UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Correlation Power Analysis (CPA) Attack on 128-bit AES Encryption

Project Report

Romain Barbason
romain.barbason@estudiantat.upc.edu

Pablo Chen
pablo.chen@estudiantat.upc.edu

Marco Corduff
marco.corduff@estudiantat.upc.edu

Oihan Irastorza
oihan.irastorza@estudiantat.upc.edu

Leonor Marques
leonor.mata.pessoa.de.carvalho@estudiantat.upc.edu

University: Universitat Politècnica de Catalunya (UPC)
Subject: Hardware Security in Embedded Systems
Degree: Master in Cybersecurity
Academic Year: 2024/2025

May 11, 2025

Contents

1 Project Description

This project simulates a practical Correlation Power Analysis (CPA) attack on an embedded system performing AES-128 encryption. The target platform is a PIC18F4520 microcontroller, a widely used low-power 8-bit MCU found in many industrial and consumer devices. In this simulated attack, a malicious actor gains physical access to the device and attempts to recover the secret AES encryption key by analyzing variations in the device's power consumption during cryptographic operations.

To replicate the conditions of a real-world side-channel attack, the system's power usage is monitored while it encrypts known plaintexts. The microcontroller operates at 250 kHz and is powered by a 5V supply. A 100 Ω resistor is placed in series with the power line to enable measurement of the current drawn by the device. An oscilloscope with a sampling rate of 25 Msamples/s captures the voltage drop across the resistor, producing high-resolution power traces corresponding to the encryption process.

The attack is based on analyzing pre-recorded datasets, which simulate this experimental setup:

- `cleartext.txt`: contains 150 plaintexts (16 bytes each) input into the AES encryption algorithm.
- `trace{N}.txt`: includes 150 power consumption traces (50,000 samples each), captured during operations involving the N^{th} key byte.
- `clock{N}.txt` (if available): contains the corresponding clock signals for each trace, aiding alignment of the power data to specific AES processing steps.

The project was carried out in two distinct stages:

- **Stage 1 – Baseline CPA:** using dataset 1, the CPA attack was applied directly without the need for preprocessing. The traces were sufficiently aligned to correlate hypothetical intermediate values from the AES SBox output with the measured power traces, resulting in successful recovery of all 16 key bytes.
- **Stage 2 – Misalignment Handling:** dataset 2 presented misaligned traces due to inconsistencies in clock signal acquisition. This required additional preprocessing steps, including realignment of traces using the clock signal and statistical correlation techniques before CPA could be effectively applied.

This project demonstrates how a determined attacker, with basic physical access and affordable equipment, can successfully extract secret cryptographic keys from an embedded system using CPA. By analyzing power traces from actual encryption runs, the attack illustrates the feasibility of key recovery under realistic conditions. The results emphasize that even robust cryptographic algorithms like AES can be undermined when hardware implementations lack appropriate side-channel resistance.

2 Data Analysis and Preprocessing

Before applying the CPA attack, the collected datasets were analyzed and prepared.

The first dataset contained 150 known plaintexts and corresponding power consumption traces, each consisting of 50,000 samples. As the traces were already well-aligned with the AES operations, preprocessing was minimal. The only steps required were consistency checks and normalization to ensure clean input for the correlation analysis.

In contrast, the second dataset presented a misalignment issue, likely caused by non-deterministic triggering or inconsistent sampling of the clock signal. To address this, preprocessing focused on analyzing the accompanying clock traces. By applying a basic thresholding technique, active regions in the clock signal corresponding to encryption operations were identified. These segments were then used to realign the power traces across all samples, ensuring that each trace captured the same portion of the AES execution. Once aligned, the traces were ready for the CPA attack.

3 Correlation Power Analysis (CPA) Methodology

The CPA attack relies on the assumption that the device's instantaneous power consumption correlates with certain intermediate values of the cryptographic algorithm. In this case, the attack targeted the output of the SBox operation during the first round of AES encryption.

To conduct the attack, hypothetical intermediate values were generated for each key byte guess by XOR-ing the known plaintext bytes with the candidate key byte and passing the result through the AES SBox. The predicted power consumption was modeled using the Hamming weight of these intermediate values. Pearson correlation coefficients between the predicted power consumption and the actual measured traces were computed, with the maximum correlation identifying the correct key byte.

