Analysis of a simple AES dataset Report

Gabriele Tam

January 23, 2025

Contents

1 Introduction

Differential Power Analysis attacks are those that leverage information on power consumption by a cryptographic device and its relation to data in order to extract sensitive information, such as encryption keys. In such side-channel attacks, the use of statistical methods is made to relate observable power traces to predicted leakage models, thereby deducing the secret keys of attackers without actual tampering of the cryptographic implementation.

This report investigates the application of DPA in cryptographic implementations, particularly the work focusing on Advanced Encryption Standard AES. This report is supported through a demo script provided to capture key components to both implement and simulate a DPA attack. This research seeks to answer the following questions:

- **Single-Bit Leakage Model:** Modify the script to attack bits instead of bytes by using a single-bit leakage model. It should include the implementation of both correlation and difference of means distinguishers. Investigate whether the use of different distinguishers impact the attack results (Section 3).
- **SubBytes Input Attack**: The script has to be extended such that an attack against the input of the SubBytes operation is performed applying a chosen distinguisher. The results in terms of attack outcomes attacking the output of the SubBytes operation have to be analyzed (Section 4).
- **Trace Requirements**: Based on the best attack strategy found, a minimum number of traces for reliably recovering the AES key with reasonable enumeration effort has to be estimated (Section 5).

For transparency and reproducibility purposes, additional files including scripts used in the project are available in a GitHub repository, accessible at the following link: https://github.com/h0r0x/Analysis-of-a-simple-AES-dataset.

The main directory structure of the repository is detailed below:

Directory Structure on Github

```
🟦 task1
 └─⬛ report_Task1_Tam_Gabriele.pdf # This file (delivered)
 └─🟥 code # Scripts used for the implementation and simulation
      of the DPA attacks
      └─🟩 attack_bit
          └─🟧 ...  # Scripts and results for the Single-Bit Leakage
                Model attack
      └─🟩 attack_SubBytes
          └─🟧 ...  # Scripts and results for the SubBytes input
                attack
      └─🟩 best_attack
          └─🟧 ...  # Scripts and results for the SubBytes output
                attack
      └─🟩 comparison
          └─🟧 ...  # Contains data and analysis identifying the
                best attack strategy and trace requirements
```

Please refer to the Github repository for details on the additional files.

2 Analysis of the Given Demo Script

The given demo script has three major files:

- `AESAttack.py`: This is where the core logics for the DPA attack are implemented.
- `TRS_Reader.py`: This provides utilities for reading power trace files in the TRS format, which includes plaintext and ciphertext data.
- `TinyAES_625Samples_FirstRoundSbox.trs`: This contains the power traces and associate data used for the attack.

Each of these files plays a certain role in realizing the attack. In the following, I look at their contents and workings in detail, with special consideration of `AESAttack.py`.

2.1 `TRS_Reader.py`

The `TRS_Reader.py` is an auxiliary script that reads in and handles TRS files. It can be used to do the following:

- `read_header()`: Reads metadata from the TRS file, including the number of traces, samples per trace, cryptographic data length and data type (integer orfloating point).
- `read_traces()`: Reads the power traces as well as plaintext and ciphertext data from the TRSfile into NumPy arrays ready for analysis.
- `read_onesample()` and `read_plainciphertext()`: These functions offer specialized utilities that can be used to extract individual samples or the plaintext/ciphertext data.

The file performs the necessary processing to handle large trace sets efficiently, providing the essential input to the attack in `AESAttack.py`.

2.2 `TinyAES_625Samples_FirstRoundSbox.trs`

This binary file contains:

- **Power Traces**: Measured power consumption data during AES encryption for 625 samples per trace.
- **Plaintext and Ciphertext**: Cryptographic input and output data associated with each trace.

The trace set is used to correlate hypothesized intermediate values with actual power consumption to recover the AES key.

2.3 `AESAttack.py`

The `AESAttack.py` file is the core of the provided script, implementing the Differential Power Analysis (DPA) attack logic. Below, I detail its key functionalities, supported by relevant code snippets for clarity.

2.3.1 Key Functions in `AESAttack.py`

The main functions in `AESAttack.py` deal with the operation of the leakage model and the attack itself. Here are the critical components:

`Sbox(X)` Applies the AES S-Box transformation to each element of the input array `X`. There is a global array holding the S-Box.

S-Box Transformation Function

```
def Sbox(self, X):
    y = np.zeros(len(X)).astype(int)
    for i in range(len(X)):
        y[i] = sbox[X[i]]
    return y
```

Description: This function converts plaintext bytes to their S-Box output, a key intermediate step in AES.

`ADK(X, k)` Implements the AddRoundKey operation by XORing the plaintext `X` with a key guess `k`.

AddRoundKey Function

```
def ADK(self, X, k):
    y = np.zeros(len(X)).astype(int)
    for i in range(len(X)):
        y[i] = X[i] ^ k
    return y
```

Description: This function generates hypotheses for intermediate values based on all possible key guesses.

`HW(X)` Simulates the leakage model by calculating the Hamming weight of each input value.

Hamming Weight Calculation Function

```
def HW(self, X):
    y = np.zeros(len(X)).astype(int)
    for i in range(len(X)):
        y[i] = bin(X[i]).count("1")
    return y
```

Description: The Hamming weight here is used an approximation of power consumption.

`maxCorr(hw, traces)` Computes the maximum Pearson correlation coefficient between the hypothesized Hamming weights (`hw`) and the power traces.

Maximum Correlation Function

```
def maxCorr(self, hw, traces):
    maxcorr = 0
    corr = np.zeros(trs.number_of_samples)
    for i in range(trs.number_of_samples):
        [corr[i], pv] = pearsonr(hw, traces[:, i])
        if abs(corr[i]) > maxcorr:
            maxcorr = abs(corr[i])
    return [maxcorr, corr]
```

Description: This function finds the most likely key guess based on the highest correlation.

`Initialise(N)` Prepares the hypotheses for all possible key bytes and key guesses.

Initialization Function

```
def Initialise(self, N):
    trs = TRS_Reader.TRS_Reader("TinyAES_625Samples_FirstRoundSbox.trs")
    trs.read_header()
    trs.read_traces(N, 0, trs.number_of_samples)
    HWguess = np.zeros((16, 256, N)).astype(int)
    for byteno in range(16):
        X = trs.plaintext[:, byteno]
        for kg in range(256):
            Y = AESAttack().ADK(X, kg)
            Y = AESAttack().Sbox(Y)
            HWguess[byteno, kg] = AESAttack().HW(Y)
    return [trs, HWguess]
```

Description: This function reads the TRS file and calculates the Hamming weights for all hypotheses.

`corrAttack(trs, ax, byteno, Nm)` Performs the correlation attack for a specific key byte and visualizes the results.

Correlation Attack Function

```python
def corrAttack(self, trs, ax, byteno, Nm):
    ax.clear()
    maxkg = 0
    maxcorr_k = 0
    for kg in range(256):
        hw = HWguess[byteno, kg]
        [maxcorr, corr] = AESAttack().maxCorr(hw[0:Nm], trs.traces[0:Nm,
        ↪  :])
        if maxcorr > maxcorr_k:
            maxkg = kg
            maxcorr_k = maxcorr
        if kg == key[byteno]:
            ax.plot(corr, 'r-', alpha=1)
        else:
            ax.plot(corr, color=(0.8, 0.8, 0.8), alpha=0.8)
    ax.set_xlim([1, trs.number_of_samples])
    ax.set_ylim([-1, 1])
    ax.title.set_text('Byte {0}=0x{1:2x}'.format(byteno, maxkg))
    ax.set_xlabel('Samples')
    ax.set_ylabel(r'$\rho$')
    return maxkg
```

Description: This function plots the correlation for each key guess, and in red highlights the correct key.

2.3.2 Workflow in `AESAttack.py`

The workflow involves the following steps:

1. Read the TRS file and initialize power traces and plaintext using `Initialise()`.
2. Generate hypotheses for all key bytes and key guesses.
3. Compute the Pearson correlation for each hypothesis using `maxCorr()`.
4. Plot the results for each key byte using `corrAttack()`.

2.3.3 Example Output

The script generates the correlation plots for all 16 AES key bytes. The key guess with the highest correlation usually gives the correct one, as can be seen from the red curve in the plots.

2.4 Conclusion

The `AESAttack.py` script s a good basis for DPA attacks, as it demonstrates how well correlation-based distinguishers and the Hamming weight leakage model work. Its modular design is easy to use to try different leakage models, distinguishers and intermediate AES states.

3 Single-Bit Attack with Multiple Distinguishers

In this section, I describe the modifications made to perform a single-bit attack instead of a byte-wise attack, using two distinguishers: correlation and the difference of means. I explain the rationale behind the changes, provide relevant code snippets, and discuss the results of the analysis.

3.1 Script Modifications

To adapt the attack script for a single-bit leakage model and integrate two distinguishers, I implemented the following changes:

3.1.1 Single-Bit Leakage Model

The original attack used a Hamming weight leakage model on bytes. For the single-bit model, I modified the script to extract specific bits from the output of the AES S-Box.

Single-Bit Extraction Function

```python
def SB(self, X, bit_index):
    """Apply S-box and extract bit 'bit_index'."""
    y = np.zeros(len(X), dtype=int)
    for i in range(len(X)):
        sbox_value = self.sbox[X[i]]
        y[i] = (sbox_value >> bit_index) & 1
    return y
```

Description: This function processes each byte, applies the AES S-Box, and extracts the desired bit based on the `bit_index` parameter.

3.1.2 Difference of Means Distinguisher

The difference of means distinguisher evaluates the statistical separation between two groups of traces corresponding to bit values 0 and 1. It computes the *T-statistic* to measure the strength of this separation.

Difference of Means Implementation

```python
def diffOfMeans(self, bit_values, traces):
    """Perform a two-sample difference of means test."""
    group0 = traces[bit_values == 0, :]
    group1 = traces[bit_values == 1, :]

    mean0 = np.mean(group0, axis=0)
    mean1 = np.mean(group1, axis=0)
    se = np.sqrt(np.var(group0, axis=0) / len(group0) + np.var(group1,
    ↪  axis=0) / len(group1))
    t_stats = (mean1 - mean0) / se
    return np.max(np.abs(t_stats)), t_stats
```

Description: This function calculates the mean and standard error for two groups of traces (bit values 0 and 1) and returns the *T-statistic*, which quantifies the separation between the groups.

T-Test and its Role The T-Test is a statistical method used to evaluate the significance of the difference between two groups. In this context, it compares the mean power consumption of traces corresponding to bit values 0 and 1:

$$T = \frac{\bar{X}_1 - \bar{X}_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_0^2}{n_0}}}$$

where:

- $\bar{X}_1, \bar{X}_0$: Mean power consumption for groups 1 and 0.
- s_1^2, s_0^2: Variance of power consumption for groups 1 and 0.
- n_1, n_0: Number of traces in each group.

