Lab 05: Side Channel Analysis Attacks (Part 2) Finding Patterns in the Power Side Channel

Jasmin Kaur and Alec Braynen
University of South Florida
Tampa, FL 33620

I. Introduction

Electronic devices can leak information about their internal processes through their power consumption at a given time. These leaks of information open these devices to side-channel analysis(SCA) attacks . [1]. One particular type of SCA attack is a timing based SCA attack. Observing and analyzing the timing patterns of the internal processes allows the inference of sensitive data such as a secret password to an attacker [2].

The main idea behind SCA is to try and identify how power consumption or timing of a device varies and when it varies with respect to some internal algorithm processing [2]. For example, the attackers could observe differences in how long it takes to complete a complex task for a timing based SCA, or one may observe the power consumption of different chip components based on data processing for a power analysis [2]. The gathered power and timing patterns are statistically analyzed to gather sensitive information and used to implement an attack.

In this work, the authors SCA a device by not only analyzing how much power is consumed but also when it is consumed [2]. A timing-based SCA attack is performed on different password checking functions to guess the correct password. A ChipWhisperer (CW) Nano board is utilized in this experiment and it's side channel information is observed and gathered as power traces. Statistical analysis is then performed on these traces and the differences in them are used to determine whether a character is the correct guess for a secret password. Then a countermeasure is implemented and the bruteforce attack is tested on this countermeasure. The results of all these experiments are shown in this paper. [2].

II. Reading Check

Question 1: How does a timing attack infer information from the physical chip?

Answer: Timing attacks allow inference when observing/utilizing the power spikes in power traces gathered from the chip and when these power spikes happen. Analyzing these patterns allows for inference about the program running on the chip. [2].

Question 2: What do timing attacks often exploit?

Answer: Timing attacks often exploit the optimizations in the code [2]. For example, when a correct character guess for a password is found, the loop exits and outputs the answer. This can be seen a power spike in a power trace. [1].

Question 3: What are some ways to defend against timing attacks?

Answer: One way to defend against timing attacks is by making sure the program always runs for a fixed number of iterations or time before outputting whether an output is correct or wrong. This could be done by adding delays before outputting the result. [1].

III. Methods

For this experiment we gather power traces of computation for password guesses from two password checking programs using CW nano board and Python. The PHS-VM includes ChipWhisperer v5.5 and Python v3.8-3.10 to interface with CW. The CW is configured for communication using the methods CW setup scripts contained in the directory "cw-base-setup" and a firmware script. The "cw-base-setup/simpleserial-base" is used as a template firmware to program the nano board.

A. Software Setup

The software setup for the experiment consists of a virtual machine running Ubuntu OS on Virtualbox. The Python and C code is implemented via Jupyter notebooks on this virtual machine. ChipWhisperer v5.5 and Python to interface with CW and CW setup library "simpleserial" is used to send and recieve data from CW.

1) The "Python" directory hold the Jupyter notebook files where the python code is executed for compiling firmware, reading CW responses, executing the brute-force attack, as well as data analysis and plotting graphs.
2) The "Firmware" directory holds the C file containing the functions that are performed by CW on the recieved password - password checkers v1.0 and v2.0. It also contains a makefile used to compile the C code and a ".hex" file that is used to configure CW.
3) The "Figures" directory holds the generated figures of the analyzed data.

The directory structure of the files in the Jupyter notebook is structured as follows:

```
PHS-Lab-04
├── Firmware
│   ├── makefile
│   ├── training-CWNANO.hex
│   ├── attack-CWNANO.hex
│   └── training.c
├── Python
│   └── Lab-05-SCA-Statistical-Analysis.ipynb
└── Figures
    ├── attack1a.pdf
    ├── attack1b.pdf
    ├── attack2a.pdf
    ├── attack2b.pdf
    ├── attack3c.pdf
    ├── trace0.pdf
    ├── trace1.pdf
    ├── trace2.pdf
    ├── trace3.pdf
    ├── trace4.pdf
    ├── trace5.pdf
    ├── trace6.pdf
    ├── trace3-0.pdf
    ├── trace3-1.pdf
    ├── trace3-2.pdf
    ├── trace3-4.pdf
    ├── trace3-5.pdf
    ├── trace3char.pdf
    └── trace3char4char.pdf
```

B. Hardware Setup

The hardware setup consists of the CW nanoboard connected via a microUSB cable to a desktop machine running Windows (Fig. 1).

