MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Implementing and Evaluating Side-Channel Attacks against JavaCard cryptographic implementations

Master's Thesis

RADOMÍR MANN

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Implementing and Evaluating Side-Channel Attacks against JavaCard cryptographic implementations

Master's Thesis

RADOMÍR MANN

Advisor: Łukasz Michał Chmielewski, PhD

Department of Computer Systems and Communications

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source. For the correction of English text, I used the Grammarly tool. I have used ChatGPT while programming the underlying implementation to find resources and documentation summaries.

Radomír Mann

Advisor: Łukasz Michał Chmielewski, PhD

Acknowledgements

I want to thank my supervisor *Łukasz Michał Chmielewski PhD.* for time and guidance during my work on this thesis, my consultant *RNDr. Milan Šorf* and my family for the extensive support during my studies.

Abstract

This thesis focuses on side-channel attacks via power analysis on cryptographic algorithms running on JavaCard. More specifically, I evaluated the software-based implementation of ASCON authenticated encryption (AEAD) using the correlation power analysis (CPA) for attack implementation and Test Vector Leakage Assessment (TVLA) for verification of key leakage. At first, I improved the readability, efficiency, and usability of the existing CPA attack scripts intended for the AES-128 encryption algorithm. Then, I implemented the CPA attack on the ASCON-128 authenticated encryption algorithm (AEAD) while reusing generic parts of the AES attack. I compared the complexity of execution and implementation for both attacks. Finally, I described possible protections against side-channel attacks through power analysis. This thesis was done in cooperation with the CRoCS laboratory.

Keywords ASCON, CRoCS, cryptographic implementations, cryptography, security, side-channel analysis

Contents

List of Tables

List of Figures

Introduction

When designing a new cryptographic protocol, many aspects have to be considered, such as semantic security, computational complexity, algorithm properties like confusion and diffusion, or durability against side-channel attacks. This thesis will primarily focus on durability against side-channel attacks by power analysis of an authenticated encryption (AEAD) from a relatively new cryptographic family called ASCON, winner of the NIST Lightweight Cryptography competition (2019–2023) [1] and CAESAR competition in the lightweight applications category [2]. As the names of both competitions suggest, ASCON cryptographic primitives are intended to be used in real-time applications with large volumes of data or devices with limited performance or power [3], which are usually small embedded devices where power consumption can often be easily measured and exploited. In our case, this device was JavaCard, a programmable chip card that internally uses a reduced version of JVM (Java Virtual Machine) called JCVM (Java Card Virtual Machine). JCVM does not have a garbage collector, multidimensional arrays, floating point arithmetic or other demanding features used in modern languages like regular Java [4].

At first, I improved and optimised the existing CPA attack on AES-128 with related scripts provided by the CRoCS laboratory initially created by my supervisor and his colleague Leo Weissbart. Improvements include adding multi-threading support, attack automation, better logging, enhanced attack configurability and a more refined structure of scripts and overall project, among others. Then, I implemented the CPA attack on ASCON-128 authenticated encryption while reusing the generic functionality from the AES attack implementation by extracting it into generic single-purpose objects and common scripts, thus creating a generic framework for any CPA attack. Furthermore, I created new utilities and scripts to make working with power traces more convenient. Finally, I implemented TVLA (Test Vector Leakage Assessment) to verify leakage and executed both attacks on power measurements captured from the JavaCard running AES-128 and ASCON-128 encryption algorithms.

AES attack is used as a baseline and compared with the ASCON attack in terms of robustness against power analysis, as well as com-

plexity of the attack implementation and execution. Neither implementation of the cryptographic algorithm in JavaCard is hardware-based and runs only as software on the general-purpose processor. As a result, this makes a side-channel attack easier because hardware implementations tend to have optimisations and protections in place. For example, Dual-Rail Random Switching, which uses two circuits with complementary logic that cancel out variations in power consumption created by the processed data [5].

The final step of this thesis is to describe potential software protections against side-channel attacks via power analysis by hiding and masking.

1 Theoretical background

1.1 Side-channel attacks

Side-channel attacks can reveal secrets or change device behaviour without flaws in the logic of the underlying algorithm. However, algorithm design can impact its robustness against this class of attacks. Software protections without hardware support often only make a side-channel attack much more difficult but still feasible [6, 7]. These attacks are less general and usually focus more on a specific algorithm, but can be very powerful if not considered [8].
They are primarily organised into two distinct classifications [8]:

1. **Logical tampering**
   (a) **Active** attacks can inject faults or change the branching of the program in a way that can be used to exploit the device.
   (b) **Passive** attacks do not change the device's actions but may be used to reveal information that should be otherwise inaccessible.

2. **Physical tampering**
   (a) **Invasive** attacks need to disassemble or somehow physically tamper with the device to access the internal circuits to be successful.
   (b) **Non-invasive** attacks only exploit information that can be gathered externally without causing irreversible damage to the device. Such as power consumption, timing, and electromagnetic emissions.

Non-invasive passive attacks are most relevant for this thesis. Here are some examples [7]:

- **Timing attack** - Executing the identical operation on a chip may take a different amount of time depending on the data processed, making these timing irregularities exploitable to attain additional knowledge about the secret.

- **Electromagnetic emissions attacks** - These attacks use electromagnetic radiation emitted by the chip, which may be linked with data and instructions executed by the processor. Therefore, it may reveal information about the internal state and instructions executed.

- **Power analysis** - Various computations and their corresponding data need a different amount of power to be performed. As a result, power consumption depends on the operations executed by the chip and the data processed, including secrets. Therefore, power consumption may reveal some properties of the underlying secret that are otherwise inaccessible.

Side-channel attacks that use power analysis are the primary focus of this thesis. Extracting information from power traces can be done using various techniques, such as correlation power analysis [7].

1.1.1 Correlation power analysis (CPA)

Correlation power analysis is a technique that can extract secrets from a device by analysing captured power consumption when the device performs sensitive operations like encryption and must use secrets stored inside the device. Different data passing through the bus inside the chip influences how much energy is needed for the transfer due to the nature of the representation of digital data in hardware (charged/discharged). As a result, there will be a correlation between data processed by the device and power consumption. We exploit this fact during the CPA attack [8].

To perform the attack, we need to predict how each possible combination of key bits influences the power consumption. As a result, we need the power model function defined as follows:

Definition 1. Generic power model function

$$y = f(x) \tag{1.1}$$

where:

- f is the power model function,
- x is processed data and
- y is theoretical power consumption.

The power model function used in a concrete attack depends on the target and cryptographic algorithm. Usually, it is Hamming Distance or Hamming Weight of an internal state [7].

Definition 2. Hamming Weight (HW) represents a count of bits set to the logical one in a binary number [7].

Definition 3. Hamming Distance (HD) represents the count of different symbols between two strings [9]. The easiest way to calculate Hamming Distance in binary representation is by using Hamming Weight.

$$HD(x, y) = HW(x \oplus y) \tag{1.2}$$

Hamming Distance can be used as a power model function because changing the bit charge inside registers consumes more power than if it remained the same. Therefore, power consumption is influenced by the number of changed bits in the corresponding register [8]. Another method uses Hamming Weight, which depends on the data transferred via a bus. Transferring charged bits (1) is more power-demanding than transferring discharged bits (0) and, as a result, can be used as a power function [8].

After choosing the power model and acquiring the power traces with a fixed key, we calculate the correlation between the theoretical and actual power consumption. Correlation is suitable because when theoretical consumption rises on a correct key guess, real consumption should increase as well and the other way around. As a result, we can distinguish between correct and incorrect key guesses with some degree of confidence [7]. Correlation cancels static noise in power consumption, which is the power necessary to keep the processor running and is independent of tasks performed and data processed [3]. With enough data, dynamic noise created by the processor's activity becomes negligible, and the correlation when using the correct key guess will be higher compared to other key guesses. Correlation is mathematically defined by this equation.

Definition 4. Pearson's Correlation Coefficient [9]

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathrm{E}[(X-\mu_X)(Y-\mu_Y)]}{\sqrt{\mathrm{E}[(X-\mu_X)^2]\,\mathrm{E}[(Y-\mu_Y)^2]}} \tag{1.3}$$

where:

- E is the expected value,
- μ the mean of the random variable and
- σ is the standard deviation.

1.2 JavaCard

JavaCard is a programmable smart card with cross-platform interoperability. JavaCard supports JCVM (Java Card Virtual Machine), a stripped version of the JVM (Java Virtual Machine). JCVM does not include all the features of the regular JVM. Most notably, it does not support multi-threading, iterators, garbage collector, multi-dimensional arrays, or floating point arithmetic [4]. Due to the architecture constraints, JavaCard only supports two primitive types, byte and short. However, newer versions of JavaCard also support the integer type. JavaCard applications are first compiled by the standard Java compiler and then into bytecode called an applet, which is executed inside the card by the JCVM [4]. JavaCard can have multiple logically isolated applets installed, but only one can run at a given time. Packages with applets are installed on a card using the Global Platform specification, which is also responsible for the security and life-cycle of a card, from initialisation to termination [10].

JavaCard uses two types of memory architecture:

- **Transient** - stored inside the card's RAM, deleted when power is lost or an applet is deselected.
- **Persistent** - stored inside the card's EEPROM, stays unchanged until manually overwritten.

Communication with JavaCard is done through APDU packets (Application Protocol Data Unit), which are defined in the standard ISO7816-4 [11]. Figure 1.1 shows structure of APDU packet.

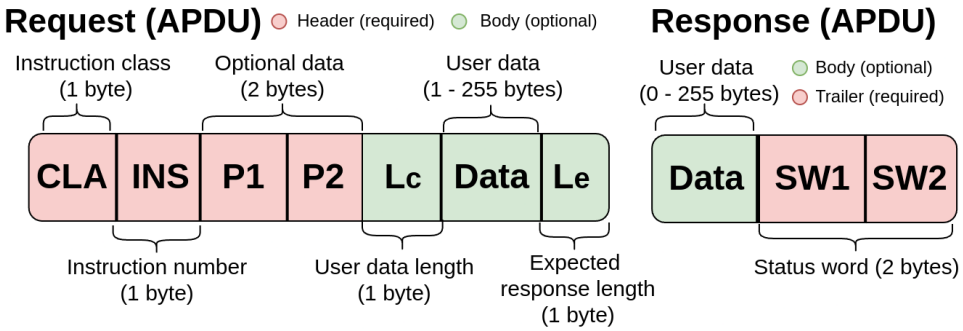


Figure 1.1: APDU packet format

1.3 AES

AES is a specification used for symmetric encryption standardised by NIST (National Institute of Standards and Technology) in 2001 and is still widely adopted [12]. The main goal of encryption is to prevent the creation of ciphertext or the acquisition of plaintext without knowledge of an encryption key, which is identical for encryption and decryption in symmetric cryptography. An encryption key in AES can be 128, 192 or 256 bits long. Longer keys increase entropy and consequently security.

AES is a block cipher, which means it encrypts plaintext in blocks of size 128 bits for any key length. As a result, padding is required to cover cases where the data length is not divisible by 128 [12]. AES consists of rounds, where the number of rounds depends on the length of the key. This can be seen in Table 1.1.

Table 1.1: AES rounds and block sizes for different key lengths [12]

Key Length	Number of rounds	Size of block
128	10	128
192	12	128
256	14	128

The AES round consists of several phases with different purposes. Before every round, a different key from the key scheduler (expander) is added (XOR) to the current state of the cipher. This key is derived

from the original key provided by the user. This key is equivalent to the original key in AES-128, during the first round [12]. Because we perform the attack within the first round, we can completely abstract the key scheduler part of AES and not consider it in the attack. The next step is the substitution layer called $Sbox$. This operation removes the linear relationship between the input and output of the $Sbox$. This step is essential because it makes some attacks that rely on algebraic analysis of ciphertext impossible [12]. Otherwise, attackers could take advantage of the linearity of all other operations in AES. Nevertheless, in our case, it allows the attack to identify the key more precisely without acquiring more potential key candidates that would have some linear relationship with the input data [3].

