NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

COLLEGE OF COMPUTING & DATA SCIENCE

NANYANG TECHNOLOGICAL UNIVERSITY

SC4015: Cyber Physical System Security

Side-Channel Analysis Lab Report

Hashil Jugjivan

U2120599F

1 Introduction

Side-channel attacks leverage unintended information leakage from physical implementations of cryptographic algorithms to recover secret data, most commonly secret encryption keys. In this lab, we aimed to perform a power side-channel attack on an AES-128 implementation. We collected multiple power traces of an embedded device during encryption and applied correlation power analysis (CPA) to deduce the secret key. The experiments provided hands-on familiarity with the vulnerabilities of various hardware-based cryptographic modules and underscored the importance of mitigating side-channel leakages and attacks.

2 Background

2.1 Target AES-128 bit Algorithm

AES (Advanced Encryption Standard) is a symmetric block cipher that operates on 128-bit blocks using keys of 128, 192, or 256 bits. **AES-128** uses a 128-bit key (16 bytes). While brute-forcing a 128-bit key is computationally infeasible, side-channel attacks circumvent brute force by exploiting physical data leaks such as power consumption. Looking at the first two operations in AES encryption, illustrated in Figure 1 in the Appendix, from Plaintext -> AddRoundKey -> SubBytes, there is some useful information we can extract:

1. AddRoundKey: Takes 16 bytes of the plaintext block and XORS it with 16 bytes of the secret round key. The XOR operation is performed byte by byte.
2. SubBytes: A non-linear substitution is performed using an 8-bit S-box, illustrated in Figure 2 of the appendix. This step is particularly important for side-channel attacks since the S-Box output is strongly data-dependent, causing notable variations in power usage. For instance, 0x34 substitution (down 3 across 4 of the S-box) yields 0x18. The information gathered from the SubBytes can be used to recover the key easily.

2.2 Power Side-Channel Attacks

Power side channel attacks are a form of cryptographic attacks that make use of the information leakage via the device's power consumption patterns rather than directly targeting the encryption algorithm. The electronic devices consume varying amounts of power when conducting operations on either logical ones or zeroes. During encryption operations, the power consumption of the devices used to encrypt data correlates with the data being encrypted with the secret key. By collecting many power traces and correlating them to the known plaintext/ciphertext pairs, an adversary can infer the secret key, which is often much faster than attempting exhaustive key searches.

2.3 Correlation Power Analysis (CPA)

CPA is done by gathering or observing the behaviour of the encryption device over several number of encryption operations (traces) that are captured. CPA is usually performed by

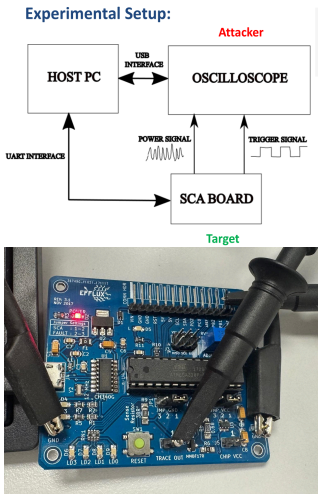
knowing the input (e.g. message input) and also knowing the output, which in this case would be the ciphertext. We will apply the following methodology to implement CPA:

1. Divide and Conquer: Retrieve each byte of the secret key, one byte at a time. Repeat this process 16 times to obtain the entire key.
2. Attack Philosophy: For each byte of the key hypotheses, we will compute an expected power consumption model, using the Hamming Weight of the targeted intermediate value, the S-Box output.
3. Correlation: We will compare, using a statistical correlation coefficient, the predicted model against the actual measured power traces over multiple encryptions. The hypothesis or model that produces the highest correlation is typically the correct key byte.

The Hamming Weight (HW) Model is defined as follows:

1. Take a value and convert it into bits.
2. The number of 1s in the bit stream determines the power consumption:
    a. 50 Decimal: 11001, HW: 3
    b. 99 Hex: 1001 1001, HW: 4 and so on.

3 Lab Proceedings



3.1 Experimental Setup

1. Hardware:
    a. Target Device: SCA Board
    b. Oscilloscope (Attacker): Captures voltage/current traces from the target device during encryption.
    c. Measurement Probe: Connected to the SCA Board pins.
    d. Trigger Signal: A digital signal used to indicated when the encryption starts, allowing the oscilloscope to precisely captures the relevant region of each encryption event.
2. Software:
    a. Serial Communication Script (Python): Sends plaintext to the target device, receives the corresponding ciphertext and coordinates with the oscilloscope.
    b. Analysis Script (Python): Parses the stored power traces and performs correlation for each key-byte hypothesis
3. Communication between Setup Components:
    a. PC ↔ Target Device: The PC runs a Python script (Appendix) that sends plaintext blocks to the target device over a serial (UART/USB) connection. The target responds with the ciphertext after performing AES encryption.
    b. Target Device ↔ Oscilloscope: The target device outputs a trigger signal (digital line) to the oscilloscope, indicating when encryption begins. This ensures the oscilloscope captures the relevant portion of the power signal.

c. Oscilloscope ↔ PC: Once triggered, the oscilloscope measures the device's power consumption and transfers the digitised trace data back to the PC (via USB). The PC's script then correlates this data with the corresponding plaintext/ciphertext for the side-channel analysis.

This is an illustration of how the wavelength would appear on the oscilloscope following the correct setup, explained and illustrated above.

Trigger Channel 2: Tells us when the encryption algorithm starts

a. Mode: Normal
b. Low: 0v, High:3.3v

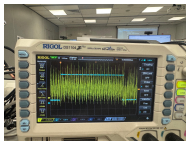
3.2 Steps Performed for Power-Trace Collection

The waveform.csv file contains Power Sidc-Channel Traces, where each row corresponds to one trace:

- Column A: Plaintext, Column B: Ciphertext
- The rest of the columns are sampled waveforms between 0 and 1, with approx. 2500 points per trace.

The oscilloscope was also configured as follows:

1. Coupling: AC/DC
2. Mode: Set to normal
3. Channels:
    a. Channel 1 was connected to measure the SCA board's power consumption.
    b. Channel 2, which is the trigger, receives a digital pulse from the target, indicating when AES encryption begins.
4. Time Base/Horizontal Scale: Adjusted to capture the entire encryption window.
5. Voltage Scale/Vertical Scale: Chosen based on the amplitude of the measured power signal.



We captured power traces in the following manner, for each encryption:

1. The signal probes are correctly connected to the SCA board.
2. The oscilloscope is set up as explained above.
3. A random or chosen plaintext block is sent to the SCA board for encryption.
4. The device begins AES encryption, and he trigger signal transitions to high, causing the scope to record the trace.
5. The scope data is then shown and analysed on the oscilloscope screen.
6. The plaintext/ciphertext pair and the captured power traces are saved.

3.3 Methodology of CPA

Hypothesis for Each Key Byte:

- Suppose we want to recover the first key byte. We guess all 256 possibilities.
- For each guess k, compute the intermediate value SBox(plaintext_byte XOR k).

Power Model (e.g., Hamming Weight):

- Convert the S-box output to binary and count the number of '1' bits. This is the estimated power consumption for that operation.

Correlation with Measured Trace:

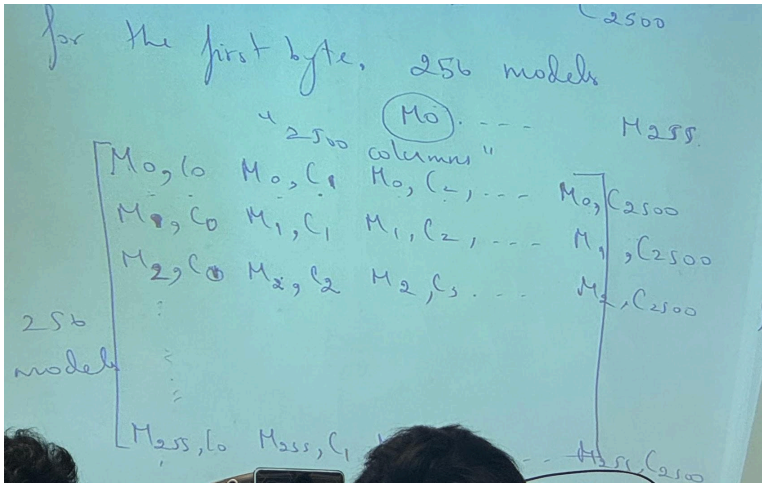
- For each of the N traces, you have:
  - Measured Power at each sample index t (e.g., 2,500 sample points).
  - Predicted Power based on plaintext[i], guess k, and the S-box operation's Hamming Weight.
- Calculate a correlation coefficient (e.g., Pearson's correlation) between the predicted and measured power across all traces at each sample index t.
- The correct guess for k typically yields a strong correlation peak at the time index corresponding to the S-box operation.

Recover the Entire 16-Byte Key:

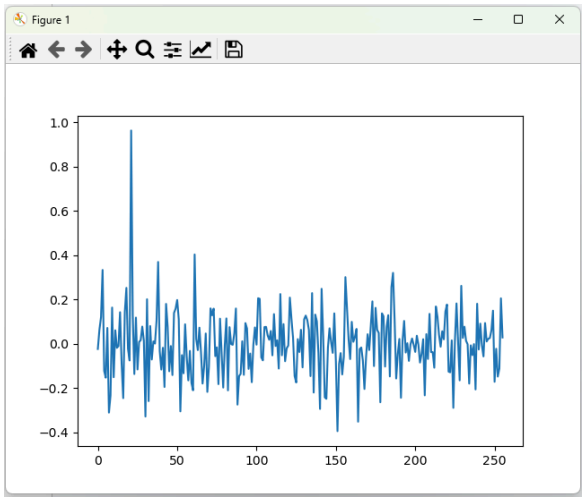
- Repeat the above steps for bytes 2 through 16. The combined results form the recovered 128-bit key.

