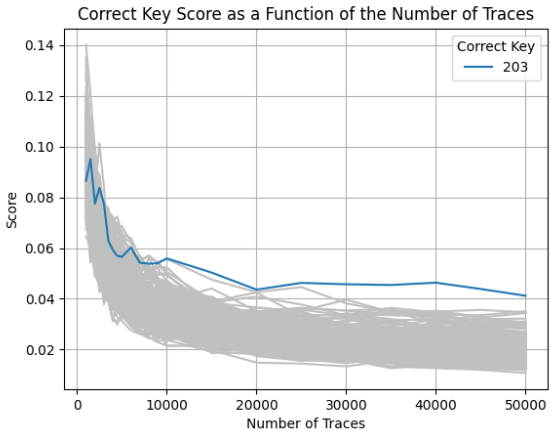

Figure 11: Score Verification

the two graphs and compare their values to the values in [6]. For example, consider the score in Figure 11 at $n = 10000$ traces. The score at this number of traces would be between 0.05 and 0.06 for the correct key. Looking at the same location in Figure 2(b) in [6] we can see that it is about the same. This can be done for any two common x-axis data points between the figures we present and the figures in [6]. Furthermore, we can use Figure 11 to definitively verify the correlation metric as mentioned previously. Since the key candidates were scored using correlation, we can think of the y-axis of Figure 11 as correlation magnitude. Since it matches Figure 2(b) in [6] we can conclude that the correlation was calculated correctly.

3.2.5 Success Rate and Guessing Entropy Implementation and Verification

The success rate and guessing entropy implementation is generally straightforward and requires that the score and rank metric be run multiple times. The metric operates in terms of experiments with each experiment having a correct key and a set of ranks. Therefore, the user needs to supply an array of correct keys for each experiment. Furthermore, the user needs to supply an array where each index is a set of ranked key candidates (this ends up being a three-dimensional array). The user also needs to supply a value for the order of the success rate calculation (i.e. within how many ranks the correct key needs to be for the success rate to be 1). Upon supplying this information the success rate and guessing entropy can be calculated. Listing 7 shows an example of how to use this metric.

Listing 7: Success Rate and Guessing Entropy Metric

```
# let's do ten experiments, assume that the correct key for each is 203
num_exp = 10
exp_ranks = []
correct_keys = [203] * num_experiments
for i in num_experiment:
    ranks = score_and_rank(...) # run score and rank
    exp_ranks.append(ranks)
s, e = success_rate_guessing_entropy(correct_keys, exp_ranks, order=1, num_exp)
```

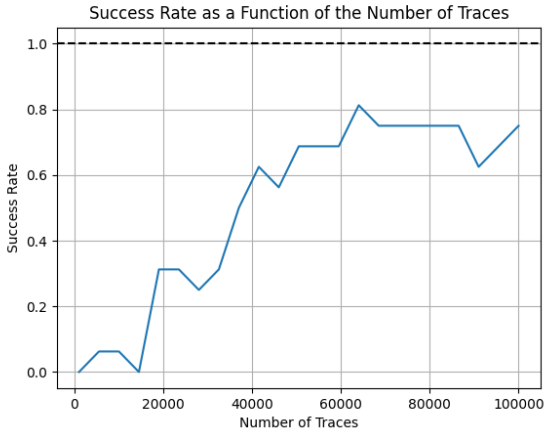

Figure 12: Success Rate Verification

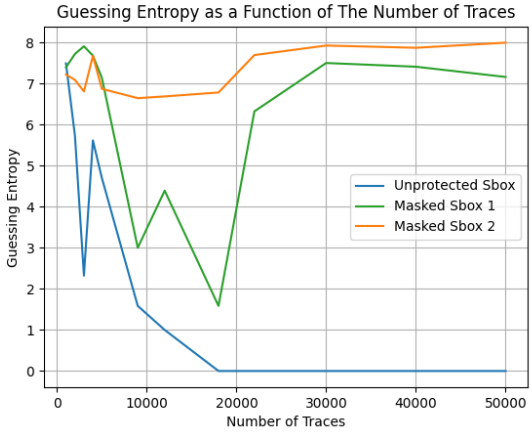

Figure 13: Guessing Entropy Verification

For verification we are faced with a very similar situation as above. We do not have access to the number of traces that the authors in [6] used to create the graph. Figure 12 and Figure 13 shown above were created by running the success rate and guessing entropy metric over varying amounts of traces. They were generated using the dataset provided by the authors in [6]. Figure 12 and Figure 13 correspond to Figure 4 and Figure 5 in [6] respectively. Figure 13, outlining the guessing entropy as a function of the number of traces, clearly matches its counterpart, Figure 5 in [6]. However, Figure 12, outlining the success rate as a function of the number of traces, is more difficult to compare to its counterpart, Figure 4 in [6]. Therefore, in an attempt to verify Figure 12 as correct, we overlaid Figure 4 in [6] with Figure 12, the result of which can be seen in Figure 14. Now that the aspect ratio is the same and the two plots are on top of each other, it is a lot easier to verify the success rate metric with the figure provided in [6]. Since the plot of the success rate is so jagged, discrepancies are most likely due to data points being taken at different values for the number of traces. However, common data points match between the two plots.