We do not use a hardware implementation of AES on JavaCard. Therefore, data must be stored in the registers or transferred over the bus after each phase. We can pinpoint the attack here and ignore the rest of the AES encryption process, such as mixing columns and shifting rows, because they do not provide any additional advantage for the attack and, on the contrary, make it more complicated.

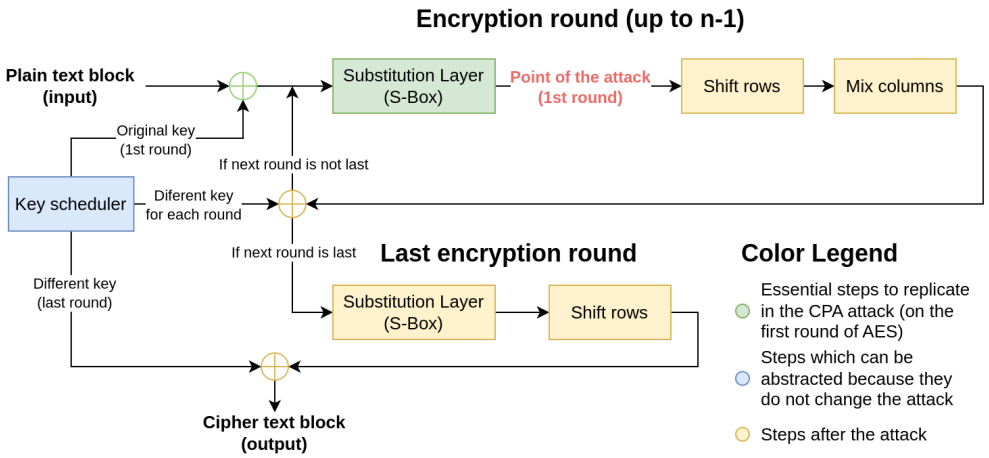


Figure 1.2: AES schema with point of the attack

1.4 ASCON

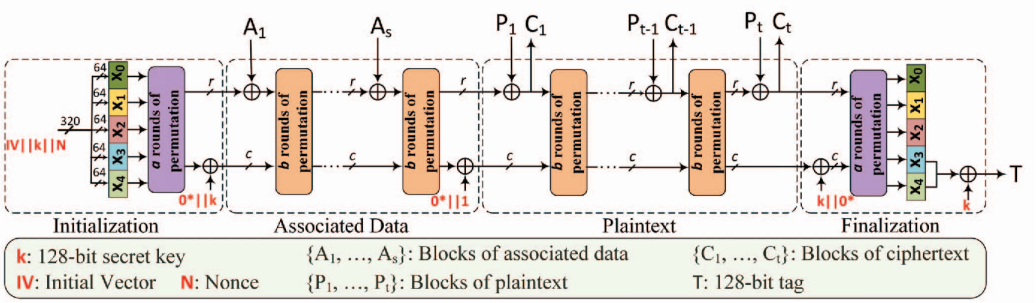


Fig. 2. Sponge-based structure of authenticated encryption in the Ascon cipher.

Figure 1.3: Structure of authenticated encryption of ASCON-128 [13]

ASCON is a family of authenticated cryptographic primitives created for high-throughput applications or devices with limited power and performance. Usually embedded systems, or, as in our case, chip cards. It offers confidentiality and authenticity of the data [14]. NIST chose ASCON as the new standard for lightweight cryptography in February of 2023 [15]. ASCON family, among other things, supports hashing and authenticated symmetric encryption with associated data (AEAD) [14].

In this thesis, I concentrate only on one version of authenticated encryption called ASCON-128. Other variants differ in the number of rounds, rate, and key size. However, these parameters do not significantly influence the attack and can be reused with minor modifications to the key extraction logic. Table 1.2 shows the differences between various AEAD variants.

Table 1.2: Parameters of different AEAD ASCON ciphers.

	ASCON-128	ASCON-128a	ASCON-80pq
Entropy	128 bits	128 bits	128 bits
Key size	128 bits	128 bits	160 bits
Rate	64 bits	128 bits	64 bits
Capacity	256 bits	192 bits	256 bits
Rounds(a,b)	(12,6)	(12,8)	(12,6)

ASCON-128 is based on sponge construction, which is divided into several phases. Each phase works with the 320-bit long internal state S, which is separated into five 64-bit segments $(x_0, x_1, x_2, x_3, x_4)$ used variously inside the permutation function p used during the encryption process [14].

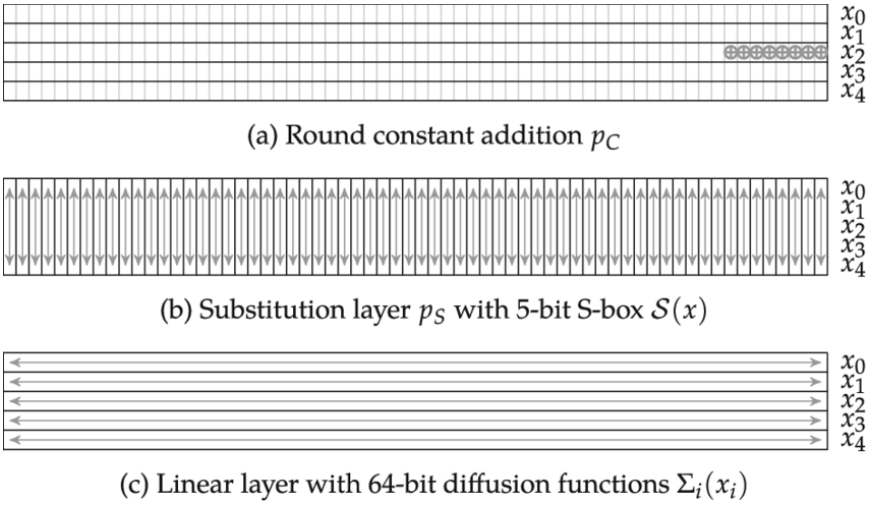
At the start of the initialisation phase, state S is filled with values in the following order [14]:

1. **Initialisation Vector** (x_0) - 64-bit constant defined in the standard of ASCON equal to *0x80400C0600000000*.

2. **Key** (x_1, x_2) - 128-bit secret value defined by the user, which recovery is the attack's objective.

3. **Nonce** (x_3, x_4) - 128-bit value that must be random and ideally unique to ensure freshness and prevent replay, chosen plain-text and other types of attacks.

After initialising the internal state, the permutation function p repeats a times. The permutation function p consists of several operations shown in Figure 1.4. These operations perform the following actions.



(a) Round constant addition p_C

(b) Substitution layer p_S with 5-bit S-box $\mathcal{S}(x)$

(c) Linear layer with 64-bit diffusion functions $\Sigma_i(x_i)$

Figure 1.4: Different operations performed by permutation function p on internal state S during the encryption process (AEAD) [16]

1. Addition of round constant to the register x_2, which changes every iteration of permutation p and is computed by the following equation:

   Definition 5. Computation of round constant [17]:

   $$RC_i = \texttt{0xF} - i \;||\; \texttt{0x0} + i \tag{1.4}$$

   where:

   i is an index of the round starting from 0 and
   $||$ is the concatenation of 2 parts.

2. The substitution layer uses a non-linear function with a 5-bit long input mapping called an *Sbox*. Input to this function is taken by bit-sliced columns from words x_0 to x_4, transformed by the *Sbox*, and written back to the state S. Like in the AES, this operation removes the linear dependency of encrypted text to the plaintext and key [14]. The *Sbox* is shown in Figure 1.5[1] but is usually implemented as a look-up table because it is faster than manual computation. On the other hand, a look-up table is more memory demanding.

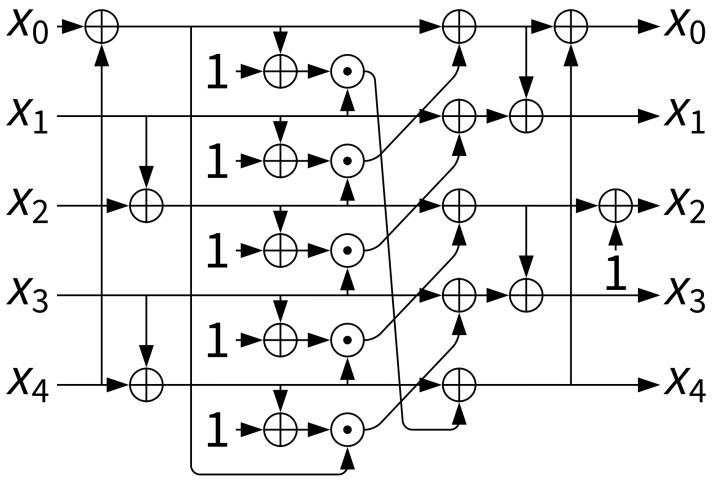


Figure 1.5: Logical representation of the ASCON *Sbox* [3]

1. In Figure 1.5 $\oplus$ represents logical XOR and $\odot$ represents logical AND.

3. The linear diffusion layer applies an Function 1.5 with different rotation values for each register x_i. These values are shown within Table 1.3 [14].

   Definition 6. Linear diffusion layer function $\sum i(x_i)$ applied on each register x_i with different y and z parameters.

   $$\sum i(x_i) = x_i \oplus (x_i >> y) \oplus (x_i >> z) \tag{1.5}$$

   where:

   $>>$ is right circular shift (right rotation).

Table 1.3: Different rotation values y and z used in the linear diffusion layer function 1.5 for each register x_i

i	y	z
0	19	28
1	61	39
2	1	6
3	10	17
4	7	41

Later parts of the ASCON encryption process, such as adding associated data or plain text and extracting ciphertext and tag, are unnecessary during the side-channel attack and can be omitted from the theoretical background.

2 Setup and techniques

2.1 Measurement setup

A measurement setup that is able to capture power consumption from JavaCard is necessary to obtain a power traces later used to perform a side-channel attack using correlation power analysis (CPA). Our setup consist of the oscilloscope Picoscope 6404D, the board LEIA-Solo 1.4, and power supplies RIGOL DP832 for AES traces and Geti GLPS3010H for ASCON traces. The power supply swap occurred because the RIGOL power supply broke and was replaced between the measurements of AES and ASCON.

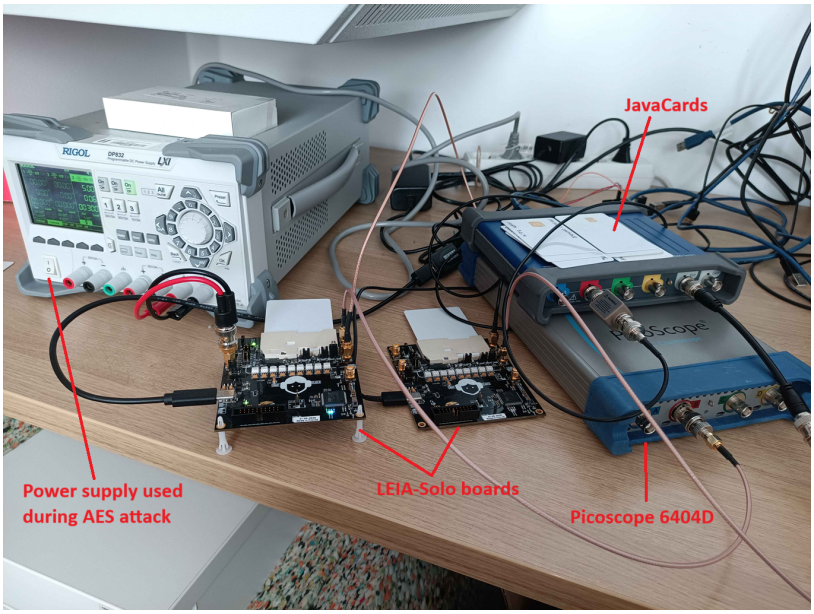


Figure 2.1: Measurement setup with power supply used during AES attack on the JavaCard

The board LEIA-Solo connects JavaCard to the oscilloscope and the power supply. Version 1.4 of the board has low resistance from the manufacturer and lowered measurement precision. Hence, the board resistance was increased by swapping the shunt resistor from 0.1Ω to 22Ω.

During measurements, we used the oscilloscope channel B configured using the Picoscope SDK[1] as follows:

- Offset: $-175mV$,
- Channel Range: 5,
- Threshold: 9753,
- Trigger AUX: 5,
- Threshold direction: 2 (Rising edge),
- Sample rate: $1.25GS/s$.