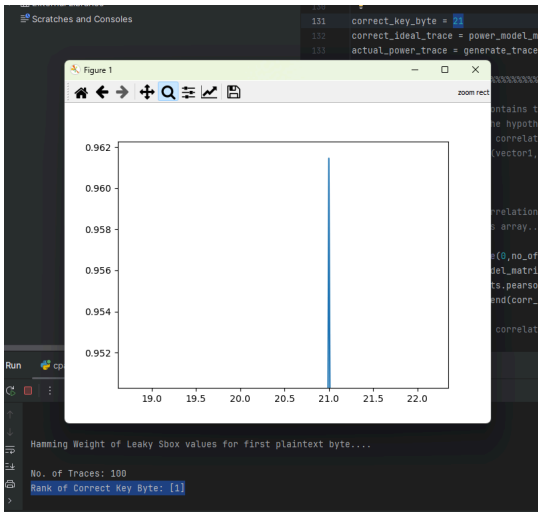
Alternatively:

1. Try each column 1 by 1 until a matching value is the highest
   1.1. ie 2500 columns, from C01, C02 to C2500
   1.2. For the first byte, create 256 models, M0, M1, M2… M255
2. Match M0 with C0
   2.1. Match M0 with C1
   2.2. Match M0 with C2
   2.3. …
   2.4. Match M0 with C2500
3. Match M1 with C0
   3.1. Match M1 with C1
   3.2. Match M1 with C2
   3.3. …
   3.4. Match M1 with C2500
4. …
5. Match M255 with C0
   5.1. Match M255 with C1
   5.2. Match M255 with C2
   5.3. …
   5.4. Match M255 with C2500



The highest matching result (correlation) will be the result of the first byte of the key, shown below in the two figures. We will then repeat this process for byte 2, byte 3 and so on, until we get the full 16-byte round key.





4 Extracting the full 16-byte key

4.1 Experiment Goal & Results

Our goal is to expand on the existing Python script used in the lab, together with the waveform.csv data, to get the first byte of the key. Using the waveform.csv data set supplied in Lab-3, we extended the reference script cpa_implementation.py to perform a full 16-byte Correlation Power Analysis (CPA) on real traces instead of simulated ones.

Our Python program, final_cpa.py, loads 110 traces × 2,500 sample points, builds leakage models for every key-byte hypothesis, and computes the Pearson correlation against every time index. The script successfully recovered all 16 bytes of the AES-128 key:

Full AES Key: 48 9D B4 B3 F3 17 29 61 CC 2B CB 4E D2 E2 8E B7, and hence the first byte of the key is **0x72** (48).

For every byte, the key value with the strongest correlation stands out clearly, and these values exactly match the real key we confirmed using an independent AES-decryption check.

4.2 Code Explanation

1. **Initialize:**
   - **1.1.** Load AES S-box (as a 1D array of 256 values)
   - **1.2.** Prepare empty lists to store plaintexts, ciphertexts, and power traces
2. **Read CSV Input File:**
   - **2.1.** For each row in the CSV:
     - **2.1.1.** Extract plaintext → store in plaintext list
     - **2.1.2.** Extract ciphertext → store in ciphertext list
     - **2.1.3.** Extract power trace (rest of the row) → store in power traces list
3. **Transpose Power Trace:**
   - **3.1.** Transpose the power traces so that each sublist corresponds to a time point across all traces
4. **For each Key Byte Position (0 to 15):**
   - **4.1.** Initialise a variable to keep track of the highest correlation and the most likely key byte guess
   - **4.2.** For each possible key guess (0 to 255):
     - **4.2.1.** For each plaintext (110):
       - XOR the plaintext byte at this position with the key guess
       - Apply AES S-box to get the intermediate value
       - Compute the Hamming weight of the S-box output → Store predicted power value
     - **4.2.2.** For each time point (2500) in the transposed power traces:
       - Extract actual power values at this time point across all traces
       - Compute the Pearson correlation between predicted power and actual power
       - If correlation is higher than the current max, update max correlation and guessed key byte
   - **4.3.** After trying all guesses, store the best guess for this key byte
5. **After all 16 bytes:**
   - **5.1.** After processing all 16 key byte positions, combine the best guesses into a full 16-byte AES key

4.3 Why “So Much Matching?”

Because CPA is basically a brute-force match:

- For each key guess (256 per byte),
    - For each time index (2500 total),
        - Compute the correlation between the model and the measured power at that time point

    That’s 256 key guesses × 2500 time points = 640,000 comparisons **per key byte**..
    ×16 bytes = **10,240,000 comparisons** for full AES key.
- Because:
    1. We don’t know which time index leaks the info
    2. We don’t know which key guess is right.

This code does NOT first pick a time point and then test all key guesses. Instead, for each key guess, it tries all 2500 time points, and remembers only the maximum correlation for that key. Then, out of the 256 max correlations (one per key guess), it picks the key guess with the highest of them all.

So, if our key guess is correct happening at the correct time index, the model (e.g., expected Hamming weights) will correlate the strongest with the actual power usage at that time index.

4.4 Why transpose power values

Because in side-channel analysis (like DPA/CPA), we analyse power at specific time points across many traces. We are asking things like:

- "At time index 320, does the power correlate with the Hamming weight of some key-related intermediate value?" We can’t ask that unless your data is structured by time index, hence the transpose.
- So, check all 2500 time points just in case the key-dependent operation happened at, say, time index 813.

Whichever time index with the highest correlation is likely when the operation happened, with the key guess that caused that is probably the real key byte.

4.4 Findings

1. Original Code Output:

Model Traces: 110, Waveform Datapoints: 2502

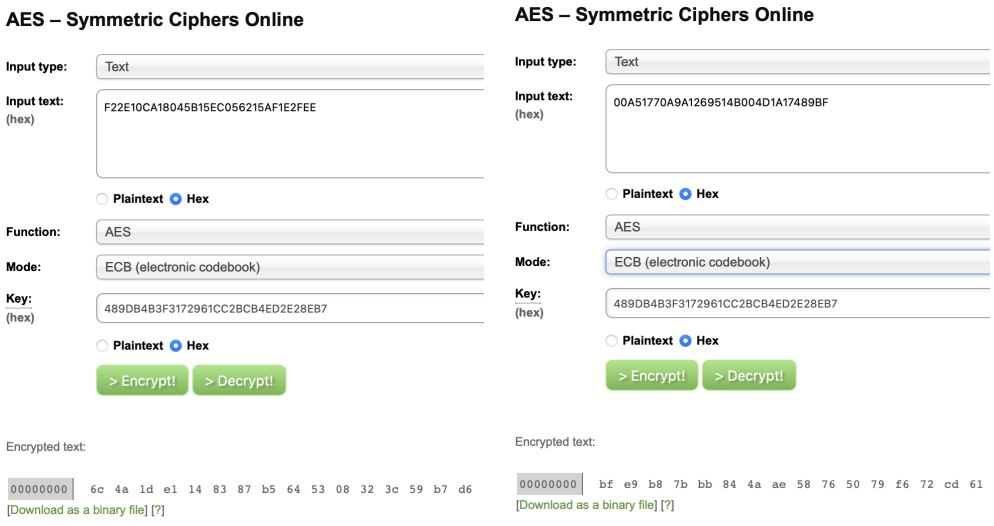
Key Byte 1: 72 (0x48) from Time Index **513** with Correlation Value 0.5725063314393224
Key Byte 2: 157 (0x9D) from Time Index **2200** with Correlation Value 0.701463295376188
Key Byte 3: 180 (0xB4) from Time Index **714** with Correlation Value 0.7081874276805077
Key Byte 4: 179 (0xB3) from Time Index **813** with Correlation Value 0.625198182450707
Key Byte 5: 243 (0xF3) from Time Index **913** with Correlation Value 0.6413699822039655
Key Byte 6: 23 (0x17) from Time Index **2250** with Correlation Value 0.6945020018894668
Key Byte 7: 41 (0x29) from Time Index **1113** with Correlation Value 0.6207291305454039
Key Byte 8: 97 (0x61) from Time Index **1213** with Correlation Value 0.6197514051006907
Key Byte 9: 204 (0xCC) from Time Index **1315** with Correlation Value 0.5052991941460786
Key Byte 10: 43 (0x2B) from Time Index **2300** with Correlation Value 0.699830649728842
Key Byte 11: 203 (0xCB) from Time Index **1514** with Correlation Value 0.5768388838813262
Key Byte 12: 78 (0x4E) from Time Index **1614** with Correlation Value 0.4894392024137064
Key Byte 13: 210 (0xD2) from Time Index **1713** with Correlation Value 0.6026759167842686
Key Byte 14: 226 (0xE2) from Time Index **1814** with Correlation Value 0.626001829906215
Key Byte 15: 142 (0x8E) from Time Index **1913** with Correlation Value 0.6015484563777957
Key Byte 16: 183 (0xB7) from Time Index **2014** with Correlation Value 0.5522224562587081

AES Key: 48 9D B4 B3 F3 17 29 61 CC 2B CB 4E D2 E2 8E B7

- Key Byte 2, Key Byte 6, Key Byte 10 stands out after analysing Time Index.
- We expected the time index to be sequential as the Plaintext is Encrypted Byte by Byte
- Attached in Appendix is the diagram for Time Index and we can observe where the Time Index is likely to be the correct for the 3 Key Bytes based on the Spikes (Correlation).

4.5 Verifying Results

But Key is Correct! Verified with: http://aes.online-domain-tools.com/



2. Corrected Code Output:

Model Traces: 110, Waveform Datapoints: 2502
Key Byte 1: 72 (0x48) from Time Index 513 with Correlation Value 0.5725063314393224
Key Byte 2: 157 (0x9D) from Time Index 613 with Correlation Value 0.6257239530117983
Key Byte 3: 180 (0xB4) from Time Index 714 with Correlation Value 0.7081874276805077
Key Byte 4: 179 (0xB3) from Time Index 813 with Correlation Value 0.625198182450707
Key Byte 5: 243 (0xF3) from Time Index 913 with Correlation Value 0.6413699822039655
Key Byte 6: 23 (0x17) from Time Index 1013 with Correlation Value 0.5603867041500399
Key Byte 7: 41 (0x29) from Time Index 1113 with Correlation Value 0.6207291305454039
Key Byte 8: 97 (0x61) from Time Index 1213 with Correlation Value 0.6197514051006907
Key Byte 9: 204 (0xCC) from Time Index 1315 with Correlation Value 0.5052991941460786
Key Byte 10: 43 (0x2B) from Time Index 1413 with Correlation Value 0.5840644224377075
Key Byte 11: 203 (0xCB) from Time Index 1514 with Correlation Value 0.5768388838813262
Key Byte 12: 78 (0x4E) from Time Index 1614 with Correlation Value 0.4894392024137064
Key Byte 13: 210 (0xD2) from Time Index 1713 with Correlation Value 0.6026759167842686
Key Byte 14: 226 (0xE2) from Time Index 1814 with Correlation Value 0.626001829906215
Key Byte 15: 142 (0x8E) from Time Index 1913 with Correlation Value 0.6015484563777957
Key Byte 16: 183 (0xB7) from Time Index 2014 with Correlation Value 0.5522224562587081

But the AES key is still the same!!! The key values didn't change, because the same key value was dominating multiple top correlation spots.