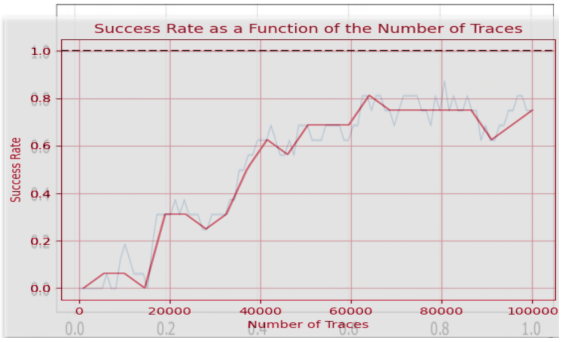

Figure 14: Success Rate Verification Overlaid

3.2.6 Metric Benchmarking

All of the above metrics were benchmarked on the WPI Turning servers. These servers run Linux and were set up to run with 16GB of RAM across a singular node utilizing a singular cluster. For each metric (except for success rate and guessing entropy) two different graphs were produced. The first shows the execution time as a function of the number of traces for varying numbers of samples per trace. The second shows the execution time as a function of the absolute file size. In both cases, we will be able to generalize the relative performance of the metrics as a function of the amount of data processed.

3.2.6.1 Signal-to-Noise Ratio Benchmark

Benchmarking was completed for SNR by performing the metric on random trace data with a range of traces from 1,000 to 100,000 and a range of samples per trace of 1,000 to 50,000. Figure 15 shows the result of this benchmark. Figure 16 shows the execution time as a function of the file size of the processed traces. We can see as the amount of data increases, the execution time increases linearly. Furthermore, Figure 16 shows similar behavior when analyzing file size. Based on these benchmarks we can estimate that the signal-to-noise ratio benchmark can process about one hundred million data points per second.

3.2.6.2 T-Test TVLA Benchmark

Benchmarking was completed for the t-test metric by running the metric on random trace data with a range of traces from 1,000 to 100,000 and a range of samples per trace of 1,000 to 50,000. The result of the benchmark can be seen in Figure 17. The execution time of the t-test metric increases linearly as the number of traces increases. Figure 18 shows the execution time as a function of the file size of the processed data. Based on Figure 17 and Figure 18, the t-test metric can process about 16 million data points per second.

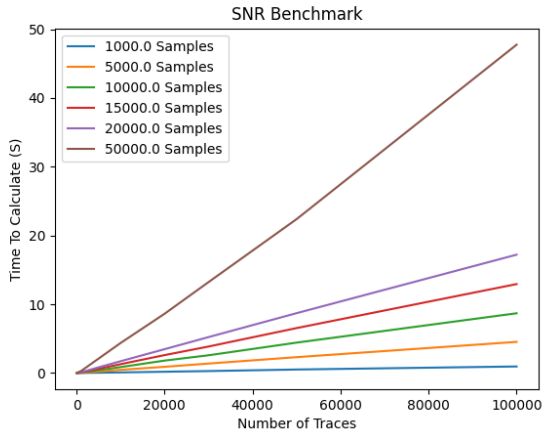

Figure 15: Signal-to-Noise Ratio Benchmark

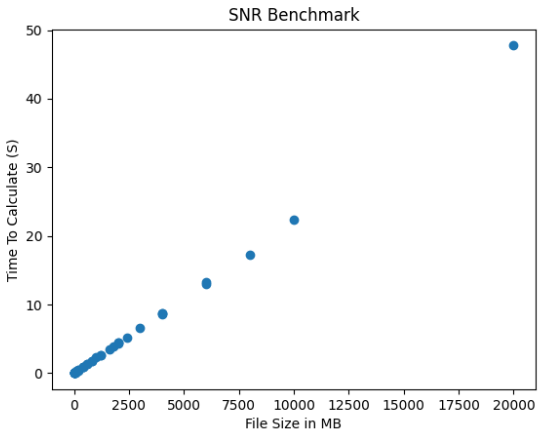

Figure 16: Signal-to-Noise Ratio Benchmark as a Function of Dataset Size

3.2.6.3 Correlation Benchmark

We performed the benchmark for correlation on random trace data with a range of traces from 1,000 to 100,000 and a range of samples per trace of 1,000 to 50,000. The benchmark result can be seen in Figure 19. There is a linear relationship between the number of traces and the execution time. Figure 20 shows the execution time of correlation as a function of the file size of the dataset provided which also increases linearly. Our correlation metric is very efficient, being able to process about 250 million data points per second. It is important to note that this benchmark does not account for the time it takes to generate modeled leakage.

3.2.6.4 Score and Rank Benchmark

We benchmarked the score and rank metric using the pre-defined `score_with_correlation` function in our library. This scoring function used the hamming distance leakage model. Furthermore, we used a set of all 8-bit numbers as key candidates. We benchmarked the score and rank metric on random trace data with a range of traces from 1,000 to 30,000 and a range of samples per trace from 1,000 to 10,000. Figure 21 and Figure 22 show the results of the benchmark. As with all of the metrics so far, there is a linear relationship between the size of the dataset and the execution time. An important thing to keep in mind regarding the results shown in Figure 21 and Figure 22 is that it is almost entirely dependent on the scoring function being used. In this case, we used correlation to score all 256 key candidates. Using a different scoring function may result in a massive difference in performance. This is because the metric itself simply iterates through all the key candidates and sorts them by the score returned by the scoring function callback. Therefore, the