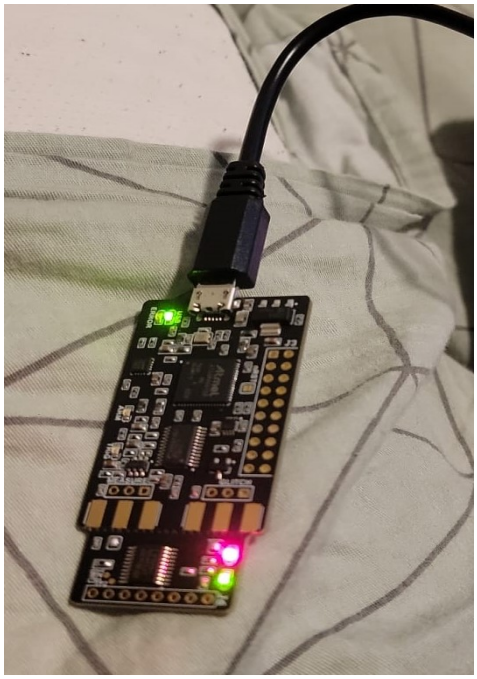


Fig. 1: CW nano board Connected to PC

C. Experimental Procedure

Part A: Firmware script

The C code for programming CW nano board is as follows:

- The ”training.c” files contains the functions naive password checking functions ”password checker v1.0”, ”password checker v2.0”.
- It additionally contains another ”password checker v3.0” with security measures against timing attacks.
- These functions are called in the main using simpleserial parameters ’a’, ’b’, and ’c’.

Part B: Training Password Attack

Obtaining the data is done by:

- Python file ””Lab-05-SCA-Statistical-Analysis.ipynb” imports chipwhisperer, matplotlib.pyplot, numpy, and scipy.stats.
- The ”training.c” code is compiled in the python file and the CW is then programmed with the C code after verifying the correctness of C code.
- A reset target function is defined to reset the target on the board.
- The password and its length are known. It uses English alphabet - both lower and upper case.
- A function is defined to capture the power trace of the board during its processing for the naive password checker algorithms for various password guesses.
- First we describe a function to guess a single correct character which returns the correct character, the power trace and the value where a power trace begins to differ.
- If no such character is found or it reaches end of the list, then it returns False. A threshold value is set above which the power trace is valid and below which it fails.
- Then each single character of the entire password is guessed and returned and checked using the naive password checking functions in ”training.c”.
- The two consecutive power traces with correct and incorrect characters are compared and the difference between them is plotted to obtain the similarities in the power traces to get the secret password.

Part C: Secret Password Attack

A secret attack is mounted on CW nano-chip to make sure our attack is successful:

- The CW nano-chip is programmed using ”attack-CWNANO.hex”, where the password and its length are unknown.

- A fully automated timing attack using Part B is mounted on the CW nano chip and the password is guessed using both of the naive password checkers.
- These waveform of the traces is then plotted to determine when a power was consumed by CW corresponding to a correct password guess.
- The t values are plotted and the mean and standard deviation of the t values are calculated.
- The percentage of statistical significance (count of p values less than 0.05) is calculated
- The board is disconnected.

Part D: Password Checker Countermeasure

The analysis of the responses is done by:

- Parts B and C are repeated for another password checker "password checker v3.0".
- The "password checker v3.0" has additionally countermeasures against timing attacks.
- The countermeasure in implemented in such a way that it ensures the code doesn't exit the loop when it encounters a correct character/password, but runs in its entirety before displaying the output. This is achieved using a XOR function.
- The waveform of the traces is plotted to make sure the timing attack fails after the countermeasure.
- The board is disconnected.

IV. Results

The results of our work is shown in Figures 2-7 respectively. In figure 2, the traces are shown for various password attempts on the CW chip. Trace 0 is the trace where there are 0 correct characters, trace 1 is the trace where there is one correct character, trace 2 has two correct characters and so on. Note that within the first 300 samples, the power trace shows meaningful spikes indicating that processing is occurring. Also note the uniformity of the traces in the last 400 samples, indicating the processing is complete and the board is within some other loop.

In figure 3, we show traces analyzed for the differential results. Trace 3 is used as the baseline trace and it is differentiated against, trace 0, trace 1, trace 2, trace 4, and trace 5. Additionally, in figure 3f and 3g we show differential traces of two different password attempts with the same number of characters correct. Again note, that the first 300 samples show occurring processing with meaningful spikes occurring in these ranges.