- A phenomenon known as "correlation clustering" in side-channel analysis
- Certain key guesses (0x9d, 0x17, 0x2b) show strong correlation at multiple time points throughout the power trace. This could be because:
    a. These operations are repeated at different points in the algorithm execution
    b. The device's power signature reflects the key byte's influence at multiple processing stages
    c. There might be architectural features (caches, pipelines) causing the same operation to appear multiple times
    d. The hardware architecture might be causing power signatures to appear at different stages (fetch, execute, store)
    e. The particular implementation of AES might have some optimization that causes these specific bytes to be processed differently

II. MULTI-LEAKAGE AND MULTI-INPUT MODEL

In this section, we first explains why a specific attack point can have multiple leakage intervals and uses different attacks as examples. Afterwards, we show the network structure of the proposed multi-input model.

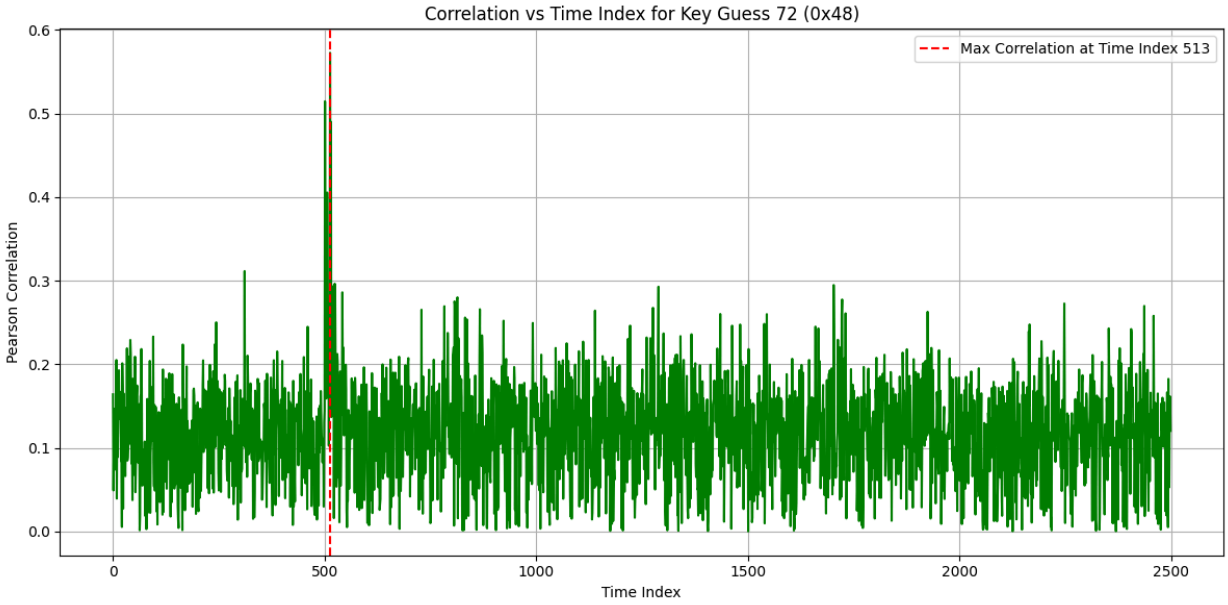
Source: https://scispace.com/pdf/multi-leak-deep-learning-side-channel-analysis-2l37uo43.pdf

3.1 The correlation coefficient when integrating within a trace

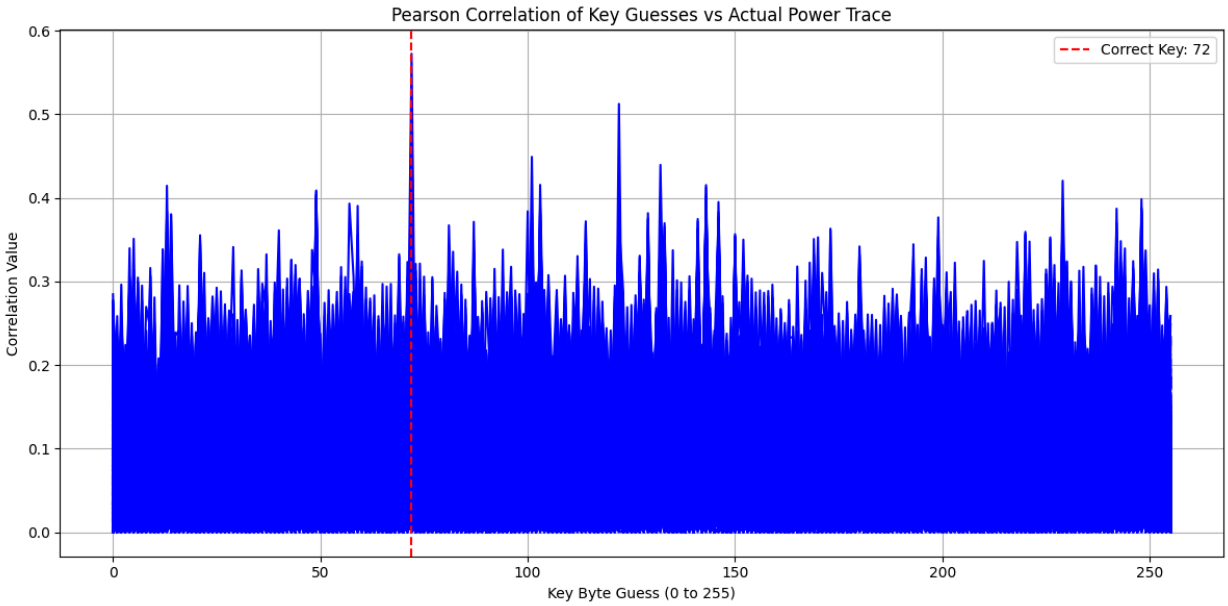
The leakage model previously mentioned in Section 2.4 assumes only a *single* leakage point within the integration window, and $r - 1$ power samples independent of the encryption key (non leakage samples). However, there might be several leakage samples in a trace. Whenever the cryptanalyst observes more than one peak in the correlation coefficient in a CPA for the correct key byte, there is more than one leakage point. This may be caused by several reasons: multiple leakage sources may exist, such as data bus leakage, address bus leakage or different electronic components' glitches which may all happen sequentially.

Source: https://eprint.iacr.org/2018/317.pdf

First Byte of Key Plots (Byte 1 of 16) → 0x48 (72)



Graph 1: First Byte of Key over Time (Time it happened)



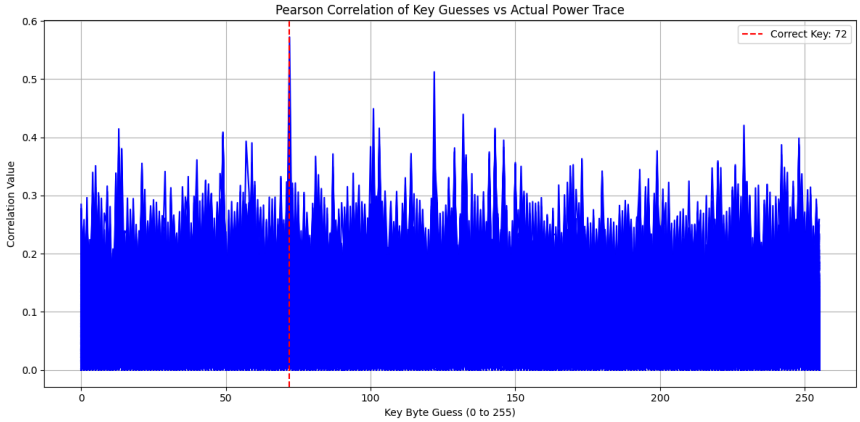
Graph 2: Most Likely Key for Byte 1 at that Single Time Index

Graph 2: Correlation value for different key guesses at the single best Time Index (513 of 2500)

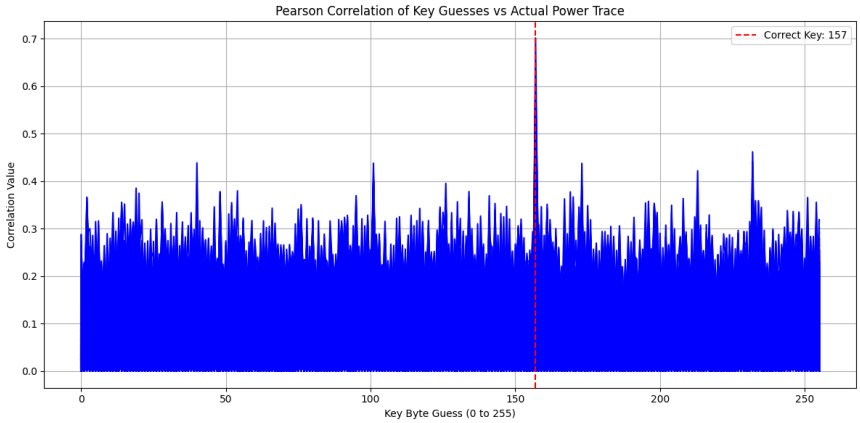
1. For each of the 256 possible values of this key byte:
    a. Code calculated predicted power values (Hamming weights) for all 110 plaintexts
    b. It computed the correlation between these predictions and the actual power measurements
    c. It found the time point (out of 2500) where this correlation was highest
2. The graph shows this maximum correlation value for each key guess
    a. The red dashed line highlights key value 72, which had the highest correlation, indicating it's likely the correct key byte value.
    b. This graph reflects the result of evaluating 256 key guesses across all 2500 time points using 110 traces. It shows the correlation values for each key guess at the single time index (513) where the correct key byte gave the strongest correlation.