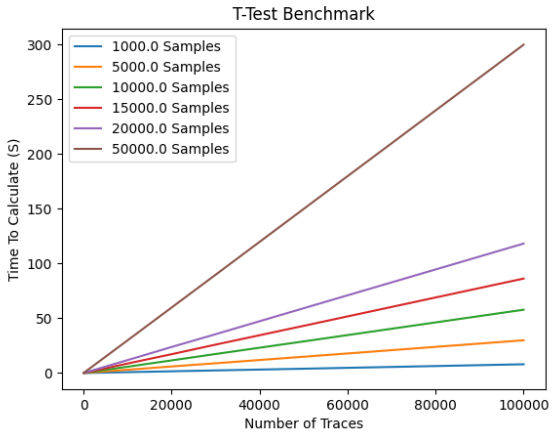

Figure 17: T-test Benchmark

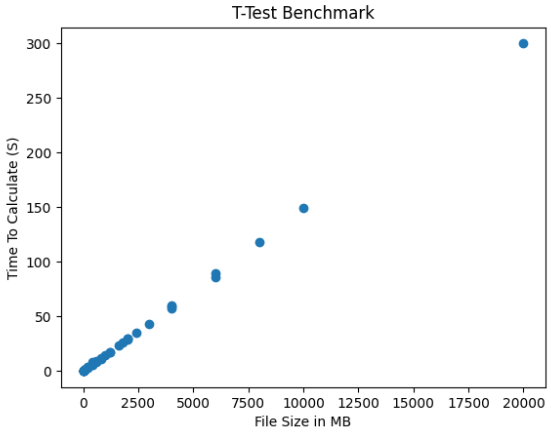

Figure 18: T-test Ratio Benchmark as a Function of Dataset Size

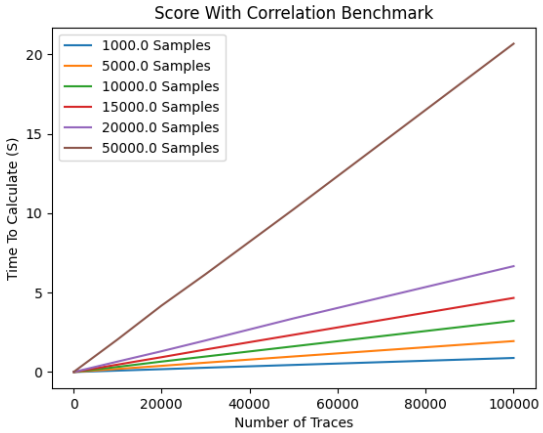

Figure 19: Correlation Benchmark

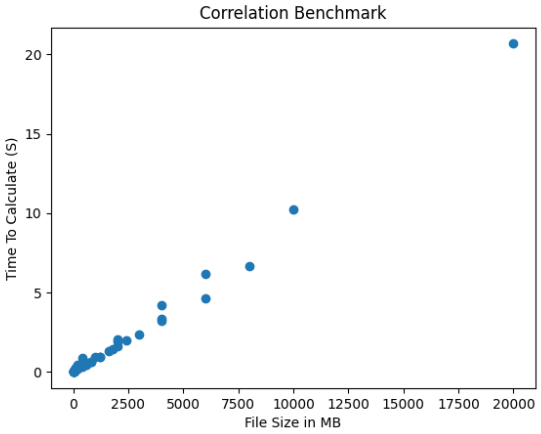

Figure 20: Correlation Benchmark as a function of Dataset Size

performance of this metric will be left up to the developer of the callback function used to score the key candidates.

3.2.6.5 Success Rate and Guessing Entropy Benchmark

Finally, we benchmarked the success rate and guessing entropy function. Success rate and guessing entropy were benchmarked differently from all of the previous metrics. This is because the metric does not need trace data directly. Instead, it is implemented in terms of experiments as defined in [6]. Therefore, the performance is independent of the number of traces. This is because the metric itself requires the score and ranks of each experiment to already have been computed. Figure 23 shows the result of the benchmark.