In figure 4, we show the traces that occurred during our bruteforce attack. Note that with each attack, which corresponds to a colored trace on the figures, there is an initial spike with each attack corresponding to the letter bruteforced. Also note, in figure 4e, that the attack trace on the countermeasure differs meaningfully from the traces in figures 2 and 3. This trace shows how the countermeasure protected the chip from the previous data leakages.

Finally in figures 5-7, we show the portions of the code in the Jupyter notebook of the bruteforce attack. The bruteforce attack is fully automated and bruteforces the password one letter or character at a time. In figure 7, due to the countermeasure, the bruteforce attack attempts to find the correct character, but cannot identify it due to the code continuing to run despite whether the correct character was entered or not.

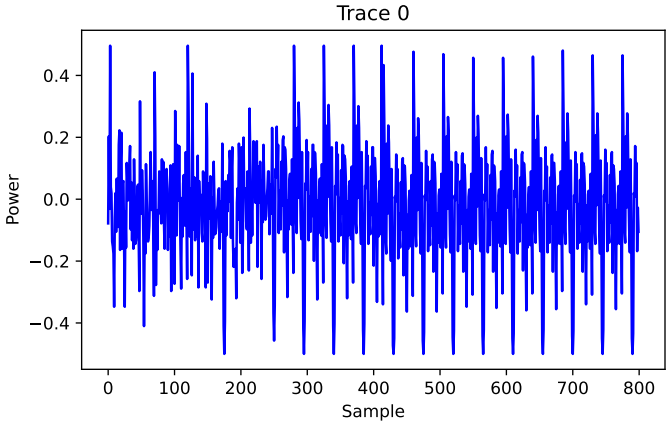
V. Discussion

These results in Section IV demonstrate that secret information is leaked by recording the time a power spike occurs in a function operation on the CW nano-board. Each naive password checker breaks under the timing attack due to their checking algorithms which immediately or with negligible delay output the check result. By noticing these patterns in the power traces the entire secret password is able to be bruteforced.

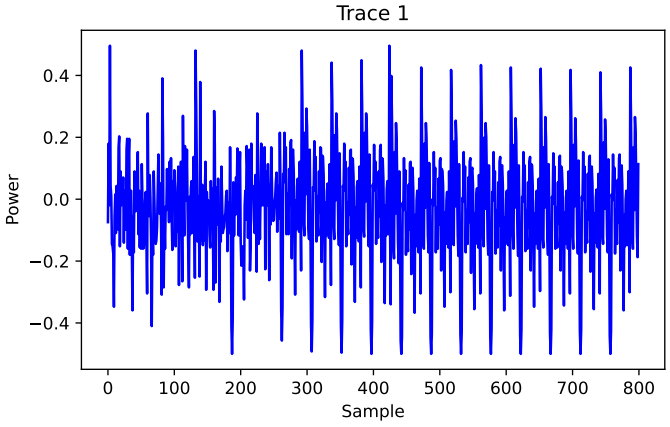
From the graphs and tables presented in Section IV, it is evident that timing attacks are one of the major SCA attacks that can be used to capture secret information as shown by this experiment using CW nano board. With a basic side channel analysis, one can determine how the optimizations of an algorithm leak side-channel information.

In the real world applications, timing attacks are especially dangerous as they are majorly mounted on public-key cryptosystems such as RSA where the execution time of its square and modular multiplication functions linearly varies with the number of high bits in the secret key, and the entire key can be recovered after extensive statistical analysis or large key inputs [3]. This work shows the importance of considering side-channels when implementing secure code. Optimizations for performance can contradict optimizations for security, and both must be considered to create robust computing solutions.