Significance in this Context:

- **High $|T|$:** Indicates a strong correlation between the guessed bit and the power trace, increasing confidence in the key guess.
- **Low $|T|$:** Suggests weaker or no correlation, implying the guessed bit is likely incorrect.

Weighted Key Guess vs. Most Frequent Key Guess The distinguisher results are processed using two approaches:

- **Weighted Key Guess:** Cumulative T-statistics across all bits are used to rank the key guesses. The key guess with the highest cumulative score is selected.
- **Most Frequent Key Guess:** For each bit, the key guess with the highest T-statistic is identified. The key guess appearing most frequently across all bits is selected.

Key Guess Implementation

```
# Weighted Key Guess
for (kg, score) in guesses_scores:
    byte_key_scores[byteno, kg] += score

# Most Frequent Key Guess
best_kg_for_bit = guesses_scores[0][0]
byte_key_counts[byteno, best_kg_for_bit] += 1
```

Description: The code above demonstrates how weighted and most frequent key guesses are calculated. Weighted guesses accumulate scores, while frequent guesses focus on bit-level dominance.

3.1.3 Correlation-Based Distinguisher

The correlation distinguisher (CPA) calculates the Pearson correlation coefficient between the guessed bit values and the observed power traces. This metric quantifies the linear relationship between these two variables, identifying the key guess that maximizes the correlation.

Correlation-Based Distinguisher

```python
def maxCorrBit(self, bit_values, traces):
    """
    Compute Pearson correlation with every sample,
    returning the maximum absolute correlation.
    """
    maxcorr = 0
    n_samples = traces.shape[1]
    for i in range(n_samples):
        corr, _ = pearsonr(bit_values, traces[:, i])
        if abs(corr) > maxcorr:
            maxcorr = abs(corr)
    return maxcorr
```

Description: The function runs over all samples in the trace and computes for every sample the Pearson correlation coefficient between the guessed bit values and the actual trace values. It returns the maximum absolute correlation, which denotes the strength of the linear relationship.

Theoretical Background The Pearson correlation coefficient, r, is given by:

$$r = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum(X_i - \bar{X})^2 \sum(Y_i - \bar{Y})^2}}$$

where:

- X_i: Guessed bit values.
- Y_i: Observed power trace values.
- $\bar{X}, \bar{Y}$: Means of X and Y, respectively.

A high absolute value of r indicates a strong linear relationship between X and Y.

Significance in CPA Attacks

- **High Correlation:** This would mean that the guessed bit value is likely correct since it correlates well with the actual leakage in the traces.
- **Low Correlation:** This suggests the guessed bit value is most likely incorrect because it is inconsistent with the true leakage.

Weighted Key Guess vs. Most Frequent Key Guess The correlation distinguisher produces scores for all key guesses, which can be processed using two approaches:

- **Weighted Key Guess:** For every key hypothesis the scores over all bits of one byte are averaged. The hypothesis that resulted in the maximum average is returned.
- **Most Frequent Key Guess:** For each bit the key guess that has the highest correlation score is found. The key guess which is found to be the "bestguess" the most number of times over all bits.

Key Guess Implementation

```
# Weighted Key Guess
for (kg, score) in guesses_scores:
    byte_key_scores[byteno, kg] += score

# Most Frequent Key Guess
best_kg_for_bit = guesses_scores[0][0]
byte_key_counts[byteno, best_kg_for_bit] += 1
```

Description: The weighted guesses look at the overall power of the key guess over all bits, whereas most frequent guesses are related to bit-level dominance.

3.1.4 Outputs and Directory Structure

In the following, the outputs are described in detail along with how the scripts can be executed along with the console output.

Execution of Scripts The scripts can be executed from the command line as follows:

Execution Commands

```
# To execute the correlation distinguisher script and save the output:
python3 new_corr.py > output_corr.txt

# To execute the mean distinguisher script and save the output:
python3 new_mean.py > output_mean.txt
```

The above commands will redirect all the execution logs to text files `output_corr.txt` and `output_mean.txt`, respectively. The console output would include:

- Status of each byte and bit being processed.
- Top 5 key guesses for each bit, along with their scores.
- Cumulative time and average time per byte.
- Summary of the final key guesses (weighted and most frequent).

This output will allow one to observe the progress of the attack and make interpretations of intermediate results.

Sample Console Output The console output snippet below is an example:

Sample Console Output

```
[Byte=1, Bit=4] Top 5 key guesses:
  #1: Key = 01, Score = 0.396806
  #2: Key = 84, Score = 0.391937
  #3: Key = 02, Score = 0.385817
  #4: Key = 28, Score = 0.379548
  #5: Key = 67, Score = 0.376893

[Nm=100] Weighted key guess:
B9 A0 79 C6 BF 40 B6 AF 19 66 4F DA 76 82 B1 3C
[Nm=100] Most frequent key guess:
12 FE C2 10 02 03 27 00 0A 03 0A 19 0B 01 30 10
[Nm=100] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

[Nm=100] Total time: 5269.29s, Avg/byte=329.33s
```

This gives some information about the status of the attack, the intermediate results, and the final result for the number of traces N_m.

Outputs Saved For each trace count N_m, the following outputs are generated:

- **Key Guess Results:**
  - `keyguess_per_bit.csv`: Provides rankings and scores for all 256 key guesses for each bit and byte.
  - `keys_for_each_byte.csv`: Summarizes the best key guesses for each byte, based on:
    - `WeightedKeyGuess`: Best guess using the weighted average approach.
    - `MostFrequentKeyGuess`: Best guess based on the most frequent approach.
- **Top-N Statistics:**
  - `statistical_results.csv`: Details the success rate of the attack for different Top-N thresholds and evaluates key recovery complexity.
- **Plots:**
  - `example_traces_bits.png`: Few examples of power traces.
  - `byte_{X}_weighted_scores.png`: The figure is a bar-plot illustrating for each guess of byte X, the value of correlation scores.
  - `byte_{X}_most_frequent_counts.png`: Bar plot of the frequency of each key guess for byte X.
  - `time_per_byte.png`: Bar plot showing time taken to analyze each byte.
  - `cumulative_time.png`: Line plot of the cumulative analysis time for all bytes.

Directory Structure The outputs are structured in a logical directory layout to allow for easy traceability and analysis. For each N_m, the following is the directory structure:

Directory Structure

```
🟦 attack_bit
└─🟦 data
   └─🟥 attack_bit_{dist_name} # e.g., corr, mean
      └─🟧 df
         ├─🟩 100 # Results for N_m = 100
         │  ├─🟥 keyguess_per_bit.csv # Rankings for all key
         │  │   guesses
         │  ├─🟥 keys_for_each_byte.csv # Summary of best guesses
         │  ├─🟥 statistical_results.csv # Success rate and
         │  │   complexity
         │  ├─🟧 example_traces_bits.png # Sample power traces
         │  ├─🟧 byte_{0}_weighted_scores.png # Weighted scores for
         │  │   byte 0
         │  ├─🟧 byte_{0}_most_frequent_counts.png # Most frequent
         │  │   counts for byte 0
         │  ├─🟧 time_per_byte.png # Time per byte
         │  └─🟧 cumulative_time.png # Cumulative analysis time
         ├─🟩 200 # Results for N_m = 200
         │  └─🟥 ...  (similar structure as above)
         ├─🟩 300
         └─🟥 ...
```

3.2 Results and Analysis

In this section, I discuss results of the single-bit attack in using two distinguishers: correlation and difference of means. Their performance is compared for various number of traces, N_m, and weighted and most frequent key guess approaches are considered.

Data Cleaning and Additional Outputs The raw outputs generated by the execution of the scripts (`python3 new_corr.py > output_corr.txt` and `python3 new_mean.py > output_mean.txt`) were cleaned and organized to facilitate analysis. As a result, two cleaned files and two CSV datasets were produced:

- `output_corr_cleaned.txt`: Cleaned version of the correlation script output.
- `Corr_Key_Guess_Analysis_Dataset.csv`: Detailed dataset summarizing key guess analysis for the correlation distinguisher.
- `output_mean_cleaned.txt`: Cleaned version of the mean script output.
- `Mean_Key_Guess_Analysis_Dataset.csv`: Table of detailed data summarizing the key guess analysis for the mean distinguisher.

The details of these files are in the Appendix.

3.2.1 Summary of Results

Following tables provide summary of performance metrics for each distinguisher:

N_m	Weighted Key Guess Success	Most Frequent Key Guess Success	Total Time (s)
100	1/16	0/16	5269.29
200	6/16	3/16	5347.99
300	13/16	7/16	5400.78
400	15/16	11/16	5469.05
500	16/16	13/16	5513.35
1000	16/16	16/16	5868.49

Table 1: Performance of the Correlation Distinguisher

N_m	Weighted Key Guess Success	Most Frequent Key Guess Success	Total Time (s)
100	1/16	0/16	33.30
200	6/16	3/16	43.35
300	13/16	7/16	53.72
400	15/16	11/16	63.21
500	16/16	13/16	71.23
1000	16/16	16/16	122.94

Table 2: Performance of the Mean Distinguisher

3.2.2 Comparison of Approaches

Weighted Key Guess: The weighted key guess approach consistently outperforms the most frequent key guess in terms of success rate. This is evident across all trace counts N_m, as seen in Tables 1 and 2, and also in the graficl rappresentation in figure 1. By accumulating scores across all bits, this approach leverages a holistic view of the leakage, achieving full key recovery (16/16) at $N_m = 500$ for both distinguishers.

Most Frequent Key Guess: The most frequent key guess method demonstrates lower success rates for small N_m, not being able to reach high accuracy since the statistical evidence is just too scarce, whereas for increasing N_m its performance steadily improves and approaches weighted approach, even reaching full key recovery for $N_m = 1000$ (16/16).

3.2.3 Time Analysis

Correlation Distinguisher: The correlation distinguisher is computation intensive; total times exceed 5000 seconds for all N_m as can be observed in Tables 1 and 2. The average time per byte is also very high as showed in the figure 2.

Mean Distinguisher: The mean distinguisher is significantly faster, with total times below 150 seconds even for $N_m = 1000$. Also the average time per byte is very low as showed in the figure 2.