Mathematically, for each plaintext P_i and key hypothesis k, the intermediate value V_i is computed as:

$$V_i = SBox(P_i \oplus k)$$

and the hypothetical power consumption is estimated as:

$$H_i = \mathrm{HW}(V_i)$$

where HW denotes the Hamming weight function. The correct key guess is the one yielding the highest correlation with the actual measured traces.

4 Implementation

The CPA serial attack was initially implemented in Python for dataset 1, and subsequently ported to Rust to enhance performance. Following this, a parallel version was developed to leverage the computational power of the device on which the program is executed, enabling faster processing and improved scalability.

Note: The code presented in the following sections corresponds to the parallel version of the CPA attack. It closely resembles the serial version, with the main differences being related to parallelization for improved performance.

4.1 Dataset 1

Dataset 1 consists of the following files: `cleartext.txt` and `traceN.txt` (where N ranges from 0 to 15). This dataset is used for a direct CPA attack, where clock alignment is not required.

Setup

The attack is performed by directly comparing the measured power consumption traces to the theoretical power consumption values predicted by the AES operations, with a particular focus on the SBox transformation. Since the traces are already well-aligned with the AES encryption process (see Fig. 2), no additional preprocessing is necessary.

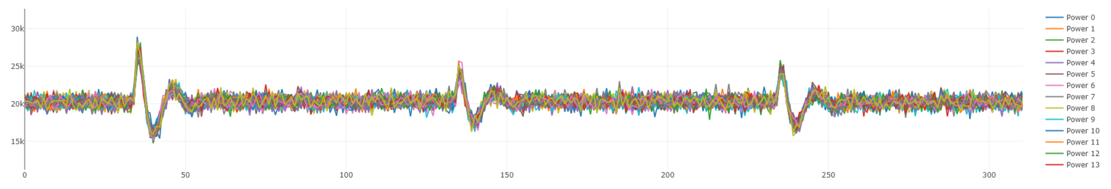


Figure 2: Dataset 1 power usage traces.

Implementation

The CPA attack against dataset 1 is implemented in the `attack_ds1` function, which executes a classical correlation power analysis on unaligned power traces. The approach consists of the following key stages:

1. **Loading the Input Data**
   - The plaintext inputs are read from `cleartext.txt` using `load_clear_text`, resulting in a 150×16 matrix (150 traces, 16 plaintext bytes each).
   - The power traces are loaded from files `trace0.txt` to `trace15.txt` using `load_power_traces`. These are stored in a $16 \times 150 \times 50000$ tensor:
     - 16: AES state bytes (one per trace file),
     - 150: number of traces,
     - 50000: samples per trace.
2. **Hamming Weight Hypothesis**
   - The function `calculate_hamming_weights` computes a hypothetical intermediate value for each key guess (0–255) using the AES S-Box and the formula:

     $$\mathrm{HW}(S(\text{plaintext} \oplus \text{key\_guess}))$$

     where S is the AES S-Box and HW is the Hamming weight.
   - The result is a $16 \times 256 \times 150$ tensor: 16 bytes, 256 key guesses, 150 traces.
3. **Correlation Analysis**
   - The `cpa_attack` function computes the Pearson correlation coefficient between the real power traces and the hypothetical Hamming weights.
   - For each byte, it selects the key guess with the highest correlation across all time samples.
   - Correlation is computed using the function `pearson_correlation`, implemented as:

     $$\rho(X, Y) = \frac{(X - \mu_X) \cdot (Y - \mu_Y)}{\|X - \mu_X\| \cdot \|Y - \mu_Y\|}$$

     where μ_X and μ_Y are the means of X and Y respectively, the numerator is the dot product of the centered vectors, and the denominator is the product of their Euclidean norms. This yields the absolute Pearson correlation coefficient, indicating the strength of linear dependence between the hypothetical power consumption (based on Hamming weight) and the actual measured trace.
4. **Result Verification**
   - The output key is printed and its byte sum is compared against the known correct sum (1712) using an assertion:

     $$\texttt{assert\_eq!(}\sum \texttt{key, 1712)}$$

   - The total execution time is measured for performance benchmarking.

4.2 Dataset 2

Dataset 2 consists of the following files: `cleartext.txt`, `clockN.txt` (where N ranges from 0 to 5000), and `traceN.txt` (where N ranges from 0 to 15). This dataset is used for a direct Correlation Power Analysis (CPA) attack, but with the added requirement of clock alignment.