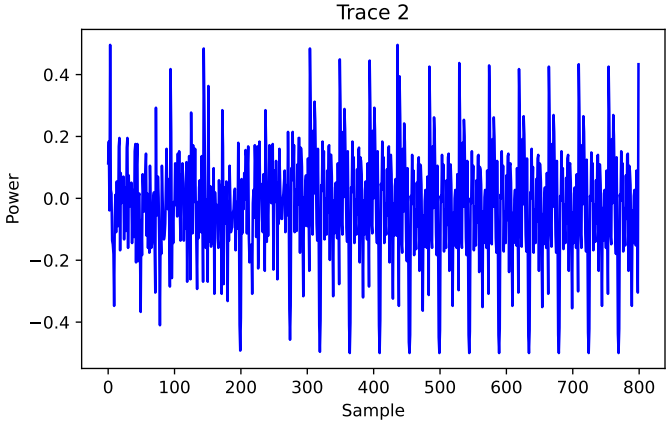
In this experiment, designing a countermeasure was difficult. The difficulty occurred due to the level of abstraction of C programming. We had to try various code implementations before we found one that worked correctly and also did not leak side channel data that allowed the bruteforce attack. Besides this, the rest of the experiment went smoothly. If we were to implement this experiment again, we would create multiple countermeasures and test them for efficiency and weakness.



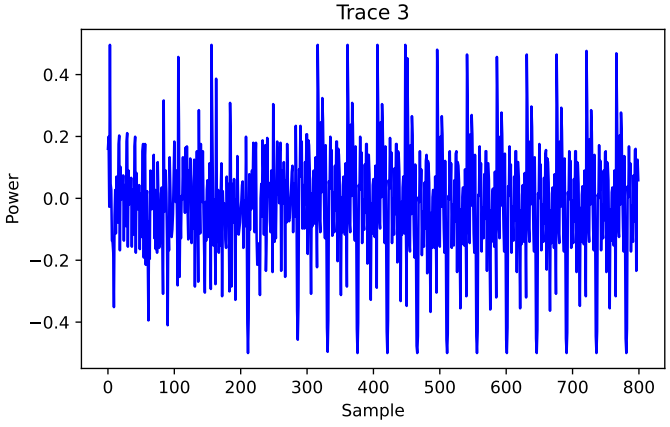
(a)



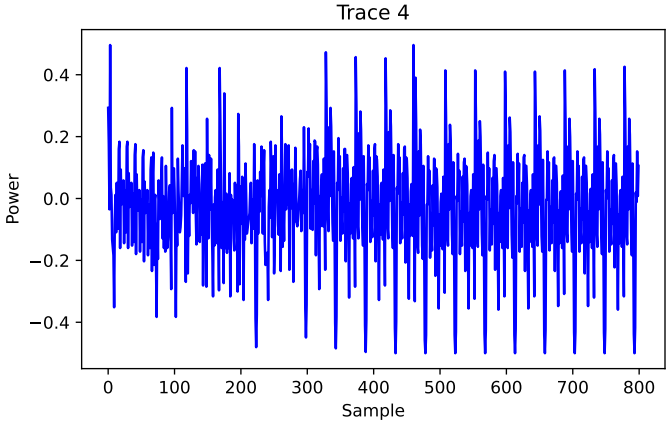
(b)



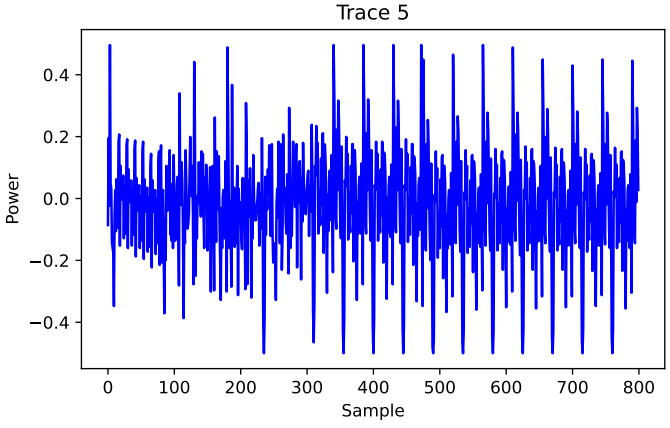
(c)



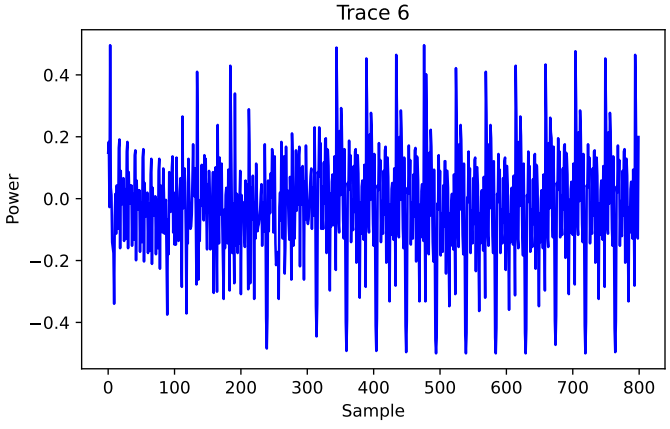
(d)