4.16 Correlation Graphs

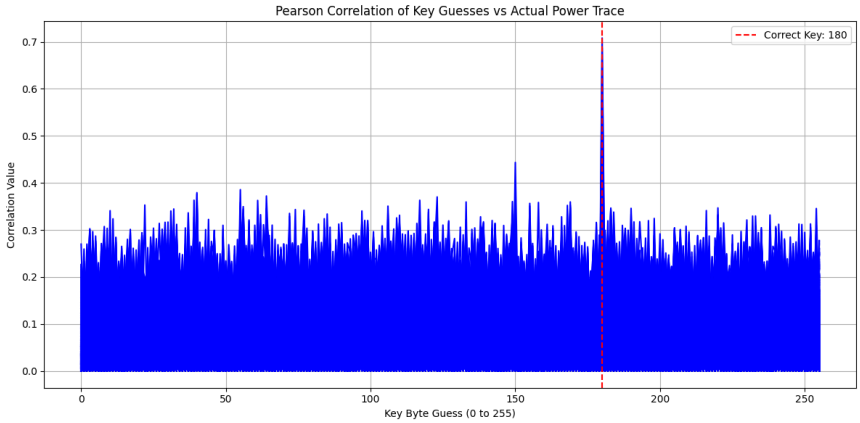
1. Correlation of Key Guess VS Actual Power Trace at a single Time Index



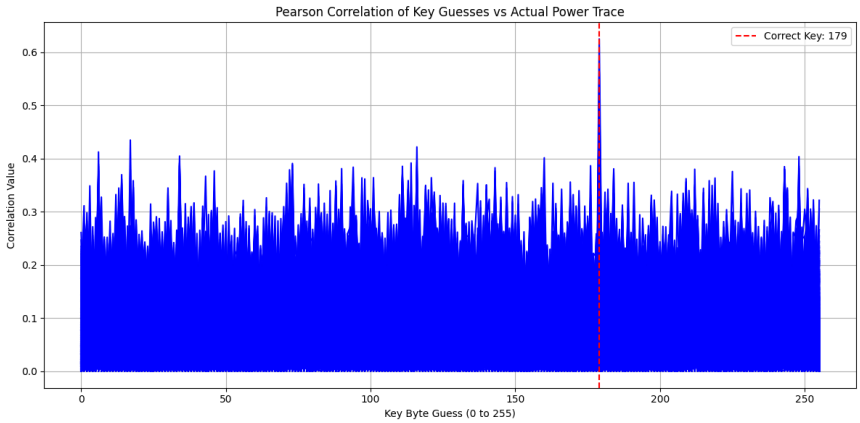
Byte 1: 0x48 (72)



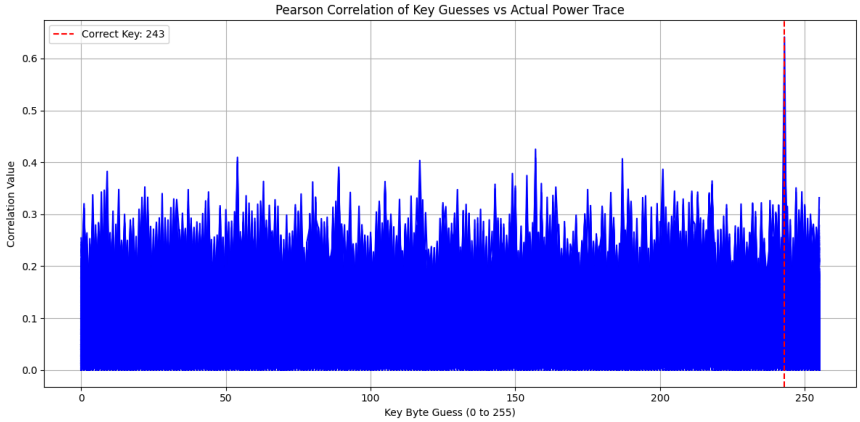
Byte 2: 0x9D (157)



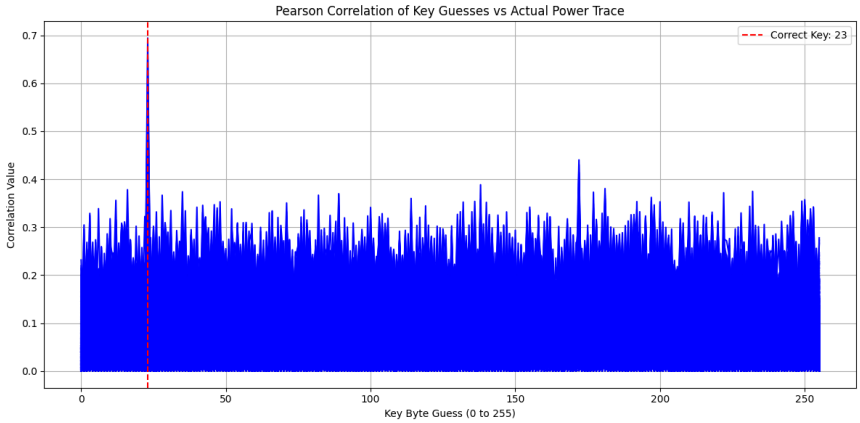
Byte 3: 0xB4 (180)



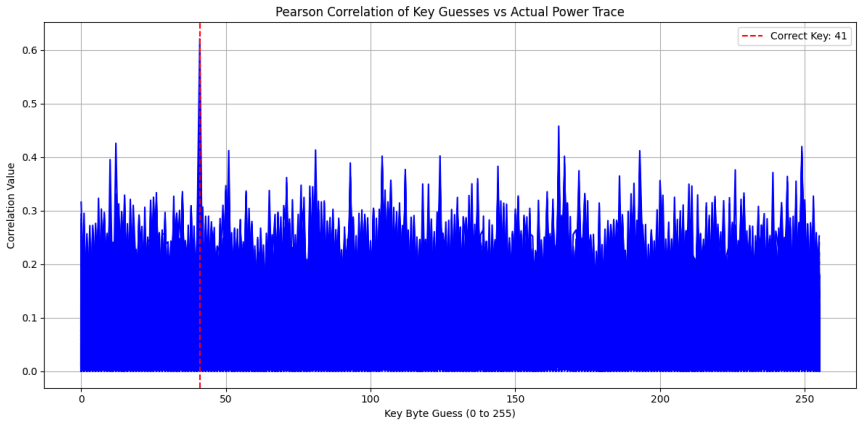
Byte 4: 0xB3 (179)



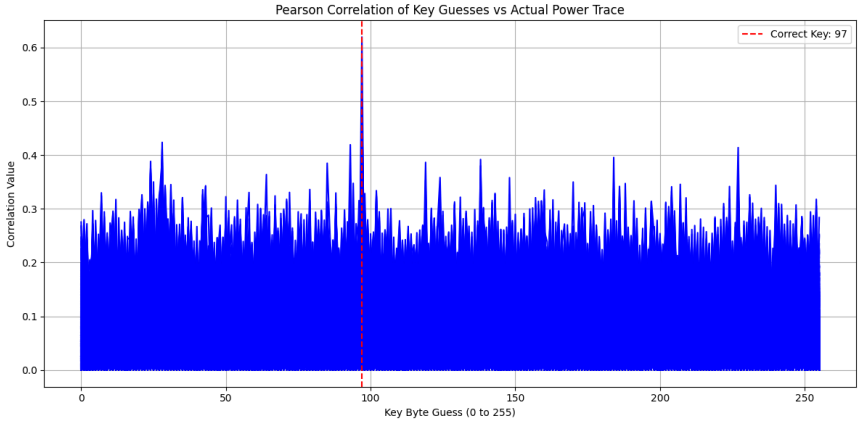
Byte 5: 0xF3 (243)



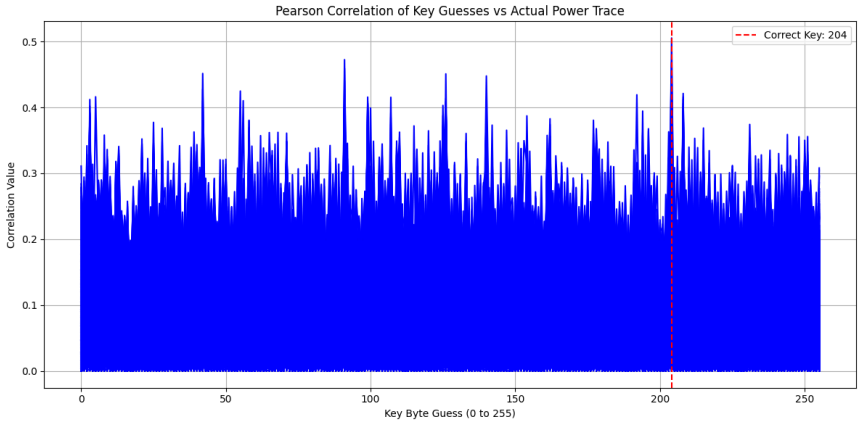
Byte 6: 0x17 (23)



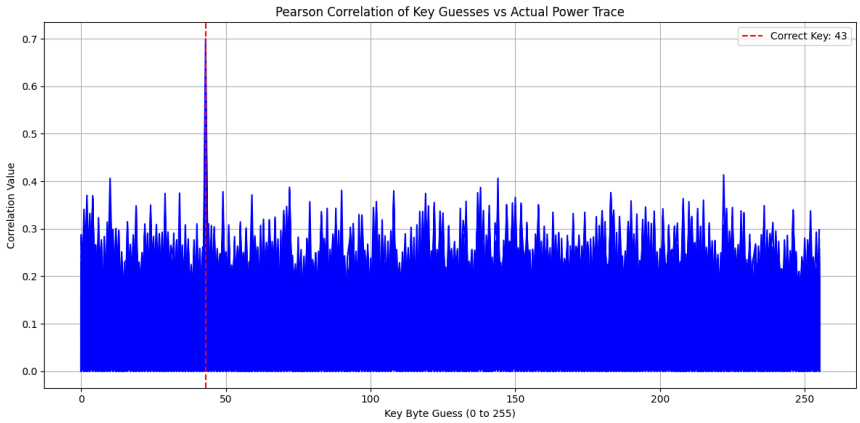
Byte 7: 0x29 (41)