JavaCard used in the setup was NXP JCOP 4 Classic Edition that supports JavaCard API version 3.0.5 and Global Platform specification 2.3. We chose this card because it was the fastest and most recent available in the CRoCS laboratory.

2.2 Measurement script

The measurement script is used to capture the traces from a JavaCard. First, the script prepares the card reader to allow communication with the JavaCard and sets up triggers. Then it selects an applet with ASCON implementation and sends the key used by the encryption algorithm. Furthermore, it needs to set up the oscilloscope with the aforementioned configuration. The measurement script then sends commands for encryption to the card, where the oscilloscope automatically captures the power trace of computation. The voltage sample within a trace is represented by an 8-bit integer produced by the Picoscope's built-in AD converter. New plaintext and nonce are sent before every measurement and later saved within the output file with the obtained trace. Knowledge of plaintext or nonce is essential during CPA attacks.

The measurement script was provided to me by Milan Šorf from the CRoCS lab. Nevertheless, I improved the script structure, added support for multiple output formats, and added an option to fix key, nonce or plaintext. Additionally, I implemented interleaving of fixed and random nonce necessary for TVLA evaluation (Test Vector Leakage Assessment) [2].

1. https://github.com/picotech/picosdk-python-wrappers
2. ~~Learn more~~ in Chapter 4.2.

Move details ave

2.3 File formats

2.3.1 Requirements for file formats

Files that contain measurements from the oscilloscope can have tenths, even hundreds of gigabytes. Therefore, these files needs to be processed in chunks that fit into regular-sized memory. The technique commonly used is called file memory mapping. It temporarily loads essential data from the file on a disk required to execute some operation, frees them when no longer needed and modifies the file content to reflect potential changes. Consequently, file formats used to save traces should be able to store multi-dimensional arrays and provide seamless memory mapping.

2.3.2 TRS file format

Traces captured by the original measurement script were saved into a proprietary file format called Inspector Trace Set with extension trs developed by Riscure [18]. The file format intended usage is for captured power traces from embedded devices or smart cards [19, 18]. Riscure developed and published a library for Python for easier processing and manipulation of TRS files by third parties [18]. Documentation for this library is available online.

The TRS file format is relatively simple. The header consists of named parameters in various formats, identical for every trace. Every trace has named metadata that may differ between traces, but their format and count must remain consistent.

The Python library allows loading a single trace from the file into memory for processing. However, it cannot fetch only relevant sections of the trace. Instead, it loads all samples within a trace into memory. This behaviour significantly slows operations that need just a section of samples from the trace, making this library and consequently the file format unsuitable for optimised processing.

2.3.3 Numpy file formats

Limited functionality of the trsfile library (especially the inability to load only part of a trace into memory) and usage of the Python numpy library within the attack implementation made me decide to use the

numpy file formats (NPY and NPZ) for the trace storage. The NPZ file format is an archive with a collection of NPY files, where NPY is just one array in binary representation with a header containing details about the array, like its size and dimension(s). Because the NPZ format is an archive, all data within arrays must be loaded in main memory before saving, making it unusable for large arrays.

These file formats support sliced memory mapping, and working with them is almost equivalent to a regular numpy array. However, they do not support metadata for each trace, so two separate arrays are needed, one for the traces and the other for related metadata. Also, metadata cannot be named and can only be distinguished by their position in the corresponding array. Still, these are minor inconveniences compared to the performance benefits.

2.3.4 Comparison of file formats

Due to the advantages mentioned previously, I have mainly used the NPY file format because I have often worked with large files. This format's main disadvantage is storing metadata and corresponding traces separately due to their incompatible dimensions. However, it is not a substantial obstacle and is only slightly more complicated to implement. Table 2.1 shows the advantages and disadvantages of the previously mentioned file formats.

Table 2.1: Advantages and disadvantages of supported file formats

File format property	NPY	TRS	NPZ
Can store multiple arrays with incompatible dimensions	X	✓	✓
Can be read and modified without loading all traces into the memory	✓	✓	✓
Can be saved in chunks, without needing all data in memory at one time.	✓	✓	X
Can read only relevant samples without loading the entire trace into the memory	✓	X	✓
Can store natively named metadata about traces	X	✓	X
Supports compression	X	X	✓

2.4 Attack techniques

2.4.1 Alignment of traces

Captured traces are unsynchronised due to several aspects that impact the timing of operations, such as clock jitter, cache, imprecise triggers and more. Therefore, an operation that happens in one trace at a specific time (sample) can be present elsewhere in another trace. This means that traces are not aligned with each other.

Aligned traces are essential for a successful attack. Otherwise, an attacker would calculate the correlation between samples representing different sections of an algorithm, consequently lowering the correlation. As a result, higher values of correlation that signify the correct key can be lost. Making the attack fail because correct key guesses are indistinguishable from incorrect ones.

Because similar computations happen in every trace, the power consumption has a linear dependency between traces where the equivalent computation occurs. As a result, correlation can be used to align the traces according to the patterns, representing the same operation.

The alignment technique takes the reference trace (often the first) and tries to align other traces using normalised cross-correlation. Normalised version of correlation reduces the dependency on the amplitude and peaks in real traces and therefore is more suitable for power measurement traces [20]. The correlation is calculated on different trace shifts, and the shift with the largest correlation is used to align the trace by reversing it. When the correlation between the aligned trace and the reference trace is low, the trace should be discarded because the trace may be faulty and can negatively impact the results. If many traces are faulty, it might signal that we should use a different reference trace or that there is an issue with the measurement setup.

2.4.2 Window Resampling

Resampling is a signal processing technique that takes a predefined number of samples from a trace called a resampling window. It calculates the window average and writes it as a new value into the output. The resampling window is then moved by a predefined step, and its average is again saved as the output sample representing the whole

window. This process is repeated until the whole trace is resampled. Resampling windows should slightly overlap to smooth transitions between output values representing the window average.

Resampling can significantly reduce the size of traces, but can impact the results of the side-channel attack because it reduces the strength of the original signal [6]. On the other hand, it can smooth out some irregularities within samples and reduce noise, thus increasing the probability of attack success [6]. Resampling also improves alignment because the aforementioned smoothing makes finding similar sections easier. Algorithm 1 shows pseudo-code for resampling.

Algorithm 1 Trace resampling

Require: Trace T with N samples
Require: Resample step - $step$
Require: Window size - $window$
Ensure: $step < window$
1: $output \leftarrow [\,]$
2: **for** $index \leftarrow 0$ up to Floor(N / ($window$ - $step$)) **do**
3: $start \leftarrow index * step$
4: $chunk \leftarrow T[start : start + window]$
5: $output[index] \leftarrow$ Average($chunk$)
6: **end for**
7: **return** $output$

2.4.3 Alignment guided by resampling

As said previously, resampling can reduce the strength of the signal and thus secret leakage as well, possibly decreasing the chances of attack success when not balanced by the benefits provided by the resampling. We should align traces even when we do not want to resample them. Resampling makes alignment easier by exposing patterns by which the traces are aligned. Therefore, we need a way to align traces without applying resampling to them while preserving the alignment benefits provided by resampling.

One solution is to create a copy of traces that is resampled with a step equal to one and a resampling window of a sufficient size to make patterns detectable. As a result, resampled traces have the

same number of samples as the original. Then, we calculate the shifts necessary to align these specifically resampled traces. Obtained shifts are then applied to the original traces, consequently aligning them without resampling. I call this approach advanced alignment.

2.4.4 Reduction of trace size

One of the main obstacles during the power analysis attack is the size of the traces, which may have tens or even hundreds of gigabytes. One solution for this problem is to shorten measurements by including only the relevant part of the encryption process containing the point of the attack, which is usually in the first round[3]. Patterns between permutation rounds are often visible in traces. Consequently, we should be able to identify the location of the first round. This location does not significantly vary between measurements. Therefore, we can measure several traces to determine the interval of the first round, which should be saved in measurements later used by CPA attacks. Patterns can be visible even within a single round, allowing us to decrease the number of preserved samples from the measurement even further.

Another technique that may help to find the location of the attack is to insert some distinctive operation visible in measured traces before and after the potential leakage location [6]. Random number generation in JavaCard has this property. However, an attacker must be able to modify the implementation or at least have access to the source code to replicate it on a different card.

3. Round repetition often does not provide anything valuable to the CPA attack and thus we usually choose the first round.

3 Attack implementation

This chapter describes the implementation of the AES attack, ASCON attack, and other related scripts. It includes a description of new features and changes from the original code, provided by the CRoCS laboratory. The code is available here.

3.1 Project structure

Implementation uses Python 3.13.2 as the programming language. I chose Python because the original scritps were also written in Python, Python supports trsfile library, and working with large matrices can be done efficiently using libraries like numpy. The main project folder contains **README.md** file that shows the necessary steps to run the implementation with a brief functionality description. The project is separated into multiple modules (folders):

Aes - Contains code specific for AES attack.

Ascon - Contains code specific for ASCON attack.

- Models - Object models used only in the ASCON attack.
- ProofOfConcept - Contains part of ASCON cryptographic algorithm with trace generator used for testing purposes, more details in Chapter 4.3.3.

AttackTools - Module for signal processing tools necessary to perform the attack (Alignment, Resampling, etc.).

Common - For generic code that can be used in any CPA attack.

- Generics - Abstract objects that contain generic functionality. More on them in Chapter 3.3.
- Maps - Specific conversions that map input to predefined output.
- Models - Generic models that can be used in any CPA attack.

Configuration - Configuration of all attacks and attack tools.

- Models - Configuration models that can be inserted into attack tools, attacks and generics to specify their behaviour.

Helpers - Module with code that performs functionality needed in various parts of the code to avoid duplication.

MeasurementScripts - Contains scripts that gather power traces from a device.

- Models - Data models used specifically by measurement scripts.

Tests - Tests that help to assess functionality and speed of specific code.

Tvla - Contains Test Vector Leakage Assessment (TVLA) implementation. For more details see Chapter 4.2.

Utilities - Module for code that performs operations on trace files like reading, converting and plotting graphs of traces.

3.2 Features

3.2.1 Automatic alignment

Aligned traces are essential to perform a CPA attack due to the reasons mentioned previously in Chapter 2.4.1. Nevertheless, traces must be aligned repeatedly because every trace measurement has slight timing differences that shift traces relative to each other. As a result, traces become unaligned after some period, even when previously aligned. Therefore, to automate the attack, we also need to realign traces automatically. Traces are attacked by intervals due to the large number of samples within one trace, which may reach tens of millions. Automatic alignment takes only relevant measurement samples to perform alignment and following attack(s)[1]. We can specify the length of the alignment interval, how often the automatic alignment should occur (for example, every second attack) and the maximum trace shift to perform alignment. The alignment interval is always centred based on the attack interval(s) before the next automatic alignment. Automatic alignment allows the discarding of traces below a predefined correlation threshold to get rid of defective ones.

Figure 3.1 shows an example of two different configurations of automatic alignment. The first example shows a large attack interval where automatic alignment is performed every attack. The attack interval is larger than the alignment interval and is therefore used as

1. Attack represents finding the best correlation between theoretical and actual power consumption on key guesses within the current attack interval (subset of samples).

the internal interval, representing currently required samples loaded from the disk. After aligning the traces, the output interval is written to a new file containing only samples used in the next attack(s). The second example shows small attack intervals where automatic alignment is performed after every second attack. However, in this case, there is an overlap between attack intervals. As a result, the alignment interval needs to be centred correctly, considering the overlaps. In this case, the alignment interval is larger than attack intervals with overlap and therefore is used as the internal interval.