Setup

The power consumption traces in dataset 2 are misaligned due to inconsistencies in the clock signal acquisition. To address this, the traces must first be pre-processed by extracting and analyzing the clock signals from the `clockN.txt` files. Fig. 3 illustrates the power traces of the first 5 bytes, while Fig. 4 illustrates the first 5 power traces along with their respective clock traces.

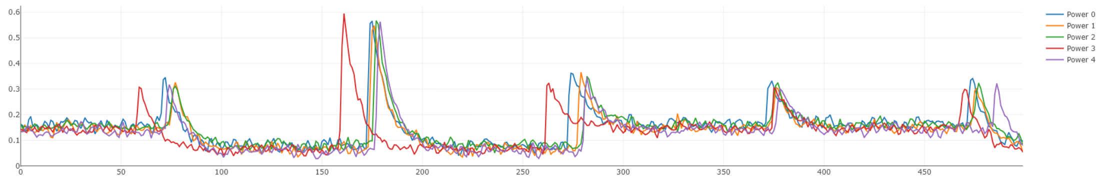


Figure 3: Representation of the first 5 power traces.

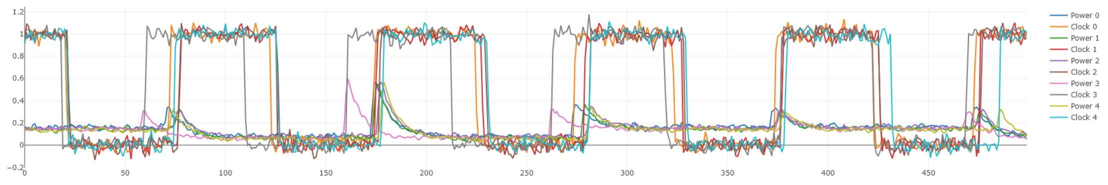


Figure 4: Representation of the first 5 power traces with their clocks.

Resampling

To align the traces, a resampling was done on the original traces using their clocks and taking a clock as reference. The following steps were followed to align the traces:

1. Choose a clock C_{ref} from the clocks as a reference.
2. Find all the rising edges for C_{ref}, $RE(C_{\text{ref}})$.
3. Take the target clock C_{target} and calculate the offsets for each rising edge, $\text{offsets}[i] = RE(C_{\text{target}})[i] - RE(C_{\text{ref}})[i]$. This will calculate how many steps each clock must take to be delayed or advanced.
4. Choose a window size for the resampling; a bigger window will capture more details but will take more time to perform the attack, and a smaller window will take less time but there is a chance that it will not retrieve the correct key.
5. Create a new sample `original_sample[resample_idx + offset[re_idx]]` by taking `window_size` samples if `resample_idx` is equal to $\text{RE}(C_{\text{ref}})[\text{re\_idx}]$ and 0 otherwise; `re_idx` is the rising edge index.

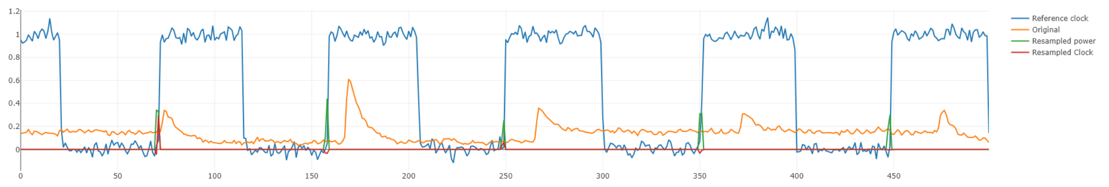


Figure 5: Synchronization of power and clock traces of the second dataset with a window size of 2 samples. The power trace (in green) is almost flat even at the rising edges of the clocks.