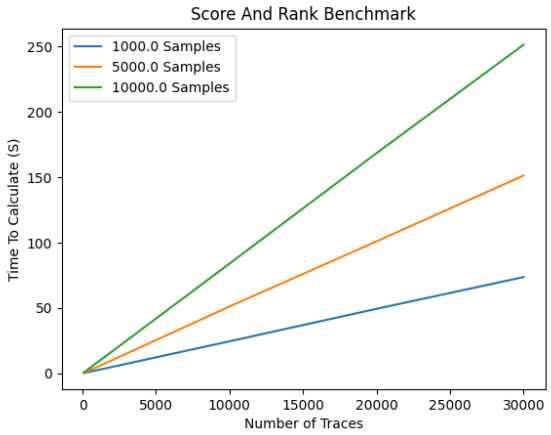

Figure 21: Score and Rank Benchmark

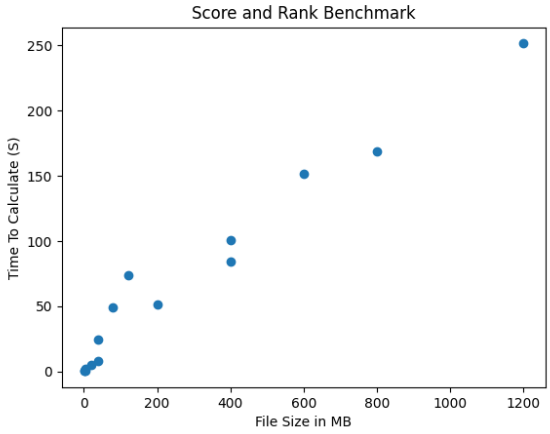

Figure 22: Score and Rank Benchmark as a Function of Dataset Size

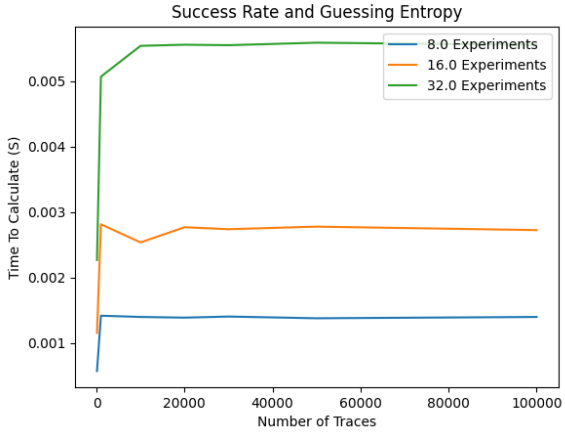

Figure 23: Benchmarking for Success Rate and Guessing Entropy

We did not generate a graph for the execution time as a function of file size as Figure 23 shows that such a benchmark would be redundant. We can see that the run time of the metric is relatively unaffected by the number of traces used to score and rank the keys. The actual difference in execution time comes from the total number of experiments.

3.3 ChipWhisperer Oscilloscope Interface Implementation

Researchers at WPI and across the country use ChipWhisperer devices to conduct side-channel analysis research. ChipWhisperer has an existing library developed to interface with their devices [22]. However, this library was improved upon by providing higher-level API calls that simplify the processes of configuring

Listing 8: Create CW Scope Object

```
firmware_path = "C:\\firmware\\simpleserial-aes-CWLITEARM.hex"
tar_type = cw.targets.SimpleSerial
programmer = cw.programmers.STM32FProgrammer
gain = 25
num_samples = 500
offset = 0
# create CWScope object
scope = CWScope(firmware_path, gain, num_samples, offset, tar_type, programmer)
```

a ChipWhisperer device and using it to collect traces. This essentially creates a layer of abstraction for the user. This library was developed using a ChipWhisperer-Lite with an ARM target board. The module is implemented as a class named `CWScope`. The constructor for the class requires a path to the firmware (usually a bit or hex file) to configure the target board. Furthermore, the programmer must specify the target type. By default, this is set to `cw.targets.SimpleSerial`. The firmware programmer must also be specified with the constructor using `STM32Programmer` by default. All other parameters are optional and are used to configure certain scope parameters. The scope parameters include gain, the number of samples per trace, and the offset. Additional scope parameters that are not included in the constructor can be modified by simply referencing `CWScope.scope` directly since a reference to the ChipWhisperer scope is kept as an attribute of the class. Listing 8 shows an example of how a `CWScope` object can be initialized. In this example, we are going to program the target board with the `simpleserial-aes` firmware which can be complied from ChipWhisperer's GitHub repository [22]. This example also uses a ChipWhisperer-Lite with an ARM target board. Therefore, the example requires that the `STM32FProgrammer` is used to configure the target board. The particular programmer to use depends on the type of target board. This information is available in the ChipWhisperer documentation [22]. Once the `CWScope` object is configured correctly, the user can use the pre-defined capture procedures. The source code for the implementation can be found in the CWScope.py file on GitHub.

3.3.1 Standard Capture Procedure

We developed a comprehensive function, defined as `standard_capture_procedure`, that implements fast capture of power traces while providing a lot of flexibility for the programmer. This capture procedure requires an initialized `CWScope` object as detailed above. The user must supply the function with the number of traces that they want to collect, the rest of the parameters are optional and will be set to default values.

Listing 9: Standard Capture Procedure

```
# initialize CWScope object
scope = CWSCope(...)
num_traces = 1000
# capture procedure using default cw.ktp
res = scope.standard_capture_procedure(num_traces, fixed_key=True, fixed_pt=False)
# capture procedure using user-defined keys and plaintexts
keys = [...] # some NumPy array of keys
texts = [...] # some NumPy array of texts
res_2 = scope.standard_capture_procedure(num_traces, keys, texts)
```