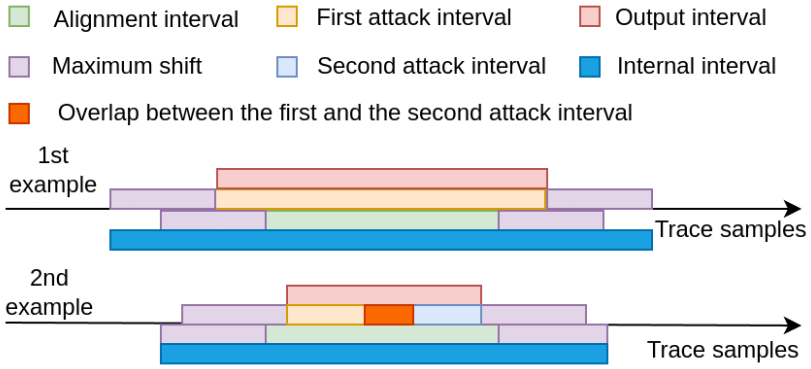


Figure 3.1: Automatic alignment

3.2.2 Multi-threading

The original code does not have multi-threading support. I added support for parallel processing using Python native ProcessPoolExecutor[2] on Resampling, Alignment and Trace Converter[3]. I have not used ThreadPoolExecutor[2] due to a limitation by Global Interpreter Lock (GIL) that prevents the interpreter from running in multiple threads simultaneously and causes some performance issues. This problem does not occur when using ProcessPoolExecutor because every process has its own instance of the Python interpreter. However, this created another problem with inter-process communication. I used global variables for memory-mapped trace files that allowed access to traces for all processes without copying them. Every process

2. See https://docs.python.org/3/library/concurrent.futures.html for more details.
3. See Chapter 3.4 for more details.

has a trace assigned, so they do not work with the same underlying data. A result from processes is stored in the shared memory and copied to the output in case of NPY/NPZ file format or returned as an instance of the Trace object from the TRS library in case of TRS output format.

A count of processes used during computation can be manually configured. Using multi-threading when working with slow mechanical hard disks is not recommended because they usually become the bottleneck even when the process count equals one. Multi-threading scatters reads from the disk, slowing it even further and making the runtime of a script longer. Scattered reads are not problematic when using Solid State Drives (SSD).

3.2.3 Logging

Preserving configuration, location and visualisation of a successful attack is necessary to assess and replicate the attack results in the future. Therefore, automatic logging of these data is helpful, but it was not present in the original implementation. When starting an attack, a log file is created, containing information like attack interval, used resampling, correlation and other information for every successful attack on separate lines. This file is named after the date and time when the attack started with the .log extension, and is saved in a folder defined in the configuration. If graph logging is enabled, a folder with the date and time in its name is created, and a log file is generated in this folder instead. This folder contains graphs of successful attacks named after the current stage[4] and attack interval.

Another logging is located in the alignment attack tool. This logging is text only and contains shift, correlation and trace index to asses the result of alignment. This logging is performed in asynchronous thread due to alignment's multi-threading support. Data for the logging thread are passed via a queue that supports writes from different processes.

Alignment logging and attack logging are optional and may be turned off if not wanted.

4. Current stage represents a key part currently attacked - byte index in the AES case, and the index of the intermediate bit needed to extract key in the ASCON case.

3.2.4 Incremental correlation

Correlation is usually calculated by two-pass algorithms that first calculate the mean and then compute the final correlation. This method is fast and stable for small datasets, but for large datasets, traversing the data twice may result in a noticeable performance hit and minor floating-point errors [21]. Furthermore, to generate graphs that show correlation evolution with an increasing number of traces, the two-pass approach needs to compute correlation for every point in the graph, causing a significant performance hit due to repeated computation for identical data. The last issue is that when calculating correlation, all traces have to be loaded into memory at once, increasing memory requirements. For these reasons, the two-pass approach is unsuitable for an efficient CPA attack.

The method we will use to compute correlation during CPA attacks is called Incremental correlation, which uses incremental statistic formulas for mean, variance and covariance [21]. This method allows loading only one trace at a time to the memory and updating the internal state with constant memory requirements. At any time, correlation can be computed from the internal state without recomputing previous data, making creating an evolution graph much less demanding. Furthermore, the algorithm traverses data only once, making it more suitable for large datasets. However, it can be slower when computing correlation on small datasets with many key guesses. A classic two-pass algorithm is recommended in that case. The implementation contains both calculation methods, so a user can choose which one to use according to circumstances.

The following formulas from 3.1 to 3.8 [21] show the correlation calculation between datasets A and B using incremental statistics. The formulas are utilized by incrementing the current index i from 0 to the samples count of both datasets. A and B must have identical number of samples.

Definition 7. Calculation of delta used later in the computation

$$\delta A_i = a_i - \text{mean}A_{i-1} \tag{3.1}$$

$$\delta B_i = b_i - \text{mean}B_{i-1} \tag{3.2}$$

where:

i	is the index of the currently processed sample,
a_i	is the current sample from dataset A,
b_i	is the current sample from dataset B,
$\text{mean}A_{i-1}$	is mean of the set A up to $i-1$ and
$\text{mean}B_{i-1}$	is mean of the set B up to $i-1$.

Definition 8. Calculation of means of A and B using incremental statistics

$$\text{mean}A_i = \text{mean}A_{i-1} + \frac{\delta A_i}{i} \tag{3.3}$$

$$\text{mean}B_i = \text{mean}B_{i-1} + \frac{\delta B_i}{i} \tag{3.4}$$

Definition 9. Calculation of variances of A and B using incremental statistics

$$\text{var}A_i = \text{var}A_{i-1} + \delta A_i \cdot (a_i - \text{mean}A_i) \tag{3.5}$$

$$\text{var}B_i = \text{var}B_{i-1} + \delta B_i \cdot (b_i - \text{mean}B_i) \tag{3.6}$$

where:

$\cdot$	is basic multiplication,
$\text{var}A_{i-1}$	is variance of the set A up to $i-1$ and
$\text{var}B_{i-1}$	is variance of the set B up to $i-1$.

Definition 10. Calculation of covariance between A and B by incremental statistic

$$\text{cov}AB_i = \text{cov}AB_{i-1} + \frac{i-1}{i} \cdot \delta A_i \cdot \delta B_i \tag{3.7}$$

where:

$\text{cov}AB_{i-1}$	is covariance between A and B up to $i-1$.

Definition 11. Calculation of correlation with n samples loaded

$$corr = \frac{\text{cov}AB_n}{\sqrt{\text{var}A_n} \cdot \sqrt{\text{var}B_n}} \tag{3.8}$$

3.2.5 Other features

Implementation provide many small features that help with overall usability and efficiency compared to the original scripts. Here is the list of these features.

- Automatic determination of input file format using extension.
- Possibility to perform an attack without knowledge of the secret key using minimal correlation and the difference between the first two guesses.
- Optional result caching of power predictions for the same input.
- Support for multiple file formats (TRS, NPY and NPZ).
- Results of the attack, including best key guess, its location and correlation, are shown in a clear table.
- Use of numba library to compile Python code into C language to increase performance and parallelisation of the code.
- Dynamic changing of the file name according to the operations performed on it (resampling, alignment, cut, etc.).
- Configuration and input checks using assert statements.
- Everything is written in a memory-friendly way and can be run with almost any size of the main memory.
- Object-oriented programming to encapsulate specific functionalities and their state.
- Use of atexit to automate file closing and shared memory release.
- Allowed cutting of traces in attack tools to decrease the number of processed samples and the output size.
- Full attack automation by attack stage swapping, result logging and automatic alignment.
- Warning a user when many traces were discarded during automatic alignment by log and console.
- Use of enums and data models to effectively organise and manage related values.
- Support for non-interactive graphs that do not stop execution.
- Capability to dynamically reduce the size of the NPY file format.
- Compatibility with various operating systems.
- Automatic discovery of related data file when using the NPY file format.
- Parameter wrapper that helps safely extract metadata about traces from numpy arrays using their names.

3.3 Generics

Tasks like loading traces from the disk, logging, correlation computation or graph generation are necessary when performing any CPA attack, and are independent of the underlying cryptographic primitive. I extracted these operations into common scripts or generic objects utilised by inheritance that provide an interface encapsulating shared functionality, or automatically perform these tasks and provide access to related data (state). Generic objects are single-purpose to avoid redundancy in a child object.

Nevertheless, some properties and functions should not be used in a child object. However, Python cannot enforce this, so I use the underscore convention '_' before the name of a function or property to signal that it is private and should not be used outside the object or by its child.

There is the use of inheritance even between generic objects, so they implement only the necessary functionality. For example, modifying traces requires loading them beforehand. However, suppose we only need to read the traces and not write them back to a disk. Writing functionality is redundant in that case, and we would like to inherit only the loading component. Therefore, these two functionalities are separated.

The Figure 3.2 shows relationships between main generic objects and their instances.

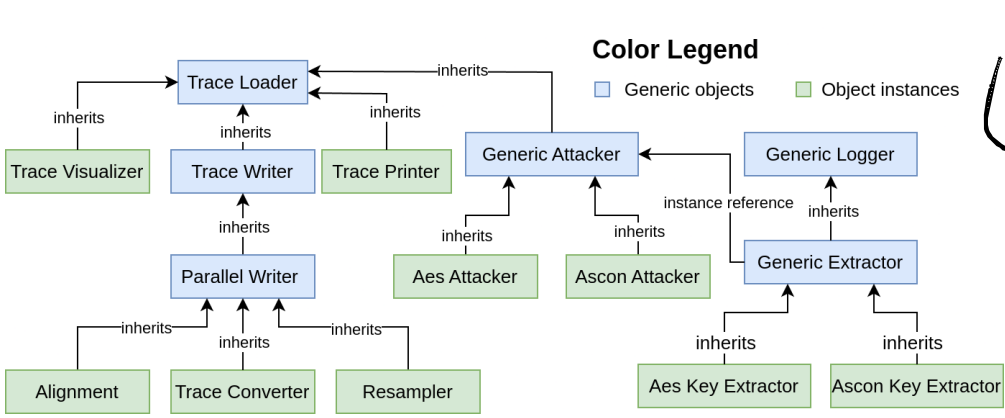


Figure 3.2: Relationships of main generics and their corresponding object instances

Follows the list of generics, their purpose and interface functions that should be used by child objects.

Trace Loader loads traces in supported formats (TRS, NPY, NPZ) and provides metadata and functionality to work with them.

- *reload_file* reloads metadata and traces from a disk to reflect potential changes. Initial loading is automatic when the object is created.

Trace Writer extends Trace Loader with support to create output files and write to them.

- *create_output* creates output files and memory objects based on the configuration.
- *save_output* saves or flushes metadata and traces from memory to a disk.

Parallel Writer extends Trace Writer with support to process and write metadata and traces in parallel.

- *copy_from_shared_memory* copies parallel computation results from shared memory to the output. Allows choosing which traces are copied by index filtering.
- *check_process_bounds* checks if the process count is larger than the remaining traces that need to be processed. If yes, lower the count of processes to match the unprocessed trace count. This method is useful when a trace count is not divisible by a process count.
- *copy_data_from_trace* copies headers from the TRS file format to a numpy array for metadata using shared memory.
- *free_shared_memory* is a helper method to free shared memory by its name.
- *allocate_shared_memory* is a helper method to allocate shared memory with the selected name and size in bytes.

Trace Creator automatically creates a new file with traces in supported formats (TRS, NPY, NPZ) without loading existing traces.

- *save_output* saves or flushes metadata and traces from memory to a disk.

Generic Attacker provides functionality needed in every CPA attack.

- *should_update_alignment* performs automatic alignment if necessary and reloads files. This method should be called before every attack interval.
- *add_trace* adds trace with its power consumption predictions for processing. Predictions depend on the cryptographic primitive used. Therefore, they cannot be generalised.
- *compute_correlation* computes current correlation between traces and related predictions provided in *add_trace* method.
- *update_evolution* computes current correlation, updates the evolution graph and returns correlation results. It internally calls the *compute_correlation* function.
- *evaluate_attack_stage* is used to evaluate attack success and possibly generate graphs, after adding all traces.
- *is_index_graph_point* decides if the evolution graph should be updated on the current trace index based on the evolution step defined in the configuration.
- *reset* is called after an attack interval is evaluated to reset the state of the Generic Attacker. The hard reset flag should be set to True when moving between attack stages.
- *attack* is an abstract method that executes the CPA attack and must be overridden by the specific attack implementation. Generic Extractor uses it to start an attack on the current attack stage.