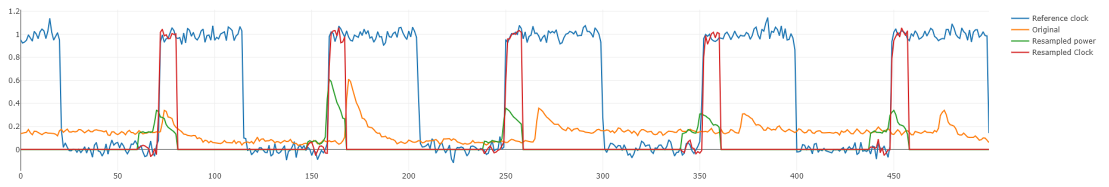


Figure 6: Synchronization of power and clock traces of the second dataset with a window size of 20 samples. The power trace (in green) is more faithful to the original sample.

Window size

A larger window size captures more details of the original sample, as seen in Fig. 6, but also increases the computation time. In contrast, a smaller window size captures fewer details of the trace, as seen in Fig. 5, resulting in faster execution due to many zero samples; however, this may lead to incorrect key recovery. An optimal window size strikes a balance between capturing sufficient signal information and maintaining reasonable runtime, ensuring both speed and successful key recovery.

Implementation

The attack on dataset 2 builds upon the same structure used for dataset 1, with modifications to handle misaligned traces. The attack is implemented in the `attack_ds2` function and proceeds as follows:

1. **Loading the Input Data**
   - Plaintexts are loaded from `cleartexts.txt`, resulting in a 150×16 matrix.
   - Power traces are loaded from `tracesN.txt` using `load_power_traces`, yielding a $16 \times 150 \times 50000$ matrix.
   - Clock traces are loaded from `clocksN.txt` using `load_clocks`, yielding a $16 \times 150 \times 50000$ matrix.
2. **Trace Realignment via Resampling**
   - Due to misalignment introduced by jitter, traces must be realigned before correlation.
   - The function `find_edges` is used to detect rising edges in the clock signal.
   - A reference clock trace (`clocks[0][0]`) is selected, and the function `arg_of_ones` extracts the indices of its rising edges.
   - Each trace is resampled using the `resample` function, which locally shifts the trace to match the reference edge positions using fixed-size windows.
   - The output is a new set of traces with reduced timing variability, suitable for CPA.
3. **Hamming Weight Hypothesis**

- Identical to dataset 1, using:

$$\text{HW}(S(\text{plaintext} \oplus \text{key\_guess}))$$

- Resulting in a $16 \times 256 \times 150$ tensor of hypothetical power consumption values.

4. **Correlation Analysis**
   - The `cpa_attack` method is applied to the resampled traces.
   - Correlation is computed using the function `pearson_correlation`, also identical to dataset 1:
   $$\rho(X,Y) = \frac{(X-\mu_X)\cdot(Y-\mu_Y)}{\|X-\mu_X\|\cdot\|Y-\mu_Y\|}$$
   - The key guess with the highest correlation is selected for each byte.
5. **Result Verification**
   - The recovered key is printed and verified against the known byte sum (1434):
   $$\texttt{assert\_eq!(}\textstyle\sum\texttt{key, 1434)}$$
   - Execution time is measured for performance comparison with dataset 1.

Parallelization Strategy

The CPA analysis is parallelized to reduce runtime, particularly in the `cpa_attack` function, where most of the execution time is spent. This function involves three nested loops: over key byte positions (16), key guesses (256), and trace samples (50000). All of these are independent and amenable to parallelization.

- Parallelization is implemented using Rust's `rayon` crate, which provides efficient data-parallel iterators via `par_iter`.
- Key byte positions and key guesses are processed concurrently, maximizing CPU utilization.
- Independent correlation computations enable parallel execution with minimal overhead or synchronization.
- This results in significant speedups (see section 5), especially with large traces or datasets.

Additional strategies for improving parallel performance are discussed in section 7.

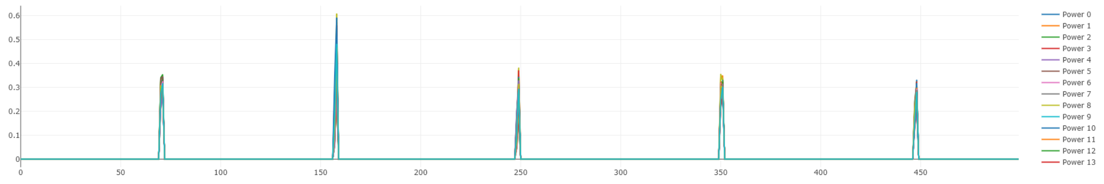


Figure 7: Resampling with a window size of 2.