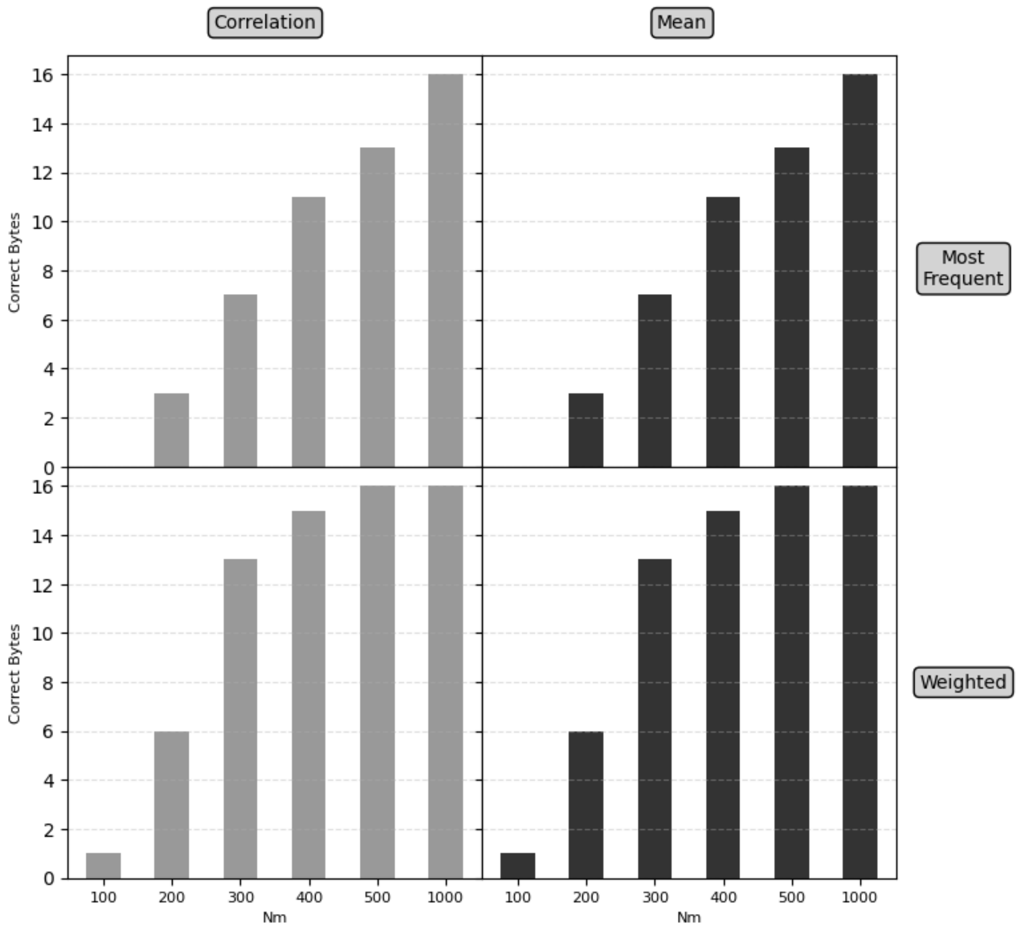

Figure 1: Corrected Bytes per approach

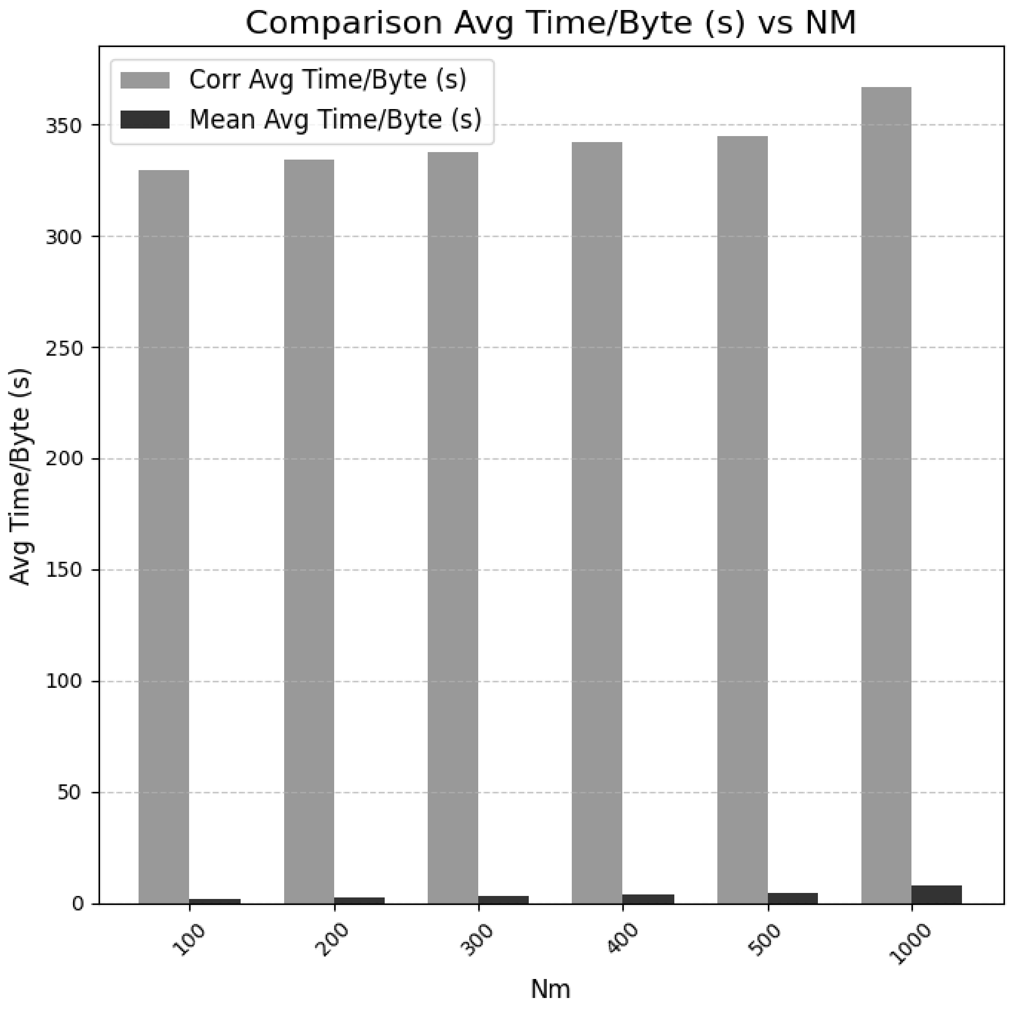

Figure 2: Time Analysis comparison

3.2.4 Key Observations

- The weighted key guess consistently outperforms the most frequent key guess.
- The correlation distinguisher is costlier in terms of computation.
- For scenarios with higher N_m or bounded computational resources, the mean distinguisher offers a time-efficient alternative.
- Both distinguishers achieve full key recovery when $N_m = 1000$ therefore confirming the efficiency of the single-bit attack model.

4 Attack on SubBytes Input

This section deals with the adaptation of the attack towards an AES attack focusing on the input of SubBytes. In the following, I detail the preprocessing, methods applied for the analysis of the traces, changes compared to the base attack and the results of the analysis.

4.1 Preprocessing Techniques

To improve the effectiveness of the attack I have used various preprocessing steps for the power traces. The following techniques are applied:

4.1.1 Savitzky-Golay Filter

The Savitzky-Golay filter is a smoothing filter that reduces noise while preserving the essential features of the signal. It operates by fitting successive polynomial functions to overlapping subsets of the trace data. For this attack:

- **Window Length**: A window length of 11 samples was chosen to balance smoothing and feature retention. A larger window would overly smooth the data, while a smaller window could leave noise unaddressed.
- **Polynomial Order**: A polynomial order of 4 was selected to provide adequate flexibility when fitting the features in the signal.

The filtered traces exhibited a considerable enhancement in the signal-to-noise ratio, thereby improving the correlation accuracy.

4.1.2 Trace Segmentation

Segmentation concentrates the focus on the parts of the power traces that are of interest while avoiding unnecessary computation. Samples ranging between 0 and 625 were selected as the range that could obtain the SubBytes operation. Empirical analysis by inspecting power trace features related to the AES operations.

4.1.3 Normalisation and Alignment

To make the traces consistent and comparable:

- **Normalization**: Each trace was normalized by subtracting its mean and dividing by its standard deviation. This removed offsets in the base line and made amplitudes consistent.
- **Alignment**: the traces are aligned using cross-correlation. Misaligned traces can dilute correlation scores; hence, this step was carried out to perfectly align the features of the signal in the various traces before subsequent processing.

4.2 Analysis Methodology

4.2.1 Hamming Weight Leakage Model

The Hamming weight leakage model assumes that the power consumption of a device is proportional to the number of bits set to `1` in the processed data. For AES, the SubBytes operation transforms plaintext bytes using a substitution box (S-box). The steps involved were:

1. For each plaintext byte, the SubBytes input was calculated as:

$$\text{SubBytes Input} = \text{Plaintext Byte} \oplus \text{Key Hypothesis}$$

2. The Hamming weight of the SubBytes input was computed for each key hypothesis.

3. The above Hamming weights were the hypothetical power consumption used for correlation analysis.

Hamming Weight Leakage Model Implementation

```
def HW(X):
    """Calculate Hamming weight of input array X."""
    return np.array([bin(x).count('1') for x in X])
```

Description: This function calculates the Hamming weight for each value in the array `X`.

4.2.2 Pearson Correlation Coefficient

To identify the correct key hypothesis, the Pearson correlation coefficient was computed between the predicted Hamming weights and the power traces. The process was as follows:

1. For each sample in the power trace, calculate the correlation coefficient with the Hamming weights.

2. Identify the maximum correlation value for each key hypothesis.

3. The hypothesis with the highest maximum correlation was considered the correct key.

Calculating the maximum Pearson correlation

```
def maxCorr(hw, traces):
    """Compute maximum Pearson correlation for given Hamming weights and
    ↪  traces."""
    max_corr = 0
    corr = np.zeros(traces.shape[1])
    for i in range(traces.shape[1]):
        corr[i], _ = pearsonr(hw, traces[:, i])
        if abs(corr[i]) > max_corr:
            max_corr = abs(corr[i])
    return max_corr, corr
```

Description: This function iterates over each sample in the power trace, calculating the correlation and returning the maximum value.

4.3 Implementation Details

This is done through the following steps of the attack script:

1. **Initialize the TRS Reader**: Loads power traces, plaintext data, and metadata from the TRS file.
2. **Preprocess the Traces**: Applying the Savitzky-Golay filter followed by the normalisation and alignment for consistency in analysis.
3. **Hypothesis Generation**: For every key-byte, calculation of the Hamming weight of the SubBytes input for all 256 word key values.
4. **Perform Correlation Analysis**: Computation of Pearson correlation between predicted Hamming weights and the power traces in order to identify the correct key.
5. **Save and Visualize Results**: Store the correlation scores and visualize the results for each byte, highlighting the correct key.