Generic Logger provides logging functionality with support for creating and updating log files, attack graphs and logging folders.

- *get_log_message* is an abstract method that needs to be implemented by the specific attack implementation because the log message can differ between various attacks.
- *get_current_graph_path* gets the file name of the graph that will be saved as a log. This name includes the current attack stage and sample interval. The graph is saved in the attack visualisation script that renders it. This method only returns the graph's location.
- *log_success_message* logs the success message by internally calling overridden *get_log_message* and writing it to the file or console.
- *log_aa_warning_message* is called when automatic alignment discards too many traces to log a warning message.

Generic Extractor extends Generic Logger and provides a way to run an attack on multiple stages with result accumulation and printing after the attack.

- *perform_result_operation* is an abstract method that should perform attack-specific operation needed to extract the key after acquiring the result of the attack on a subset of samples (attack interval).
- *extract_specific_stage* performs a CPA attack on a specific stage using the attacker instance. Automatically resets the state after the attack and calls *perform_result_operation*.
- *extract_key* extracts the whole key by executing all attack stages.

3.4 Utilities

Utilities provide extra functionality to make analysing and processing of traces more convenient. They can assist in finding the leakage's approximate location by trace visualization and reduce the trace size.

Trace Converter allows to convert traces between supported file formats (NPY, NPZ and TRS) without applying any additional processing. The conversion from numpy formats (NPY and NPZ) back to TRS format is not implemented due to its lack of usefulness in this thesis. Furthermore, Trace Converter allows manual filtering of potentially faulty traces using their indexes and reducing trace length so that the output only contains relevant samples to the current attack. Processing of the traces can be performed in parallel to increase performance.

Trace Printer is a basic script that prints headers and trace samples for all supported files. Printing parameters like format, trace index, and sample interval can be specified in the configuration situated above the script. An attacker can use this script to verify that the trace file contains correct values and is not corrupted.

Trace Visualizer - Updated visualization script that my supervisor provided to me. It can render multiple traces simultaneously in one graph or separately. Graphs are interactive when Tkinter library is installed for example via the pip package manager.

3.5 Configuration

Original scripts were configured using global parameters defined directly within the code. This approach had several drawbacks, the most problematic being the difficulty of using the same functionality from multiple sources with different configuration needs. As a result, global parameters had to be frequently modified to adhere to the currently executed script. I removed these global parameters and replaced them with configuration models that encapsulate all settings influencing the script or its parents, if there are any. These configuration models are injected into the corresponding object implementing the script, allowing different sources to use different configurations without interfering with each other. Configuration models are automatically created within configuration files located in a designated directory.

The following sections contain non-trivial parameters used in various configuration models with Python type notation. Star is naming a wildcard when the configuration parameter occurs multiple times.

3.5.1 General parameters

This section contains parameters without binding to specialised functionality utilised in various objects.

1. Input-Output
   - *_TARGET_PATH: str - Location of file with traces.
   - *_DATA_PATH: str - Location of file with associated data for the traces, such as key, nonce or plaintext. Only used when using NPY file format, due to the separation of data and traces. If empty, automatically determine name from *_TARGET_PATH.
   - *_OUTPUT_FOLDER_PATH: str - Folder where script saves the output. If empty, then use the same folder where the input is located.
   - *_OUTPUT_TYPE: enum - Output file format.
   - *_INCLUDE_CUT_OUTPUT_NAME: bool - When the script reduces trace length, the output file name will reflect it.
   - *_OUTPUT_COMPRESSED: bool - Specify whether to use output compression when using the NPZ file format as the output.

- *_DATA_PARAMETERS: object - Object that defines the length and position of named metadata parameters. Must be specified.

2. Generic location

- *_SAMPLE_START: int - Sample index where the processing should start.
- *_SAMPLE_END: int - Sample index where the processing should end (None to get the index of the last sample).
- *_TRACE_START: int - Index of the first trace to process.
- *_TRACE_END: int - Index of the last trace to process (None to end with the last trace).
- *_TRACE_COUNT: int - Alternative way to restrict processed traces using trace count from the beginning. Not used in the same configuration with *_TRACE_START and *_TRACE_END.

3. Multi-threading

- *_PROCESS_COUNT: int - Number of threads (processes) used in the script computation. I recommend specifying the number corresponding to the count of CPU threads. However, when using slow mechanical hard drives to store traces, use only one process to avoid scattered reads from the disk.

3.5.2 Alignment properties

This section contains parameters specific to alignment.

- ALIGNMENT_DRY_RUN: bool - Calculate alignment without saving to a disk. Consequently, saving time and avoiding disk wear (SSD) when only evaluating the alignment performance.
- ALIGNMENT_REFERENCE_SAMPLE_INDEX: int - Index of a trace to use as the reference trace.
- ALIGNMENT_START: int - Sample index where to start alignment with the reference trace.
- ALIGNMENT_END: int - Sample index where to end alignment with the reference trace.
- ALIGNMENT_LOG_ENABLE: bool - Create a log file with shift and correlation for every trace.

- ALIGNMENT_THRESHOLD: float - The minimum correlation expected to consider the alignment successful. When not met, the trace is discarded.
- ALIGNMENT_MAX_SHIFT: int - Maximum shift of trace allowed.
- ALIGNMENT_SHOW_GRAPH: bool - Show an interactive graph to evaluate the alignment result.
- ALIGNMENT_GRAPH_TRACE_COUNT: int - Number of traces to show in the graph.

3.5.3 Resampling parameters

This section contains parameters specific to resampling.

- RESAMPLER_WINDOW_SIZE: int - Number of samples to include in the window.
- RESAMPLER_OVERLAP: int - Number of samples which overlap in neighbouring windows.
- RESAMPLER_ABS: bool - Samples transformed to absolute value before processing.
- RESAMPLER_USE_FLOAT: bool - If true, averages from resampling are saved as floats (increases the output size while increasing sample precision).

3.5.4 Attack parameters

This section contains parameters specific to any CPA attack without an explicit attack prefix. Star represents the name of the particular attack.

here it is clear

1. Attack features

   - *_USE_AVERAGER: bool - Average traces with the identical data. Not recommended when an attack has only a few possible data combinations and not enough information is preserved.
   - *_USE_INCREMENTAL_CORRELATION: bool - Use when attacking larger amounts of traces with few key bits candidates attacked in one attack stage.
   - *_USE_PREDICTION_CACHE: bool - Use cache with already computed predictions to avoid recalculation. Not recommended

when there are numerous possible input combinations for prediction because it considerably increases memory demand.

- *_ADVANCED_ALIGNMENT_LENGTH: int - Length of the resampling window used during advanced alignment. This feature was described in 2.4.3. Zero to not use advanced alignment.
- *_ADVANCED_ALIGNMENT_USE_ABS: bool - Use absolute value on resampling intermediate result during advanced alignment. It can help with resampling accuracy.

2. Attack location

- *_ATTACK_LENGTH: int - Number of samples to attack in one iteration.
- *_ATTACK_STEP: int- Step size between attack iterations. When larger than ATTACK_LENGTH, some samples are skipped. If smaller, attack iterations will overlap.

3. Attack success

- *_USE_CORRECT_KEY: bool - Use the correct key to determine attack success.
- *_MIN_CORRELATION: float - Minimum correlation to consider the attack successful, set it to zero to avoid this feature.
- *_CORRELATION_DIFFERENCE: float - Minimum correlation difference between the first two candidates to consider the attack successful, specify zero not to use this feature.

4. Automatic alignment

- *_AUTOMATICALLY_UPDATE_ALIGNMENT_ON: int - Defines how often automatic alignment occurs between attack iterations. Use zero to disable automatic alignment. Described in more detail in Chapter 3.2.1
- *_ALIGNMENT_INTERVAL_LENGTH: int - Number of samples compared with the reference trace.

- *_ALIGNMENT_INTERMEDIATE_TYPE: enum - Intermediate file format created by automatic alignment.

5. Extractor configuration

- *_ATTACK_STAGE: int - Specify the attack stage which determines which part of the key is attacked, use -1 to attack all stages and try to recover the entire key.
- *_STOP_ON_FIRST: bool - Stop the execution on first successful attack.

6. Graph generation

- *_SHOW_GRAPH_EVERY_ATTACK: bool - Show attack graph for every iteration, even when the attack is unsuccessful. Useful for debugging reasons.
- *_SHOW_INTERACTIVE_GRAPH: bool - Use an interactive graph that pauses the attack execution.
- *_SHOW_GRAPH_ATTACK_SUCCESSFUL: bool - Show graph if the attack was successful.
- *_EVOLUTION_STEP: int - Number of traces between correlation points in the correlation evolution graph.

7. Logging

- *_LOG_FOLDER_PATH: str - Location in the file system where the log from the CPA attack is saved.
- *_LOG_RESAMPLER_USED: bool - Log includes the current configuration of the resampler.
- *_LOG_SUCCESS_ATTEMPTS: bool - Enable logging of successful attempts to the log file. Otherwise, logging into the text file is disabled.
- *_LOG_INCLUDE_GRAPHS: bool - Log graphs of successful attacks for future use in the attack examination.
- *_PRINT_LOG: bool - Enable printing log to the console.

3.5.5 Utilities configuration

Utilities are single-purpose scripts. Therefore, their configuration parameters are situated directly in the script above the implementation, just after imports. Configuration models are created at the bottom, where the utility object is initialised and its entry point called.

1. Trace Converter non-generic configuration parameters.
   - CONVERTER_TRACES_TO_SKIP: set[int] - Set of integers representing indexes of traces that will be skipped and not copied to the output.

2. Trace Printer non-generic configuration parameters.
   - PRINTER_TRACE_INDEX: int - Index of the trace whose samples and metadata are printed by the script.
   - PRINTER_SHOW_DATA_HEX: bool - Printed data are represented in hexadecimal format.

3. Trace Visualizer non-generic configuration parameters.
   - VISUALIZER_GRAPH_TRACE_COUNT: int - Number of traces plotted in one graph.
   - VISUALIZER_GRAPH_COUNT: int - Total number of graphs rendered by the script.
   - VISUALIZER_TOGETHER: bool - Render all graphs simultaneously in one window. Otherwise, render them one by one.
   - VISUALIZER_SPACING: float - Space size that separates graphs in simultaneous visualisation (VISUALIZER_TOGETHER is True). Use zero not to modify default spacing.
   - VISUALIZER_X_LABEL: str - Label for x-axis in graphs.
   - VISUALIZER_Y_LABEL: str - Label for y-axis in graphs.
   - VISUALIZER_DISPLAY_LABELS: str - Display labels for the first N traces.
   - VISUALIZER_LABEL_LOCATION: str - Location of trace labels, check the Pyplot documentation for more information.

4 Attack execution

CPA attacks rely on leakage of the secret key through the device's power consumption, as described in Chapter 1.1.1. We need a power model influenced by a combination of key and value that changes every run of the encryption process and is known to the attacker. We calculate the correlation between the theoretical power consumption obtained by the power model for every key guess and the captured traces. When the correlation is high enough compared to other key guesses, the key guess should represent part of the secret key with some degree of confidence.

Usually, data transferred over the bus affects power consumption. To create power model predictions, we need to compute an intermediate value in the encryption process, dependent on the key.

4.1 AES

4.1.1 Attack description

CPA attack on AES-128 usually regard only the first two operations executed during the encryption process to perform the attack as defined in Chapter 1.3. The first operation is the XOR between plaintext and key. The second operation is the *Sbox* function calculated. *Sbox* is a byte mapping computed 16 times for each byte of the current AES block. Therefore, we can split the attack into multiple stages for each *Sbox* calculation, extracting one byte of a key at a time with 256 possible combinations. Thus, reducing the required brute-force attack to obtain a key from 2^{128} combinations to just $16 * 2^8$.

Definition 12. Power model used to extract one byte of AES encryption key[9]

$$PM_{AES} = HW(Sbox(K_s \oplus I_s)) \tag{4.1}$$

where:

K_s	is the key guess for the current stage,
I_s	is the byte of user input relevant to the current stage,
Sbox	is the *Sbox* byte mapping function and
HW	is hamming weight.