(e)

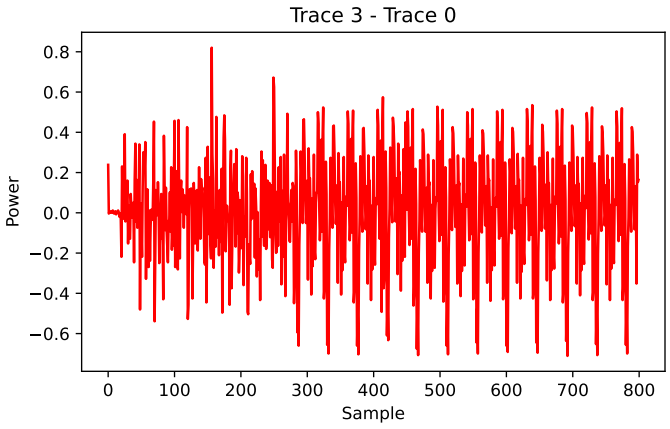


(f)

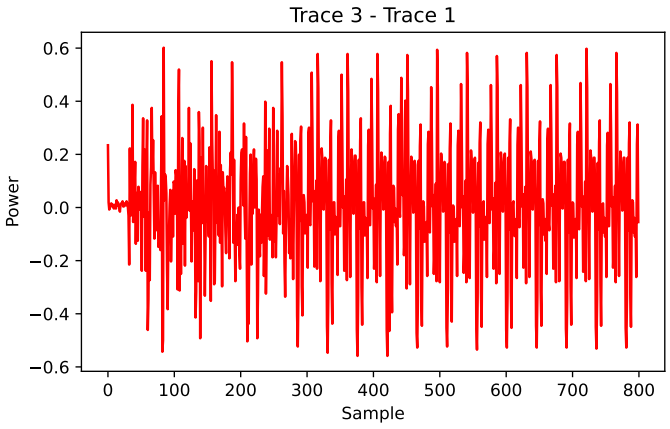


(g)

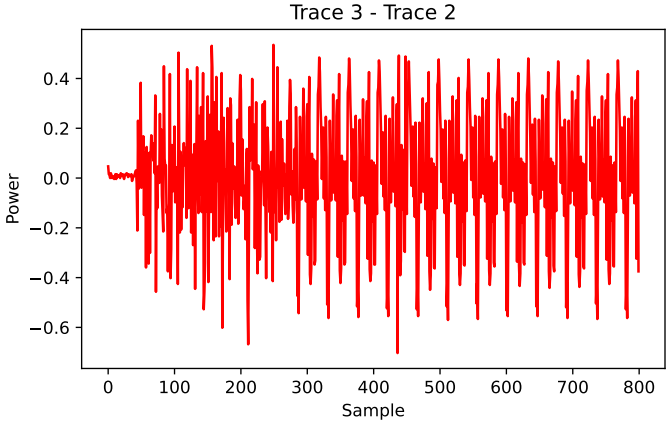
Fig. 2: Power Traces of Various Password Attempts



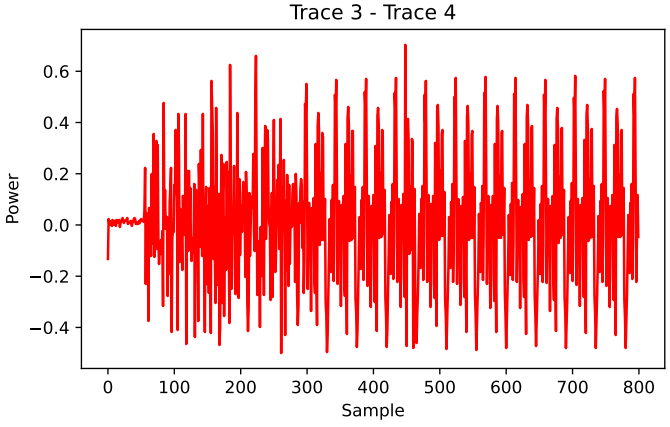
(a)



