Hardware Security Lab Report

1 February 2025

Ernest Adjei, Luca Volonterio, Mohmmad Qasim Hussaini

Contents

Introduction

The main goal of this course was to understand how side-channel attacks and fault injection attacks work and to learn how to protect devices from them. Through hands-on experiments, we explored real-world attacks that hackers can use to extract sensitive data or bypass security mechanisms.

One of the key topics we studied was side-channel attacks, which do not directly break encryption but instead use information from the physical behavior of a device, such as power consumption or electromagnetic emissions. In our experiments, we used power analysis to reveal password characters by observing how a device's energy use changed with different inputs. We also explored Differential Power Analysis (DPA), where small differences in power consumption helped us recover encryption keys, such as those used in AES encryption.

Another major topic was fault injection attacks, which involve disturbing a device's normal operation to make it behave unexpectedly. We used clock glitching to shorten clock cycles and bypass security checks, and we tested password bypass techniques to exploit weaknesses in authentication systems. These experiments showed us how attackers can manipulate hardware to gain unauthorized access to systems.

This course helped us understand that hardware security is just as important as software security. While strong encryption algorithms exist, their implementation in hardware can still be vulnerable to physical attacks.

1 Method

What is ChipWhisperer?

ChipWhisperer is a full open-source toolchain designed to help learn about side channel attacks on embedded devices and test how resistant these devices are to such attacks. Specifically, ChipWhisperer focuses on power analysis, which uses information from a device's power consumption to carry out an attack. It also deals with voltage and clock glitching attacks, which temporarily interfere with a device's power or clock to make it behave in unexpected ways (such as bypassing a password check).

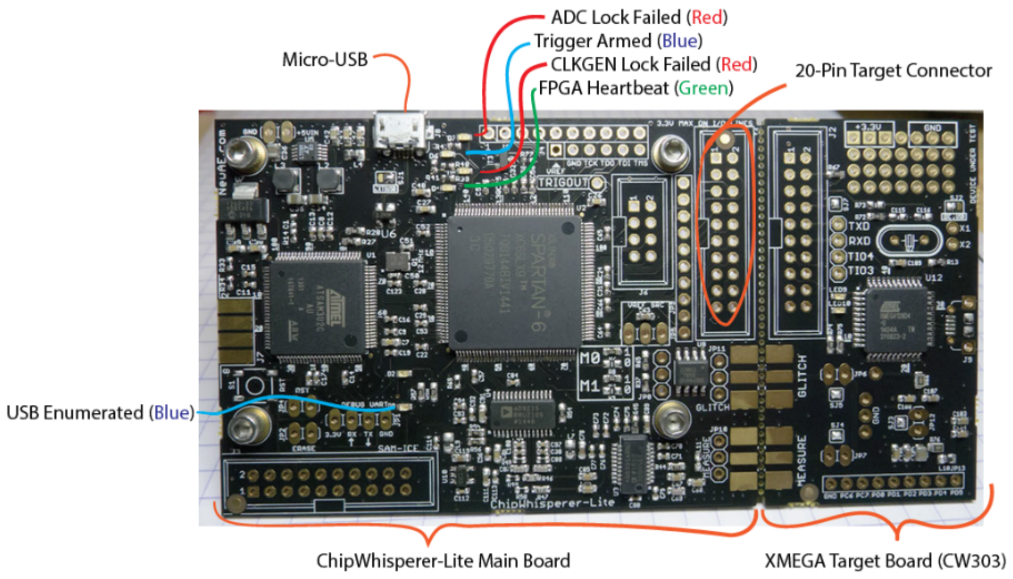
Components and Setup:

ChipWhisperer consists of four layers of open-source components:

Hardware

The scope board captures the side-channel data from the target device. It measures signals such as voltage changes or current flow. By analyzing these signals, attackers can gather important information about the target device, such as secret keys or other private data. The scope board is designed to record this data accurately, making it effective for side-channel analysis.

The target board is the embedded device that is being tested. This device performs tasks, such as encryption or decryption, which are vulnerable to side-channel attacks. The target board could be a microcontroller or another small computer. During these tasks, the device may leak important information through signals like power consumption or electromagnetic emissions.



Firmware

ChipWhisperer also includes open source firmware for both scopes and targets.

Software

The ChipWhisperer software controls the setup, including the target and scope boards. It allows us to start and manage side-channel attacks, such as Differential Power Analysis (DPA) or fault injection. The software automates these attacks, making it easier to repeat tests and examine different security weaknesses in the target device.

Tutorial

ChipWhisperer includes Jupyter Notebook tutorials that teach us about side-channel attacks. These notebooks provide step-by-step guides to perform various attacks and show us how to use the ChipWhisperer Python API.

CWLITEXMEGA Setup

In the CWLITEXMEGA setup, the CWLITEXMEGA board serves as the target board. It often uses a microcontroller which performs tasks such as cryptographic operations. The scope board records side-channel data from the target device. The ChipWhisperer software controls the process, running attacks, collecting data, and testing how secure the device is against these attacks.

2 Side Channel Attacks

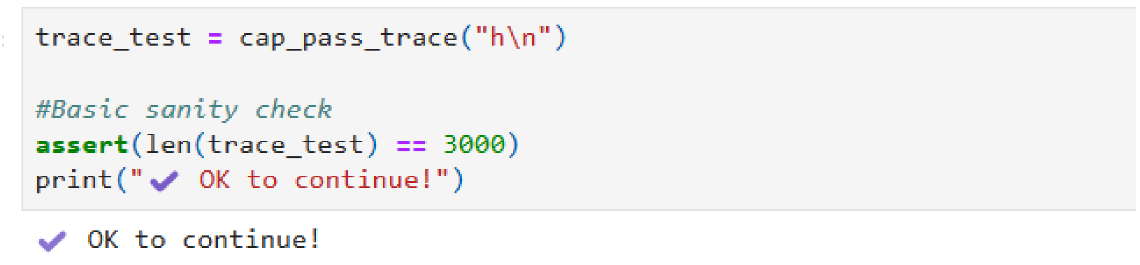
In cryptography, side channel attacks are another way to get secret information. These attacks do not target the mathematical design of the encryption but instead use physical clues like timing, power use, or electromagnetic signals. Most devices are implemented using semiconductor logic gates that are built out of transistors. Electrons move throughout the silicon substrate when charge is applied to, or removed from, a transistor's gate. This flow of electrons consume power and generate electromagnetic radiation.

2.1 Power Analysis for Password Bypass

In this lab, we examine how hardware components are vulnerable to power analysis attacks in password verification. Our objective is to analyze power consumption patterns to identify when a device performs specific operations and potentially bypass authentication.

2.1.1 Target and Attack Methodology

Power Trace Gathering: To perform power trace capture, we used ChipWhisperer hardware. The trace was collected directly from the physical device. Our setup was validated using a basic sanity check to ensure the correctness of the acquired trace, 3000 samples:



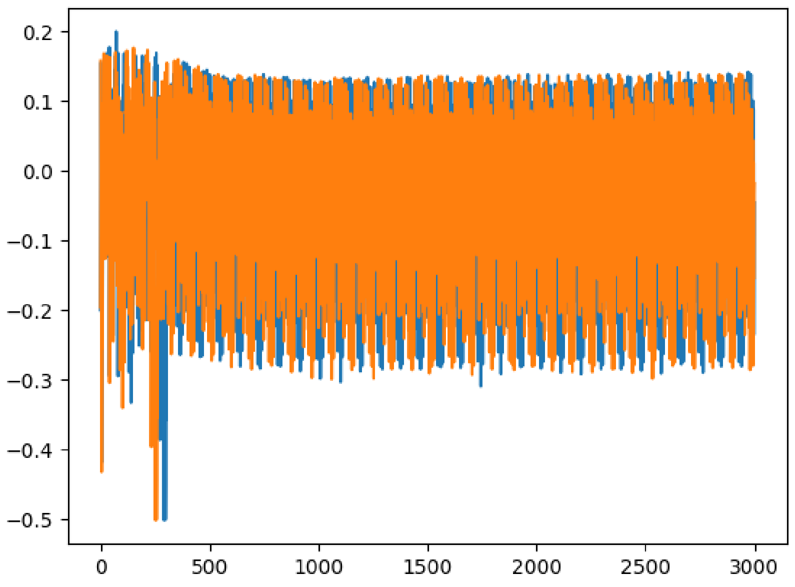
```
trace_test = cap_pass_trace("h\n")

#Basic sanity check
assert(len(trace_test) == 3000)
print("✔ OK to continue!")
```

```
✔ OK to continue!
```

2.1.2 Exploration

I) Known Password Prefix "h" Against "0"

We initially examined power traces for different input characters, focusing on a known password prefix ('h') to identify significant variations. A comparative plot was generated by capturing traces for different input values:

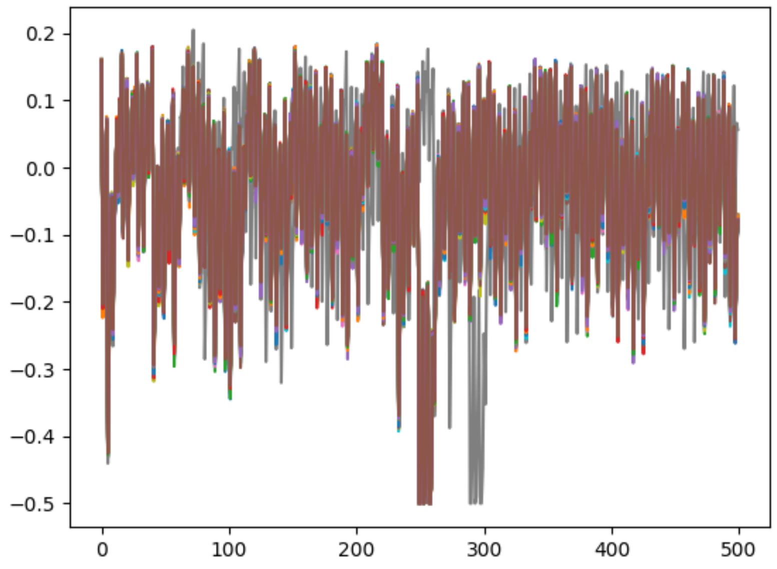


Observation

We observed that both inputs, ”h” and ”0,” exhibited identical power trace waveforms, indicating similar computational processing within the target system. This suggests that these characters may be valid components of the five-character password.