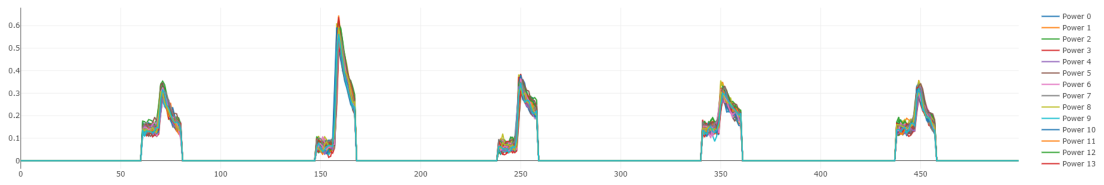
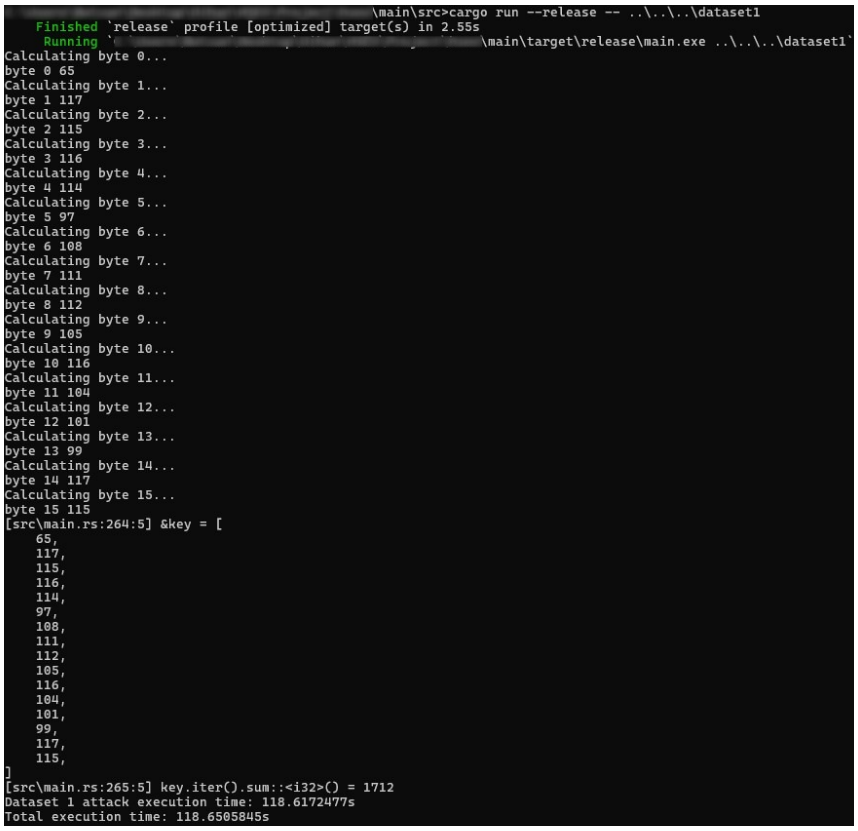


Figure 8: Resampling with a window size of 20.

5 Results

5.1 Dataset 1

The output from the serial execution of the program on dataset 1 is shown in Fig. 9.



```
[redacted]\main\src>cargo run --release -- ..\..\..\dataset1
    Finished `release` profile [optimized] target(s) in 2.55s
     Running `[redacted]\main\target\release\main.exe ..\..\..\dataset1`
Calculating byte 0...
byte 0 65
Calculating byte 1...
byte 1 117
Calculating byte 2...
byte 2 115
Calculating byte 3...
byte 3 116
Calculating byte 4...
byte 4 114
Calculating byte 5...
byte 5 97
Calculating byte 6...
byte 6 108
Calculating byte 7...
byte 7 111
Calculating byte 8...
byte 8 112
Calculating byte 9...
byte 9 105
Calculating byte 10...
byte 10 116
Calculating byte 11...
byte 11 104
Calculating byte 12...
byte 12 101
Calculating byte 13...
byte 13 99
Calculating byte 14...
byte 14 117
Calculating byte 15...
byte 15 115
[src\main.rs:264:5] &key = [
    65,
    117,
    115,
    116,
    114,
    97,
    108,
    111,
    112,
    105,
    116,
    104,
    101,
    99,
    117,
    115,
]
[src\main.rs:265:5] key.iter().sum::<i32>() = 1712
Dataset 1 attack execution time: 118.6172477s
Total execution time: 118.6505845s
```

Figure 9: Serial program execution for dataset 1.