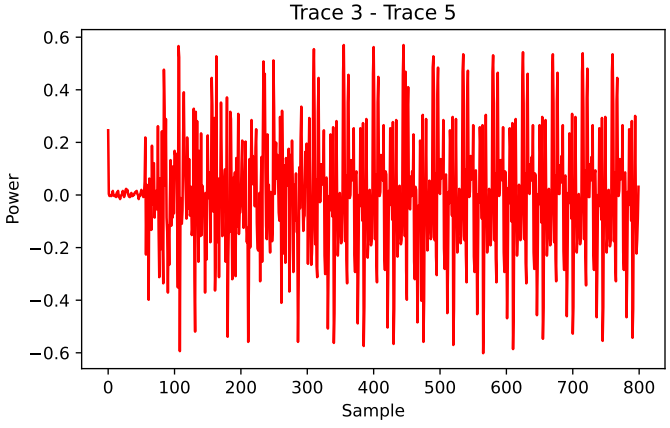
(b)



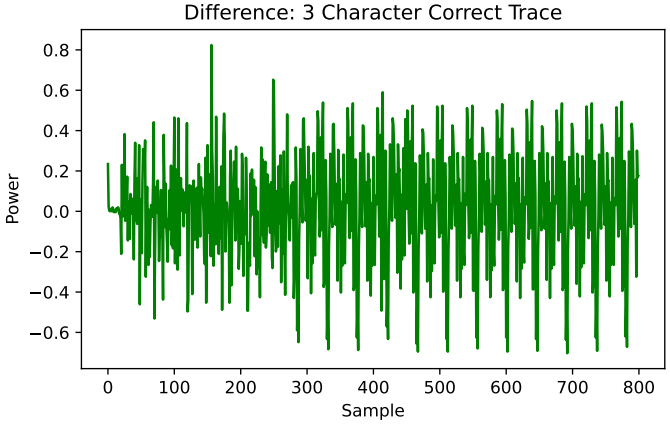
(c)



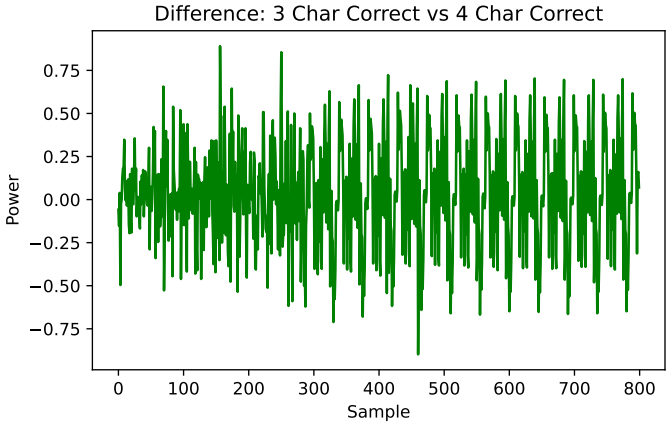
(d)



(e)

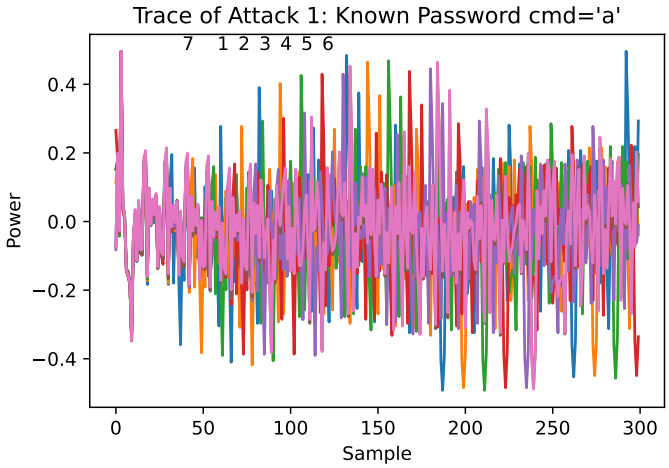


(f)