Key Hypothesis Initialization

```
def Initialise(N, trs_file):
    """Initialize traces and compute Hamming weight guesses for SubBytes
    ↪  input."""
    trs = TRS_Reader(trs_file)
    trs.read_header()
    trs.read_traces(N, 0, trs.number_of_samples)

    HWguess = np.zeros((16, 256, N), dtype=int)
    for byteno in range(16):
        plaintext_byte = trs.plaintext[:, byteno]
        for kg in range(256):
            subbytes_input = plaintext_byte ^ kg  # Modify to calculate
            ↪  SubBytes input
            HWguess[byteno, kg] = HW(subbytes_input)  # Compute Hamming
            ↪  weight of input
    return trs, HWguess
```

4.4 Comparison with Base Attack

Compared to the base attack that targeted the SubBytes output, the following modifications were implemented:

- **Target Shift**: While the base attack targeted the Hamming weight of the SubBytes output, computed as

$$\text{SubBytes Output} = \text{S-box}(\text{Plaintext Byte} \oplus \text{Key Hypothesis})$$

The new attack targets the SubBytes input:

$$\text{SubBytes Input} = \text{Plaintext Byte} \oplus \text{Key Hypothesis}$$

- **Hypothesis Generation**: The 'Initialise' function was changed to predict hypotheses for the SubBytes input rather than the output.
- **Preprocessing Enhancements**: Traces were smoothened, normalised, and aligned to provide a better signal to noise ratio which was less stressed in the base attack.

4.4.1 Comparison with Alternative Strategies

Alternative strategies tried:

- **Mean and T-Test**: This considered the statistical difference between groups of traces for different key hypotheses. But it delivered lower efficiency than the correlation analysis.
- **Bit-Level Attacks**: Similar to previously performed attacks on single bits, this approach again demonstrated worse efficiency than the one focusing on the SubBytes input.

4.5 Results and Observations

4.5.1 Concept of *topN*

In the context of the attack, ***topN*** is defined as the number of candidate keys (per byte) retained after correlation analysis. The attack calculates the correlation between the predicted Hamming weight vectors (for each possible key value, from 0 to 255) and the measured power traces. The Classical CPA strategy selects only the key with the highest correlation, (*top1*). However, in case of an insufficient number of traces or high noise level the correct key may not reach the top position.

Using *topN*, the single most correlated key hypothesis is extended to the N top key candidates having the highest correlation in each byte.

On the one hand, increasing *topN* gives a larger probability of having the correct key; on the other hand, it increases the search complexity due to testing the combination of the key candidates from *topN*.

4.5.2 Dependency Between *topN*, Number of Traces, and Correct Bytes

We may observe following dependences of number of *Correct Bytes* and the *number of traces* for different values of *topN*:

- For very small *topN* values (e.g., 1, 2, or 3) if the number of traces is low the correct correlation often doesn't rank high enough. Consequently, a limited number of bytes can be correctly identified. More traces are needed so that the ranking of the true key exceeds the noise and enters *top1*, *top2*, etc.
- Increasing *topN* raises the chances of the correct key being present in the first N candidates with less traces. Indeed, results indicate that for a moderated *topN* value, such as 20, 25, or 30, a fair number of bytes can be correctly recovered without the need to collect up to 1000 traces.

However, as the data demonstrates, even with high *topN* values, the number of bytes *correctly* ranked within the top N may not reach 16/16 if the traces are insufficient or if the signal is too noisy. In that case, the correct key remains out of the *topN*, and those bytes cannot be recovered.

4.5.3 Complexity Calculation and Impact of *topN*

The disadvantage of having a high value of *topN* is that the **overall complexity** increases, as more combinatorial search has to be performed for the variants of possible keys. This is shown in the second plot, where the x-axis is topN and the y-axis is the total complexity (on a logarithmic scale, base 2). The code performing these calculations defines complexity as:

- **Correct Bytes:** If, for a given byte, the correct key is within *topN*, that byte is effectively "circumscribed" to N possible values. For example, if c bytes are correctly recovered, the partial complexity (only for those bytes) is:

  $$(topN)^c,$$

  meaning that the *topN* values for each correctly identified byte must be combined.

- **Missing Bytes:** The remaining $16 - c$ bytes (where the correct key is not even within *topN*) has to be brute-forced completely, with 2^8 combinations for each byte:

  $$2^{8(16-c)}.$$

- **Total Complexity:** The sum (or, in some definitions, the product) of these two components represents the *effective search* that must be performed. The chosen formula in the code is:

  $$\underbrace{(topN)^c}_{\text{partial key complexity}} \quad + \quad \underbrace{2^{8(16-c)}}_{\text{remaining key complexity}} \quad .$$

  In *log2*, the total complexity becomes:

  $$c \cdot \log_2(topN) + 8(16 - c).$$

  If c is high (i.e., many bytes are correct within *topN*), the term $2^{8(16-c)}$ drastically decreases, reducing complexity. However, if *topN* is large, the first term $(topN)^c$ can increase rapidly.

Reference Complexities. To put these results into perspective, I also include two reference levels in our plots:

- **Full Brute-Force Complexity:** For a 128-bit AES key, a complete brute-force attack corresponds to 2^{128}. This is infeasible in practice with current computational resources.

- **Feasibility Threshold:** I additionally mark a lower reference threshold (e.g., 2^{64}) as a rough boundary beyond which an attack might still be conceivable in high-resource scenarios (though it remains expensive).

These reference lines help illustrate whether a given partial complexity is closer to a practical attack or remains prohibitively large.

4.5.4 Tables of Results for Different *topN* Values

Below are the tables summarizing the results for various *topN* values and different numbers of traces.

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	2	3	2	2	5	5
Bytes to Guess	14	13	14	14	11	11
Partial Key Complexity	1^2	1^3	1^2	1^2	1^5	1^5
Remaining Key Complexity	2^{112}	2^{104}	2^{112}	2^{112}	2^{88}	2^{88}
Total Complexity	$2^{112.0}$	$2^{104.0}$	$2^{112.0}$	$2^{112.0}$	$2^{88.0}$	$2^{88.0}$

Table 3: Results for *topN* = 1 (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	3	5	6	5	8	8
Bytes to Guess	13	11	10	11	8	8
Partial Key Complexity	2^3	2^5	2^6	2^5	2^8	2^8
Remaining Key Complexity	2^{104}	2^{88}	2^{80}	2^{88}	2^{64}	2^{64}
Total Complexity	$2^{107.0}$	$2^{93.0}$	$2^{86.0}$	$2^{93.0}$	$2^{72.0}$	$2^{72.0}$

Table 4: Results for *topN* = 2 (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	3	6	8	7	10	11
Bytes to Guess	13	10	8	9	6	5
Partial Key Complexity	3^3	3^6	3^8	3^7	3^{10}	3^{11}
Remaining Key Complexity	2^{104}	2^{80}	2^{64}	2^{72}	2^{48}	2^{40}
Total Complexity	$2^{108.8}$	$2^{89.5}$	$2^{76.7}$	$2^{83.1}$	$2^{63.8}$	$2^{57.4}$

Table 5: Results for *topN* = 3 (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	3	6	8	10	12	13
Bytes to Guess	13	10	8	6	4	3
Partial Key Complexity	4^3	4^6	4^8	4^{10}	4^{12}	4^{13}
Remaining Key Complexity	2^{104}	2^{80}	2^{64}	2^{48}	2^{32}	2^{24}
Total Complexity	$2^{110.0}$	$2^{92.0}$	$2^{80.0}$	$2^{68.0}$	$2^{56.0}$	$2^{50.0}$

Table 6: Results for *topN* = 4 (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	3	7	8	10	12	11
Bytes to Guess	13	9	8	6	4	5
Partial Key Complexity	5^3	5^7	5^8	5^{10}	5^{12}	5^{11}
Remaining Key Complexity	2^{104}	2^{72}	2^{64}	2^{48}	2^{32}	2^{40}
Total Complexity	$2^{110.9}$	$2^{88.3}$	$2^{82.6}$	$2^{71.2}$	$2^{59.9}$	$2^{65.5}$

Table 7: Results for $topN = 5$ (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	4	10	10	12	12	13
Bytes to Guess	12	6	6	4	4	3
Partial Key Complexity	10^4	10^{10}	10^{10}	10^{12}	10^{12}	10^{13}
Remaining Key Complexity	2^{96}	2^{48}	2^{48}	2^{32}	2^{32}	2^{24}
Total Complexity	$2^{109.3}$	$2^{81.2}$	$2^{81.2}$	$2^{71.9}$	$2^{71.9}$	$2^{67.2}$

Table 8: Results for $topN = 10$ (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	6	13	11	12	14	14
Bytes to Guess	10	3	5	4	2	2
Partial Key Complexity	20^6	20^{13}	20^{11}	20^{12}	20^{14}	20^{14}
Remaining Key Complexity	2^{80}	2^{24}	2^{40}	2^{32}	2^{16}	2^{16}
Total Complexity	$2^{105.9}$	$2^{80.2}$	$2^{87.5}$	$2^{83.9}$	$2^{76.5}$	$2^{76.5}$

Table 9: Results for $topN = 20$ (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	7	13	13	14	14	15
Bytes to Guess	9	3	3	2	2	1
Partial Key Complexity	25^7	25^{13}	25^{13}	25^{14}	25^{14}	25^{15}
Remaining Key Complexity	2^{72}	2^{24}	2^{24}	2^{16}	2^{16}	2^8
Total Complexity	$2^{104.5}$	$2^{84.4}$	$2^{84.4}$	$2^{81.0}$	$2^{81.0}$	$2^{77.7}$

Table 10: Results for $topN = 25$ (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	7	13	13	14	14	15
Bytes to Guess	9	3	3	2	2	1
Partial Key Complexity	30^7	30^{13}	30^{13}	30^{14}	30^{14}	30^{15}
Remaining Key Complexity	2^{72}	2^{24}	2^{24}	2^{16}	2^{16}	2^8
Total Complexity	$2^{106.3}$	$2^{87.8}$	$2^{87.8}$	$2^{84.7}$	$2^{84.7}$	$2^{81.6}$

Table 11: Results for $topN = 30$ (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	11	15	13	14	14	15
Bytes to Guess	5	1	3	2	2	1
Partial Key Complexity	50^{11}	50^{15}	50^{13}	50^{14}	50^{14}	50^{15}
Remaining Key Complexity	2^{40}	2^{8}	2^{24}	2^{16}	2^{16}	2^{8}
Total Complexity	$2^{102.1}$	$2^{92.7}$	$2^{97.4}$	$2^{95.0}$	$2^{95.0}$	$2^{92.7}$

Table 12: Results for *topN* = 50 (Transposed)

Number of traces	**100**	**200**	**300**	**400**	**500**	**600**
Correct Bytes	14	15	15	15	16	16
Bytes to Guess	2	1	1	1	0	0
Partial Key Complexity	100^{14}	100^{15}	100^{15}	100^{15}	100^{16}	100^{16}
Remaining Key Complexity	2^{16}	2^{8}	2^{8}	2^{8}	2^{0}	2^{0}
Total Complexity	$2^{109.0}$	$2^{107.7}$	$2^{107.7}$	$2^{107.7}$	$2^{106.3}$	$2^{106.3}$

Table 13: Results for *topN* = 100 (Transposed)

4.5.5 Graph Interpretation

- **Graph: "Correct Bytes vs. Traces" for Various *topN* (Figure 3)**. For small (number) traces, eg 100 and small *topN*(1-3), the amount of correctly identified bytes is very small. When this value of traces increases to 500, 700, or 1000, there is a progressive improvement, and most bytes enter the *topN* (especially for higher *topN* values). Total recovery (16/16) remains rare unless both many traces and high *topN* values are combined.
- **Graph: "Complexity (log2) vs. TopN" for Various Trace Counts (Figure 4)**. This graph highlights the trade-off explicitly:
  - For small *topN* values, the overall complexity can remain low. However, if c (the number of bytes correctly included in the *topN*) is small, the remaining brute-force complexity ($2^{8(16-c)}$) stays close to 2^{128}, which is impractical.
  - For large *topN* values (e.g., 50 or 100), the probability of including the correct byte in the *topN* increases even with fewer traces, so c increases. Consequently, the term $2^{8(16-c)}$ decreases significantly. However, the term $(topN)^c$ can still become very large, especially for high c values.