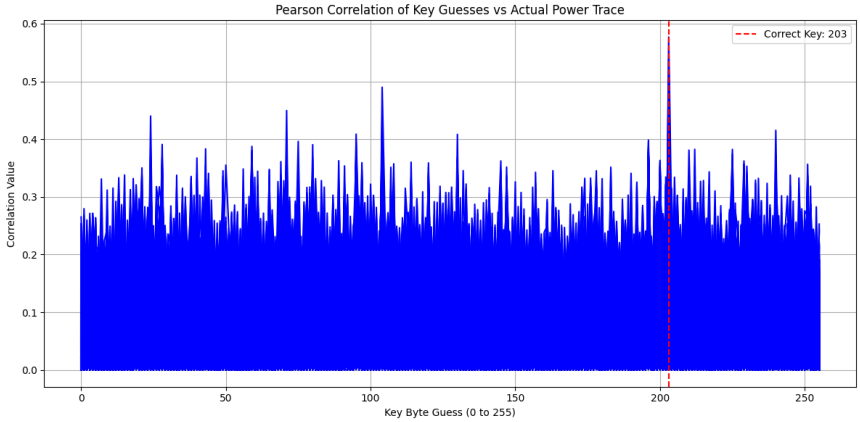
Byte 8: 0x61 (97)



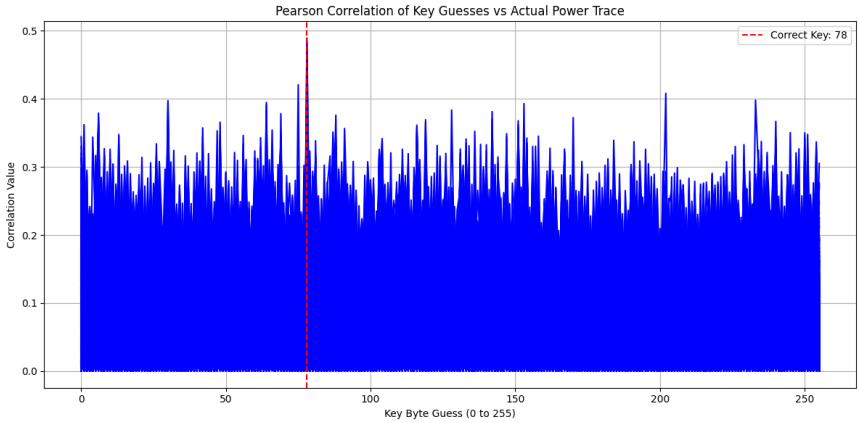
Byte 9: 0xCC (204)



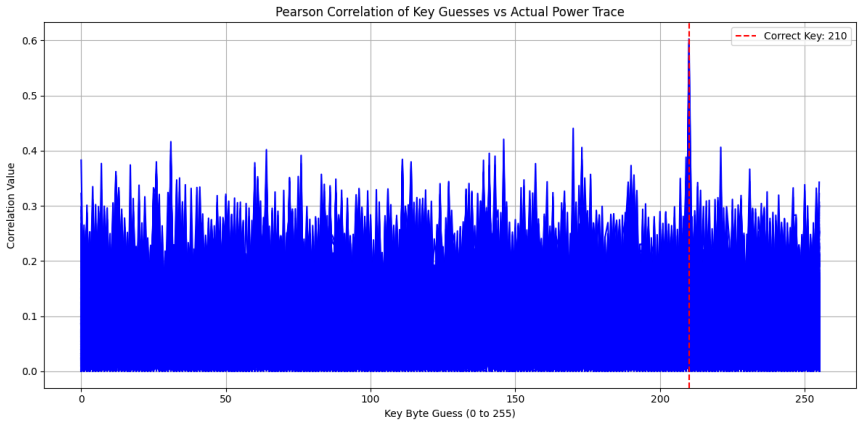
Byte 10: 0x2B (43)



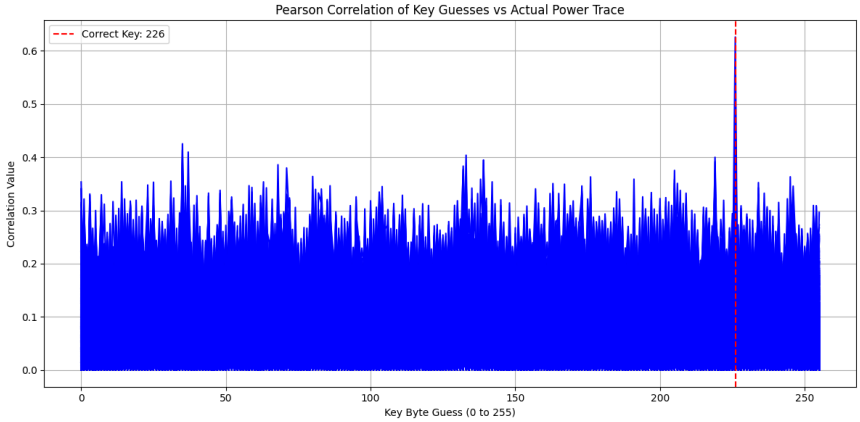
Byte 11: 0xCB (203)



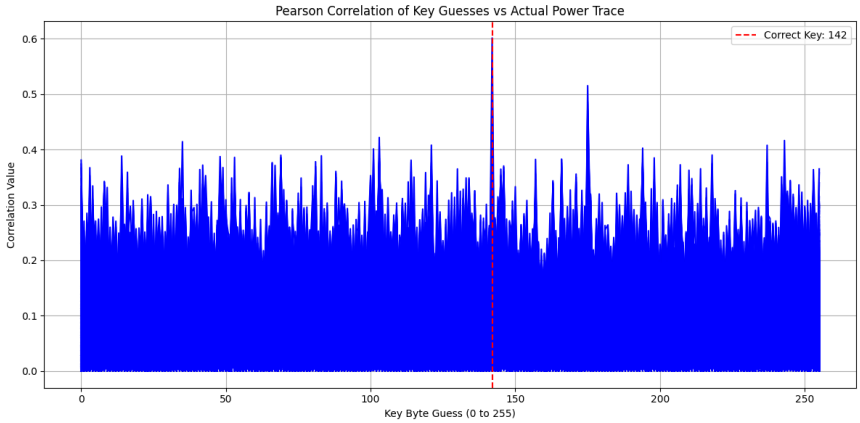
Byte 12: 0x4E (78)



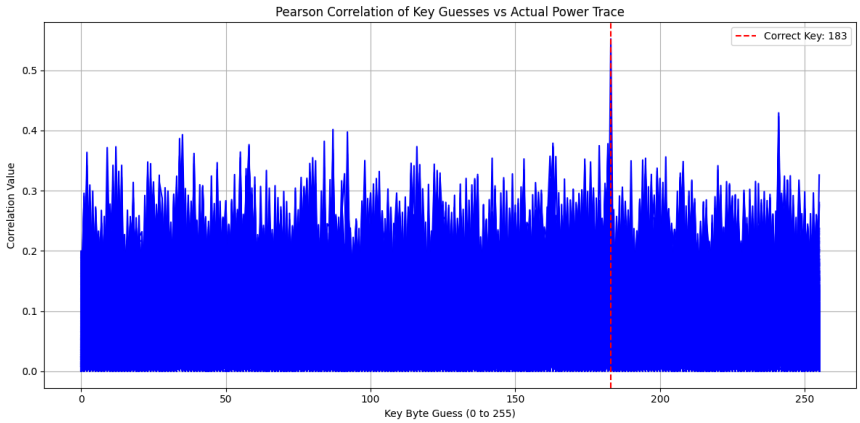
Byte 13: 0xD2 (210)



Byte 14: 0xE2 (226)

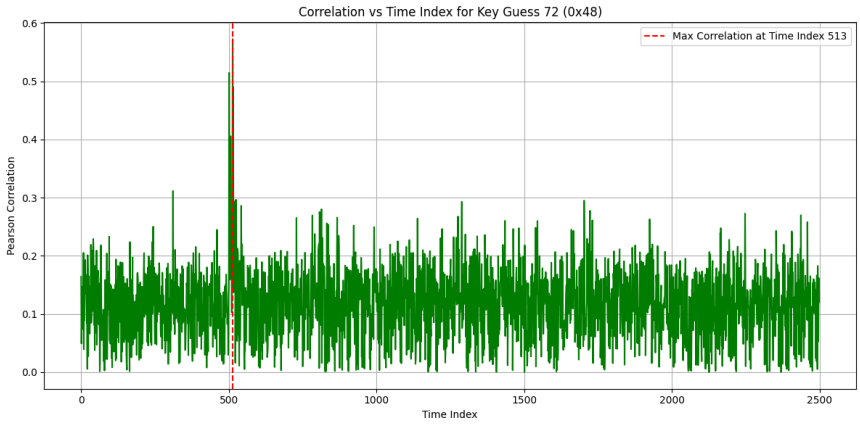


Byte 15: 0x8E (142)