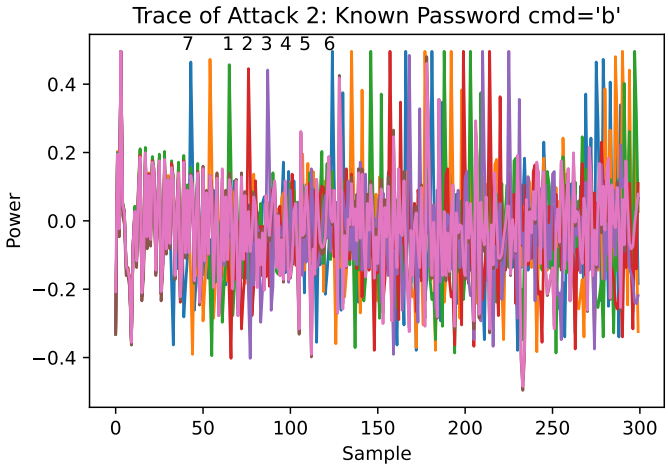


(g)

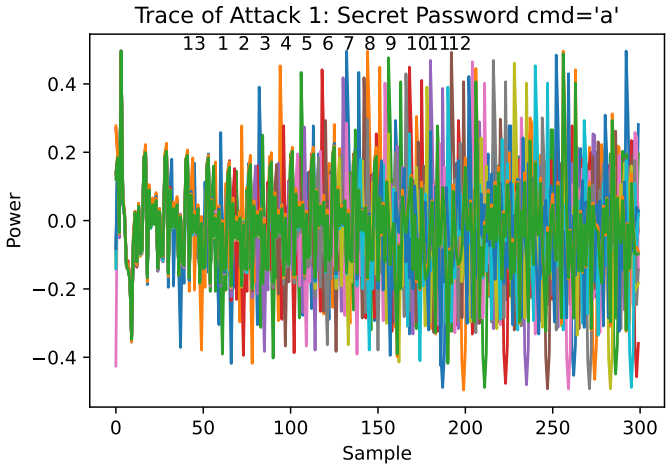
Fig. 3: Differential Power Traces of Various Traces and Correct Character Traces



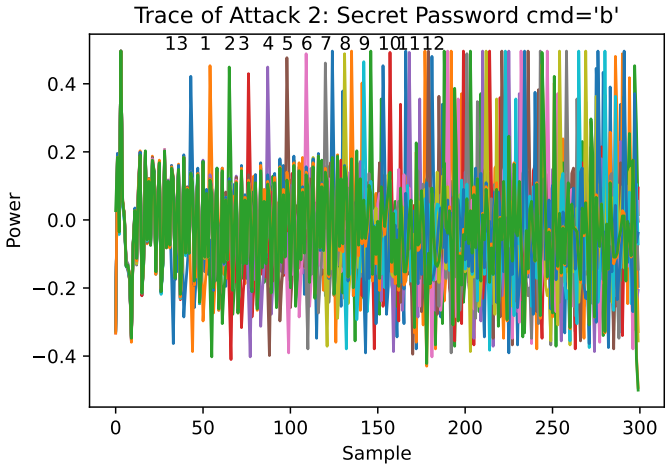
(a)



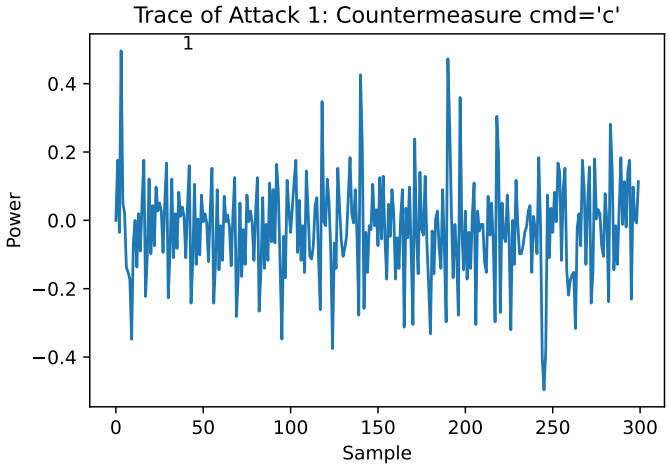
(b)



(c)



(d)



(e)

Fig. 4: Attack Power Traces

6.1.1 Test with cmd='a'

Find the password with `password_checker_v1`

Complete missing parts

In [38]:
```
# "training-CWNANO.hex"; cmd='a'
password, trace_list, pos_list = find_password("a")

print("Password:", password)

# Verify password
msg = check_password(password, "a")
print(f"Confirm {password}: {msg}")

# do not change these lines below:
assert msg == "ACCESS GRANTED"
print("✔ OK to continue!")
```

```
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
```

```
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
```

```
EO Charlist
Password: USFCSE
Confirm USFCSE: ACCESS GRANTED
✔ OK to continue!
```