II) Testing All Characters Within the Defined Set for our Password Recognition

We systematically tested all possible password characters (a-z, 0-9) to map the target's power response. By limiting the sample range to the first 500 data points, we refined the analysis to the most relevant sections of the trace, improving efficiency and clarity.

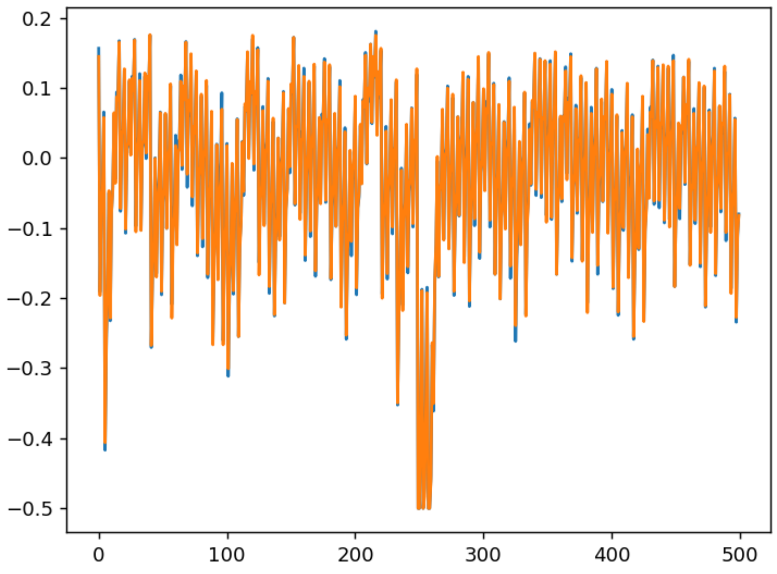


Observation

Upon analyzing the power traces of all tested characters (a-z, 0-9), we observed that the majority of characters follow a consistent, identifiable pattern. However, a notable outlier trace deviates from the expected trajectory, suggesting the presence of a correct component of the password.

III) Automating an Attack on a Single Character

To automate password extraction, we compared power traces of valid and invalid characters. Using a null byte ('00') as a baseline reference, we measured differences in power traces:

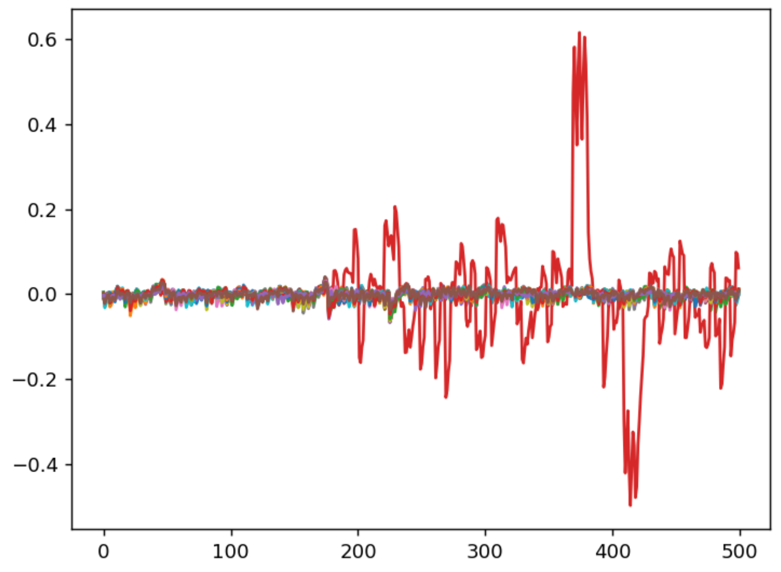


Observation

The consistent waveform patterns indicate that neither 0x00 nor c contributes to a valid password, as both activate the same rejection mechanism. This implies that the system treats them as incorrect components, emphasizing that a valid password adheres to a specific processing path. This approach revealed clear deviations in power consumption, allowing us to distinguish valid password inputs from incorrect ones.

IV) Analyzing the Difference Between a Known Reference and Varied Captured Traces

To conduct a more in-depth analysis of the password verification process, we compare the power traces of a known reference input (h0p00) with those of various character inputs (h0pX, where X ranges from a to z and 0 to 9). By plotting the differences between each captured trace and the reference trace, we can highlight variations in power consumption. These variations may indicate distinct processing behaviors, which could reveal patterns that differentiate correct password components from incorrect ones.

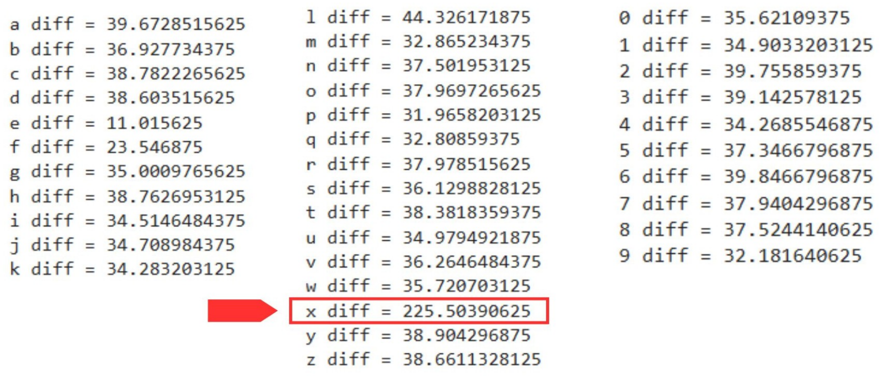


Observation

The comparison of power trace differences between the reference input (h0p00) and various test inputs (h0pX, where X ranges from a to z and 0 to 9) reveals a consistent pattern across the majority of inputs. Most traces exhibit minor deviations around a near-zero baseline, suggesting a uniform processing behavior for incorrect password components. However, a notable outlier is observed in the red trace, which shows significant deviations from the baseline within the 200 to 450 sample range, marked by pronounced peaks and fluctuations. This outlier strongly indicates that the corresponding character in the test input activates a distinct computational pathway, likely signifying a correct password component.

V) Using Large Differences to Determine Character Variations in Password Traces

We begin with a reference trace generated from the input ”h0p00” and sequentially compare it to traces captured from various alphanumeric characters (a-z, 0-9) added to the string ”h0p.” By calculating the absolute differences in power consumption between each trace and the reference, we seek to identify patterns that may reveal the distinct processing behaviors linked to valid and invalid password components.



```
a diff = 39.6728515625
b diff = 36.927734375
c diff = 38.7822265625
d diff = 38.603515625
e diff = 11.015625
f diff = 23.546875
g diff = 35.0009765625
h diff = 38.7626953125
i diff = 34.5146484375
j diff = 34.708984375
k diff = 34.283203125
l diff = 44.326171875
m diff = 32.865234375
n diff = 37.501953125
o diff = 37.9697265625
p diff = 31.9658203125
q diff = 32.80859375
r diff = 37.978515625
s diff = 36.1298828125
t diff = 38.3818359375
u diff = 34.9794921875
v diff = 36.2646484375
w diff = 35.720703125
x diff = 225.50390625
y diff = 38.904296875
z diff = 38.6611328125
0 diff = 35.62109375
1 diff = 34.9033203125
2 diff = 39.755859375
3 diff = 39.142578125
4 diff = 34.2685546875
5 diff = 37.3466796875
6 diff = 39.8466796875
7 diff = 37.9404296875
8 diff = 37.5244140625
9 diff = 32.181640625
```

Observation

The majority of differences remain within a narrow range, indicating a similar computational behavior when processing incorrect password components. Nonetheless, a prominent outlier is observed with the character ’x’, highlighted in red. The difference value for ’x’ (225.50390625) is considerably larger than those of all other characters, which fall below 50. This outlier suggests that the target performs additional or distinct processing when ’x’ is present, indicating that it is likely a correct component of the password. The increased power consumption may be associated with internal operations, such as the verification logic that identifies valid characters.

VI) Running a Full Attack

At this stage, we refine our approach to brute-force each character of the password. We begin with an initial guess and capture the power trace for the partially constructed password combined with a null byte (’00’). By iterating through potential character candidates and comparing the resulting traces to a reference, we calculate the absolute differences in power consumption. If the difference exceeds a predefined threshold, we update our guessed password and continue the process. This method enables us to progressively identify each character of the password based on variations in power consumption.

Observation

By progressively constructing the password based on observed power trace variations, we successfully bypassed authentication mechanisms.

Conclusion and Countermeasures

This experiment demonstrated the feasibility of power analysis attacks on password-based authentication. By leveraging variations in power consumption, we successfully identified valid input sequences. These findings highlight the need for countermeasures such as power obfuscation, noise injection, and constant-time execution strategies to mitigate such vulnerabilities.

2.2 Large Hamming Weight Swings

In this lab, we investigate how specific data values impact power consumption, highlighting potential security vulnerabilities. We demonstrate techniques for modeling device behavior using power measurements, detecting single-bit values, and conducting Differential Power Analysis (DPA) to compromise AES encryption.

2.2.1 Target and Attack Methodology

Power Trace Gathering: We collected power traces using ChipWhisperer hardware, which allowed us to measure the power consumption directly from the physical device.