Key Retrieval

The following key bytes were recovered:

0x41, 0x75, 0x73, 0x74, 0x72, 0x61, 0x6C, 0x6F, 0x70, 0x69, 0x74, 0x68, 0x65, 0x63, 0x75, 0x73

Their sum is:

$$1712$$

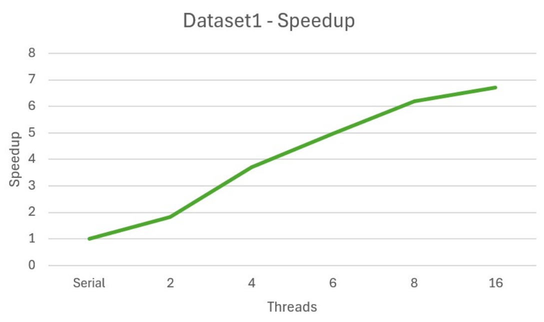
If we translate these bytes to ASCII text, we see that the used key was **"Australopithecus"**.

Performance Metrics

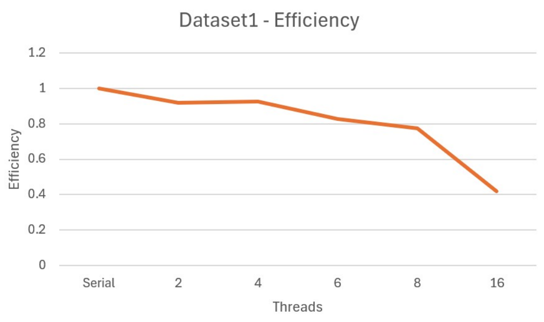
Table 1 summarizes the performance of the CPA attack on Dataset 1 using different thread configurations.

Dataset 1			
	Time (s)	**Speedup**	**Efficiency**
Serial	117.36	1	1
2	63.89	1.84	0.92
4	31.63	3.71	0.93
6	23.63	4.97	0.83
8	18.93	6.2	0.77
16	17.52	6.7	0.42

Table 1: CPA program execution summary for dataset 1.



(a) CPA attack for dataset 1 - Speedup



(b) CPA attack for dataset 1 - Efficiency

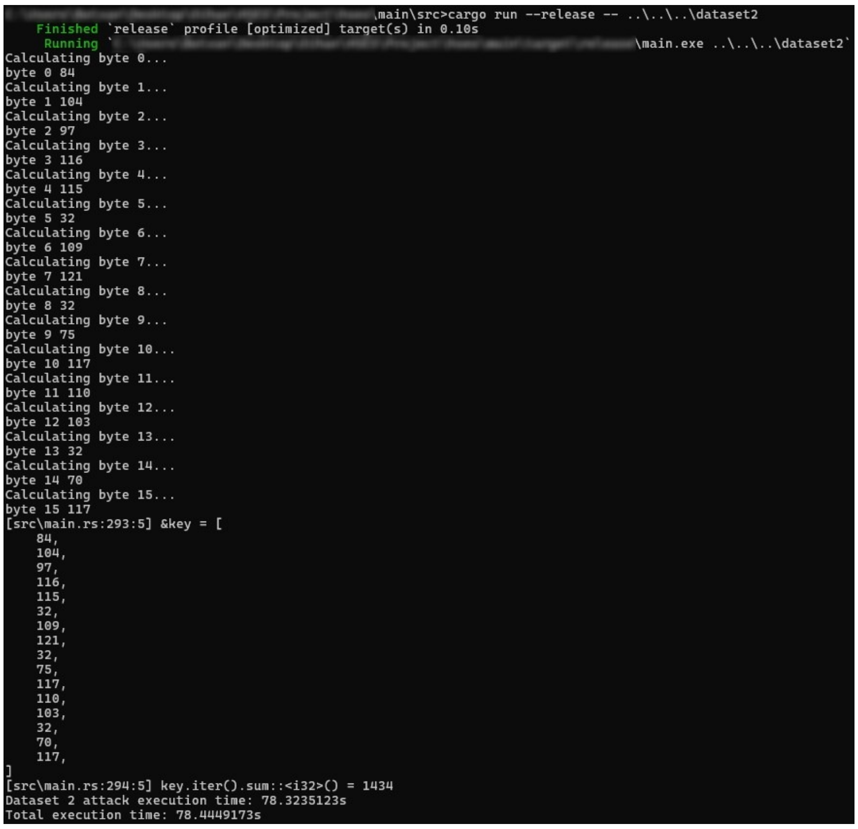
Figure 10: Performance metrics for dataset 1