(a)

6.1.2 Test with cmd='b'

Find the password with `password_checker_v2`

Complete missing parts

In [41]:
```
# "training-CWNANO.hex"; cmd='b'
password, trace_list, pos_list = find_password("b")

print("Password:", password)

# Verify password
msg = check_password(password, "b")
print(f"Confirm {password}: {msg}")

# do not change these lines below:
assert msg == "ACCESS GRANTED"
print("✔ OK to continue!")
```

```
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
EO Charlist
Password: USFCSE
Confirm USFCSE: ACCESS GRANTED
✔ OK to continue!
```

(b)

Fig. 5: Pictures of the Bruteforcing Attack on Known Password

7.1.1 Test with cmd='a'

Find the password with `password_checker_v1`

Complete missing parts

In [45]:
```
# "attack-CWNANO.hex"; cmd='a'
password, trace_list, pos_list = find_password("a")

print("Password:", password)

# Verify password
msg = check_password(password, "a")
print(f"Confirm {password}: {msg}")

# do not change these lines below:
assert msg == "ACCESS GRANTED"
print("✔ OK to continue!")
```

```
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
EO Charlist
Password: TimingSCAgzt
Confirm TimingSCAgzt: ACCESS GRANTED
✔ OK to continue!
```

(a)

7.1.2 Test with cmd='b'

Find the password with `password_checker_v2`

Complete missing parts

In [47]:
```
# "attack-CWNANO.hex"; cmd='b'
password, trace_list, pos_list = find_password("b")

print("Password:", password)

# Verify password
msg = check_password(password, "b")
print(f"Confirm {password}: {msg}")

# do not change these lines below:
assert msg == "ACCESS GRANTED"
print("✔ OK to continue!")
```

```
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
WARNING:root:SAM3U Serial buffers OVERRUN - data loss has occurred.
Correct Letter Found
Bruteforcing Character.
Correct Letter Found
Bruteforcing Character.
EO Charlist
Password: TimingSCAszi
Confirm TimingSCAszi: ACCESS GRANTED
✔ OK to continue!
```

(b)

Fig. 6: Pictures of the Bruteforcing Attack on Secret Password

8.2 Attempt attack on countermeasure

Attempt to attack `password_checker_v3` with your same full password attack.

Complete missing parts

In [53]:

```
# "training-CWNANO.hex"; cmd='c'
password, trace_list, pos_list = find_password("c")

print("Password:", password)

# Verify password
msg = check_password(password, "c")
print(f"Confirm {password}: {msg}")

# do not change these lines below:
assert msg == "WRONG PASSWORD"
print("✔ OK to continue!")


```

```
Bruteforcing Character.
EO Charlist
Password:
Confirm : WRONG PASSWORD
✔ OK to continue!
```

(a)

Fig. 7: Pictures of the Bruteforcing Attack on Countermeasure

VI. Conclusion

In this lab, we implemented a SCA timing attack on password checking algorithms to crack passwords by analysing the power traces and determining when a power spike is observed. The attack included first to bruteforce a correct character of the password and then it was scaled up to get all the correct characters of a password by capturing and analysing the power traces for each valid character. After these attacks we attempt to bruteforce another password checking algorithm with countermeasures such as delays and loops that do not break on correct characters to protect against timing SCA attacks. This countermeasure prevented the bruteforce attack from working.

This experiment shows that timing attacks are powerful attacks that can be mounted on chips and it is important to consider these attacks when creating secure code. In this experiment we learned how to bruteforce a FPGA device and how to utilize side-channel analysis for a meaningful attack on hardware. With these things in mind, we would like to further learn of various coding countermeasures and their tradeoffs in preventing SCA attacks.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Experiment 4: Side Channel Analysis Attacks (Part 1)," in *Practical Hardware Security Course Manual*. University of South Florida, Aug 2022.

[2] R. Karam, M. Mozaffari-Kermani, and S. Katkoori, "Experiment 5: Side Channel Analysis Attacks (Part 2)," in *Practical Hardware Security Course Manual*. University of South Florida, Aug 2022.

[3] S. Karthik, "Introduction to timing attacks," 2020. [Online]. Available: https://medium.com/spidernitt/introduction-to-timing-attacks-4e1e8c84b32b