2.2.2 Exploration

I) Grouping Traces based on High Hamming Weight and Low Hamming Weight

We hypothesize that manipulating the bits on data lines influences power consumption. To investigate this, we examine the target device as it encrypts data with high (0xFF) and low (0x00) Hamming weights. Given that the target device runs AES, we expect to observe a discernible difference in power traces corresponding to these variations, thereby confirming the effect of data-dependent power consumption.

a) Separating Traces into Two Groups: one list and zero list

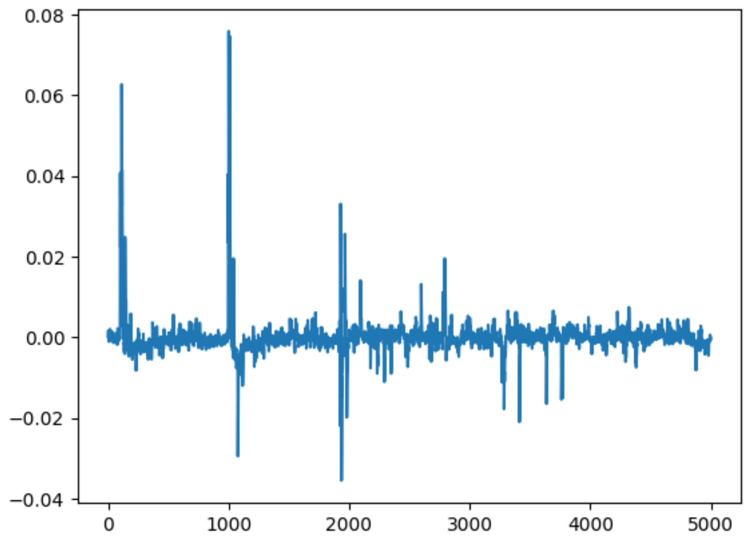
To examine the correlation between power consumption and data manipulation, we categorize the collected power traces into two distinct groups: one list and zero list. This classification is based on the hamming weight of the input data, with traces corresponding to an input byte of 0x00 assigned to one list, while all other traces are placed in zero list.

b) Finding average of the traces in time

Next, we calculate the mean of two distinct sets of traces: one corresponding to 0xFF transmissions and the other to 0x00 transmissions. Given that the assignment of traces to these groups was random and potentially uneven, averaging across each time point helps maintain consistency in the analysis. This method reduces noise, minimizes the impact of outliers, and enhances the clarity of the signal.

c) Using difference of means and plotting the resulting data:

Lastly, we calculate and visualise the difference between two averaged datasets, one avg and zero avg.

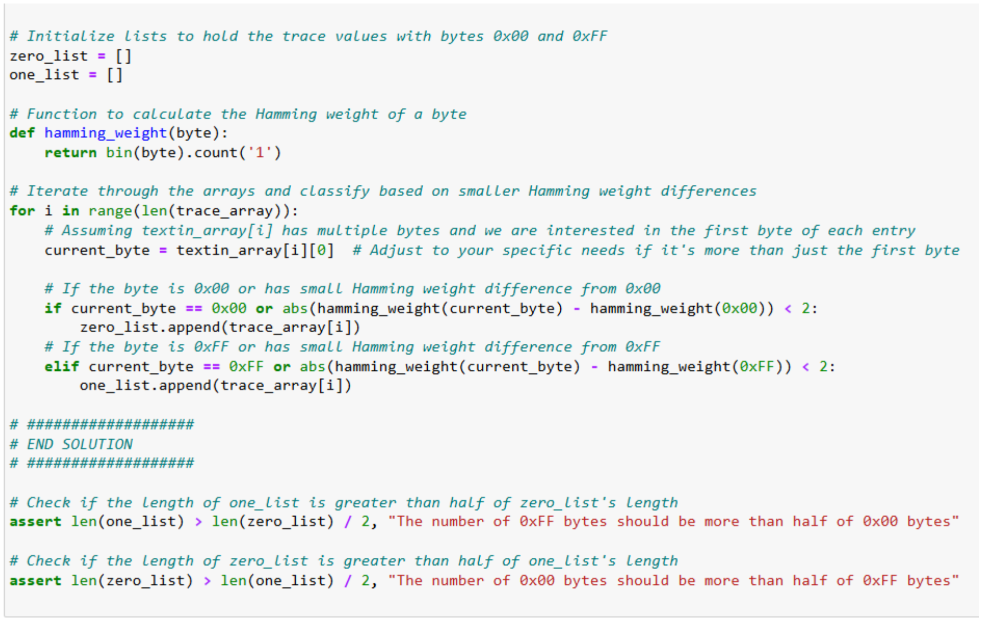


Observation

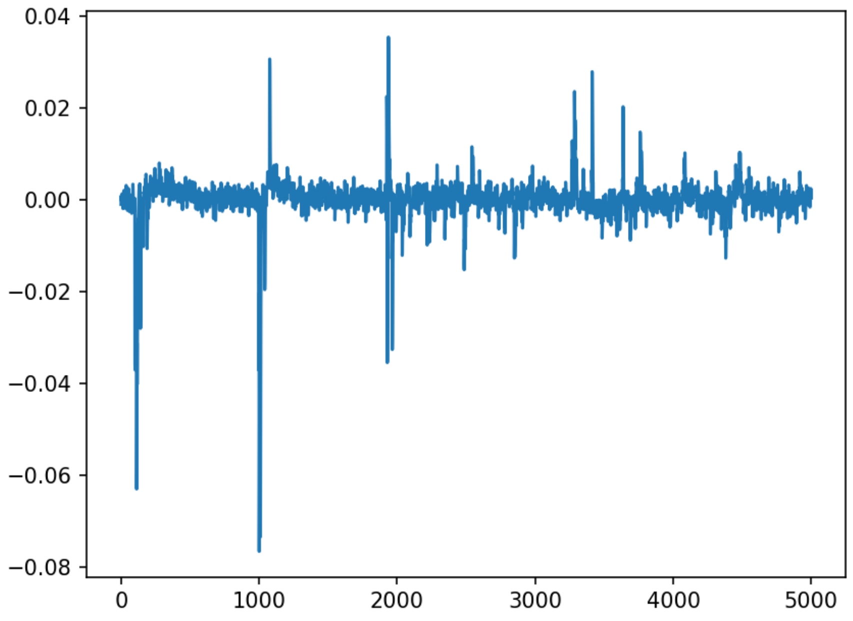
By grouping the traces into those with high (0xFF) and low (0x00) Hamming weights, we can observe a measurable difference. Significant spikes, particularly in the initial segment of the trace, suggest the presence of data-dependent power variations. This indicates that the microcontroller's power consumption is influenced not only by the execution of instructions but also by the specific data being processed.

II) Using smaller hamming weight differences

By categorizing traces according to smaller Hamming weight differences from 0x00 to 0xFF, the experiment seeks to determine whether the observed spikes remain distinct.



```python
# Initialize lists to hold the trace values with bytes 0x00 and 0xFF
zero_list = []
one_list = []

# Function to calculate the Hamming weight of a byte
def hamming_weight(byte):
    return bin(byte).count('1')

# Iterate through the arrays and classify based on smaller Hamming weight differences
for i in range(len(trace_array)):
    # Assuming textin_array[i] has multiple bytes and we are interested in the first byte of each entry
    current_byte = textin_array[i][0]  # Adjust to your specific needs if it's more than just the first byte

    # If the byte is 0x00 or has small Hamming weight difference from 0x00
    if current_byte == 0x00 or abs(hamming_weight(current_byte) - hamming_weight(0x00)) < 2:
        zero_list.append(trace_array[i])
    # If the byte is 0xFF or has small Hamming weight difference from 0xFF
    elif current_byte == 0xFF or abs(hamming_weight(current_byte) - hamming_weight(0xFF)) < 2:
        one_list.append(trace_array[i])

# ###################
# END SOLUTION
# ###################

# Check if the length of one_list is greater than half of zero_list's length
assert len(one_list) > len(zero_list) / 2, "The number of 0xFF bytes should be more than half of 0x00 bytes"

# Check if the length of zero_list is greater than half of one_list's length
assert len(zero_list) > len(one_list) / 2, "The number of 0x00 bytes should be more than half of 0xFF bytes"
```

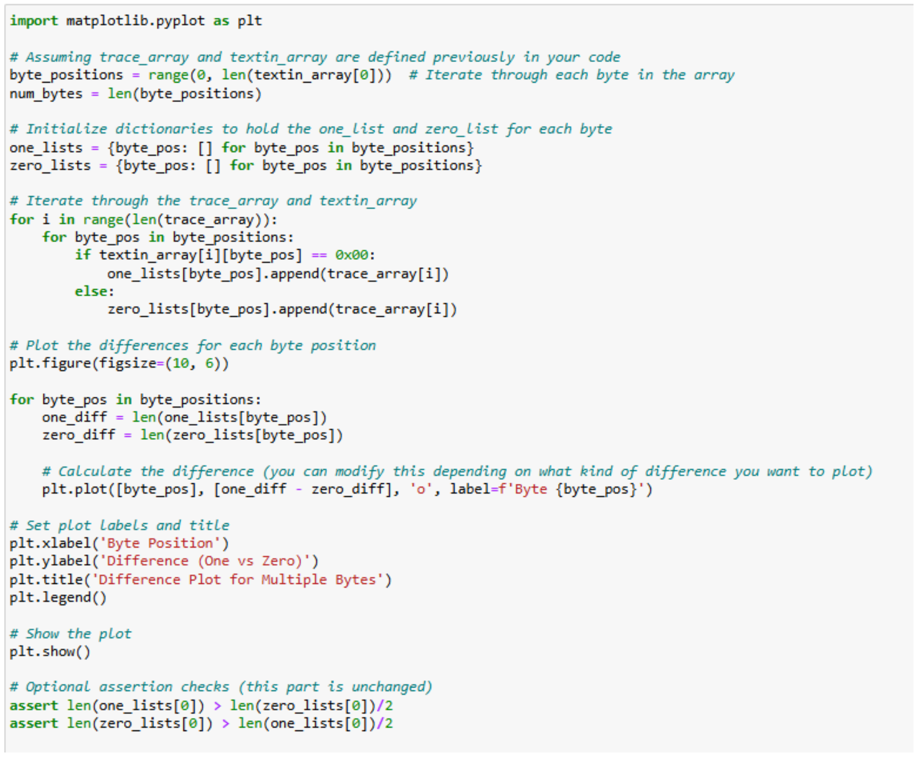


Observation

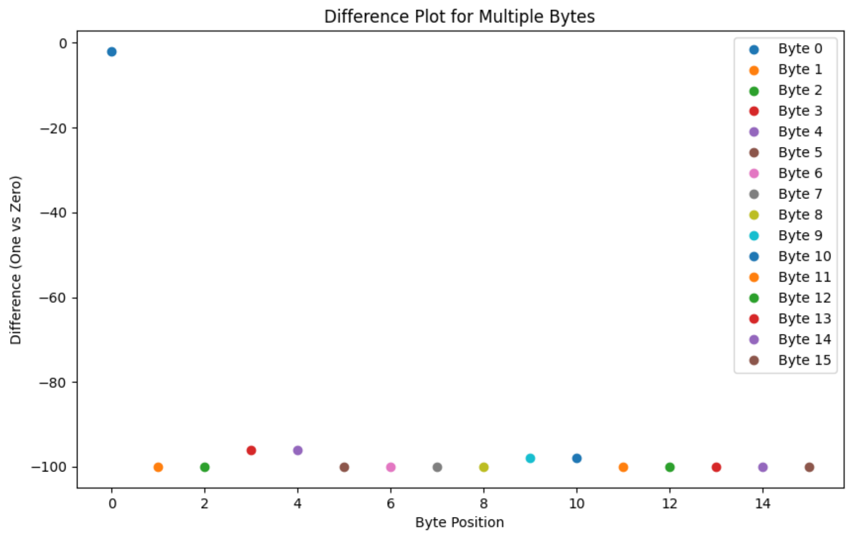
Compared to the previous analysis, which showed clear distinctions due to large Hamming weight differences (0xFF vs. 0x00), the current study with smaller Hamming weight differences presents more subtle power variations. Notably, prominent spikes are found in the initial segment of the trace on the negative axis, indicating that while reduced Hamming weight contrast lowers the visibility of power discrepancies, it does not eliminate data-dependent variations. These findings highlight the continued importance of side-channel analysis in evaluating cryptographic implementations, even with lower Hamming weight differences.