The speedup plot of the first dataset, as seen in Fig. 10a, exhibits a steady increase initially, but begins to show diminishing returns beyond 6 threads. The curve noticeably flattens after this point, and the gain from 8 to 16 threads is minimal—only about 0.3 times.

Correspondingly, the efficiency plot of the same dataset, shown in Fig. 10b, remains high up to 6 threads, peaking at 0.93 with 4 threads, but drops sharply to 0.42 at 16 threads. This significant decline suggests increasing parallel overhead and inefficient thread utilization at higher core counts.

5.2 Dataset 2

The output from the serial execution of the program on dataset 2 is shown in Fig. 11.



```
[redacted]\main\src>cargo run --release -- ..\..\..\dataset2
    Finished `release` profile [optimized] target(s) in 0.10s
     Running `[redacted]\main.exe ..\..\..\dataset2`
Calculating byte 0...
byte 0 84
Calculating byte 1...
byte 1 104
Calculating byte 2...
byte 2 97
Calculating byte 3...
byte 3 116
Calculating byte 4...
byte 4 115
Calculating byte 5...
byte 5 32
Calculating byte 6...
byte 6 109
Calculating byte 7...
byte 7 121
Calculating byte 8...
byte 8 32
Calculating byte 9...
byte 9 75
Calculating byte 10...
byte 10 117
Calculating byte 11...
byte 11 110
Calculating byte 12...
byte 12 103
Calculating byte 13...
byte 13 32
Calculating byte 14...
byte 14 70
Calculating byte 15...
byte 15 117
[src\main.rs:293:5] &key = [
    84,
    104,
    97,
    116,
    115,
    32,
    109,
    121,
    32,
    75,
    117,
    110,
    103,
    32,
    70,
    117,
]
[src\main.rs:294:5] key.iter().sum::<i32>() = 1434
Dataset 2 attack execution time: 78.3235123s
Total execution time: 78.4449173s
```

Figure 11: Serial program execution for dataset 2.

Key Retrieval

The following key bytes were recovered:

$$0x54, 0x68, 0x61, 0x74, 0x73, 0x20, 0x6D, 0x79, 0x20, 0x4B, 0x75, 0x6E, 0x67, 0x20, 0x46, 0x75$$

Their sum is:

$$1434$$

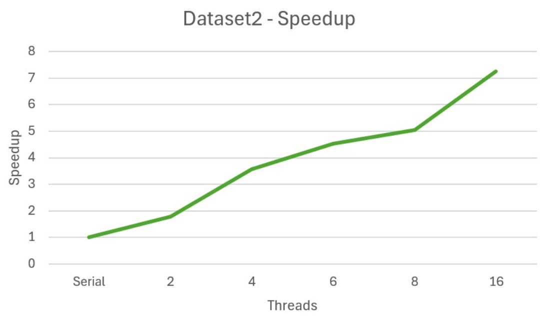
If we translate these bytes to ASCII text, we see that the used key was **"Thats my Kung Fu"**.

Performance Metrics

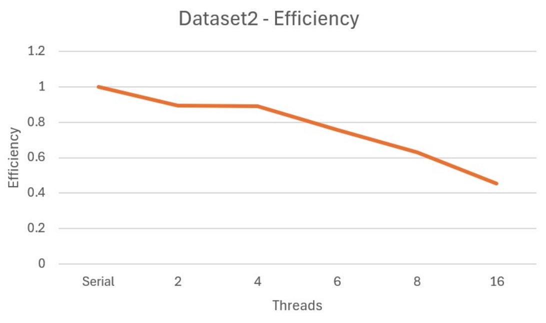
Table 2 summarizes the performance of the CPA attack on dataset 2 using different thread configurations.

dataset 2			
	Time (s)	**Speedup**	**Efficiency**
Serial	62.86	1	1
2	35.19	1.79	0.89
4	17.63	3.57	0.89
6	13.85	4.54	0.76
8	12.44	5.05	0.63
16	8.67	7.25	0.45

Table 2: CPA program execution summary for dataset 2.