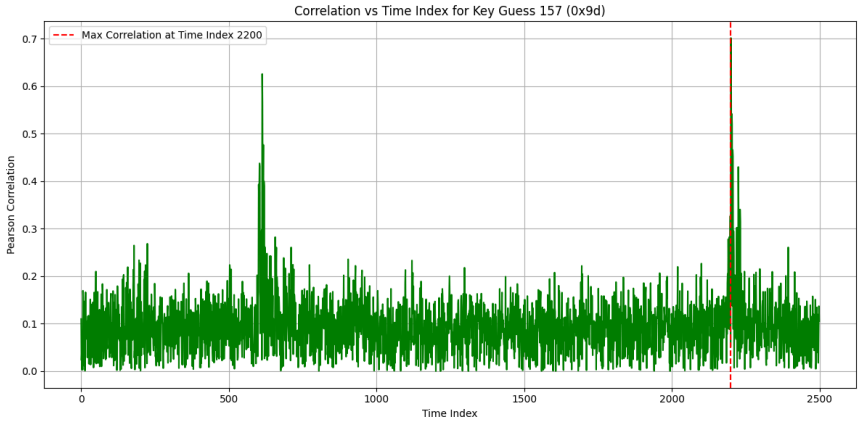


Byte 16: 0xB7 (183)

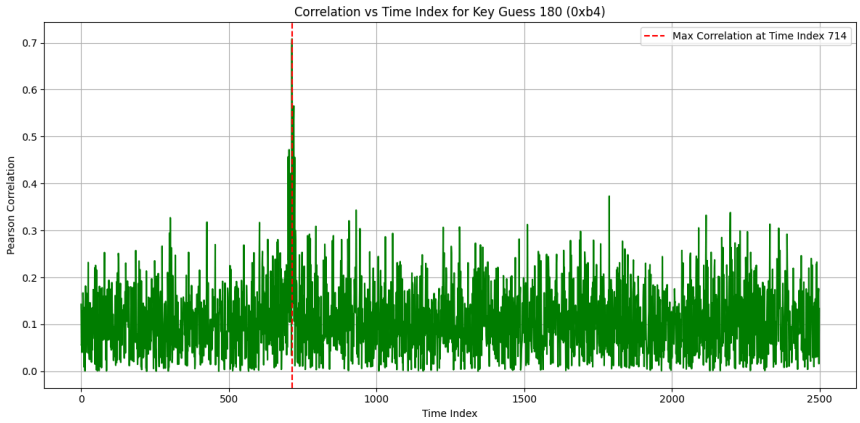
2. Correlation over 110 Samples at each Time Index



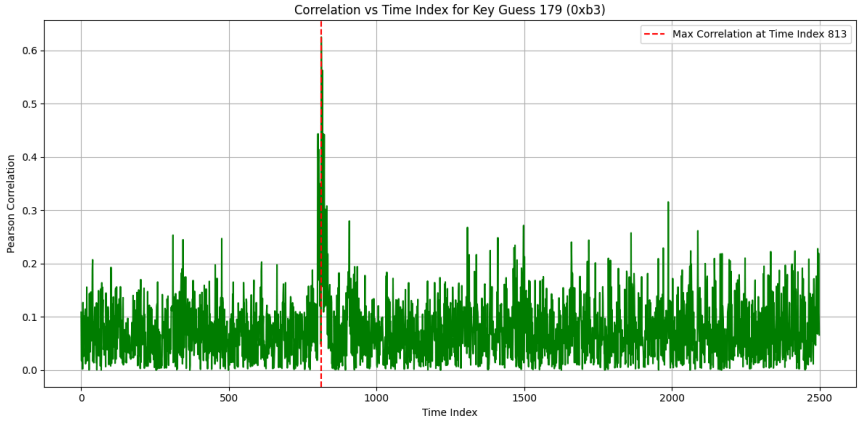
Byte 1: 0x48 (72)



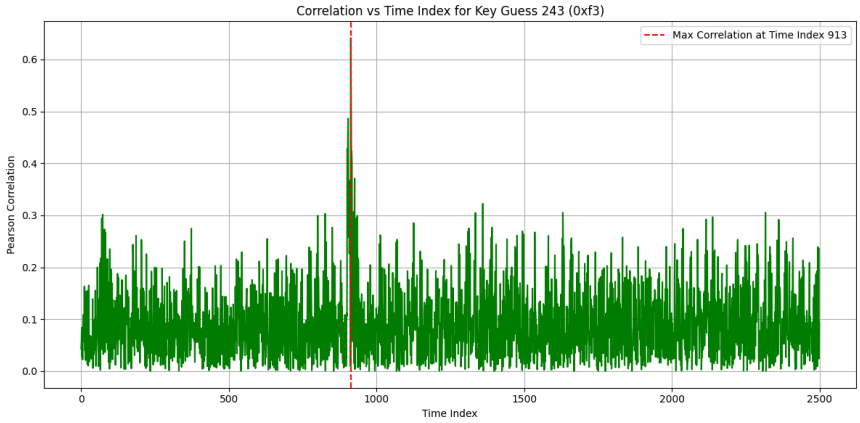
Byte 2: 0x9D (157)



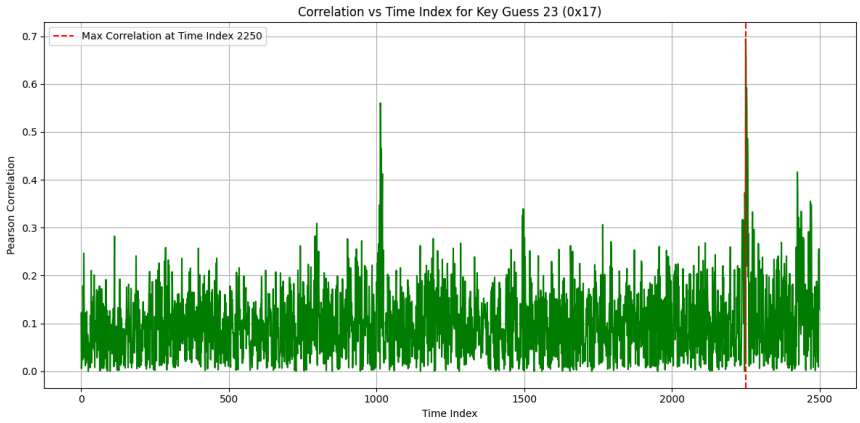
Byte 3: 0xB4 (180)



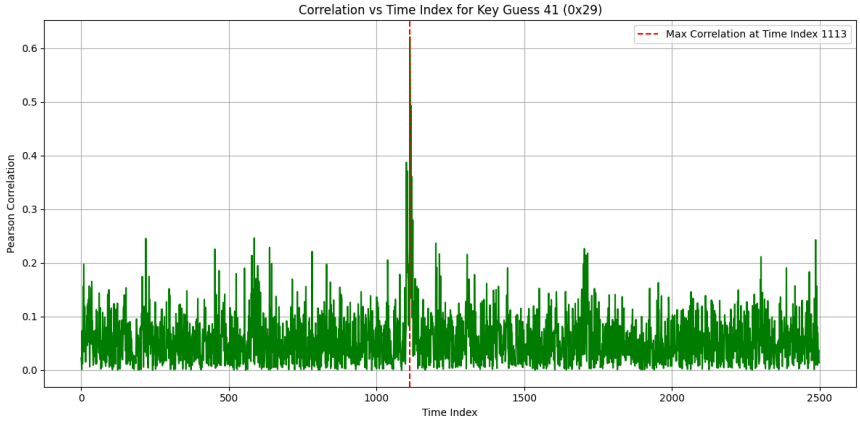
Byte 4: 0xB3 (179)



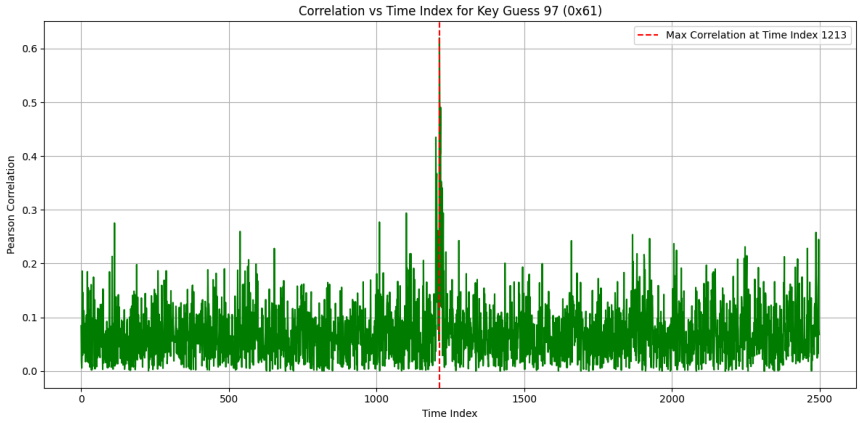
Byte 5: 0xF3 (243)



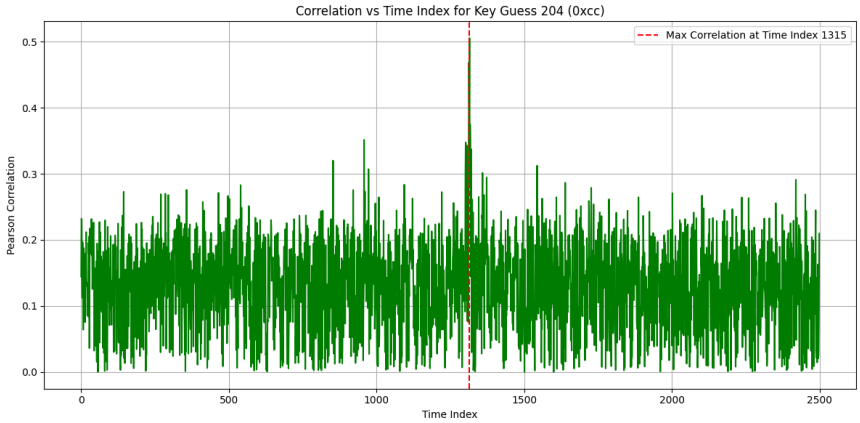
Byte 6: 0x17 (23)