III) Comparing Power Consumption Across Multiple Different Bytes

In contrast to previous experiments that focused on individual bytes, this comparative analysis illustrates how power consumption fluctuates among multiple bytes. Power traces were gathered for each byte as it transitioned between Hamming weight states of one and zero. The differences in power consumption between these states were computed for each byte and visualized in a single plot.



```python
import matplotlib.pyplot as plt

# Assuming trace_array and textin_array are defined previously in your code
byte_positions = range(0, len(textin_array[0]))  # Iterate through each byte in the array
num_bytes = len(byte_positions)

# Initialize dictionaries to hold the one_list and zero_list for each byte
one_lists = {byte_pos: [] for byte_pos in byte_positions}
zero_lists = {byte_pos: [] for byte_pos in byte_positions}

# Iterate through the trace_array and textin_array
for i in range(len(trace_array)):
    for byte_pos in byte_positions:
        if textin_array[i][byte_pos] == 0x00:
            one_lists[byte_pos].append(trace_array[i])
        else:
            zero_lists[byte_pos].append(trace_array[i])

# Plot the differences for each byte position
plt.figure(figsize=(10, 6))

for byte_pos in byte_positions:
    one_diff = len(one_lists[byte_pos])
    zero_diff = len(zero_lists[byte_pos])

    # Calculate the difference (you can modify this depending on what kind of difference you want to plot)
    plt.plot([byte_pos], [one_diff - zero_diff], 'o', label=f'Byte {byte_pos}')

# Set plot labels and title
plt.xlabel('Byte Position')
plt.ylabel('Difference (One vs Zero)')
plt.title('Difference Plot for Multiple Bytes')
plt.legend()

# Show the plot
plt.show()

# Optional assertion checks (this part is unchanged)
assert len(one_lists[0]) > len(zero_lists[0])/2
assert len(zero_lists[0]) > len(one_lists[0])/2
```



Observation

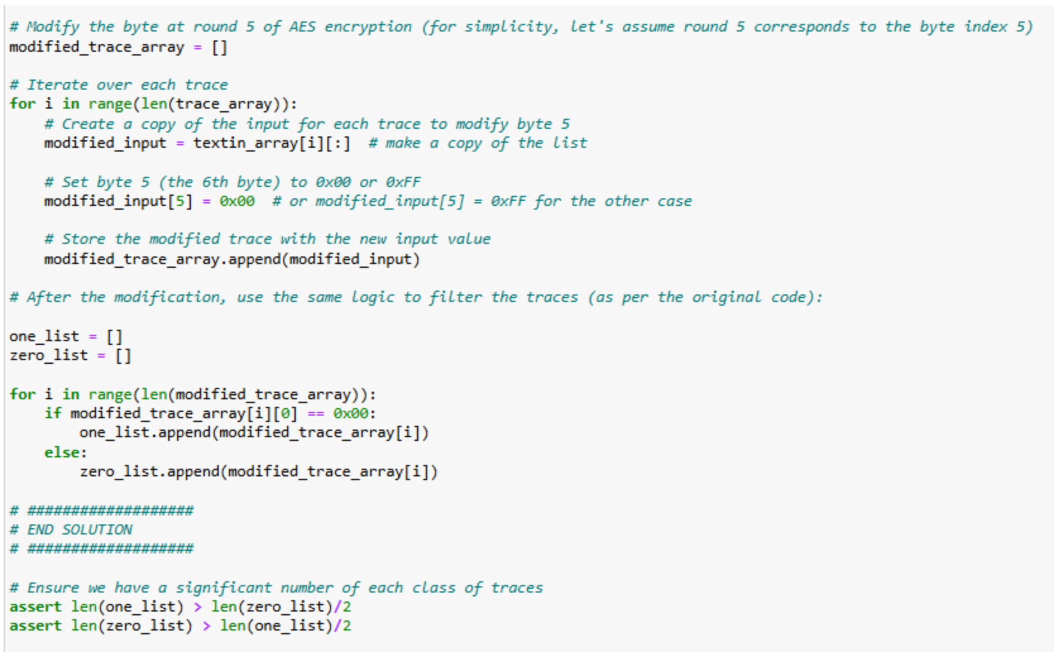
While most bytes show considerable negative differences—indicating a notable reduction in power consumption—Byte 0 demonstrates a distinct behavior, exhibiting a near-zero difference. This suggests that Byte 0 may undergo a different processing mechanism. Also, the magnitude of the differences varies across byte positions, highlighting that not all bytes contribute equally to variations in power consumption. These findings emphasize the importance of analyzing multiple bytes in power analysis, as certain byte positions may be more prone to information leakage than others.

IV) Modifying the byte at round 5 of AES encryption

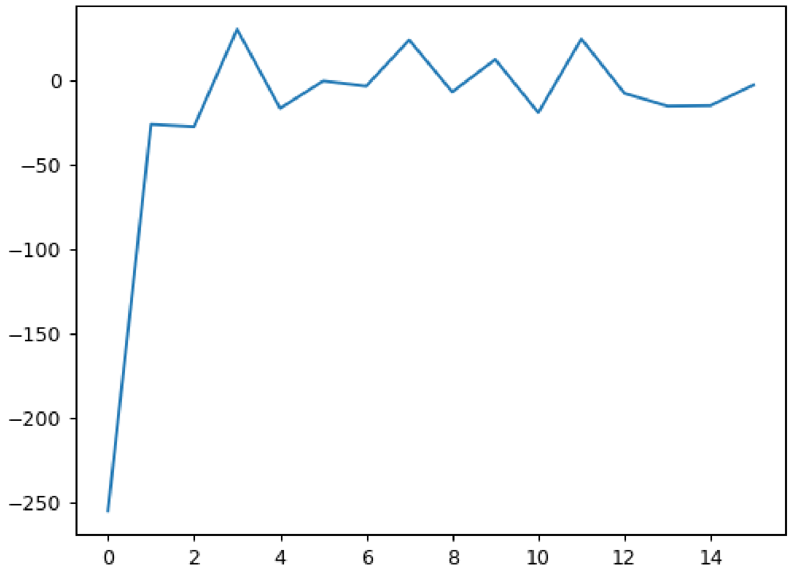
Focus on Round 5: The analysis specifically targets a later round (round 5) of the AES encryption.

Byte Modification: A single byte is modified to either 0x00 or 0xFF to induce controlled changes in the encryption process.

Power Trace Analysis: The resulting power traces are analyzed for variations in the location and characteristics of spikes, comparing them to observations from previous experiments.



```
# Modify the byte at round 5 of AES encryption (for simplicity, let's assume round 5 corresponds to the byte index 5)
modified_trace_array = []

# Iterate over each trace
for i in range(len(trace_array)):
    # Create a copy of the input for each trace to modify byte 5
    modified_input = textin_array[i][:]  # make a copy of the list

    # Set byte 5 (the 6th byte) to 0x00 or 0xFF
    modified_input[5] = 0x00  # or modified_input[5] = 0xFF for the other case

    # Store the modified trace with the new input value
    modified_trace_array.append(modified_input)

# After the modification, use the same logic to filter the traces (as per the original code):

one_list = []
zero_list = []

for i in range(len(modified_trace_array)):
    if modified_trace_array[i][0] == 0x00:
        one_list.append(modified_trace_array[i])
    else:
        zero_list.append(modified_trace_array[i])

# ###################
# END SOLUTION
# ###################

# Ensure we have a significant number of each class of traces
assert len(one_list) > len(zero_list)/2
assert len(zero_list) > len(one_list)/2
```