(a) CPA attack for dataset 2 - Speedup



(b) CPA attack for dataset 2 - Efficiency

Figure 12: Performance metrics for dataset 2

The speedup plot of the second dataset, seen in Fig. 12a, demonstrates stronger scaling performance, maintaining an almost linear speedup up to 8 threads. It ultimately reaches a higher overall speedup at 16 threads—7.25 times better compared to 6.7 times better in dataset 1—indicating more effective parallelization.

The efficiency plot, shown in Fig. 12b, follows a similar trend to dataset 1 but with a more stable profile. Efficiency remains consistently high through mid-range thread counts and declines more gradually, reaching 0.45 at 16 threads. This smoother drop suggests better workload balance and lower overhead, even as parallelism increases.

6 Challenges Encountered

The most significant challenge in this project was aligning the power traces in Dataset 2. Initially, we attempted to use the first rising edge of the first trace as a reference point and align all other traces by shifting them to match this edge. However, this approach proved ineffective. The reason is that the traces are not uniformly misaligned—each trace can exhibit variations in clock offset due to jitter, noise, or slight inconsistencies in the acquisition process. As a result, using a single global reference point led to poor alignment across the dataset. We initially misattributed the misalignment to incorrect choices of window size or initial offset, which caused us to spend considerable time fine-tuning parameters that ultimately had little impact.

Another key challenge was the initial performance of the serial implementation in Python. Although the logic of the CPA attack was correctly implemented, the execution time was impractically long—taking several hours to complete. This performance bottleneck was likely due to Python's interpreted nature, lack of low-level memory control, and limitations in handling large numerical computations efficiently without specialized libraries. To address this, we reimplemented the attack logic in Rust, which provided much better performance thanks to its compiled execution, strong optimization capabilities, and efficient memory management. The result was a substantial speedup, reducing runtime from hours to just minutes.

7 Future work

A greater speedup could be achieved by removing the zeros from the resampled trace, reducing the inner loop from 50,000 iterations to just 500, corresponding to the number of rising edges in the clock. This would result in a significantly faster execution time than what we achieved with the current implementation. However, implementing this improvement is challenging, as the number of rising edges may vary across different clocks.

Also, the performance on the first dataset could be further enhanced; however, since we do not have the clocks, we must find the rising edges using only the power traces. This can be achieved by using a simple thresholding or more robust methods such as windowing with a dynamic threshold.

Regarding parallelization, our implementation relies on Rust's `rayon` crate to automatically handle data-parallelism. However, more fine-grained manual parallelization could be explored. Since the CPA attack operates independently on each key byte—meaning there are no shared variables between computations—these byte-wise tasks could be explicitly divided among a number of threads that evenly divide the 16 key bytes (e.g., 2, 4, 8, or 16 threads). Each thread would process an independent and balanced subset of bytes, enabling a theoretically ideal linear speedup. Nonetheless, this level of manual control was considered beyond the scope of the project, so we opted for a more straightforward approach using `rayon`.

8 Conclusions

This project successfully demonstrated the vulnerability of AES-128 encryption implemented on embedded systems to CPA attacks. By analyzing power consumption traces, we recovered the complete secret encryption key from both aligned and misaligned datasets using relatively simple equipment.

Our key findings include:

1. **Attack Effectiveness:** The CPA methodology successfully recovered all 16 key bytes from both datasets, confirming that strong algorithms can be compromised through side-channel leakage.
2. **Alignment Challenges:** The misaligned traces in Dataset 2 required additional preprocessing through clock-based resampling, with window size selection proving crucial for balancing accuracy and performance.
3. **Performance Optimization:** Reimplementing the attack in Rust with parallelization yielded significant speedups—6.7x for Dataset 1 and 7.25x for Dataset 2 with 16 threads—though efficiency declined notably beyond 6-8 threads.

These results underscore the critical need for implementing countermeasures against power analysis attacks in security-sensitive embedded systems. Techniques such as random delays, constant-time operations, masking, and secure hardware elements should be considered essential components of cryptographic implementations rather than optional additions.