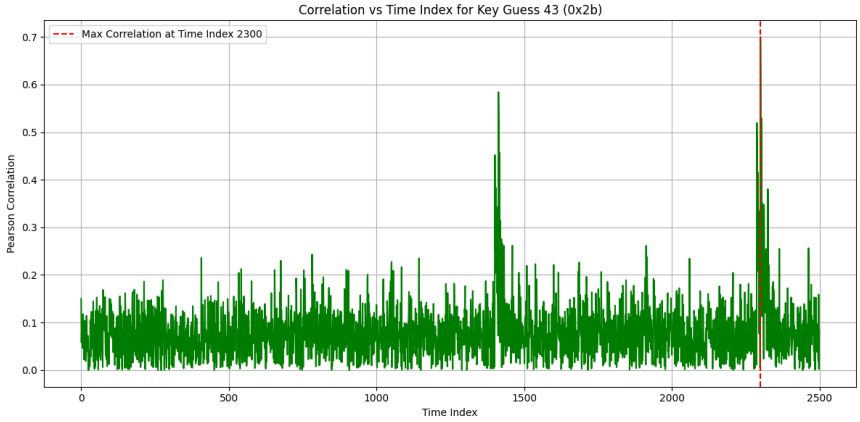
Byte 7: 0x29 (41)



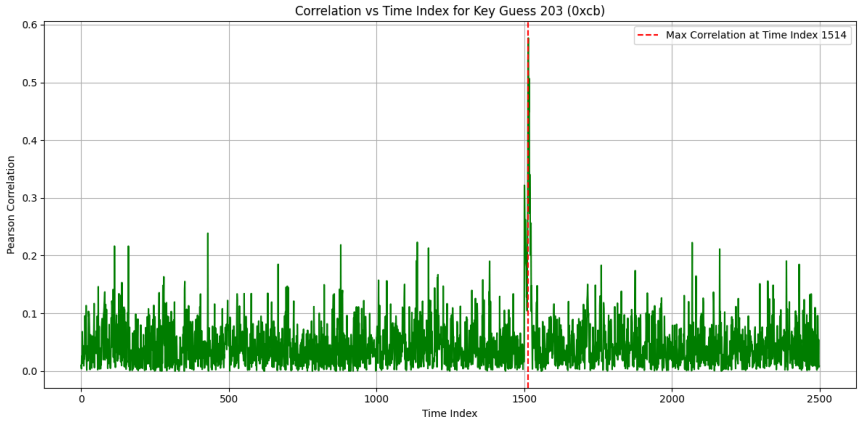
Byte 8: 0x61 (97)



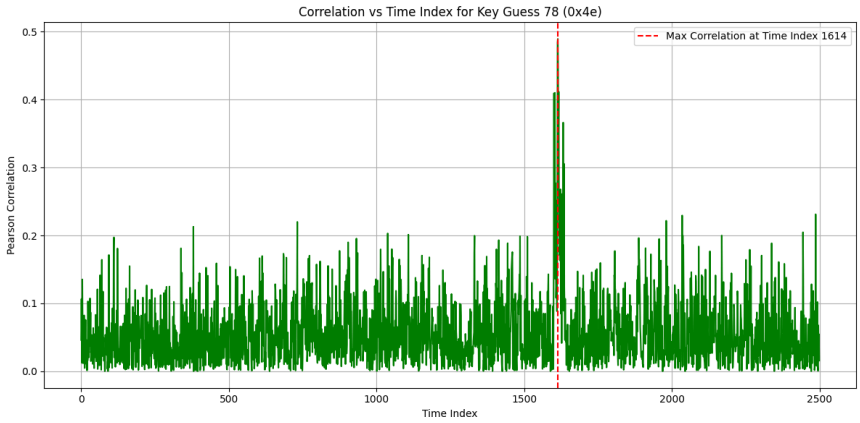
Byte 9: 0xCC (204)



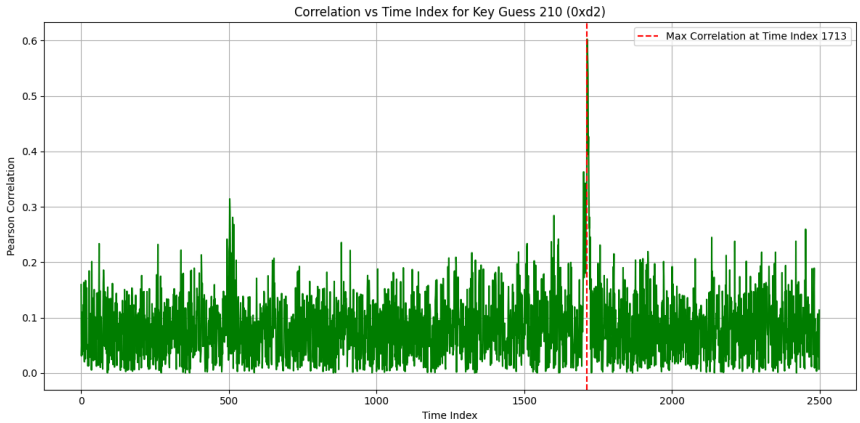
Byte 10: 0x2B (43)



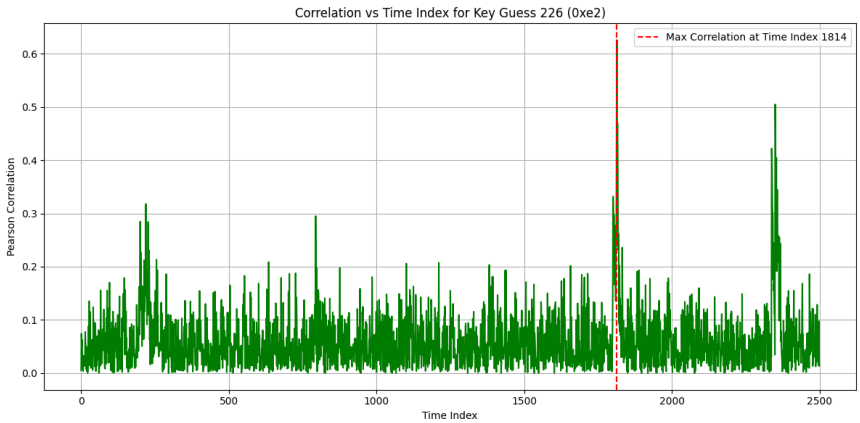
Byte 11: 0xCB (203)



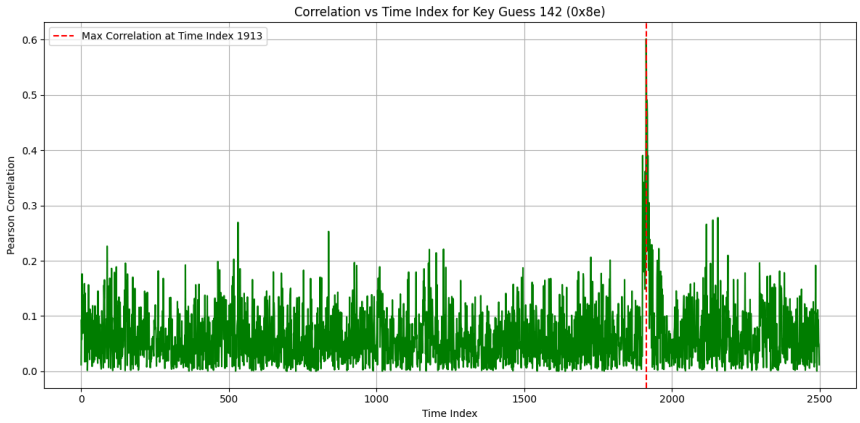
Byte 12: 0x4E (78)



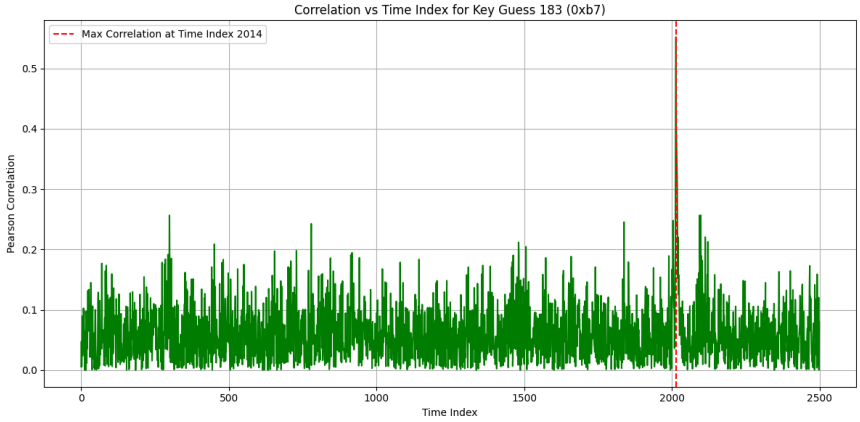
Byte 13: 0xD2 (210)



Byte 14: 0xE2 (226)



Byte 15: 0x8E (142)



Byte 16: 0xB7 (183)

5 Inference & Conclusion

5.1 Brief Summary of the Experiments

- We configured an oscilloscope to record power consumption traces from an AES-128 encryption device.
- We manually and automatically set up the instrument to ensure reliable triggering and high-fidelity captures.
- We collected numerous trace sets, each tagged with the corresponding plaintext and ciphertext.
- We applied **Correlation Power Analysis** (CPA) to systematically guess each key byte and confirmed how strong correlations reveal the correct key.

5.2 Implications and Importance

1. **Security Awareness**: Even robust encryption algorithms like AES can be vulnerable if the hardware design leaks data through side channels.
2. **Hardware Countermeasures**: Techniques such as masking, hiding, and clock randomisation can mitigate these power leakages.
3. **Real-World Impact**: Side-channel attacks pose threats to smart cards, IoT devices, and secure embedded applications. Proper device design must incorporate side-channel protection, especially in high-security contexts like payment systems.

5.3 Main Takeaways

- **Practical Vulnerability**: Cryptographic security depends not only on mathematical strength but also on physical implementation.
- **CPA Methodology**: Systematically correlates predicted power usage to actual traces, a small leak in data-dependent power consumption can reveal the secret key.
- **Future Work**: Consider investigating more advanced side-channel attack variants (e.g., differential power analysis), exploring additional countermeasures, or repeating the process on more complex hardware to gauge the effectiveness of protections.