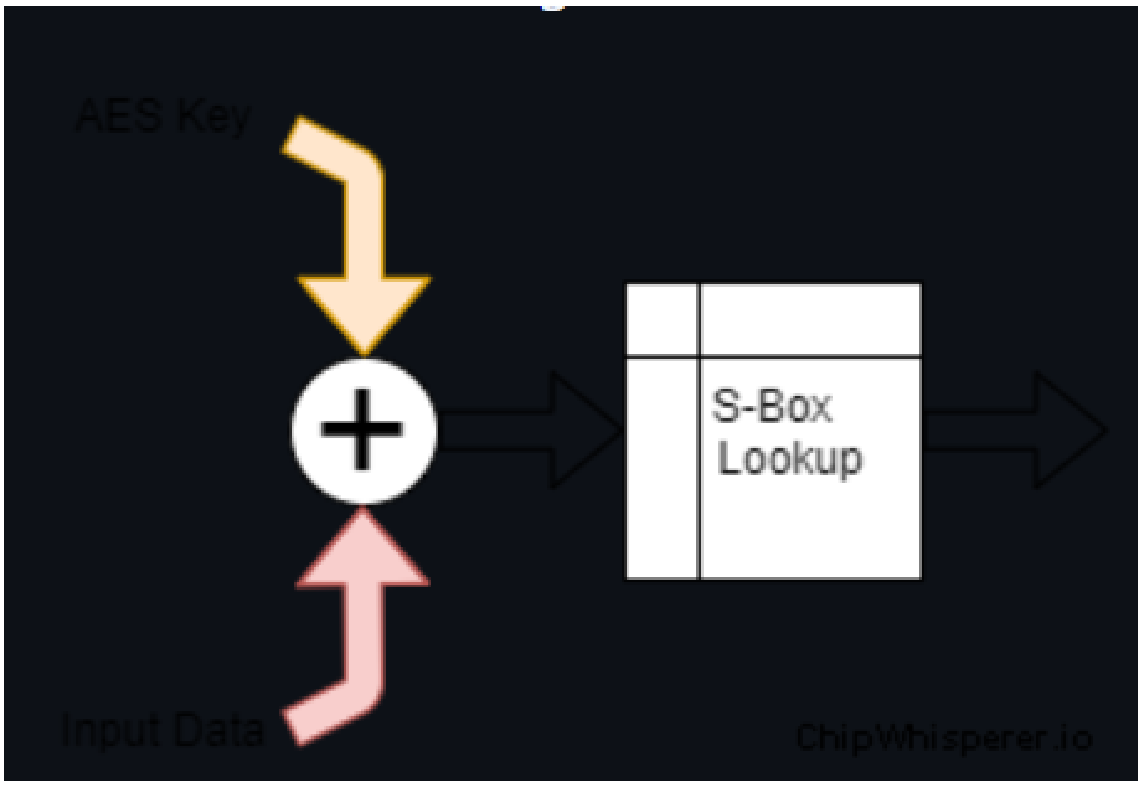
Observation

The power trace illustrated in the graph shows notable variations in spike locations, indicating that altering byte 5 in round 5 of AES encryption significantly impacts power consumption patterns. When compared to the baseline traces, the observed shifts in spike positions imply that modifications made in later rounds propagate through the encryption process, influencing subsequent computations. This supports the hypothesis that the characteristics of power leakage are sensitive to specific byte modifications.

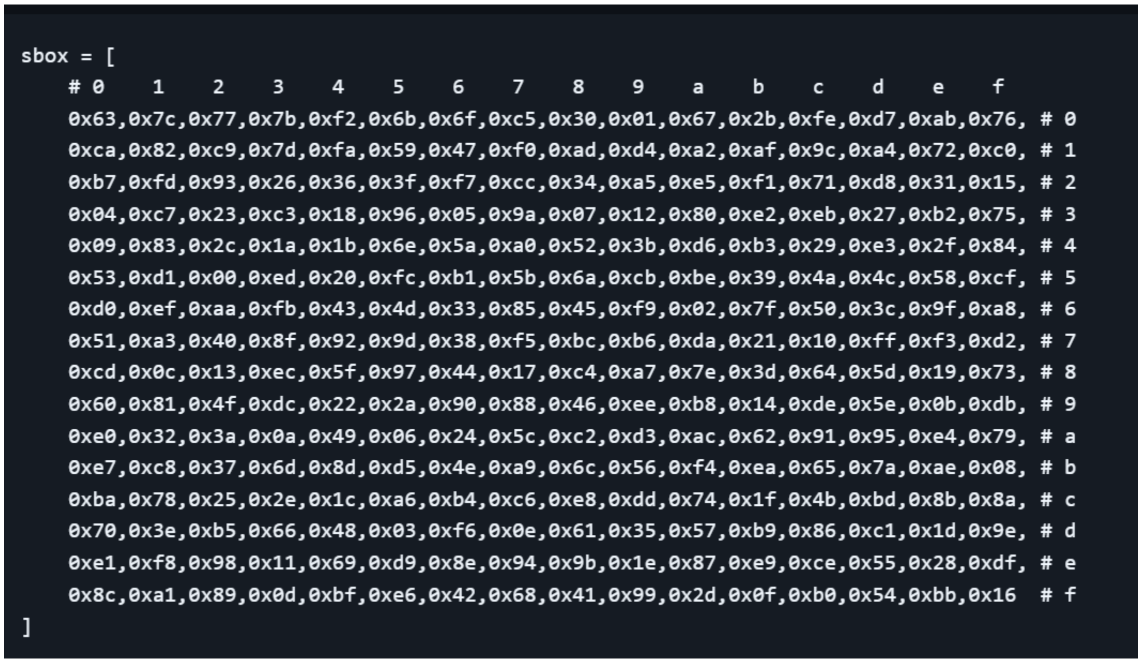
2.3 Recovering Data from a Single Bit

2.3.1 Introduction

This lab looks at recovering data from a single bit in an encryption system. It shows how this attack works when observation accuracy changes. We will see how a single bit of leakage can transform to a full key reveal. Also, we will see how to sort and rank the list.



First, we are going to build a simple function to X'OR our secret key with our input data and, we perform a lookup in the s-box table. Then, we run the test vector.



```
sbox = [
    # 0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, # 0
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0, # 1
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15, # 2
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75, # 3
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84, # 4
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf, # 5
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8, # 6
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2, # 7
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73, # 8
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb, # 9
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79, # a
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08, # b
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a, # c
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e, # d
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf, # e
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16  # f
]
```

2.3.2 Target and Attack Methodology

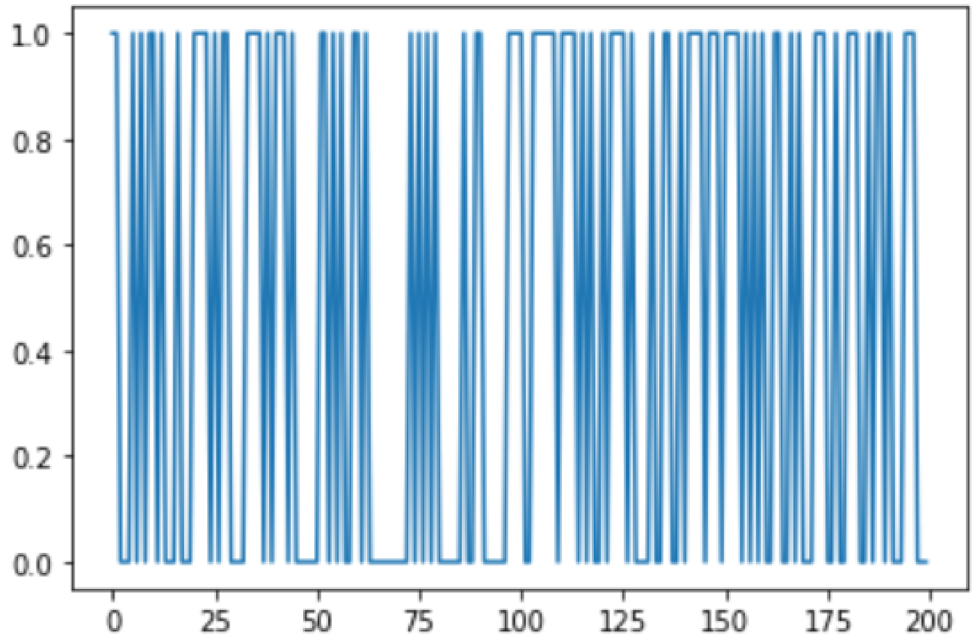
The target of this attack is a cryptographic system that leaks physical clues, especially through a single-bit output. The steps of the attack is:

1. Collecting Data: Record a single bit from many encryption tries.
2. Changing Accuracy: Try different observation accuracies, from low (around 20 percent) to high (close to 100 percent).
3. Analyzing Recovery: Measure how many encryption tries are needed to guess the key based on the observation accuracy.
4. Creating Graphs: Use the data to make graphs that show the connection between accuracy and attempts needed.

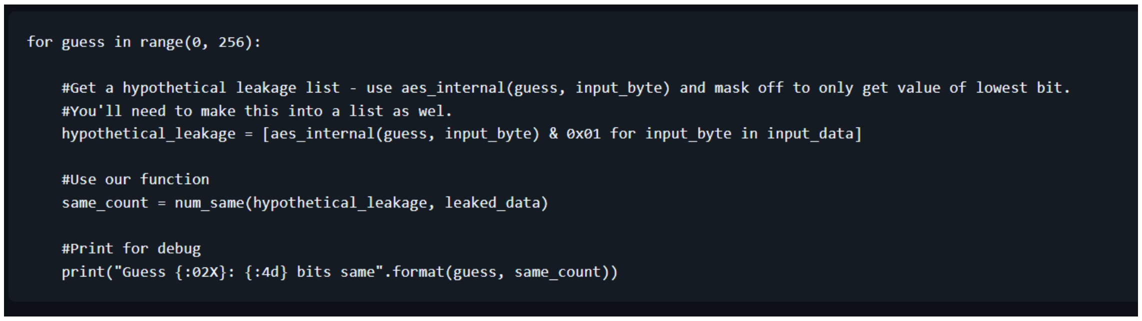
For our first attack, we are going to assume that we can't observe a single bit of this entire value. Let's just observe a single bit of this value. Imagine that someone put a probe down inside the chip to get this. We can do this by simply throwing away all the other data besides a single bit and only expose that single bit to the observer. The watcher is going to observe a single bit of data. We will need to build lists of input to feed to the algorithm (we are going to send in 1000 random bytes that get's encrypted) as well as lists to hold our observation.

2.3.3 Result and Analysis

We feed all those inputs through the aes_secret function because this is a secret function we are going to observe a single bit of output (the leakage). leaked_data contains just the 1 or 0 status of the lowest bit. The analysis focused on two graphs: Bit Observation Over Time: This graph shows how a single-bit output changes during many encryption attempts. The changes are random and noisy.