Attacking all samples from every trace simultaneously is not possible in a regularly-sized memory if traces are numerous and large. Furthermore, alignment of traces diverges over time and should be performed repeatedly. Therefore, the attack stage is further separated into iterations. Every iteration takes only a subset of samples from every trace, processes them and evaluates the results.

4.1.2 AES attack results

Attacking AES via power analysis was reasonably straightforward. Byte leakage via Hamming Weight from the bus is sufficient to distinguish the correct key. The leakage of the JavaCard's internal state is substantial, and an attacker can extract the secret even with large resampling (1000 step, 900 overlap) in less than 200 traces while using automatic alignment (10000 alignment interval, 5000 max shift). Figure 4.1 shows the representative result using the configuration above via three graphs, where the red line represents the correct key.

1. The top graph depicts the correlation between the power model prediction and the actual power consumption.
2. The bottom left graph depicts the evolution of correlation with an increasing amount of traces.
3. The bottom right graph depicts the ranking of the correct key.

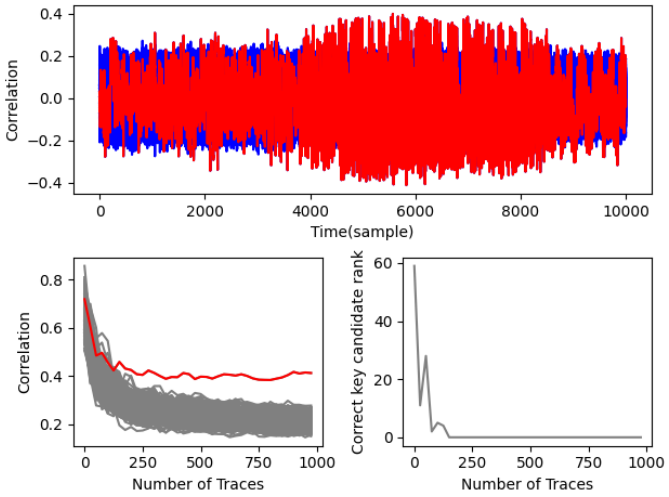


Figure 4.1: Sample attack result of the CPA attack on the AES-128

We can utilise these graphs without knowledge of the secret key. Nevertheless, the graph will not highlight the correct key, and the bottom right graph would be empty.

4.2 Test Vector Leakage Assessment (TVLA)

4.2.1 Description of the TVLA

Test Vector Leakage Assessment (TVLA) [22] is a technique used to determine if a device leaks the secret value through power consumption without the necessity to bind to a specific attack. It uses a statistical method called Welch's t-test that determines if two datasets are substantially different or come from comparable distributions. It uses sample mean and variances to evaluate both datasets and establishes the null hypothesis that samples are drawn from a similar distribution [22]. Due to the distinct variances, the degree of freedom(DOF) should be calculated and used in cumulative function to precisely identify the p-value for the test. However, for the sake of simplicity only t-value with threshold $|t| > 4.5$ is used to evaluate leakage. When the t-value is larger than the absolute value of 4.5, it leads to a confidence larger than 0.9999 to reject the null hypothesis [22]. Nevertheless, both sample sets must have at least 500 samples and comparable sizes.

Definition 13. The calculation of t-value used in the Welch's t-test

$$\frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{4.2}$$

where:

μ_0	is sample mean of the first set,
μ_1	is sample mean of the second set,
s_0	is sample variance of the first set,
s_1	is sample variance of the second set,
n_0	is size of the first set and
n_1	is size of the second set.

4.2.2 TVLA methods

There are many methods of TVLA that differ in the selection of the compared datasets. Follows the list of these methods [22].

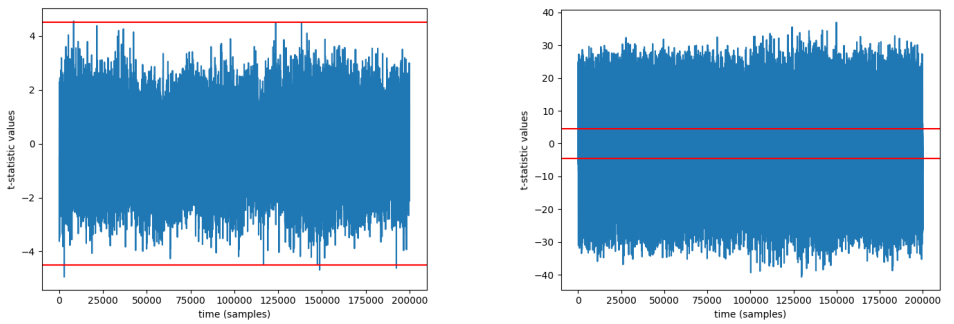
1. Specific - This method depends on the underlying cryptographic algorithm because it uses an intermediate value to evaluate the leakage. Traces are grouped into two datasets based on the result of the intermediate value.
2. Fixed vs random - This approach does not depend on the underlying cryptographic algorithm because it only fixes the associated data used in the attack in one dataset, like nonce or plaintext, whereas they are kept random in the other dataset.
3. Semi-fixed vs random - Extension of fixed vs random method where instead of using specific constant for the associated data, we calculate all possible values that lead to the certain intermediate value. As a result, it depends on the assessed cryptographic implementation.

4.2.3 TVLA on ASCON

I executed TVLA on ASCON to evaluate if nonce leakage is substantial enough to influence the card's power consumption. I used the fix vs random technique of TVLA. Measurements of traces with fixed nonce were interleaved with measurements using a random nonce. Interleaving prevents some device characteristics like value caching that may influence the captured traces [22]. I aligned both traces, which is vital for correct evaluation by TVLA. Otherwise, the TVLA test would not compare samples representing the same operation, making the results useless.

Figure 4.2 shows two graphs. The graph on the left is the baseline, where both trace sets used a random nonce. This graph depicts how the result would look if the value of nonce does not influence the power consumption. On the right graph is the result of TVLA using the fixed vs random method. We can see many occurrences where the t-value is larger than the 4.5 threshold represented by the red lines. Therefore, we reject the null hypothesis that both trace sets come from the same distribution, proving that nonce leaks as expected. Datasets used in the TVLA contained 500 traces in each dataset.

justify why you used this method. simplicity is a good argument :)



(**a**) TVLA only random nonce (**b**) TVLA fixed vs random nonce

Figure 4.2: TVLA between trace sets with random nonce and fixed vs random nonce

4.3 ASCON

4.3.1 Attack description

The main goal of the ASCON CPA attack is to extract the secret key initialised in registers x_1 and x_2 of the internal state S. Attacker knows IV[1](x_0), constant and random nonce (x_3, x_4), which needs to be public to enable decryption of a ciphertext. More details in Chapter 1.4. Attack position needs to be after *Sbox* transformation to avoid linear dependencies between the nonce and the key [3].

Sbox is calculated column-wise, increasing the difficulty of the attack because it needs to be performed bit by bit. Before the *Sbox* calculation is the addition (XOR) of the round constant to the register x_2 that can be easily reverted by repeating the same operation and obtaining the original key. ~~The~~ Definition 14 shows *Sbox* output represented in algebraic normal form for input registers x_0 - x_4 of the internal state S, each having 64 bits [3].

1. IV is abbreviation for Initialisation Vector.

Definition 14. *Sbox* algebraic normal form (ANF) [3]

$$y_0 = x_1 \wedge (x_4 \oplus x_2 \oplus x_0 \oplus 1) \oplus x_3 \oplus x_2 \oplus x_0 \tag{4.3}$$
$$y_1 = (x_3 \oplus 1) \wedge (x_2 \oplus x_1) \oplus x_2 \wedge x_1 \oplus x_4 \oplus x_0 \tag{4.4}$$
$$y_2 = x_4 \wedge (x_3 \oplus 1) \oplus x_2 \oplus x_1 \oplus 1 \tag{4.5}$$
$$y_3 = (x_0 \oplus 1) \wedge (x_4 \oplus x_3) \oplus x_2 \oplus x_1 \oplus x_0 \tag{4.6}$$
$$y_4 = x_1 \wedge (x_4 \oplus x_0 \oplus 1) \oplus x_3 \oplus x_4 \tag{4.7}$$

where:

- $\wedge$ is and logical operation,
- $\oplus$ is XOR logical operation,
- 1 represents 64-bit value with all bits set to 1,
- x_i is register on index i of internal state S and
- y_i is register transformed by *Sbox* on index i of internal state S.

The correlation within CPA attack exploits power consumption changes between traces with a fixed key. Constant values like IV (x_0) and key itself (x_1, x_2) do not influence the power consumption across multiple traces and can be omitted from the evaluation. As a result, we can simplify these equations by removing IV and key parts that do not depend on a random nonce (x_3, x_4) that changes every trace and is used to extract the key [3].

Definition 15. Simplified *Sbox* algebraic normal form (ANF) [3]

$$y_0 = x_1 \wedge x_4 \oplus x_3 \tag{4.8}$$
$$y_1 = x_3 \wedge x_2 \oplus x_3 \wedge x_1 \oplus x_3 \oplus x_4 \tag{4.9}$$
$$y_2 = x_4 \wedge x_3 \oplus x_4 \tag{4.10}$$
$$y_3 = x_0 \wedge x_4 \oplus x_0 \wedge x_3 \oplus x_4 \oplus x_3 \tag{4.11}$$
$$y_4 = x_1 \wedge x_4 \oplus x_3 \oplus x_4 \tag{4.12}$$

From these equations we can learn that to attack the key part x_1 we may use transformed registers y_0 and y_4 due to their dependence

on nonce and the first part of the key (x_1). With knowledge of x_1, we may continue to extract the second part of the key (x_2) using the intermediate register y_1. Other registers do not have a relationship between key and nonce and cannot be used in the attack [3].

We could pinpoint our attack just after the *Sbox*. However, the linear diffusion layer of ASCON improves the attack by combining three key bits into a single bit, thus allowing an attacker to extract three bits of the secret key by attacking just one bit of the intermediate state, consequently accelerating the key extraction.

Definition 16. Bits of the intermediate state after the linear diffusion layer of *Sbox* result. Rotations used in the linear diffusion layer are defined in Table 1.3.

$$z_{0\_i} = y_{0_i} \oplus y_{0\_i+(64-19)} \oplus y_{0\_i+(64-28)} \tag{4.13}$$

$$z_{1\_i} = y_{1_i} \oplus y_{1\_i+(64-61)} \oplus y_{1\_i+(64-39)} \tag{4.14}$$

$$z_{4\_i} = y_{4_i} \oplus y_{4\_i+(64-7)} \oplus y_{4\_i+(61-41)} \tag{4.15}$$

where:

- z_{s_i} is bit on index i of register s after linear diffusion layer,
- y_{s_i} is bit on index i of *Sbox* output in register s.

From the Definition 16 we can compute a single bit of the intermediate state that depends on three bits of the key. Hamming Weight of a single bit is identity. Therefore, using this bit as a power model is viable. I added support to combine multiple bits as Hamming Weight, but it increases the key size that needs to be brute-forced and introduces correlation peaks that do not represent the correct key guess. Therefore, I used only the single bit leakage in the attack, as in the original paper.

4.3.2 Attack implementation

I use the shared functionality of the Generic Attacker for the ASCON CPA attack. The Generic Attacker requires data for the prediction calculation to be represented by a single value for various reasons, like prediction caching, intermediate value storage, conditional averaging, etc. However, in the ASCON case, the data necessary to perform the

attack on a single bit is distributed across the nonce. Furthermore, when attacking the second key part x_2, we also need 3 bits from the first part of the key x_1. For these reasons, I created a compact representation of the required data called a nonce number.

The Figure 4.3 shows a nonce number for one bit of intermediate state. When multiple bits are attacked at once, this representation is just duplicated. The bit of key is ignored when attacking x_1 but is still present to preserve the structure.