4.6 Conclusions

The combined analysis of these graphs and the tabular data reveals several key points:

- **Trade-Off Between Accuracy and Complexity:**
  - A high *topN* increases the likelihood that the correct key is captured among the top candidates, but also raises the combinatorial search complexity in successive phases;
  - A low *topN* reduces complexity, but can only be effective if a sufficient number of traces are available to push the true key into *top1*, *top2*, etc.

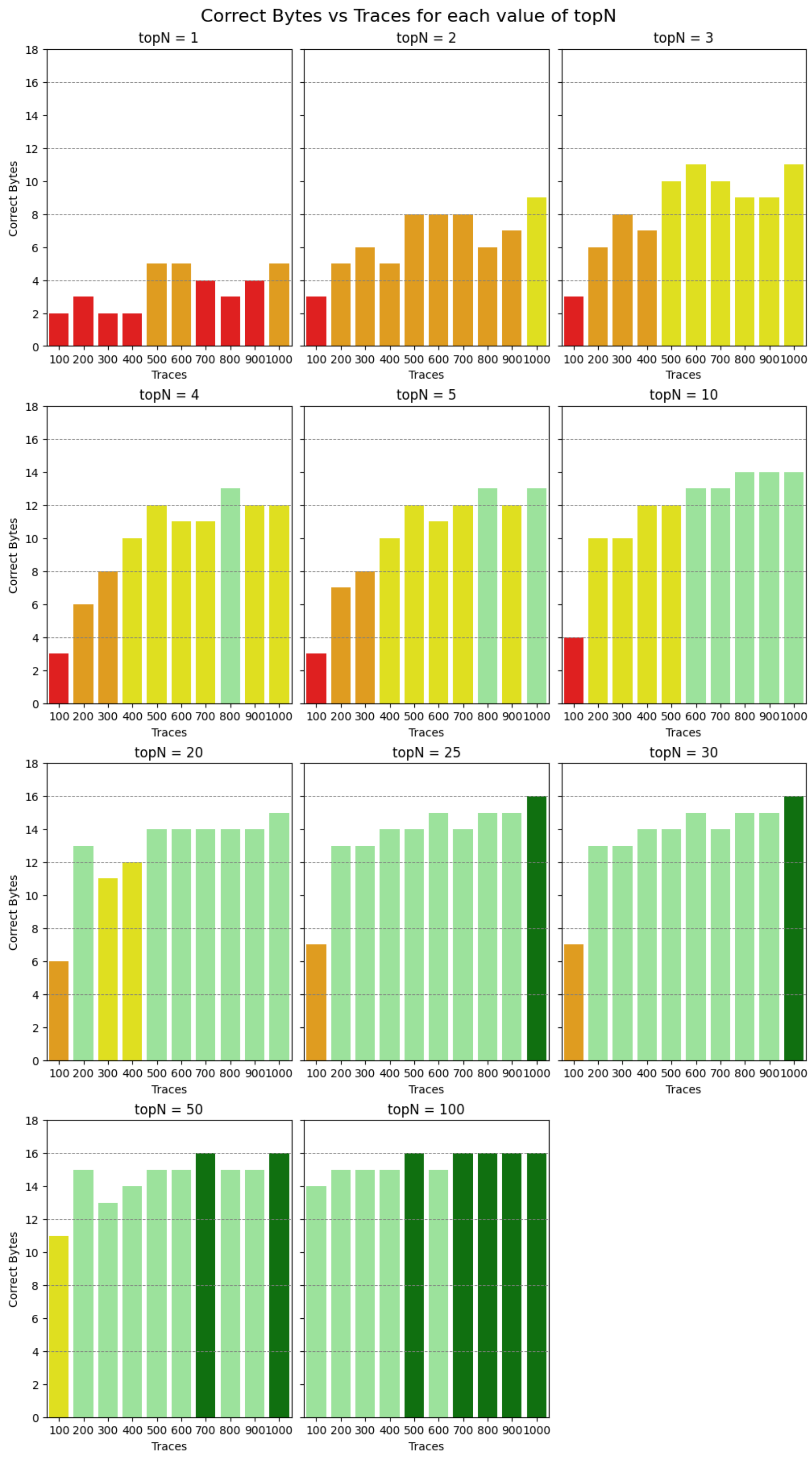

Figure 3: Graph: "Correct Bytes vs. Traces" for Various *topN*.

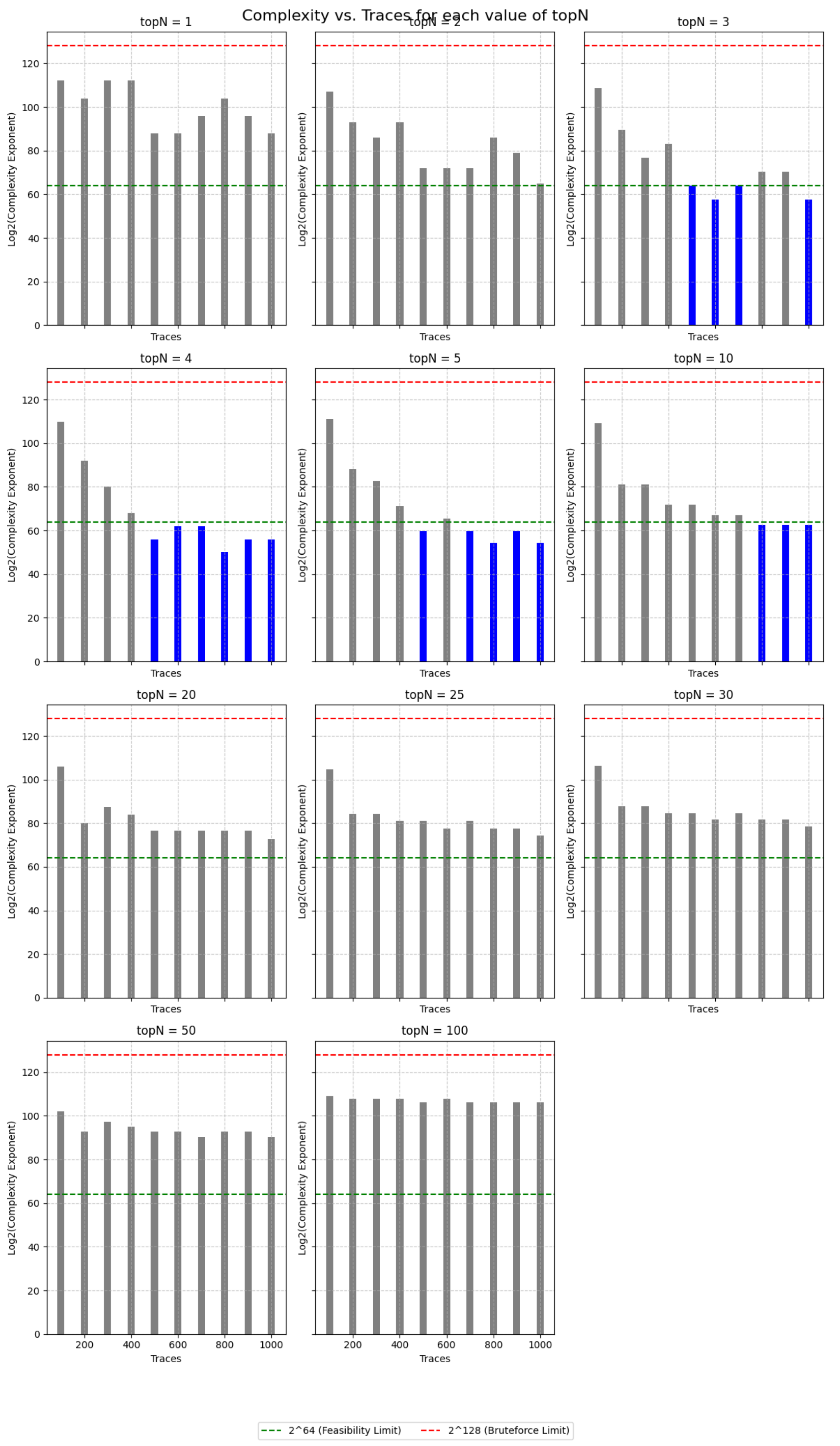

Figure 4: Graph: “Complexity (log2) vs. TopN” for Various Trace Counts.

- **Need for Many Traces for Small *topN*:** If I aim to keep *topN* small (e.g., hoping to "guess" the correct key in the first position), many more traces are required to clearly distinguish the correct correlation. The data shows that with only 100-200 traces, fewer than half the bytes typically fall within *top1*, *top2*, or *top3*.
- **Intermediate Strategies:** Moderate *topN* values (e.g., 5 or 10) provide a good trade-off between byte recovery and total complexity, particularly when the number of traces is limited.

In summary, the results clarify that a right balance between the *quantity of data* (traces) with the *breadth* of the *topN* selection width is required to achieve considerable reduction of the residual brute-force complexity. The statistical analyses and graphs (especially "Correct Bytes vs. Traces" and "Complexity vs. TopN") clearly demonstrate that there exists an *optimal balance* between having a sufficiently large *topN* to include the correct key and avoiding excessive search complexity.

One of the important observations to be made from this analysis is that taking only *top1* (i.e., the most probable key for each byte) never allows for full key recovery, even with a high number of traces. That can be clearly seen from Table 4.5.4, which never reaches the number of correct bytes to 16/16.

4.7 Discussion: Linearity, SubBytes Input vs. SubBytes Output

When performing a Correlation Power Analysis (CPA) attack on the AES algorithm, one often has a choice of which intermediate variable to target. Two common targets include:

1. The **SubBytes output**:

$$\text{SubBytesOutput} = S\big(\text{PlaintextByte} \oplus \text{KeyByte}\big),$$

   where $S(\cdot)$ is the non-linear S-box.