The following block is the most important because we will need to apply the leakage function. That is for each known input byte, pass it through the aes_internal (input_date, key_guess) function.



```
for guess in range(0, 256):

    #Get a hypothetical leakage list - use aes_internal(guess, input_byte) and mask off to only get value of lowest bit.
    #You'll need to make this into a list as wel.
    hypothetical_leakage = [aes_internal(guess, input_byte) & 0x01 for input_byte in input_data]

    #Use our function
    same_count = num_same(hypothetical_leakage, leaked_data)

    #Print for debug
    print("Guess {:02X}: {:4d} bits same".format(guess, same_count))
```

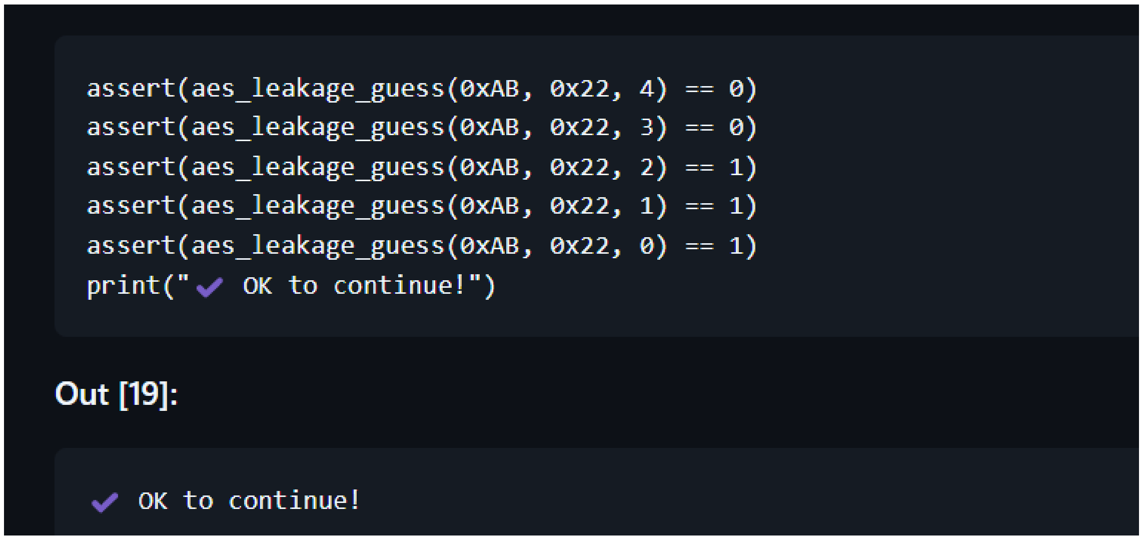
Sorting argument

```
sorted_list = np.argsort(guess_list)[::-1]
```

np.argsort (guess_list) return an array of indices, sorted from lowest to highest match count. $[:: -1]$ reverses the array so the best key guess (highest match count) come first.

For this case, we know that bit '0' was the leakage. The question is what if we did not know that? Then, we need to align our leakage function to take in a bit number which is leaking. For this, we are going to define a function that returns the value of a bit being 1 or 0.

```python
def aes_leakage_guess(keyguess, inputdata, bit):
    return get_bit(aes_internal(keyguess, inputdata), bit)
```



```python
assert(aes_leakage_guess(0xAB, 0x22, 4) == 0)
assert(aes_leakage_guess(0xAB, 0x22, 3) == 0)
assert(aes_leakage_guess(0xAB, 0x22, 2) == 1)
assert(aes_leakage_guess(0xAB, 0x22, 1) == 1)
assert(aes_leakage_guess(0xAB, 0x22, 0) == 1)
print("✓ OK to continue!")
```

Out [19]:

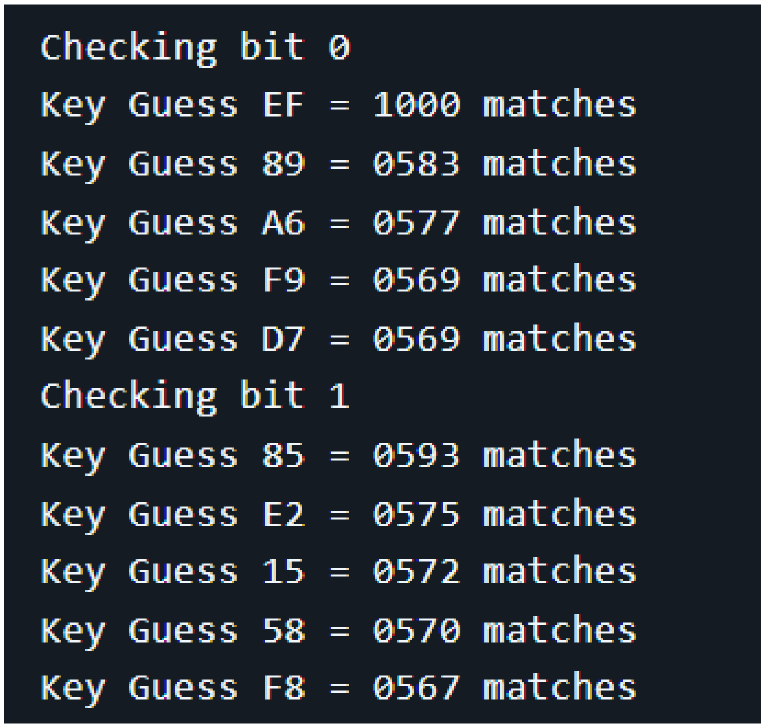
```
✓ OK to continue!
```

Hopefully we see that only the matching bit shows the full number of matches.

```python
    for bit_guess in range(0, 8):
    guess_list = [0] * 256
    print("Checking bit {:d}".format(bit_guess))
    for guess in range(0, 256):

        #Get a hypothetical leakage for guessed bit (ensure returns 1/0 only)
        #Use bit_guess as the bit number, guess as the key guess, and data from input_data
        hypothetical_leakage = [aes_leakage_guess(guess, input_byte, bit_guess) for
    input_byte in input_data]

        #Use our function
        same_count = num_same(hypothetical_leakage, leaked_data)

        #Track the number of correct bits
        guess_list[guess] = same_count

    sorted_list = np.argsort(guess_list)[::-1]

    #Print top 5 only
    for guess in sorted_list[0:5]:
            print("Key Guess {:02X} = {:04d} matches".format(guess, guess_list[guess]))
```

Listing 1: Bitwise Guessing Loop

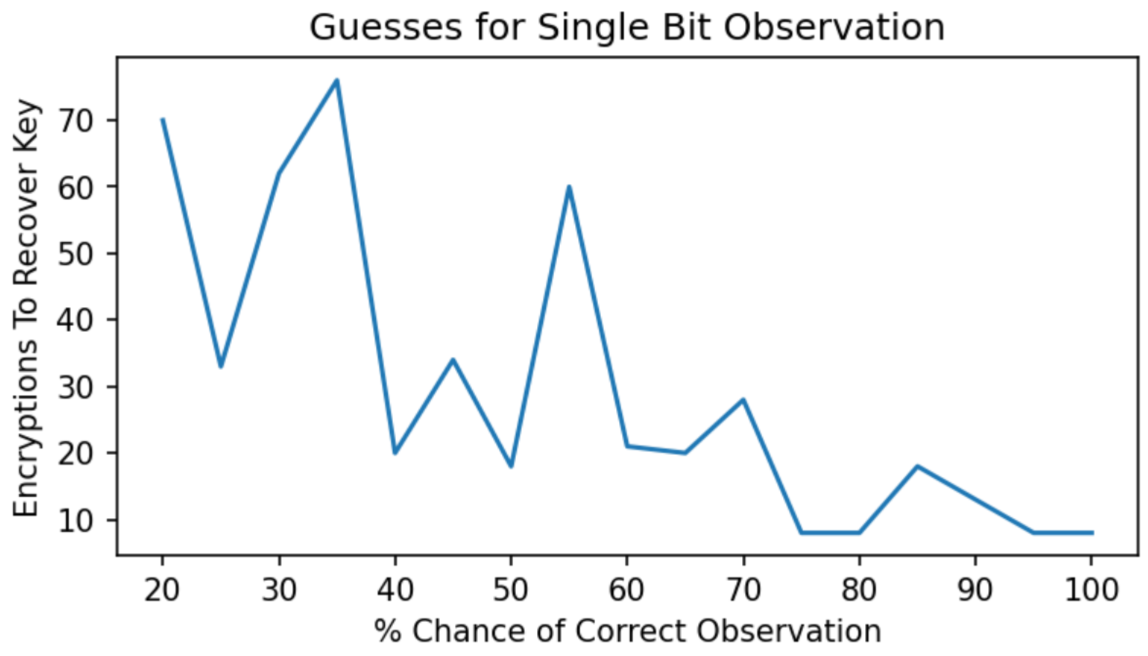
The loop tests different key guesses and compares predicted leakage with real leaked data. The loop identifies key guesses that closely match the real leaked data, helping to reduce the number of possible encryption keys. This method helps find cryptographic keys by analyzing side-channel leakage. It shows how attackers can use leaked data to break encryption.



```
Checking bit 0
Key Guess EF = 1000 matches
Key Guess 89 = 0583 matches
Key Guess A6 = 0577 matches
Key Guess F9 = 0569 matches
Key Guess D7 = 0569 matches
Checking bit 1
Key Guess 85 = 0593 matches
Key Guess E2 = 0575 matches
Key Guess 15 = 0572 matches
Key Guess 58 = 0570 matches
Key Guess F8 = 0567 matches
```

The output shows the top five key guesses for each bit, ranked by match count. Some key guesses have significantly higher matches, indicating they are more likely to be correct. For example: Bit 0: Key guess EF had 1000 matches, making it a strong candidate. Bit 7: Key guess 18 had 595 matches, the highest for that bit.

Guessing Attempts vs. Observation Accuracy: This graph shows how the number of encryptions needed to guess the key depends on the accuracy of the observation. When accuracy is low (below 30 percent), many encryption tries are needed often more than 70. When accuracy is higher than 70 percent, the required encryption tries drop a lot, to around 10 when accuracy reaches 90 percent or more.



These results showed that more accurate observations make the attack easier and faster. Small improvements in observation accuracy can save a lot of effort when trying to get cryptographic keys.

2.3.4 Conclusion

This lab shows how cryptographic systems can be attacked through side channel attacks methodologies, even with only a single bit observation. The results prove that high observation accuracy makes key recovery easier. To stop these attacks, systems need better protections like noise, masking techniques, or improved hardware.

2.4 DPA on Firmware Implementation of AES

By combining the techniques from the previous side-channel techniques, we are now able to directly attack a naive implementation of AES using Hamming weights to guess the S-Box output bit we attacked previously. The main idea is that power consumption can be influenced by a particular bit being one or zero in a specific point in time.

For this example, we are attacking one single bit of the S-box output by monitoring the power consumption of 2500 random input, one key byte at the time. Eventually, the full key could be recovered.