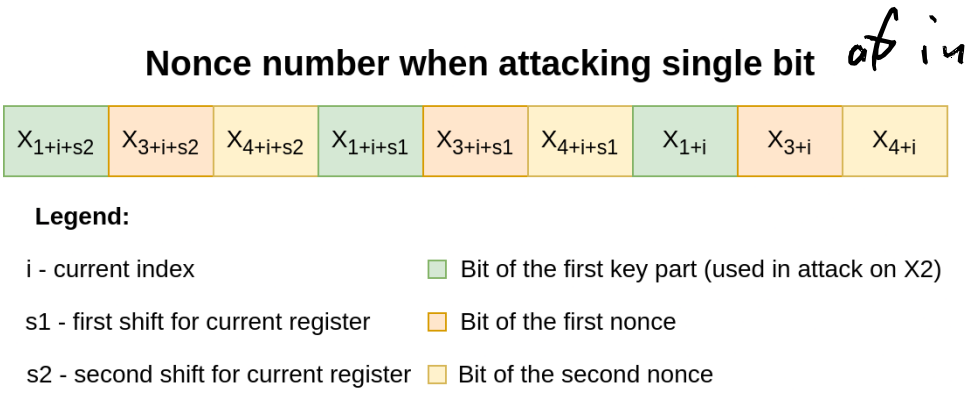


Figure 4.3: Nonce number structure when attacking on one bit

Extracted key bits by a successful attack may overlap between attack stages (bits). These overlaps are caused by different rotation values of corresponding registers that are unevenly distributed. We want to minimise these overlaps and use a minimal set of indexes i necessary to extract all 128 key bits. A suitable approach for discovering these indexes is presented in [3] that removes avoidable overlapping.

Table 4.1 shows minimal sets of indexes for different registers of internal state S. We can see that the best combination to extract the whole key is register z_0 and z_1 with a total number of 58 bits that need to be extracted. Reducing the required brute-force from 2^{128} to just $58 * 2^3 = 464$, which is lower amount than in AES case ($8 * 2^8 = 2048$). However, in contrast with AES, a single bit leakage is more sensitive to signal noise, unaligned traces or large resampling. Furthermore, detecting leakage is much harder than the related brute-force attack.

Table 4.1: List of best column indexes i for key extraction [3]

Register	Indexes	Length
z_0	0, 1, 2, 3, 4, 5, 6, 7, 8, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35	28
z_1	0, 1, 2, 7, 8, 9, 14, 15, 16, 20, 21, 22, 28, 29, 30, 35, 36, 37, 42, 43, 44, 45, 49, 50, 51, 52, 56, 57, 58, 63	30
z_4	0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 16, 17, 18, 20, 21, 22, 24, 26, 34, 37, 42, 46, 50, 51, 52, 53, 54, 55, 56, 58	33

4.3.3 Proof of concept

Verifying the code that implements the ASCON attack using real traces is hard because they contain millions of samples, and the leakage can be hidden by noise even when the code is correct. For these reasons, I implemented a Trace generator that simulates the behaviour of a real trace by using random samples from a normal distribution with a predefined mean and standard deviation. Furthermore, we can configure noise, leakage intensity, sample count, trace count, leakage location, index of leakage bit and random shifting[1] of generated traces.

Leakage is created by calculating the ASCON intermediate state by executing the relevant part of the ASCON-128 AEAD algorithm. The result of the first round of the permutation function p is added to the sample at the specified location within the generated traces, simulating the leakage.

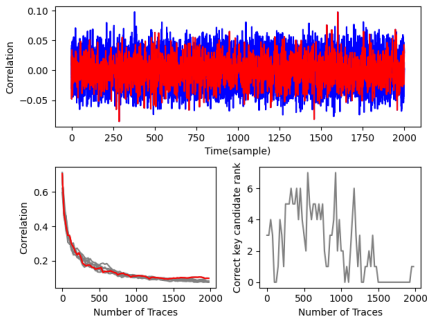
This approach is correct because the power model is the identity of the leakage bit. Therefore, we should be able to detect the leakage and extract the relevant key part if the attack code is correct.

In the Figure 4.4 with two images, we can see that leakage is correctly detected if the noise is reasonable. Otherwise, it can blend with invalid key guesses, and the attack code is unable to detect the leakage.

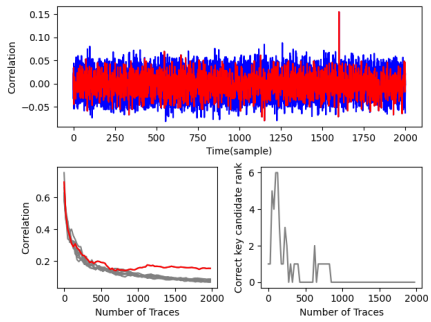
1. Trace shifting can be used to test alignment of traces.

The image on the left is configured with more noise than on the right, using a standard deviation equal to 20, leakage amplification[2] equal to 4 and random noise equal to 3. The image on the right used identical parameters except for the standard deviation, which was set to 10 to simulate less noise.

We can see that there is still a peak at 1600 in the top graph, where the leakage was inserted. However, this leakage is too small to be detected as a successful attack. When we look at the graph on the bottom right[3] of the left image, we can see that at around 1900 processed traces, it even got back in the second place behind the false positive key guess. We can clearly detect the leakage in the image on the right. Therefore, noise can significantly impair the attack and make it unfeasible to detect the leakage.



(**a**) Attack on generated traces with significant noise.



(**b**) Attack on generated traces with a reasonable noise.

Figure 4.4: Comparison of leakage detection when noise rises on the generated traces

4.3.4 Key leakage location

Single power trace of the first round of ASCON permutation function p on our JavaCard takes about 65 million samples to measure on 1.25 GS/s[4]. This amount of samples is expensive to process, and we would

2. Leakage amplification represents the number added to the trace when the intermediate state equals a logical one for the correct key.
3. This graph shows correct key position compared to other key guesses.
4. GS/s is abbreviation for giga samples per second.

like to pinpoint the leakage location more precisely to avoid unnecessary computations and thoroughly analyse the potential leakage location. Patterns in traces are better visible when aggressive resampling is applied.

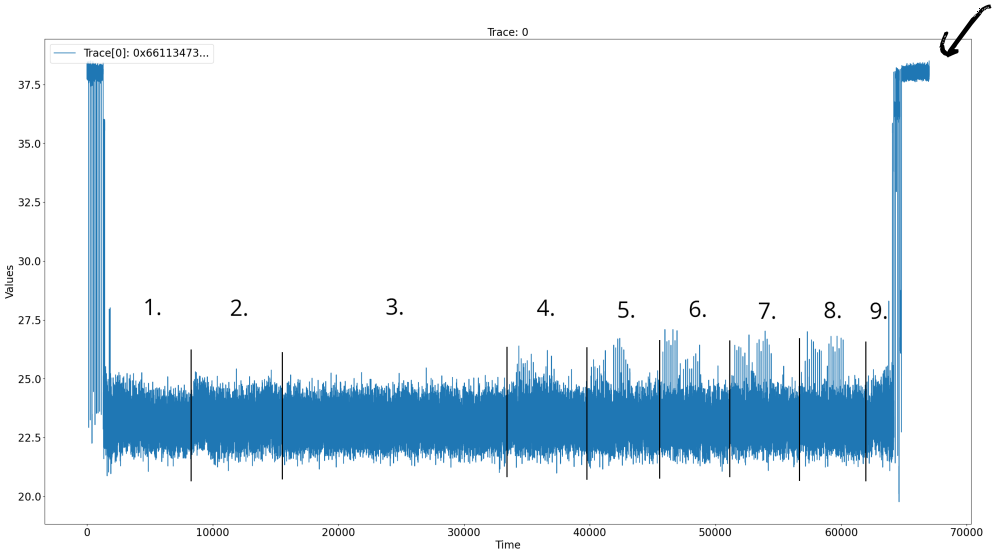


Figure 4.5: Measured trace of first ASCON round with recognized patterns

Figure 4.5 shows samples of one trace during the first round of ASCON. On this trace, resampling with a window of 2000 and an overlap of 1000 was applied, reducing the total size of the samples to only 65 thousand from about 65 million. We can see several patterns that can help us find the leakage location. Here is a description of each segment and its probable underlying computation.

1. Processing APDU packet, padding plaintext and associated data initializing ASCON internal state S and constants.[5]
2. Creating temporary internal state S' copied from S and performing round constant addition.
3. Performing *Sbox* calculation without usage of look-up table.

5. Nonce and plaintext sending is not measured together with card initialization.

4-8. Performing rotation and xor operation on five internal registers (y_0 - y_4) of S' in this order (Linear diffusion layer).
9. Copying temporary internal state S' back to the original registers S. Sending a response from the card and waiting for further instructions.

Leakage established in Chapter 4.3.1 is located after performing one round of the permutation function p on the corresponding internal state register. Therefore, we can see that the leakage is approximately located at the end of the corresponding segment where the rotation by the linear diffusion layer occurred for this register. As a result, we can reduce the attack only to about two million samples for each register. Furthermore, there is another potential leakage of all registers simultaneously at segment 9, where the temporary internal state S' containing leakage bits is copied to the original one. Therefore, we can choose where to perform the power analysis.

4.3.5 Attack results

Detecting a single bit leakage can be significantly impaired by noise within the signal, wrong alignment and inadequate resampling (too aggressive or too small). Therefore, finding the leakage on real traces proves to be challenging in ASCON. Features like automatic alignment, advanced alignment[6] and trace discarding may help find the key bit leakage.

I have managed to recover several bits of the key using the following configuration.

1. Resampling applied to the traces configured with a window of size 10 and step 5. Reducing the total number of samples within the traces by five times.
2. Attack interval and step equal to 15000.
3. Automatic alignment performed every attack interval with alignment interval length equal to 15000 and max shift equal to 7500.
4. Threshold[7] for trace discarding after alignment set to 0.55.
5. Advanced alignment was not used because basic alignment could find patterns even without additional resampling.

6. Advanced alignment is a technique described in Chapter 2.4.3.
7. Minimal correlation between reference trace and the aligned trace.

Figure 4.6 shows one example of successful leakage discovery using the settings above. Leakage represents the first bit of the fourth internal register (z_{4_0}). This leakage occurs at the interval from 61412500 to 61487500, where both types of leakages defined in Chapter 4.3.4 could occur. Note that due to the applied resampling, the interval length was reduced five times to 15000. In this interval, 8000 traces were below the threshold and discarded from the total of 30000. As a result, the attack was performed with about 22 thousand traces.

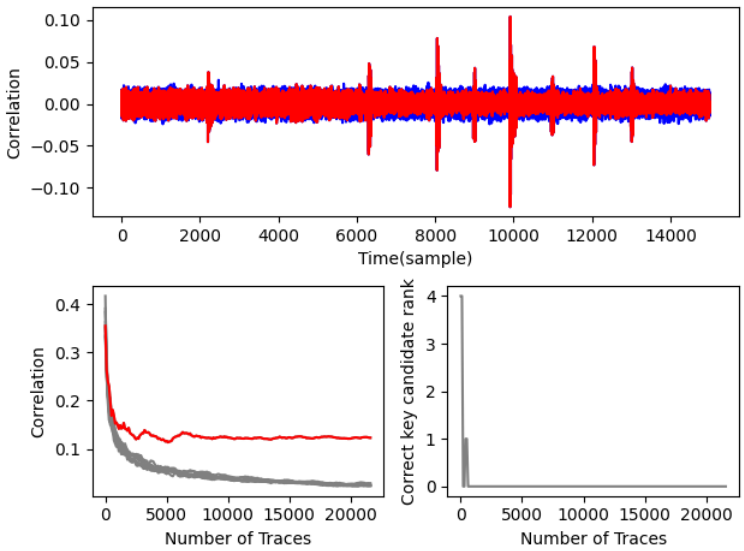


Figure 4.6: Leakage of the ASCON bit z_{4_0}

The top graph shows multiple occurrences of the bit leakage because the card executed many operations that caused the bit to be transmitted over the bus with a maximum correlation equal to 0.123, making it clearly distinguishable as the correct key compared to the average of 0.02 for incorrect key guesses.

4.4 Comparison of CPA attacks on AES and ASCON

The main difference between attacks on AES and ASCON is the power model. The AES power model is the Hamming Weight of one byte of the intermediate register containing the *Sbox* result. Finding this

leakage results in the recovery of one byte of the key. On the other hand, in the ASCON case, it is just one bit of the intermediate state that allows obtaining only three bits of the secret key. A leakage of one bit is harder to find and is much more sensitive to non-optimal alignment and resampling, making it harder to detect. We need to extract at least 58 bits to obtain the whole key, contrasting with AES, where leakage needs to be detected only 16 times for each key byte. Furthermore, six nonce bits, necessary to extract one key bit, are scattered across the nonce and are harder to collect.