2. The **SubBytes input**:

$$\text{SubBytesInput} = \text{PlaintextByte} \oplus \text{KeyByte}.$$

Empirical Findings. In our experiments:

- Attacking the *output of SubBytes* produces strong correlations that allow recovering the entire 128-bit key using *top-1* for each byte with as few as 500 power traces (as I see attacking the output of subbytes using single bit leakage model in section 3). This means that for each byte, the correct key guess yields the highest correlation, even in a noisy environment, once I have at least a few hundred traces.
- Conversely, when attacking the *input of SubBytes* (i.e., plaintext XOR key), many more traces are required to push the correct key guesses into the top positions; even then, I often need to enlarge the *top-N* (e.g., up to 20 or more) for each byte to avoid losing the correct hypothesis. Hence, I either use far more traces (e.g., 500–1000) or accept a bigger *top-N* to maintain a good chance of capturing the correct key in the top candidates.

4.7.1 Why Attacking the SubBytes Input Is Harder

Linearity Considerations. The SubBytes operation is inherently *non-linear* due to the S-box. When I attack the SubBytes *output*, I are effectively correlating with:

$$HW\big(S(\text{PlaintextByte} \oplus \text{KeyByte})\big),$$

which often gives a stronger leakage signature. Empirically, S-box outputs exhibit more distinctive bit patterns (i.e., Hamming weights) that can yield higher CPA correlation peaks, especially once sufficient traces are gathered. The non-linear transformation introduced by the S-box can emphasize certain bit transitions, separating correct from incorrect key guesses more sharply.

When attacking the *input* of SubBytes, I are correlating with:

$$HW\big(\text{PlaintextByte} \oplus \text{KeyByte}\big).$$

This value is *linear* ($\oplus$ is a linear operation in GF(2) arithmetic). Because the SubBytes input lacks the non-linear S-box mapping, its distribution of Hamming weights can be closer to uniformly random. As a result, the power consumption differences across various key guesses can be more subtle and more susceptible to noise. In practice, this means:

- I need more traces to reliably elevate the correct key's correlation above the rest (i.e., to overcome noise).
- The ranking distribution for incorrect key hypotheses is narrower, so the correct hypothesis might not always appear in *top-1*. I often must look deeper into *top-N* to capture it.

Impact on *top-N* and Total Complexity. Attacking the SubBytes input thus shifts the "correlation advantage" away from the correct key. With the S-box output, even a small number of traces can yield a peak correlation that is clearly distinguishable (allowing a *top-1* success). With a linear target (the XOR input), the correct key guess can remain close to other guesses unless I collect significantly more traces.

As a result, to reliably find the correct key byte, I might choose a larger *top-N* for each byte. Unfortunately, this also increases the key search complexity, because I then need to consider $(topN)^c$ partial combinations for the c bytes that appear in *top-N*, plus $2^{8(16-c)}$ for the remaining $16 - c$ bytes that were not identified. In other words, I either gather enough traces so that the correct key guess reliably rises to *top-1* or I must accept a larger *top-N* —either way the cost (in traces or complexity) is higher compared to attacking the SubBytes output.

4.7.2 Summary of the Trade-Off

In summary, the non-linear transformation of the S-box appears to amplify leakage signals, making the SubBytes *output* an easier CPA target. By contrast, attacking the SubBytes *input*, which is just a linear XOR of plaintext and key, requires more traces (to improve the signal-to-noise ratio) or a larger *top-N* to ensure that the correct key remains in our candidate set. This comes with a dual cost:

1. Larger *top-N* $\implies$ bigger partial-key search space.

2. More traces $\implies$ longer acquisition time or more measurement effort in practice.

While the SubBytes input attack might still succeed eventually, it typically cannot match the efficiency achieved by directly attacking the SubBytes output.

5 Estimating the Minimal Number of Traces

In this section, I compare four attack strategies regarding the minimal number of traces that have to be used to successfully recover a key in a DPA attack. The analysis points out the relationship of the number of traces and key recovery rates including the complexity of key enumeration for various *top-N* (see Section 4.5.3).

1. **Bit-level correlation attack *after* the SubBytes step using the correlation distinguisher** (see Section 5.2.1).
2. **Bit-level correlation attack *after* the SubBytes step using the mean distinguisher** (see Section 5.2.2).
3. **Byte-level correlation attack *before* the SubBytes step using the correlation distinguisher** (see Section 5.2.3).
4. **Byte-level correlation attack *after* the SubBytes step using the correlation distinguisher** (see Section 5.2.4).

5.1 Methodology

- **Trace Collection:** I collected simulated traces for each attack, varying the total number of available traces (e.g., 100, 200, 300, etc.).
- **Attack Simulation:** For each attack strategy, simulations are performed by using traces issued by the specified number and measure the number of correctly recovered key bytes.
- **Key Ranking Analysis:** The correct key rank has been analyzed against all possible key guesses under different *top-N* metrics.
- **Complexity Estimation:** Overall, the attack complexity is estimated by considering the bytes recovered within the *top-N* along with the remaining brute-force search in unrecovered bytes.

5.2 Results and Observations

5.2.1 Bit-Level Correlation Attack After SubBytes Using the Correlation Distinguisher

This attack has been presented in 3.1.3 and yielded to a good result for bit-level attacks.

- **Minimum Traces for Full Key Recovery:** Recovery of all 16 key bytes was possible using about 500 traces for the *top-1* setting.
- **Impact of *top-N*:** Increasing the *top-N* value significantly decreased the number of traces required for full key recovery. For example, for *top-1*, *top-2*, *top-3* , *top-4* , *top-5* and *top-10*, full recovery was possible with approximately 300 traces with a complexity under 2^{64} (the practical complexity threshold).
- **Relationship with Complexity Thresholds:** As Figure 5 depicts I analyzed the attack complexity with respect to critical thresholds:

  - The green threshold (2^{64}) corresponds to computational complexity considered practical for real-world attacks. A remarkable number of the tests were below this threshold, especially for $Nm \geq 300$.
  - The red threshold (2^{128}) corresponds to brute-force complexity, which is considered infeasible. All tests were below this threshold.

- **Overall Effectiveness:** For attacks with a high trace count ($Nm = 500$ and $Nm = 1000$), the complexity is the same for all the *top-N*. Hence $Nm = 500$ is the ideal setting for practical scenarios.

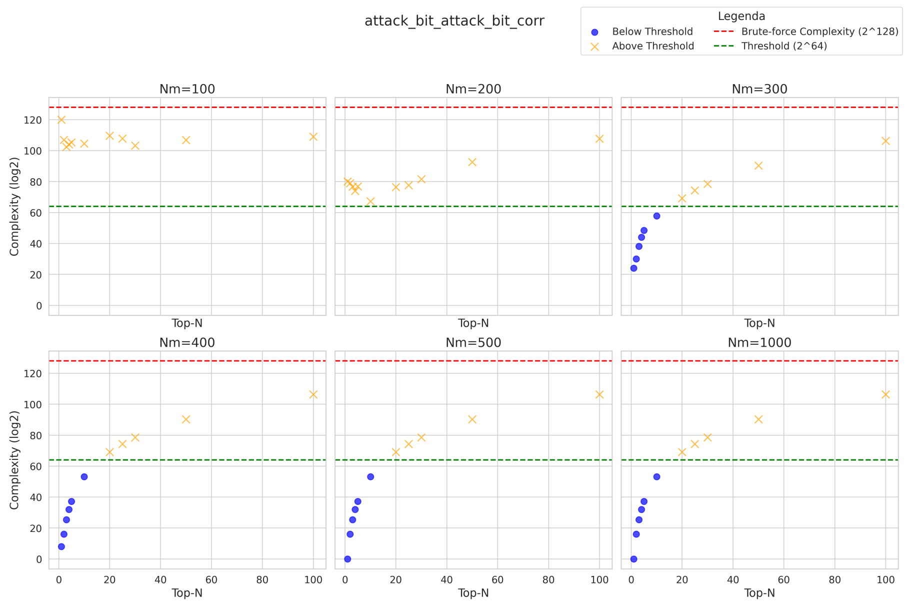

Figure 5: Complexity of the attack **"Bit-Level Correlation Attack After SubBytes Using the Correlation Distinguisher"** (5.2.1) as a function of *top-N* in logarithmic scale for different trace count (Nm) values. Green threshold (2^{64}) denotes practical complexity while the red threshold (2^{128}) shows brute-force complexity which is impractical. All points above or below these thresholds are colored differently.

In conclusion, the bit-level correlation attack after the SubBytes step using the correlation distinguisher was very effcient and practical providing a goof trade off between number of traces and computational complexity.

5.2.2 Bit-Level Correlation Attack After SubBytes Using the Mean Distinguisher

This is described in 3.1.2 and was successful in terms of key recovery complexity and efficient as the correlation distinguisher.

- **Minimum Traces for Full Key Recovery:** The performed attack needed around 500 traces to recover all 16 key bytes under a *top-1* setting and it is the same result that correlation distinguisher has.

- **Impact of *top-N*:** The same findings and observation that could be made in the correlation distinguisher settings can be made here.

- **Relationship with Complexity Thresholds:** As depicted in Figure 6, the attack complexity is evaluated with respect to relevant thresholds:
  - The green threshold (2^{64}) denotes a practical level of computational complexity for real-world attacks. For $Nm \geq 300$, several tests fell below this threshold, particularly for low *top-N* settings.
  - The red threshold (2^{128}) indicates brute-force complexity, deemed infeasible. With $Nm \geq 300$, all points were comfortably below this threshold.
- **Overall Effectiveness:** When using higher numbers of traces ($Nm = 1000$), nearly all data points for low *top-N* values (*top-1* and *top-5*) were below the green threshold and hence feasible in practice under high-trace settings.

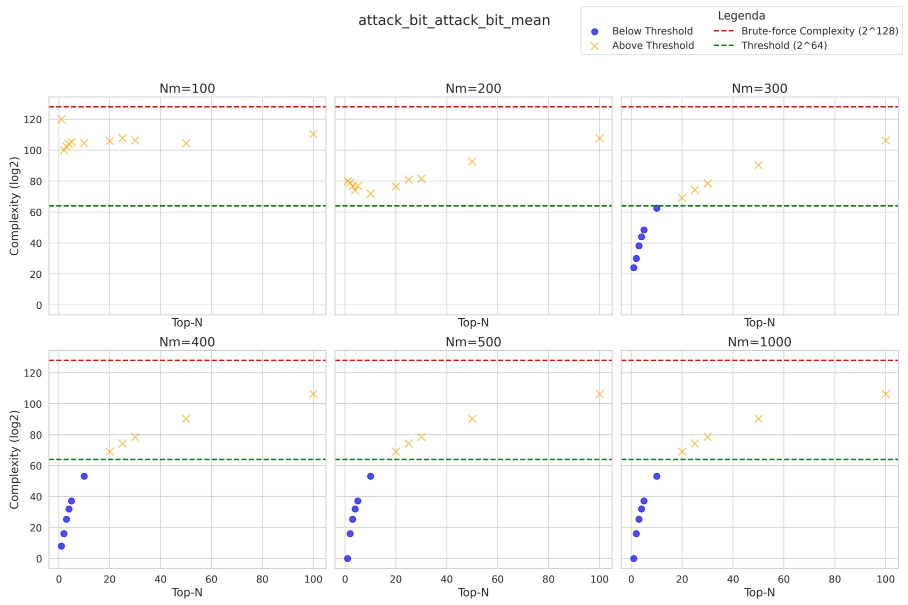

Figure 6: Complexity of the attack **"Bit-Level Correlation Attack After SubBytes Using the Mean Distinguisher"** (5.2.2) as a function of *top-N* in logarithmic scale for different trace count (Nm) values. Green threshold (2^{64}) denotes practical complexity while the red threshold (2^{128}) shows brute-force complexity which is impractical. All points above or below these thresholds are colored differently.