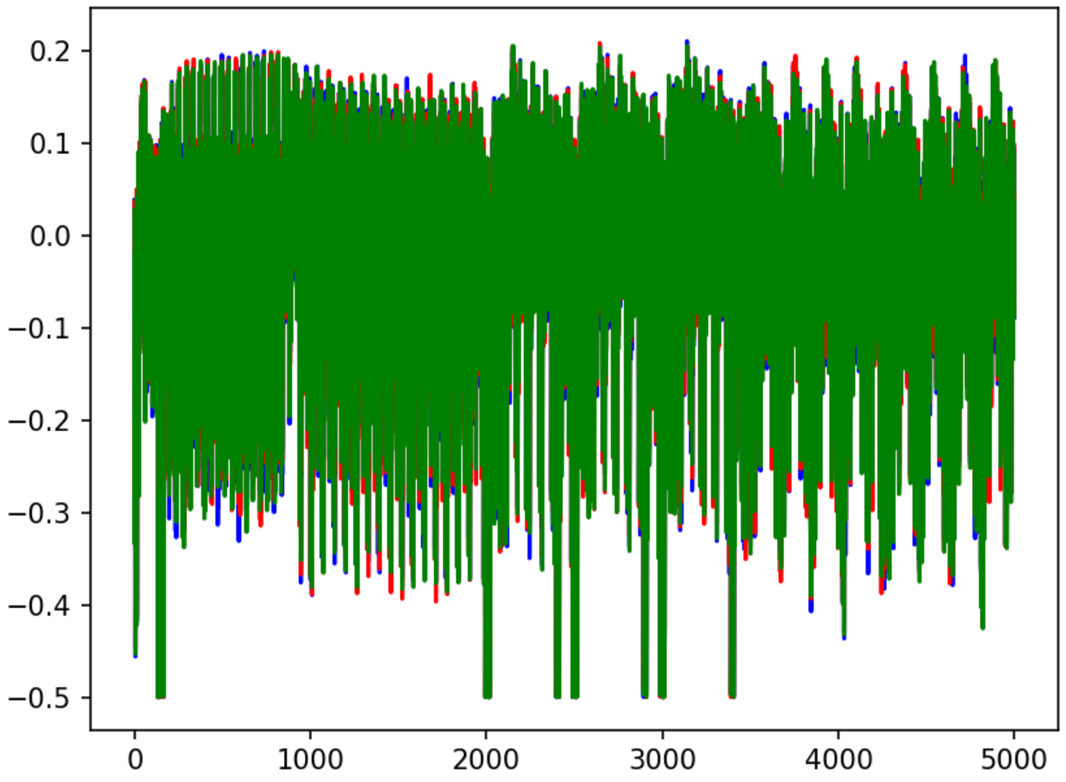


Figure 1: AES Power Trace

Figure 1 shows the overlayed power consumption of three different encryptions (with the same key). It can be seen that there are small differences in power consumption that can be used to gain some information about the processed information, particularly the first S-box output.

To carry out the DPA attack, we need to monitor some power traces related to random inputs and divide them into groups based on the target S-box bit values related to a specific key. If the difference between the two groups is large enough (in term of average power consumption), we know the separation is correct and we can thus conclude that the key byte we chose is correct.

Code 2 performs the division and distance calculation between the two cited groups, based on a given key guess and the chosen leakage model. A high distance value would suggest a good key guess.

```python
def calculate_diffs(guess, byteindex=0, bitnum=0):
    """Perform a simple DPA on two traces, uses global `textin_array` and `trace_array` """

    one_list = []
    zero_list = []

    for trace_index in range(numtraces):
        hypothetical_leakage = aes_internal(guess, textin_array[trace_index][byteindex])

        #Mask off the requested bit
        if hypothetical_leakage & (1<<bitnum):
            one_list.append(trace_array[trace_index])
        else:
            zero_list.append(trace_array[trace_index])

    one_avg = np.asarray(one_list).mean(axis=0)
    zero_avg = np.asarray(zero_list).mean(axis=0)
    return abs(one_avg - zero_avg)
```

Listing 2: Code for distance calculation

In code 3, for each byte of the key (line 7) and for each possible key value (line 10), the traces are divided based on the value of the attacked bit according to the leakage model. The possible key values are then sorted and ranked on the basis of the Hamming distance. As a high distance suggests a good guess, the top 5 candidates are shown for each key byte.

```
1  from tqdm import tnrange
2  import numpy as np
3
4  #Store your key_guess here, compare to known_key
5  key_guess = []
6
7  for subkey in tnrange(0, 16, desc="Attacking Subkey"):
8      max_diffs = [0]*256
9      full_diffs = [0]*256
10     for guess in range(0, 256):
11         full_diff_trace = calculate_diffs(guess, subkey)
12         max_diffs[guess] = np.max(full_diff_trace)
13         full_diffs[guess] = full_diff_trace
14
15     #Get argument sort, as each index is the actual key guess.
16     sorted_args = np.argsort(max_diffs)[::-1]
17
18     #Keep most likely
19     key_guess.append(sorted_args[0])
20
21     #Print results
22     print("Subkey %2d - most likely %02X (actual %02X)"%(subkey, key_guess[subkey],
       known_key[subkey]))
23
24     #Print other top guesses
25     print(" Top 5 guesses: ")
26     for i in range(0, 5):
        g = sorted_args[i]
        print("    %02X - Diff = %f"%(g, max_diffs[g]))

    print("\n")
```

Listing 3: Code for byte guessing

This ranking gives provides some alternative key candidates in case the most likely key does not work. As can be seen in figure 4, the difference in distance for the guesses is greater for some values (E.G. Subkey 0) than for others (E.G. Subkey 1). The subkeys with the smallest difference would be the first one to check in case the guess turns out wrong.

```
Subkey  0 - most likely 2B (actual 2B)
 Top 5 guesses:
   2B - Diff = 0.015402
   73 - Diff = 0.007788
   35 - Diff = 0.006993
   15 - Diff = 0.006764
   69 - Diff = 0.006722


Subkey  1 - most likely 7E (actual 7E)
 Top 5 guesses:
   7E - Diff = 0.012779
   92 - Diff = 0.012288
   77 - Diff = 0.011720
   1B - Diff = 0.010098
   04 - Diff = 0.009978


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.009492
   89 - Diff = 0.006490
   CA - Diff = 0.005791
   9E - Diff = 0.005735
   78 - Diff = 0.005497
[...]
```

Listing 4: Key Bytes guessing

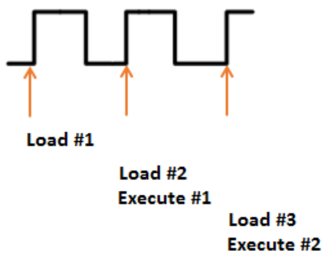
3 Fault injection

While side channel techniques aim to recover some data from a software running on device, fault injection attacks are instead focused on modifying the data or execution flow in order to predict unintended behavior.

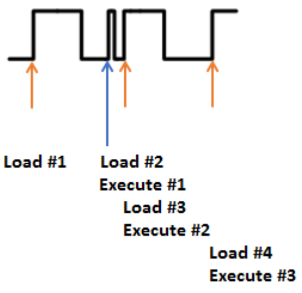
This can be done by modifying the operating conditions of a device such as supply voltage. Outside of the manufacturer-specified boundaries, electronic components will begin to malfunction, with more extreme violations causing the device to stop entirely or even become damaged. By going outside these operating conditions for very small amounts of time, we can cause a variety of temporary malfunctions.

3.1 Clock Glitching

The purpose of clock glitching is to shorten the duration of a clock cycle by sending a voltage glitch to the clock bus before the cycle actually ends. In this scenario, the distance between the inserted glitch and the next legit clock edge is less than half the clock period. This causes the instruction that was loaded on the glitch cycle to skip in some cases, and it can be very powerful depending on the code that is running.



(a) Normal clock signal



(b) Glitched clock signal

Figure 2: Clock time graph

The Chip Whisperer platform makes it possible to create custom clock glitches to carry out specific attacks. The most important characteristic of a clock glitch is (from docs):

- *offset*: where in the output clock to place the glitch. Can be in the range $[-48.8, 48.8]$. Often, we will want to try many offsets when trying to glitch a target.
- *width*: How wide to make the glitch. Can be in the range $[-50, 50]$, though there is no reason to use widths < 0. Wider glitches more easily cause glitches, but are also more likely to crash the target, meaning we'll often want to try a range of widths when attacking a target.
- *ext_offset*: the number of cycles from the trigger before the glitch is injected.

The Python module also contains a glitch controller, to make searching for a good set of parameters easier. Basically, we can specify a range of possible values for each setting as well as a set of outcome labels (E.G. *success, reset, normal*) that can be filled by the developer according to the measured outcome of the glitch.

```python
import chipwhisperer.common.results.glitch as glitch
gc = glitch.GlitchController(groups=["success", "reset", "normal"], parameters=["width", "
    offset", "ext_offset"])
```

```python
# Setting parameters range
gc.set_range("width", 2.25,2.4)
gc.set_range("offset", -10,-6)
gc.set_range("ext_offset", 20, 70)
gc.set_global_step(0.1)

for glitch_settings in gc.glitch_values():
    scope.glitch.offset = glitch_setting[1]
    scope.glitch.width = glitch_setting[0]
    scope.glitch.ext_offset = glitch_setting[2]
    scope.arm()
    target.simpleserial_write('g', bytearray([]))
```

Listing 5: Simple glitch code

The code 5 shows a basic usage of the *Glitch controller* to try a set of specified parameters in a given step. By adding some more logic to check for the output state, and accounting for crashes in the board, it is possible to map the parameter domain to visually see which glitches resulted in a program crash and which in a modification of the behavior.

3.1.1 Proof of concept attack

It's really easy to test a clock glitch on the following code 6. If the return value is one, we automatically know that some instructions (we don't care which ones) of the for cycle have been skipped.

```c
uint8_t glitch_loop(uint8_t* in)
{
    volatile uint16_t i, j;
    volatile uint32_t cnt;
    cnt = 0;
    trigger_high();
    for(i=0; i<50; i++){
        for(j=0; j<50; j++){
            cnt++;
        }
    }
    trigger_low();
    simpleserial_put('r', 4, (uint8_t*)&cnt);
    return (cnt != 2500);
}
```

Listing 6: Simple target code

Using *Glitch controller* similarly to Code 5, we can perform a parameter scan and plot a map of the glitches for the shown code.