Appendix

Figure 1: AES Encryption Sequence

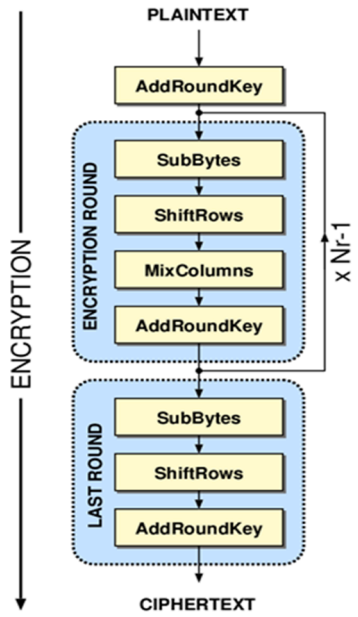


Figure 2: AES SBox

AES SBOX:

Input: $a_7a_6a_5a_4a_3a_2a_1a_0$ **Output:** $b_7b_6b_5b_4b_3b_2b_1b_0$

$a_3a_2a_1a_0$

$a_7a_6a_5a_4$	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 3: Python Script for Plain and Ciphertext Communication

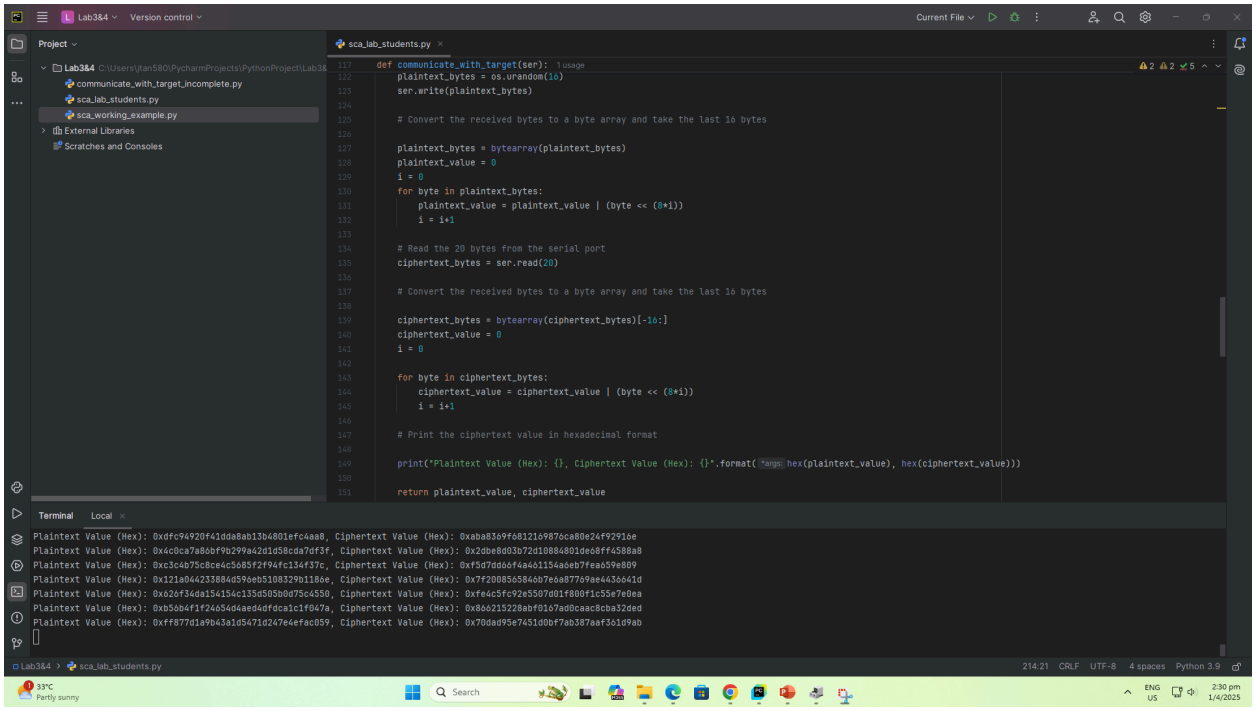


Figure 4: CPA Analysis Python Script

```
#!/usr/bin/env python


"""

Download data from a Rigol DS1052E oscilloscope and graph with matplotlib.

By Ken Shirriff, http://righto.com/rigol



Based                                                                    on
http://www.cibomahto.com/2010/04/controlling-a-rigol-oscilloscope-using-linux-
and-python/

by Cibo Mahto.

"""
```

```
import numpy as np
import matplotlib.pyplot as plot
import sys
import pyvisa as visa
import time
import serial
import struct
import os

# Define the bytes to send

fixed_bytes = bytes([0x01, 0x23, 0xab, 0xcd])

def connect_to_target():

   # Open the serial port to the target...
   ser = serial.Serial('COM4', baudrate=38400)
   time.sleep(2)
   return ser


def connect_to_scope():
   rm = visa.ResourceManager()
   instruments = rm.list_resources()
   # Get the USB device, e.g. 'USB0::0x1AB1::0x0588::DS1ED141904883'
   usb = list(filter(lambda x: 'USB' in x, instruments))
   print(usb)
     scope  =  rm.open_resource(usb[0],  timeout=1000000,  chunk_size=1024000)    #
bigger timeout for long mem
   return scope


def configure_scope(scope):

   # Setting unlimited Bandwidth...
   scope.write(":CHAN1:BWL OFF")

   # Turning Display for both channels..
   scope.write(":CHAN1:DISP ON")
   scope.write(":CHAN2:DISP ON")

   # Setting AC Coupling on Signal Channel...
   scope.write(":CHAN1:COUP AC")

   # Setting the Scale...
   scope.write(":CHAN1:SCAL <Enter value here in volts>")
   scope.write(":CHAN2:SCAL <Enter value here in volts>")

   # Setting the offset...
```

```
    scope.write(":CHAN1:OFFS <Enter value here in volts>")
    scope.write(":CHAN2:OFFS <Enter value here in volts>")

    # Setting the Acquisition Properties...
    scope.write(":ACQ:MDEP <Enter number of samples you want to capture>")
    scope.write(":ACQ:TYPE NORM")

    # Setting the timescale properties...

    scope.write(":TIM:MAIN:SCAL <Enter the scale in seconds>")
    scope.write(":TIM:MAIN:OFFS <Enter the offset in seconds>")

    # Setting Trigger Mode...
    scope.write(":TRIG:MODE EDGE")

    # Setting Trigger Mode to Channel 2 Edge...
    scope.write(":TRIG:EDG:SOUR CHAN2")

    # Setting Trigger Mode to Positive Edge...
    scope.write(":TRIG:EDG:SLOP POS")

    # Setting Trigger Level...
    scope.write(":TRIG:EDG:LEV 1.98")

    # Setting Trigger Coupling: DC
    scope.write(":TRIG:COUP DC")

    # Setting Trigger to Single Mode...
    scope.write(":TRIG:SWE SING")

    # Setting the waveform for acquisition...
    scope.write(":WAV:SOUR CHAN1")
    scope.write(":WAV:MODE RAW")
    scope.write(":WAV:FORM ASCII")

    # Setting waveform startpoint...
    scope.write(":WAV:STAR <Enter start sample index you want to capture")

    # Setting waveform endpoint...
    scope.write(":WAV:STOP <Enter final sample index you want to capture>")


def arm_scope(scope):
    # Set the scope in Single Mode...

    scope.write(":SING")
    print("Waiting to be armed")
    time.sleep(2)
    print("Wait completed...")
```

```
def communicate_with_target(ser):
   ser.write(fixed_bytes)

   # Generate 16 random bytes

   plaintext_bytes = os.urandom(16)
   ser.write(plaintext_bytes)

   # Convert the received bytes to a byte array and take the last 16 bytes

   plaintext_bytes = bytearray(plaintext_bytes)
   plaintext_value = 0
   i = 0
   for byte in plaintext_bytes:
       plaintext_value = plaintext_value | (byte << (8*i))
       i = i+1

   # Read the 20 bytes from the serial port
   ciphertext_bytes = ser.read(20)

   # Convert the received bytes to a byte array and take the last 16 bytes

   ciphertext_bytes = bytearray(ciphertext_bytes)[-16:]
   ciphertext_value = 0
   i = 0

   for byte in ciphertext_bytes:
       ciphertext_value = ciphertext_value | (byte << (8*i))
       i = i+1

   # Print the ciphertext value in hexadecimal format

         print("Plaintext   Value   (Hex):   {},   Ciphertext   Value   (Hex):
{}".format(hex(plaintext_value), hex(ciphertext_value)))

   return plaintext_value, ciphertext_value


def store_trace(scope):
   # Now, you need to store the trace...

   scope.write(":WAV:SOUR <Enter channel you want to capture>")
   scope.write(":WAV:MODE RAW")
   scope.write(":WAV:FORM BYTE")

   # Setting waveform startpoint...
   scope.write(":WAV:STAR <Enter start sample index you want to capture")
```

```
    # Setting waveform endpoint...
    scope.write(":WAV:STOP <Enter final sample index you want to capture>")

    rawdata = scope.query(":WAV:DATA? <Enter channel you want to capture>")
    print(rawdata)

    data_size = len(rawdata)
    timescale = []

    for i in range(0, data_size):
        timescale.append(i)

    plot.plot(timescale, rawdata)
    plot.show()

    return rawdata


def string_to_numpy_array(input_string):
      #  Split  the  input  string  by  commas  and  strip  any  leading/trailing
whitespaces

    numbers_str = input_string.split(',')
    numbers_str = [num.strip() for num in numbers_str]
    # Convert the string numbers to floating-point numbers
    numbers_float = [float(num) for num in numbers_str]
    # Convert the list of floating-point numbers to a NumPy array
    result_array = np.array(numbers_float)
    return result_array


# Main Function Starts here...

def main():

    # Connect to target...
    ser_target = connect_to_target()

    # Connect to Oscilloscope
    # scope = connect_to_scope()

    # Configure Oscilloscope
    # configure_scope(scope)

    num_traces = 100000000000
    for i in range(0, num_traces):
        # Make scope wait for trigger....Put scope in wait state...
        # arm_scope(scope)
```

```
        # Communicate with Target to send plaintext and receive Ciphertext...
        plaintext, ciphertext = communicate_with_target(ser_target)
        time.sleep(1)

        # Store the captured trace on your PC
        # trace_got = store_trace(scope)

main()
```