Finally, this attack was proved to be effective as the correlation distinguisher and showed similar results.

5.2.3 Byte-Level Correlation Attack Before SubBytes Using the Correlation Distinguisher

This attack approach, performed at the byte level before the SubBytes step as described in Section 4, showed a bad performance compared to other considered approaches.

- **Minimal Number of Traces for Full Key Recovery:** For this strategy, is never the case in which we can achieve full key recovery with a *top-1* setting.

- **Impact of *top-N*:** Increasing the *top-N* and the number of traces ($Nm \geq 500$) it is possible to recover the full key in a complexity time that is uder the practical boundary.
- **Relationship with Complexity Thresholds:** As shown in Figure 7:
  - The green threshold (2^{64}) represents the practical boundary for computational complexity. For $Nm \geq 500$, a few tests with low *top-N* values fell below this threshold, indicating partial success.
  - The red threshold (2^{128}) corresponds to brute-force complexity, which is infeasible. Most points with $Nm \geq 200$ were below this thresholds but this doesen't mean that are feasable and utilizable approaches.
- **Overall Effectiveness:** For higher trace counts ($Nm = 1000$ and $Nm = 500$), multiple points fell below the green threshold for lower *top-N* settings (from *top-1* to *top-5*). This demonstrates that this approach can achieve practical key recovery in scenarios with sufficient traces.

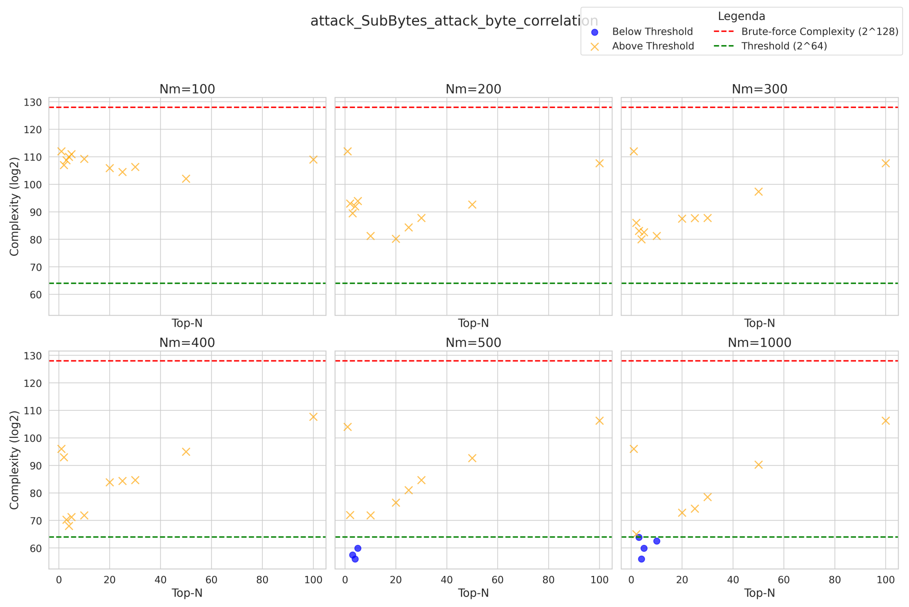

Figure 7: Complexity of the attack **"Byte-Level Correlation Attack Before SubBytes Using the Correlation Distinguisher"** (5.2.3) as a function of *top-N* in logarithmic scale for different trace count (Nm) values. Green threshold (2^{64}) denotes practical complexity while the red threshold (2^{128}) shows brute-force complexity which is impractical. All points above or below these thresholds are colored differently.

In conclusion, the byte-level correlation attack before the SubBytes step using the correlation distinguisher requires more traces compared to bit-level strategies to achieve comparable performance.

5.2.4 Byte-Level Correlation Attack After SubBytes Using the Correlation Distinguisher

The attack work in the same way as the one presented in Section 4: all the preprocessing techniques that are described in this section were applied and the attack is performed against **SubBytes output**:

$$\text{SubBytesOutput} = S\big(\text{PlaintextByte} \oplus \text{KeyByte}\big),$$

where $S(\cdot)$ is the non-linear S-box.

This attack strategy demonstrated high efficiency compared with the previous approaches:

- **Minimal Number of Traces for Full Key Recovery:** Approximately 100 traces were sufficient to recover all 16 key bytes with a *top-1* setting. This makes it one of the most efficient strategies analyzed, requiring fewer traces compared to attacks performed before the SubBytes step.
- **Impact of *top-N*:** A larger increase in the value of *top-N* and value of *Nm* doesn't make any significant note. Since with *Nm = 100* and *top-1* we are capable to recover the full key is not clever to increase the value of *top-N* or the *Nm*.
- **Relationship with Complexity Thresholds:** As shown in Figure 8:
  - The green threshold (2^{64}) represents the practical boundary for computational complexity. Most tests for all *Nm* with low *top-N* values fell below this threshold, indicating practical success.
  - The red threshold (2^{128}) corresponds to brute-force complexity, which is infeasible. All tests remained well below this threshold.
- **Overall Effectiveness:** the best strategy is achived with *Nm = 100* and *top-1* and allow to recover the full key with a small *Nm* using the first value predicted for the key.

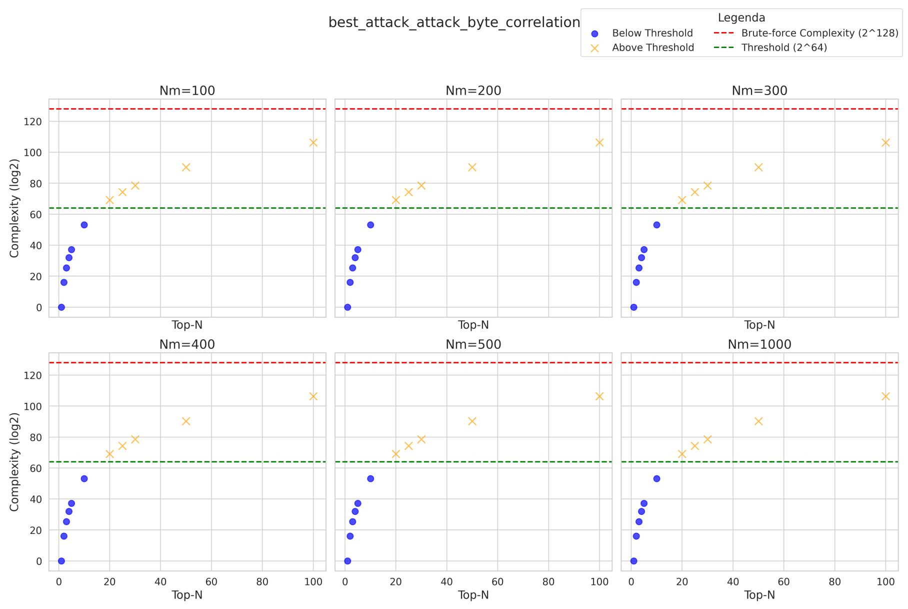

Figure 8: Complexity of the attack **"Byte-Level Correlation Attack After SubBytes Using the Correlation Distinguisher"** (5.2.4) as a function of *top-N* in logarithmic scale for different trace count (Nm) values. Green threshold (2^{64}) denotes practical complexity while the red threshold (2^{128}) shows brute-force complexity which is impractical. All points above or below these thresholds are colored differently.

In conclusion, the byte-level correlation attack after the SubBytes step using the correlation distinguisher is the most effective and practical approach that allows key recovery with fewer traces compared to other analyzed approaches.

5.3 Minimum Number of Traces for Byte-Level Correlation Attack After SubBytes Using the Correlation Distinguisher

This subsection focuses on determining the minimum number of traces (Nm) required to successfully recover the key using the byte-level correlation attack performed after the SubBytes step, leveraging the correlation distinguisher. The analysis also includes identifying the point where the complexity reaches 0, indicating full key recovery.

5.3.1 Analysis for *Top-N = 1*

As shown in Figure 9, the complexity of the attack decreases significantly with increasing Nm. For *Top-N = 1*, the attack achieves:

- **Practical success (complexity below 2^{64}):** This occurs when Nm reaches approximately 55 traces.
- **Full key recovery (complexity = 0):** The complexity stabilizes at 0 when Nm reaches 100 traces. This indicates that all key bytes are correctly recovered at this point, with no remaining brute-force complexity.

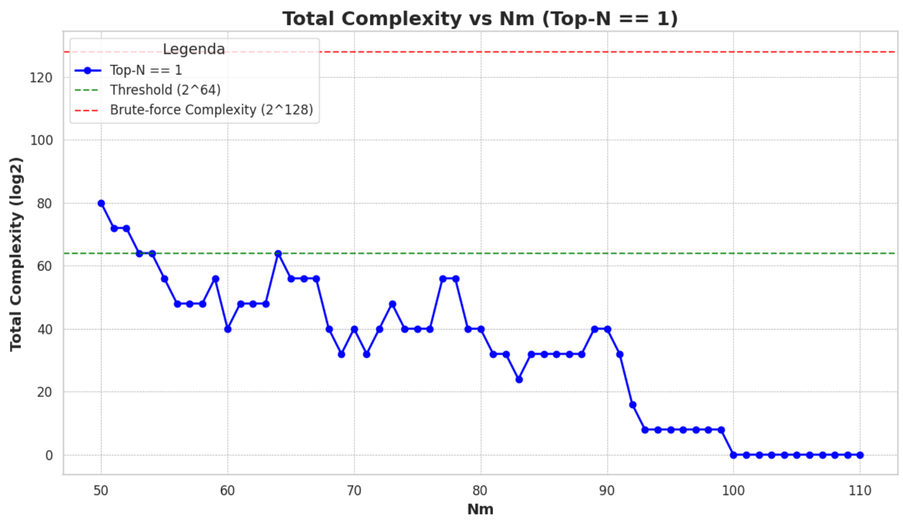

Figure 9: Total complexity (log scale) as a function of Nm for *Top-N = 1*. The green threshold (2^{64}) represents practical complexity, while the red threshold (2^{128}) corresponds to brute-force complexity. Complexity = 0 indicates full key recovery.