The four optional parameters are `experiment_keys`, `experiment_texts`, `fixed_key`, and `fixed_pt`. The `experiment_keys` and `experiment_texts` allow the user to specify a list of keys and plaintexts to be used by each encryption. If they are not specified, the program will use the default ChipWhisperer key-text pair generation algorithm (`cw.ktp`) as specified in the ChipWhisperer library [22]. This key-text pair generation creates 128-bit encryption keys and plaintexts that can be used with various implementations of the AES-128 encryption algorithm. However, oftentimes researchers are not working with AES-128 are may have encryption keys and plaintexts that are not necessarily 128-bits. Therefore, by specifying a Python list (not NumPy array) for both `experiment_keys` and `experiment_texts`, the researcher can supply the capture procedure with custom plaintexts and keys for the encryption. The `fixed_key` and `fixed_pt` parameters are only used if a list of keys and plaintexts are not supplied. These parameters are Boolean variables and dictate the behavior of the default ChipWhisperer key-text pair generation algorithm. If `fixed_key` is set to true, then the encryption key will remain the same for all traces collected. Similarly, if `fixed_pt` is set to true, then the plaintext remains the same for all encryptions. Listing 9 shows an example of using the standard capture procedure using both default and custom keys and plaintexts. The results of the capture procedure in Listing 9 are stored in the variables `res`, and `res2`. The standard capture procedure returns a tuple containing the power traces, the encryption keys, the plaintexts, and the ciphertexts. Therefore, in this example, `res[0]` would be a NumPy array containing all of the power traces, `res[1]` would be all of the keys, `res[2]` the plaintexts, and `res[3]` the ciphertexts. The performance of the standard capture procedure varies depending on the ChipWhisperer device, the encryption algorithm being used, and the specification of the computer running the Python script. Figure 24 shows the performance of a ChipWhisperer-Lite using the `SimpleSerial2` version of `simple-aes` found in [22]. The computer running the experiment has an eight-core Intel processor and is running Windows 11. We can see that the execution time of the capture procedure increases linearly as the number of traces increases. The slope of this line indicates the rate at

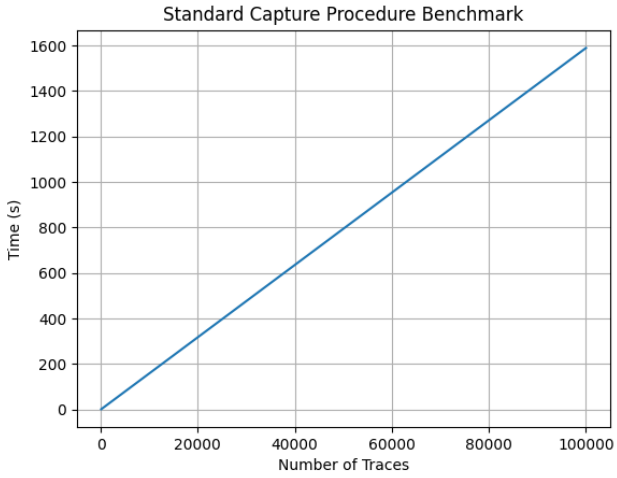

Figure 24: Standard Capture Procedure Benchmark

which the traces were collected. In this experiment, about 67 traces per second were collected. Again, it is important to keep in mind that the speed will vary depending on the encryption algorithm used and the ChipWhisperer device itself. The important takeaway from the benchmark conducted above is that the relationship between the number of traces and the execution time is linear.

3.3.2 T-Test TVLA Capture Procedure

The t-test TVLA capture procedure acts as an easy way for users to collect the fixed and random trace sets for the t-test metric. Both sets of traces are collected in one loop and returned together to decrease the amount of programming needed to collect these traces. Similar to the standard capture procedure, the only parameter that is strictly required is the number of traces that the user wants to collect. If nothing else is specified, the two datasets are collected using the `cwtvla.ktp.FixedVRandomText` key-text pair algorithm. This uses 128-bit keys with `ktp.next_group_A` returning the key-text pair for the fixed group and `ktp.next_group_B` returning the pair for the random group. Furthermore, the user can specify a Python list of keys and texts for the fixed and random groups if they do not want to use the default key-text pair generation algorithm. The `group_a_keys` and `group_a_texts` parameters correspond to the keys and plaintexts to be used to generate the fixed trace set. The `group_b_keys` and `group_b_texts` parameters correspond to the keys and plaintexts for the random trace set. Listing 10 shows an example of how this capture procedure can be used. The keys, plaintexts, and ciphertexts for the capture are not returned along with the traces because they are typically not used in t-test experiments. In terms of performance, we can expect results that are 50% slower than the benchmark in Figure 24 since we are performing two encryptions per loop iteration rather than one.

Listing 10: T-Test TVLA Capture Procedure

```
scope = CWSCope(...)
num_traces = 1000
# run capture procedure using default key-text generation
fixed, random = scope.capture_traces_tvla(num_traces)
# run capture procedure using custom keys and texts
keys = [...] # some list of keys
fixed_texts = [] # some list of fixed plaintexts
rand_texts = [...] # some list of random plaintexts
fix1, rand1 = scope.capture_traces_tvla(num_traces, key, fixed_texts, keys, rand_texts)
```

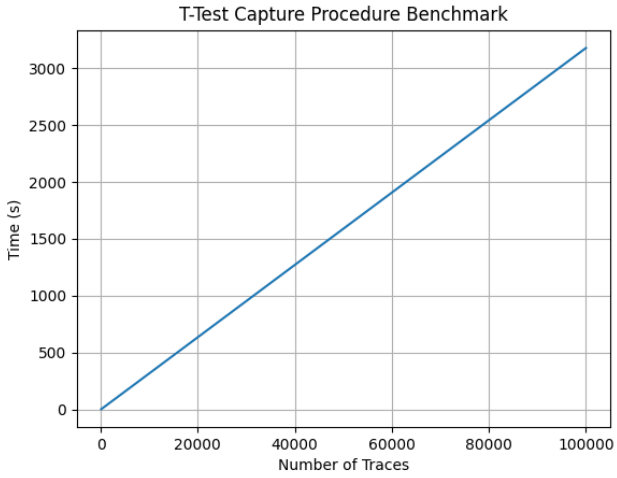

Figure 25: T-Test TVLA Capture Procedure Benchmark

The result of the same benchmark performed in Section 3.3.1 can be seen in Figure 25. As expected, the relationship between execution time and the number of traces is linear. However, the speed of the procedure is cut in half to about 31 traces per second due to the additional encryptions.