Additionally, when attacking the second part of the ASCON key x_2, we must already have extracted the first part of the key x_1 because the attacked intermediate value depends on it. This requirement makes attacking the second part of the key more complicated when we have only partially recovered the first part of the key.

One round of AES relevant to the attack is shorter than one round of ASCON. Therefore, measured traces are larger and more demanding to process. Furthermore, ASCON needs smaller resampling windows to enable detection of the bit leakage, decreasing the reduction of trace length from resampling. In the AES case, resampling can decrease the size of traces up to 100 times or even more, in contrast with 10 times in the ASCON case.

For all these reasons, attacking ASCON with a CPA attack proved to be much more complex than the original attack on AES. This result demonstrates that the ASCON algorithm design is somewhat robust against side-channel attacks, even without any additional protections being applied, and it proves that the designers of ASCON concentrated on making the side-channel analysis harder.

↑ nice analysis, but write what more would be necessary for full attack. It can be vague if one paragraph

4.5 Software countermeasures against CPA attacks

We extracted key bits of AES and ASCON using a CPA attack on power traces and made the extraction with less than 1000 traces in some cases. This success shows that we cannot leave the cryptographic implementation unprotected in the JavaCard when security is concerned, even where algorithm design is mindful of side-channel attacks, like in the ASCON case. However, complete protection against side-channel attacks using only software protections is not possible and usually only

makes the attack execution harder, requiring more traces and more expensive computations on them [7]. Nevertheless, they may discourage the attacker from trying if the attack becomes too expensive.

These protections come at varying costs on the performance of the cryptographic algorithm. Therefore, the balance between the level of protection and usability is crucial.

4.5.1 Hiding

Hiding is one of the two main approaches of protection against side-channel attacks via power analysis. The main goal of hiding is to make the leakage location harder to find and to make aligning traces more challenging. This result is achieved by randomising the computation by unpredictably shuffling the order of operations that can be swapped and randomly introducing operations that do not influence the encryption result [7].

Because randomness is independent of the fixed key, it can be averaged out with more power measurements. As a result, this approach only increases the signal-to-noise ratio and hence the difficulty of the attack [7]. A number of new operations and randomisation possibilities make the calculation slower but increase the complexity of the side-channel attack, achieving a more secure implementation.

this may be already implemented by JavaCard

4.5.2 Masking

Masking is a more complex countermeasure against power consumption analysis. This approach takes the sensitive variable, usually key, and splits it into several shares by introducing random masks $s_1, ..., s_d$ applied to the key. The number of masks d is called a masking order creating $d + 1$ shares [7]. Equation 4.16 shows how the additional share is obtained to make this equation hold $k = s_0 \oplus s_1 \oplus ... \oplus s_d$.

Definition 17. Calculation of the first share s_0 [7]

$$s_0 = k \oplus s_1 \oplus ... \oplus s_d \tag{4.16}$$

where:

- k is the sensitive value,
- s_i is the share with index i,
- s_d is the last share with index equal to the masking order.

Because only the shares created by the random masks are processed by the device the real key does not influence the power consumption directly. Attacking masked implementation is done using higher-order attacks. Attacker must recover all shares that may occur at different times to extract the key.

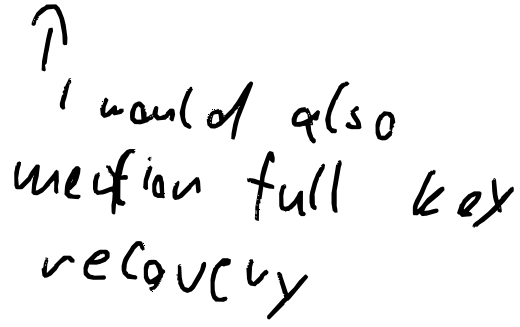
Key masking must not change the computation result. Therefore, some linear and non-linear operations within a cryptographic algorithm need to be adjusted. Linear operations can process the shares separately, then combine the results to gain the correct output. Non-linear operations like $Sbox$ must take a different strategy involving more complex computations out of the scope for this thesis [7].

4.6 Future work

Protected implementations require a substantial number of power measurements and advanced trace processing techniques to mount the attack [7]. Gathering traces and writing the code for the CPA attack on a protected implementation requires considerable time investment.

Time constraints and the extensive scope of the thesis left me with an insufficient amount of time to complete the implementation and evaluation of CPA attacks against protected implementations of ASCON via hiding or masking. This part of the thesis was optional and will be performed in subsequent work.

The follow-up work will include extending the implementation to support attacks on protected implementations of ASCON-128, as well as their execution and comparison with the unprotected version.



5 Conclusion

Work on this thesis started by updating basic scripts that performed a side-channel attack on AES-128 via power consumption using correlation power analysis (CPA). These scripts were provided to me by the CRoCS laboratory. The original scripts were poorly structured, lacked automation, were difficult to read, and suffered from significant performance inefficiencies. I mitigated these issues by substantially improving the performance, configurability, usability, structure, and readability of these scripts. Additionally, I introduced several new features, including automated attack execution, file memory mapping for constant memory usage, logging, more efficient incremental calculation of correlation, and enhanced leakage detection.

I developed a reusable CPA Framework from this improved codebase by abstracting shared functionality into generic objects and scripts. This framework simplifies the implementation of other CPA attacks on various cryptographic algorithms by providing a modular and extensible architecture. I utilised this framework while implementing the CPA attack on the ASCON-128 authenticated encryption algorithm.

I implemented the Test Vector Leakage Assessment (TVLA) technique to assess leakage of nonce on power traces collected from a JavaCard using the fixed vs random approach. The measurement script needed to be updated to support the interleaved fixing of a nonce necessary during TVLA evaluation.

I successfully implemented a state-of-the-art CPA attack on the ASCON-128 authenticated encryption and identified exploitable leakage in the power traces, enabling recovery of the key bits. This key bit recovery demonstrates that the framework and attack implementation are correct and can extract the secret using the leakage of the content of internal registers via power consumption.

At last, I described potential software-based countermeasures to decrease vulnerability against side-channel attacks through power analysis. Evaluating the effectiveness of these countermeasures by implementing and executing the attack on protected implementations and comparing their resilience against the unprotected version is subject to future work.

Bibliography

1. *Lightweight Cryptography* [online]. USA: NIST, 2017-01-03 [visited on 2025-01-02]. Available from: https://csrc.nist.gov/projects/lightweight-cryptography.
2. *Cryptographic competitions: CAESAR submissions* [online]. 2019-02-20. [visited on 2025-02-01]. Available from: https://competitions.cr.yp.to/caesar-submissions.html.
3. WEISSBART, Léo; PICEK, Stjepan. Lightweight but not easy: side-channel analysis of the ascon authenticated cipher on a 32-bit microcontroller. *Cryptology ePrint Archive* [online]. 2023 [visited on 2025-01-10]. Available from: https://eprint.iacr.org/2023/1598.pdf.
4. *Java Card™ Platform* [online]. USA: Oracle, 2017-01-03 [visited on 2025-01-11]. Available from: https://docs.oracle.com/javacard/3.1/related-docs/JCVMS/JCVMS.pdf.
5. CHEN, Zhimin; ZHOU, Yujie. *Cryptographic Hardware and Embedded Systems - CHES 2006*. Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side Channel Leakage. Ed. by GOUBIN, Louis; MATSUI, Mitsuru. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. ISBN 978-3-540-46561-4.
6. CHMIELEWSKI, Łukasz Michał. *Exploiting Horizontal Leakage in Public Key Cryptosystems* [online]. Nijmegen, 2019 [visited on 2025-02-01]. Available from: https://repository.ubn.ru.nl/bitstream/handle/2066/207470/207470.pdf. PhD thesis. Radboud University Nijmegen.
7. PETR SOCHA, Vojtěch Miškovský; NOVOTNÝ, Martin. A Comprehensive Survey on the Non-Invasive Passive Side-Channel Analysis. *Sensors* [online]. 2022, vol. 22, no. 21 [visited on 2025-05-01]. ISSN 1424-8220. Available from DOI: 10.3390/s22218096.
8. STANDAERT, François-Xavier. *Introduction to Side-Channel Attacks* [online]. 2024. [visited on 2025-02-01]. Available from: https://perso.uclouvain.be/fstandae/PUBLIS/42.pdf.

9. *Correlation Power Analysis* [online]. USA: MediaWiki, 2018-05-01 [visited on 2025-01-03]. Available from: http://wiki.newae.com/Correlation_Power_Analysis.

10. *GlobalPlatform Technology, Card Specification, Version 2.3.1* [online]. USA: Oracle, 2018-03-01 [visited on 2025-01-11]. Available from: https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf.

11. *Organization, security and commands for interchange* [online]. USA: ISO/EIC, 2020-05 [visited on 2025-01-03]. Available from: https://www.iso.org/standard/77180.html.

12. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)* [online]. USA: NIST, 2001-11-26 [visited on 2025-01-04]. Available from: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

13. KEYVAN RAMEZANPOUR P. Ampadu, William Diehl. *A Statistical Fault Analysis Methodology for the Ascon Authenticated Cipher* [online]. 2019-05-05. [visited on 2025-01-05]. Available from: https://www.semanticscholar.org/paper/A-Statistical-Fault-Analysis-Methodology-for-the-Ramezanpour-Ampadu/8dd20c014a38e03798d65342ad082c6b69e52f99/figure/1.

14. DOBRAUNIG, Christoph; EICHLSEDER, Maria; MENDEL, Florian; SCHLÄFFER, Martin. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *Journal of Cryptology* [online]. 2021, vol. 34, no. 3, p. 33 [visited on 2025-01-05]. ISSN 1432-1378. Available from DOI: 10.1007/s00145-021-09398-9.

15. *Lightweight Cryptography* [online]. USA: NIST, 2023-02-07 [visited on 2025-01-02]. Available from: https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon.

16. MARIA EICHLSEDER Florian Mendel, Martin Schläffer. *Ascon v1.2: Lightweight Authenticated Encryption and Hashing* [online]. 2019. [visited on 2025-01-05]. Available from: https://www.semanticscholar.org/paper/A-Statistical-Fault-Analysis-Methodology-for-the-Ramezanpour-Ampadu/8dd20c014a38e03798d65342ad082c6b69e52f99/figure/1.

17. SAMWEL, Niels; DAEMEN, Joan. *DPA on hardware implementations of Ascon and Keyak* [online]. 2017. [visited on 2025-01-05]. Available from: https://nielssamwel.nl/papers/cf2017_dpa.pdf.

18. *Inspector Trace Set .trs file support in Python* [online]. Riscure [visited on 2025-01-11]. Available from: https://trsfile.readthedocs.io/en/latest/.

19. *Inspector Side Channel Analysis* [online]. Riscure [visited on 2025-01-11]. Available from: https://www.riscure.com/security-tools/inspector-sca/.

20. FERREIRA, André; LI, Jianning; POMYKALA, Kelsey L.; KLEESIEK, Jens; ALVES, Victor; EGGER, Jan. GAN-based generation of realistic 3D volumetric data: A systematic review and taxonomy. *Medical Image Analysis* [online]. 2024, vol. 93, p. 103100 [visited on 2025-01-15]. ISSN 1361-8415. Available from DOI: https://doi.org/10.1016/j.media.2024.103100.

21. SOCHA, Petr; MISKOVSKY, Vojtech; KUBATOVA, Hana; NOVOTNÝ, Martin. *Optimization of Pearson correlation coefficient calculation for DPA and comparison of different approaches* [online]. 2017-04. [visited on 2025-04-18]. Available from DOI: 10.1109/DDECS.2017.7934563.

22. IT-SECURITY, Horst G¨ortz Institute for; RUHR-UNIVERSIT¨AT BOCHUM, Germany. *Leakage Assessment Methodology: a clear roadmap for side-channel evaluations* [online]. 2015. [visited on 2025-04-01]. Available from: https://eprint.iacr.org/2015/207.pdf.