5.3.2 Impact of Different *Top-N* Settings

Figure 10 provides a detailed breakdown of the attack complexity for various *Top-N* settings. The results highlight the following trends:

- **Practical Success (Complexity below 2^{64}):** The attack becomes practical as soon as the complexity falls below 2^{64}, represented by the green threshold. The number of traces required for practical success decreases with higher *Top-N* values:

  - *Top-N = 1:* Practical success is achieved with approximately 53 traces.
  - *Top-N = 3:* Practical success is achieved with approximately 52 traces.
  - *Top-N = 5:* Practical success is observed with around 56 traces.
  - *Top-N = 10:* Practical success is achieved with around 60 traces.

- **Full Key Recovery (Complexity = 0):** Full recovery occurs when the complexity stabilizes at 0, indicating that all key bytes have been correctly recovered. This stabilization also varies with *Top-N*:
  - *Top-N = 1:* Full recovery is achieved with approximately 100 traces.
  - *Top-N = 3:* Full recovery is achieved with approximately 85 traces.
  - *Top-N = 5:* Full recovery requires about 90 traces.
  - *Top-N = 10:* Full recovery is observed with 75 traces.

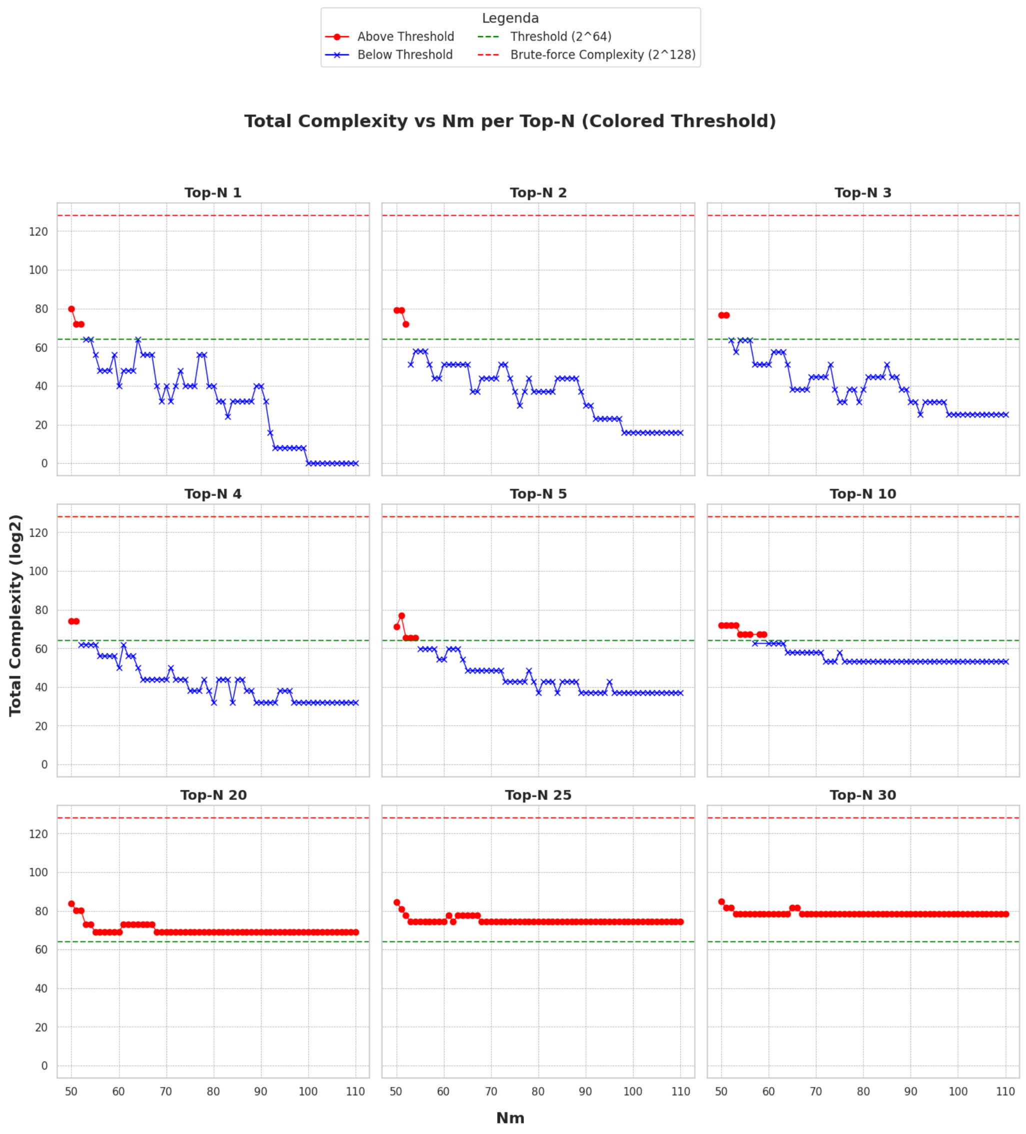

Figure 10: Total complexity (log scale) for various *Top-N* settings as a function of *Nm*. Red points indicate complexity above the practical threshold (2^{64}), while blue points fall below this threshold. Complexity = 0 indicates full key recovery.

5.3.3 Observations on Efficiency and Scalability

The results clearly indicate that relaxing the *Top-N* constraint improves both efficiency and scalability:

- Higher *Top-N* values (e.g., *Top-N = 10*) allow the attack to succeed with fewer traces, as the key ranking becomes less restrictive.
- However, for *Top-N* values above 10 (e.g., *Top-N = 20, 25, 30*), the complexity frequently exceeds the practical threshold (2^{64}), indicating diminishing returns for higher *Top-N* settings.
- The sweet spot appears to be in the range of *Top-N = 5 to 10*, balancing the trace requirement and practical success rates.
- If focusing exclusively on *Top-N = 1* (considering only the most likely key candidate), a minimum of 100 traces is required to ensure full key recovery (complexity = 0).

A Data and Outputs

This appendix provides an overview of the files and datasets generated during the analysis for the Section 3.

A.1 Cleaned Outputs

The raw outputs from the scripts were processed and saved in cleaned formats for easier analysis. The cleaned output files are as follows:

- `output_corr_cleaned.txt`: Contains the results of the correlation distinguisher for various values of N_m. Each section includes:
  - Weighted key guesses.
  - Most frequent key guesses.
  - Real keys.
  - Execution time details.
- `output_mean_cleaned.txt`: Contains the results of the mean distinguisher, formatted similarly to the correlation output.

A.1.1 output_corr_cleaned.txt

output_corr_cleaned.txt

```
[Nm=100] Weighted key guess:
B9 A0 79 C6 BF 40 B6 AF 19 66 4F DA 76 82 B1 3C
[Nm=100] Most frequent key guess:
12 FE C2 10 02 03 27 00 0A 03 0A 19 0B 01 30 10
[Nm=100] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=100] Total time: 5269.29s, Avg/byte=329.33s
Output stored in: data/attack_bit_corr/df/100

[Nm=200] Weighted key guess:
95 7E F8 3F 28 40 E4 1B 56 03 09 DA 09 CF 4F 3C
[Nm=200] Most frequent key guess:
03 13 0E 09 17 40 15 1B 14 08 12 16 0D CF 4F 3C
[Nm=200] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=200] Total time: 5347.99s, Avg/byte=334.25s
Output stored in: data/attack_bit_corr/df/200

[Nm=300] Weighted key guess:
87 7E 0A 3F 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=300] Most frequent key guess:
17 30 15 10 02 2F 05 1B 1E F7 15 65 09 CF 4F 3C
[Nm=300] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=300] Total time: 5400.78s, Avg/byte=337.55s
Output stored in: data/attack_bit_corr/df/300

[Nm=400] Weighted key guess:
C0 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=400] Most frequent key guess:
2A 7E 15 0F 28 AE 05 A6 06 F7 15 62 09 CF 4F 3C
[Nm=400] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=400] Total time: 5469.05s, Avg/byte=341.82s
Output stored in: data/attack_bit_corr/df/400

[Nm=500] Weighted key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Most frequent key guess:
0F 7E 17 10 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Total time: 5513.35s, Avg/byte=344.58s
Output stored in: data/attack_bit_corr/df/500

[Nm=1000] Weighted key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Most frequent key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Total time: 5868.49s, Avg/byte=366.78s
Output stored in: data/attack_bit_corr/df/1000
```

A.1.2 output_mean_cleaned.txt

output_mean_cleaned.txt

```
\textcolor{red}{easily}
[Nm=100] Weighted key guess:
B9 A0 EF C6 BF 40 B6 AF 19 FF 4F DA 76 82 B1 3C
[Nm=100] Most frequent key guess:
12 01 C2 10 02 03 21 1B 0A 03 0A 19 0B 01 08 10
[Nm=100] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=100] Total time: 33.30s, Avg/byte=2.08s
Output stored in: data/attack_bit_mean/df/100

[Nm=200] Weighted key guess:
95 7E FB 3F 28 40 E4 1B 56 03 09 DA 09 CF 4F 3C
[Nm=200] Most frequent key guess:
03 13 0E 09 17 40 15 1B 14 08 12 16 0D CF 4F 3C
[Nm=200] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=200] Total time: 43.35s, Avg/byte=2.71s
Output stored in: data/attack_bit_mean/df/200

[Nm=300] Weighted key guess:
87 7E 0A 8A 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=300] Most frequent key guess:
17 7E 15 10 01 2F 05 1B 1E 07 15 65 09 CF 4F 3C
[Nm=300] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=300] Total time: 53.72s, Avg/byte=3.36s
Output stored in: data/attack_bit_mean/df/300

[Nm=400] Weighted key guess:
C0 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=400] Most frequent key guess:
2A 7E 15 0F 28 AE 05 A6 06 F7 15 62 09 CF 4F 3C
[Nm=400] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=400] Total time: 63.21s, Avg/byte=3.95s
Output stored in: data/attack_bit_mean/df/400

[Nm=500] Weighted key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Most frequent key guess:
0F 7E 17 10 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=500] Total time: 71.23s, Avg/byte=4.45s
Output stored in: data/attack_bit_mean/df/500

[Nm=1000] Weighted key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Most frequent key guess:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Real key:
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
[Nm=1000] Total time: 122.94s, Avg/byte=7.68s
Output stored in: data/attack_bit_mean/df/1000
```

A.2 CSV Datasets

The cleaned data was further organized into CSV files to facilitate analysis and visualization:

- `Corr_Key_Guess_Analysis_Dataset.csv`: Summarizes the key guess analysis for the correlation distinguisher. It includes columns such as N_m, weighted key guesses, most frequent key guesses, and execution times.
- `Mean_Key_Guess_Analysis_Dataset.csv`: Provides a similar summary for the mean distinguisher.