3.3.3 ChipWhisperer to File Framework Adapter

To facilitate easy capture and storage of power traces, we created a function that allows for traces collected on ChipWhisperer devices to be directly stored in our custom file framework. The function, defined as `cw_to_file_framework`, uses the standard capture procedure in our library to collect traces and their associated keys, plaintexts, and ciphertexts which can be added as a dataset to a new or existing experiment. Listing 11 shows an example of using this adapter. In this example, the traces, keys, plaintexts, and ciphertexts are collected from the standard capture procedure and are all saved to an experiment named

Listing 11: CW to File Framework Adapter

```
scope = CWScope(...)
file = FileParent(...)
num_traces = 1000
keys = [...] # some list of keys
texts = [...] # some list of texts
scope.cw_to_file_framework(num_traces, file, "TestExperiment", keys, texts)
```

`"TestExperiment"`. If this experiment does not exist in the `FileParent` object specified by the `file` variable, a new experiment with that name will be created where these datasets will be saved.

3.4 Existing Code Integration

In addition to the code we developed in the aforementioned sections, Vernam lab also had existing code that they used to conduct side-channel research. However, this code had no centralized location and only existed in locally stored Python notebooks. Furthermore, this code was poorly documented and not optimized for performance. Therefore, we created two additional modules to our library to integrate this existing code.

3.4.1 Differential Power Analysis (DPA)

Differential Power Analysis is a type of side-channel attack that analyzes the power consumption of a cryptographic device. The basic premise of a DPA attack is that a variation in the power consumption of a system can be used to determine information about the processing that is occurring within a system. DPA attacks can be used to not only extract secret keys but also weights for ML algorithms or other pieces of information a developer would wish to keep secret. Vernam lab had existing DPA code, however, it was not well integrated with any existing libraries and had performance issues. The given code was first integrated into a function called `calculate_dpa` which combined both the second and first-order calculations into one function. This function requires power traces and intermediate values. This function also has a wealth of customizable parameters. The order parameter dictates the order of DPA which you would like to calculate. The `key_guess` parameter represents the current key guess which is used in a DPA attack. The `Window_size_fma` parameter represents the window size of the moving average calculation. Typically when performing a second-order calculation there is an intermediate step where the trace array is expanded. This means that a trace array of dimensions `(T, S)` will be expanded to size `(T, S^2/2)`. To reduce memory usage, the previously mentioned function divides the traces into windows before performing this expansion, however, this comes at the cost

Listing 12: Lecroy Capture Sequence

```
#Configure Scope
scope = scope_setup()
#Configure DUT
dut = dut_setup(board="CW305")
#Capture
trace = capture_cw305(scope, dut)
```

of accuracy. To fix this issue another function, named `calculate_second_order_dpa_mem_efficient`, was created that instead performs windowing across the entire `(T, S^2/2)` space. This allows for a ram efficient option that still maintains accuracy. The source code for each implementation can be found in DPA.py on GitHub.

3.4.2 Lecroy Scope Interface

Vernam Lab had an existing Python notebook used for interfacing with the Lecroy scope in the lab. Researchers at Vernam lab would greatly benefit from making this code accessible in our library and documenting its usage. Before being implemented into the SCAPEgoat library, the code to use the Lecroy scope was scattered across multiple Jupiter notebooks and provided no consistent way to set up and use the Lecroy scope. A user would have to modify magic numbers or other hard-coded values to adjust the scope setup to fit their needs. This would require knowledge about the serial communication protocol that the scopes use. This placed a burden on less experienced researchers who were not entirely familiar with the code itself. In the process of integrating the code with our library, we parameterized many different values that were previously hard-coded in the Jupiter notebooks and removed redundant or unused code. Now the library includes a Lecroy Scope class that groups a variety of miscellaneous functions like `set_trigger`, `get_trigger`, `reset`, and `setup`. Additionally, functions were added to simplify command-line calls meaning that to add functionality to the library one doesn't need to know the specific commands to use for common actions. Listing 12 shows a common setup using the Lecroy scope interface. The integrated source code can be found in the LecroyScope.py file. Upon testing the library implementation, it was found that it behaves identically to the un-integrated code given to us by the researchers meaning that no functionality was lost in the integration process.