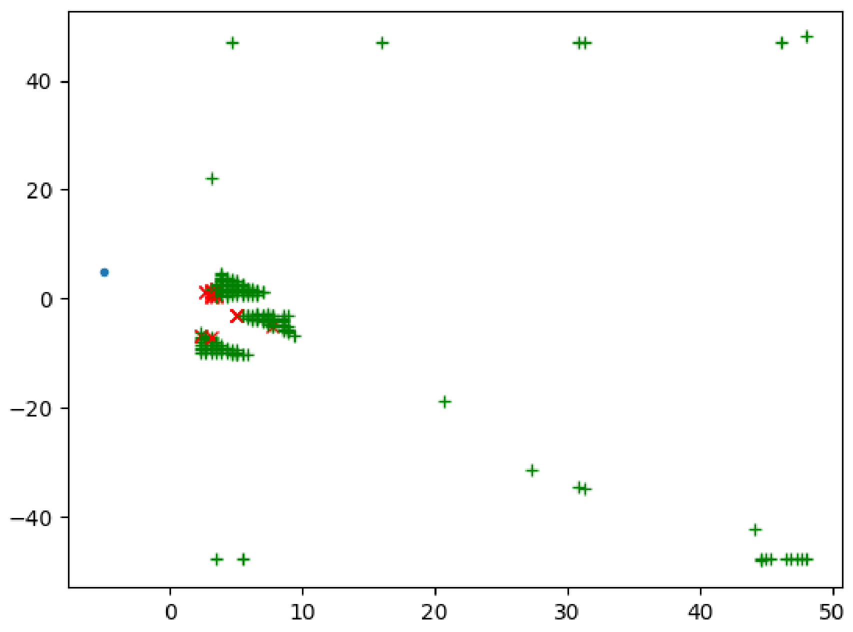


Figure 3: Glitch map (red = crash, green = success)

As demonstrated by the 3 plot, this search was particularly successful and many favorable glitches have been found for this really easy case.

3.2 Password bypass

With this knowledge, we want to try and attack a naive password verification algorithm shown in 7. The idea is that if we can skip the heading of *for* cycle (line 10), the program will not check the correctness of the password and return a positive outcome anyways, because the *passok* variable was initialized to 1.

```c
1  uint8_t password(uint8_t* pw)
2  {
3      char passwd[] = "touch";
4      char passok = 1;
5      int cnt;
6  
7      trigger_high();
8  
9      //Simple test - doesn't check for too-long password!
10     for(cnt = 0; cnt < 5; cnt++){
11         if (pw[cnt] != passwd[cnt]){
12             passok = 0;
13         }
14     }
15
16      trigger_low();
17
18      simpleserial_put('r', 1, (uint8_t*)&passok);
19      return passok;
20  }
```

Listing 7: Realistic target code

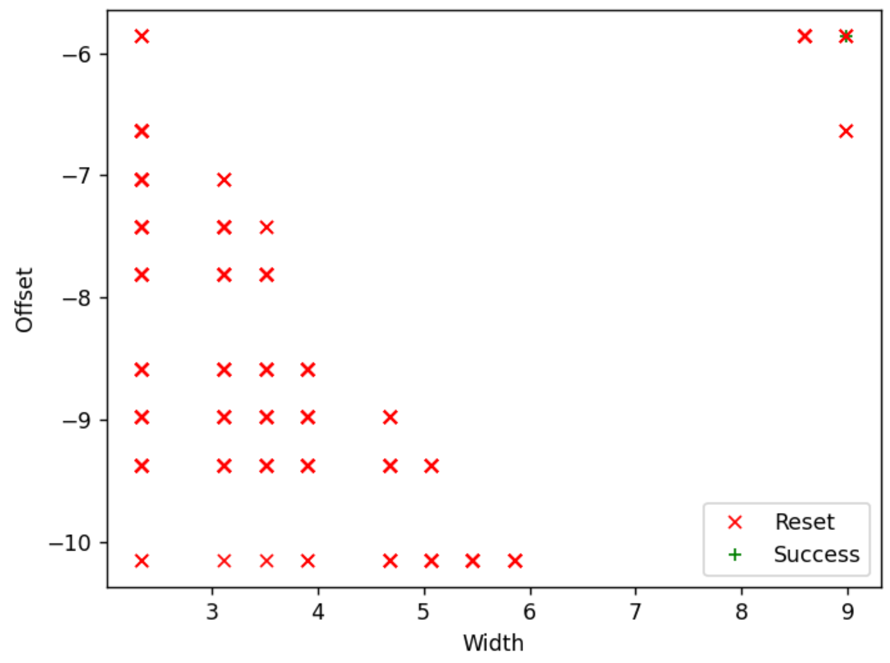
For this experiment, we set up a *Glitch controller* like previously and try to map the glitches and an empty password (*00000*) was sent to the board and the results have been sorted based on the boolean response (*passok* variable, sent in line 18) as well as the *valid* attribute provided by the *SimpleSerial* protocol.

The parameters interval we chose for the test are:

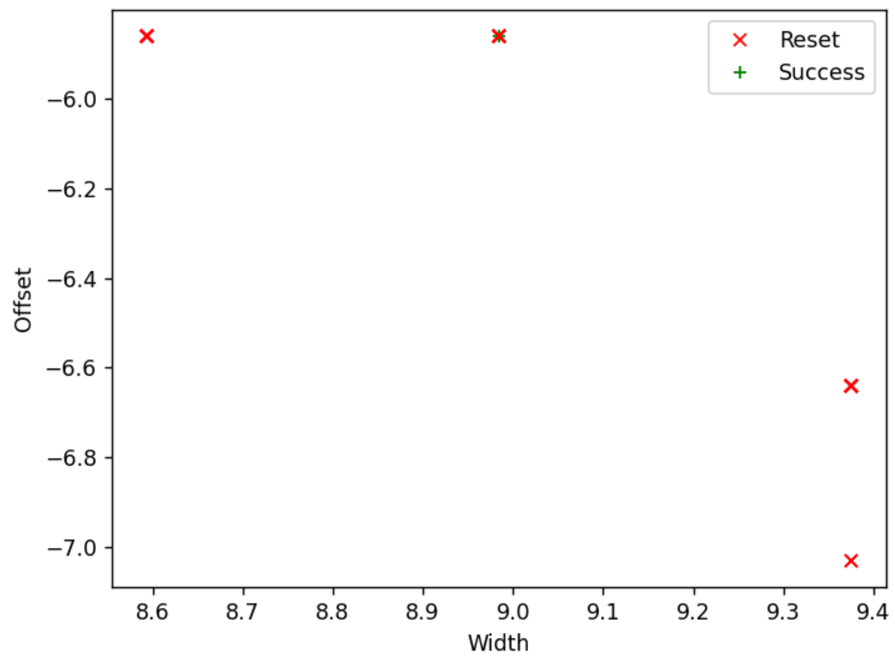
- width: $[2, 10]$
- offset: $[-11, -5]$
- ext_offset: $[20, 70]$

The test results shown in Listing 4 show us several crashes - occurred when either the board becomes unresponsive, or it replies with the *invalid* flag - and one overlapped success. This is hard to interpret because in one single scenario we managed to get the desired outcome (password verification success) but the response was invalid, probably due to a communication issue.

Looking at SimpleSerial source code at line 471 shown in Listing 8, it can be seen the the invalid flag could be either caused by Incorrect Response Length, Command Mismatch (line 6) or invalid Payload value (line 18). Therefore, it is not clear how we obtained a 1 for the payload value for the success glitch (parameters value: $[8.984375, -5.859375, 54]$).



(a) Overall interval



(b) Success detail

Figure 4: Password bypass glitch diagram

```
def simpleserial_read_witherrors(self, cmd, pay_len, end="\n", timeout=250, glitch_timeout
    =8000, ack=True):
    [...]
    valid = False
    rv = None

    if len(response) != recv_len or response[0:cmd_len] != cmd:
        # Switch to robust mode - likely a glitch happened. Get all response first...
        response += self.read(1000, timeout=glitch_timeout)
        payload = None
    else:
        valid = True
        idx = cmd_len
        for i in range(0, pay_len):
            try:
                payload[i] = int(response[idx:(idx + 2)], 16)
            except ValueError as e:
                payload = None
                valid = False
                break
                #target_logger.warning("ValueError: {}".format(e))
    [...]
```

Listing 8: SimpleSerial source code

This highlights the difficulty of crafting a fault injection attack, even on seemingly simple code. The last experiment required approximately four hours to complete, and within this time frame, we were unable to fully explore the desired range of parameter values. This suggests that additional time could potentially uncover more results

4 Conclusion and Countermeasures

This lab report examined different hardware security vulnerabilities through side-channel attacks and fault injection techniques. We used ChipWhisperer, an open-source tool, to analyze power consumption and manipulate device behavior to bypass security measures.

The first part focused on side-channel attacks, particularly power analysis for password bypass. By analyzing power traces linked to different inputs, we identified valid password characters, showing how attackers can extract authentication details. Power variations were used to automate the attack, confirming the possibility of exploiting such weaknesses.

Next, we examined Hamming weight analysis to see how data values affect power consumption. By comparing high and low Hamming weight inputs, we showed that cryptographic systems like AES have measurable power variations, making them vulnerable to Differential Power Analysis (DPA). We refined the analysis by considering smaller Hamming weight differences and comparing power use across multiple bytes.

The single-bit data recovery experiment demonstrated how cryptographic systems could be attacked by observing a single leaking bit. We showed that increasing observation accuracy significantly reduced the number of encryption attempts needed to recover the full cryptographic key. This highlighted the importance of minimizing side-channel leaks in secure systems.

We then applied DPA techniques to an AES firmware implementation, successfully retrieving key bytes by analyzing power traces. By sorting traces based on S-Box output bits and ranking potential key values using a leakage model, we confirmed the effectiveness of DPA in breaking encryption schemes.

Finally, we explored fault injection attacks, including clock glitching and password bypass. By altering a device's working conditions, we induced malfunctions to bypass authentication checks. A proof-of concept attack was performed on a password verification function, demonstrating how glitching techniques can compromise embedded systems.

To conclude, this lab demonstrated the importance of hardware security and countermeasures like constant time execution, power noise injection, and fault detection. Our findings emphasize the need for strong cryptographic implementations that resist side-channel and fault injection attacks to protect embedded systems.