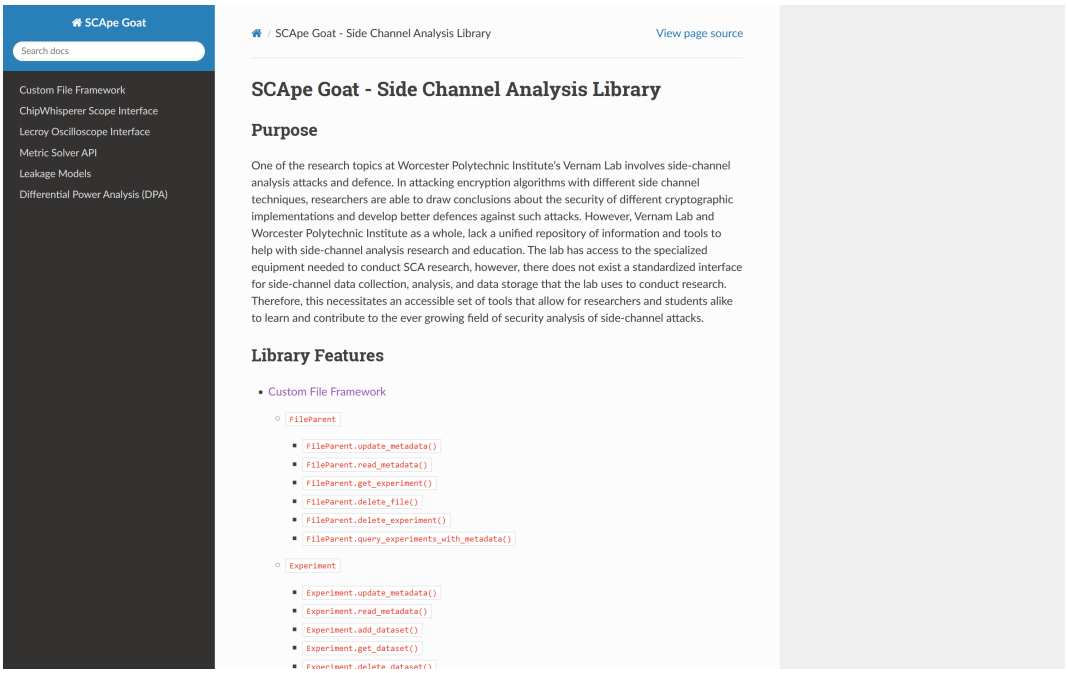

Figure 26: Documentation Website Homepage

3.5 Read The Docs Online Documentation

To document the code for researchers at WPI and around the world, we created a documentation website[3] that provides information about how to use the library we developed. The documentation uses the Sphinx documentation generator with the Read the Docs theme. As such the documentation itself is written in the reStructuredText markup language and compiled into HTML using GitHub actions once a developer pushes to the main branch. This was configured using a template provided by Read the Docs [30]. Figure 26 shows the home page of the documentation website. We can see that the Read the Docs theme takes care of the HTML layout. Each module documentation is accessible via the sidebar. All of the documentation itself is stored in the `doc` directory in the repository. Each class and method in the library is documented. Each method has a description of what it does, a description of the parameters, and a description of what it returns. Furthermore, the types of the parameters and return type are also specified. The documentation available on this website is also available in the source code itself as a docstring. Providing this documentation will not only make it easier for researchers to get accustomed to the library but also allow students who want to learn more about side-channel analysis to be able to use this library for personal use.

[3] https://vernamlab.org/SCApeGoat/

4 Conclusion

4.1 Project Outcome

We developed a very useful set of open-source tools for researchers of side-channel analysis to utilize in laboratories at WPI and around the world. Our custom file framework provides an extremely flexible and easy-to-use interface for organizing and saving side-channel analysis experiment data. The development of this framework is a much-needed addition to laboratories at WPI conducting this research. Our metric solver centralizes common side-channel analysis metrics by providing an efficient Python implementation. The ChipWhisperer oscilloscope interface provides an easy and fast way to collect traces for researchers utilizing ChipWhisperer devices to conduct side-channel analysis research. Furthermore, we integrated and improved upon existing code used at WPI's Vernam lab. The Lecroy scope interface was documented and integrated into our GitHub repository. The differential power analysis code was integrated into our library and also received performance improvements compared to the original implementation that the lab had been using. What makes our library so great is that each module can be together or independently of each other. If a group of researchers wants to only use the metric solver, they do not need to have their data in our custom file framework. Similarly, data collected in the oscilloscope interface can be saved in any file type that the user wants. While the library is best when used together, our ultimate goal was flexibility for the needs of side-channel analysis researchers. We anticipate that SCAPEgoat will be a very useful tool for side-channel analysis researchers for years to come.

4.2 Future Work

As an open-source project, the idea is that researchers and future MQP teams can improve upon the library developed for this project. Some aspects of the library that can be improved with future work include the following.

- **Information-Theoretic Metrics**: The authors in [6] discuss the theory behind information-theoretic metrics for side-channel analysis. Implementing these metrics would be very useful for side-channel analysis researchers. Once implemented, the library will contain all of the metrics discussed by the authors in [6].
- **Advanced File Framework Compression:** The custom file framework currently uses NumPy binaries to store and compress experiment data. Future project groups can research more advanced file

compression techniques and integrate them with the existing file framework structure. The framework was developed in a way such that changing the way data is saved would be relatively easy.

- **PicoScope Interface:** Another oscilloscope commonly used by laboratories is the PicoScope 3000. Creating an interface for this scope and integrating it with this library would allow for more flexibility in trace collection. The PicoScope API is in the C programming language. Therefore, to implement this in Python, the project group would most likely need to write C code that can be called from a Python development environment.
- **Formal User-Study of Researchers:** Future project groups can conduct a moderated or unmoderated user-study of researchers at other universities. They can formally collect feedback on the usability of the library and see what additional features are wanted by researchers at different institutions.

These are just a few items that can future project groups can work on. Predictably, bugs and other issues with the current implementation may arise. Future project groups can address these issues and make the code more refined than it is currently. Continued development of this open-source library will be significant to the field of side-channel analysis research at WPI and beyond.

References

[1] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full aes," *Advances in Cryptology – ASIACRYPT 2011*, pp. 344–371, 2011.

[2] D. Gstir and M. Schläffer, "Fast software encryption attacks on aes," *Progress in Cryptology*, p. 359–374, 2013.

[3] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer, 2010.

[4] European Commission, "The framework programme for research and innovation." https://ec.europa.eu/programmes/horizon2020/en, 2020.

[5] National Institute of Standards and Technology, "Threshold cryptography project." https://csrc.nist.gov/projects/threshold-cryptograph, 2020.

[6] 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, p. 1–38, 2023.

[7] W. J. B. Owen Lo and D. Carson, "Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa)," *Journal of Cyber Security Technology*, vol. 1, no. 2, pp. 88–107, 2017.

[8] J. v. Woudenberg and C. O'Flynn, *The hardware hacking handbook: Breaking embedded security with hardware attacks*. No Starch Press, Inc., 2022.

[9] A. Heuser, M. Kasper, W. Schindler, and M. Stöttinger, "A new difference method for side-channel analysis with high-dimensional leakage models," *Lecture Notes in Computer Science*, p. 365–382, 2012.

[10] C. Jin and Y. Zhou, "Enhancing non-profiled side-channel attacks by time-frequency analysis," *Cybersecurity*, vol. 6, Aug 2023.

[11] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Advances in Cryptology CRYPTO' 99*, p. 388–397, 1999.

[12] C. Whitnall and E. Oswald, "A critical analysis of iso 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules')," *Cryptology ePrint Archive*, 2019.

[13] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side-channel resistance validation," 2011.

[14] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[15] T. Schneider and A. Moradi, "Leakage assessment methodology," *Journal of Cryptographic Engineering*, vol. 6, p. 85–99, Feb 2016.

[16] S. Mangard, "Hardware countermeasures against dpa – a statistical analysis of their effectiveness," *Topics in Cryptology – CT-RSA 2004*, p. 222–235, 2004.

[17] A. Thillard, E. Prouff, and T. Roche, "Success through confidence: Evaluating the effectiveness of a side-channel attack," *Cryptographic Hardware and Embedded Systems - CHES 2013*, p. 21–36, 2013.

[18] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," *Advances in Cryptology - EUROCRYPT 2009*, p. 443–461, 2009.

[19] J. L. Massey, "Guessing and entropy," *Proceedings of 1994 IEEE International Symposium on Information Theory*, pp. 204–, 1994.

[20] J. Beguinot, W. Cheng, S. Guilley, and O. Rioul, "Be my guess: Guessing entropy vs. success rate for evaluating side-channel attacks of secure chips," *2022 25th Euromicro Conference on Digital System Design (DSD)*, Aug 2022.

[21] D. Ros, O. Glamočanin, and M. Stojilović, "MetriSCA: A library of metrics for side-channel analysis." https://doi.org/10.5281/zenodo.5778947, Dec. 2021.

[22] NewAETech, "Chipwhisperer 5.7.0." https://github.com/newaetech/chipwhisperer, Jan. 2023.

[23] G. Brandl, "Sphinx documentation," *URL http://sphinx-doc. org/sphinx. pdf*, 2021.

[24] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, pp. 263–273, International World Wide Web Conferences Steering Committee, 2016.

[25] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, and et al., "Array programming with numpy," *Nature*, vol. 585, p. 357–362, Sep 2020.

[26] E. Prouff, R. Strullu, R. Benadjila, E. Cagli, and C. Canovas, "Study of deep learning techniques for side-channel analysis and introduction to ascad database," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 53, 2018.

[27] D. M. Mehta, M. Hashemi, D. S. Koblah, D. Forte, and F. Ganji, "Bake it till you make it: Heat-induced leakage from masked neural networks," *Cryptology ePrint Archive, Paper 2023/076*, 2023. https://eprint.iacr.org/2023/076.

[28] The HDF Group, "Hierarchical data format, version 5." https://github.com/HDFGroup/hdf5.

[29] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[30] M. Kaufmann, "Read the docs tutorial template." https://github.com/readthedocs/tutorial-template, Sept. 2023.

Authorship Table

Section	Author(s)	Editor(s)
Abstract	Samuel Karkache	Samuel Karkache
1.1 Project Motivation	Trey Marcantonio	Samuel Karkache
1.2 Side Channel Analysis Background	Samuel Karkache, Trey Marcantonio	Samuel Karkache
2.1 Custom File Framework	Samuel Karkache Trey Marcantonio	Samuel Karkache
2.2 Side-Channel Analysis Metric Solver	Samuel Karkache	Samuel Karkache
2.3 Oscilloscope Interface	Samuel Karkache	Samuel Karkache
2.4 Online Documentation	Samuel Karkache	Samuel Karkache
3.1 File Framework Implementation	Samuel Karkache	Samuel Karkache
3.2 Metric Solver Results	Samuel Karkache, Trey Marcantonio	Samuel Karkache
3.3 ChipWhisperer Oscilloscope Interface Implementation	Samuel Karkache	Samuel Karkache
3.4 Existing Code Integration	Trey Marcantonio	Trey Marcantonio
4.1 Project Outcome	Samuel Karkache	Samuel Karkache
4.2 Future Work	Samuel Karkache	Samuel Karkache

Table 1: Authorship Table