Lab 07: Side Channel Analysis Attacks (Part 4) Template Matching Attacks on RSA

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 laboratory experiment, we delve into the intriguing world of side-channel attacks, a critical aspect of cybersecurity, and its relevance to Internet-of-Things (IoT) technology. IoT is a groundbreaking paradigm, set to revolutionize the way we interact with and perceive the world around us. It encompasses various domains, from smart healthcare to intelligent transportation, with the promise of enhancing our daily lives. However, with great innovation comes an equally significant concern - security. The experiment aims to address the vulnerability of IoT systems, particularly the edge nodes, which serve as the initial data collection points in IoT applications. At the heart of this experiment is the investigation of side-channel attacks on a public key encryption algorithm that could be employed within an edge node to establish secure connections with the IoT cloud. RSA, a widely used public key encryption algorithm, serves as the focal point of this study. RSA security relies on the computational complexity of factoring large prime numbers, making it a formidable candidate for encryption. Our primary objective is to explore the vulnerability of RSA, particularly in a small microcontroller environment, where computational resources are constrained. To achieve this, we delve into the square-and-multiply (SAM) algorithm, used for modular exponentiation in small devices. The experiment utilizes SAM's characteristics and employs side-channel analysis techniques to recover a critical secret key, 'd,' from the power trace.

The experiment introduces the concept of template matching and the Sum of Absolute Differences (SAD) technique. These tools are employed to uncover patterns in the power trace data and recover the secret key, a feat that underscores the importance of security considerations in IoT device design. Through this exploration, we aim to highlight the potential vulnerabilities of IoT systems and the necessity of robust countermeasures. Furthermore, the experiment underscores the significance of designing secure algorithms and countermeasures for IoT applications to ensure the confidentiality and integrity of sensitive data.

II. Reading Check

Question 1: What vulnerability is present in the SAM algorithm as presented?

Answer: The vulnerability present in the SAM (Square-and-Multiply) algorithm, as presented in the experiment, is related to the potential disclosure of the secret key 'd' used in the RSA encryption process [2]. RSA is a public key encryption algorithm that relies on the computational complexity of factoring the product of two large prime numbers. In this algorithm, modular exponentiation is a critical operation that involves raising a base 'b' to the power of a secret exponent 'd' modulo a public key modulo 'm.'

Listing 1: Square-and-Multiply (SAM)

```
// b: base
// m: modulo
// n: number of bits it takes to represent the exponent
// exp_bin: exponent represented as an array of 'n' bits
// r: result from the modular exponentiation

r = b;
i = n - 1; /* skips MSB '1' */
while(i > 0)
{
    r = (r * r) % m;
    if(exp_bin[--i] == 1)
    {
            r = (r * b) % m;
    }
}
return r;
```

Fig. 1: SAM algorithm

The SAM algorithm, designed for efficient modular exponentiation, utilizes the binary representation of the exponent. It performs one or two operations per bit of the exponent, depending on whether the bit is 0 or 1. This design characteristic allows attackers to exploit the power side-channel to potentially recover the secret key 'd.' By analyzing subtle variations in power consumption during the

SAM algorithm's execution, knowledgeable attackers can deduce information about the number of bit operations, the bits themselves, and eventually, the secret exponent 'd.'

This vulnerability highlights the importance of safeguarding cryptographic operations against side-channel attacks, as even seemingly secure algorithms like RSA can be susceptible to information leakage through power traces. In the context of IoT security, where resource-constrained devices may be used, understanding and addressing these vulnerabilities is crucial to ensure data confidentiality and system integrity.

Question 2: What is template matching? How does SAD help you identify patterns in a signal?

Answer:

Template matching is a technique used to identify specific patterns or segments within a signal, typically a waveform or data stream. In the context of the experiment, template matching is employed to find a specific pattern within the power traces generated during the execution of the SAM (Square-and-Multiply) algorithm. This pattern corresponds to the part of the signal where specific mathematical operations are performed, such as the multiplication of base 'b' or the additional multiplication when the current bit of the exponent is 1.

The Sum of Absolute Differences (SAD) is a mathematical metric used to quantify how similar one waveform (or template) is to another, by measuring the absolute differences between corresponding data points in the two waveforms. In the context of the experiment, SAD is used to compare the power trace from the SAM algorithm's execution with a predefined template that represents the expected pattern of power consumption during specific mathematical operations. The SAD metric measures the degree of similarity between the template and the corresponding segment of the power trace [3].

By calculating the SAD between the template and each segment of the power trace as the template "slides" along the trace, it becomes possible to identify the segment of the power trace that most closely matches the template. This closest match represents the region where the desired mathematical operations, like modular exponentiation, occur. SAD helps in pinpointing this pattern within the signal by producing lower SAD values when the template closely aligns with the trace data and higher values when they differ significantly. As a result, SAD assists in identifying the regions in the signal where key operations are executed, which is essential for recovering the secret key 'd' from the power traces.

Question 3: How can template matching be used to attack implementations of algorithms where random delays have been used as a countermeasure

Answer: Template matching is a powerful technique used to identify patterns in power traces and recover sensitive information in side-channel attacks. However, its effectiveness may be reduced when dealing with countermeasures like random delays. In the presence of random delays, the power consumption patterns become less predictable, making it challenging to create accurate templates for matching. Determining suitable thresholds for template matching can be complicated due to the variability introduced by these delays. Additionally, attackers may face a higher risk of false positives and false negatives in their analyses. To overcome these challenges, attackers may need to employ more advanced methods, including machine learning, to adapt to the presence of random delays and improve the accuracy of their attacks. This underscores the need for robust countermeasures and secure design practices to protect against side-channel attacks, even in the presence of template matching [4].

III. Methods

In this experiment, we are utilizing ChipWhisperer (CW) Nano board., as depicted in Fig 2. 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.

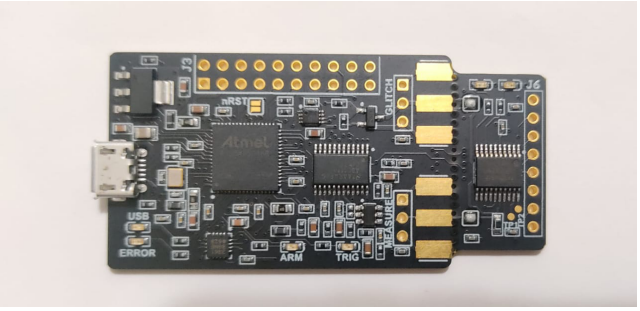


Fig. 2: ChipWhisperer Nano Board

A. Hardware and Software Setup

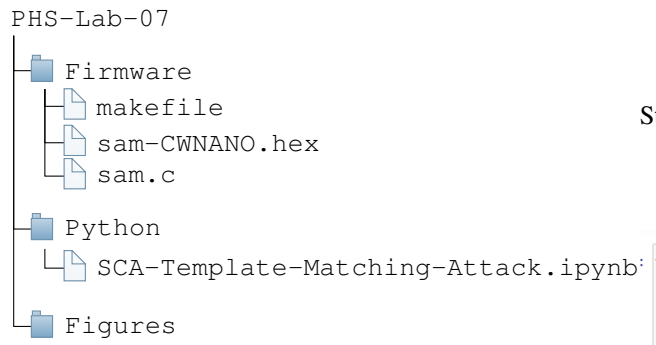
Our procedure for Hardware and Software setup will take the following steps:

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 3.

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-07" directory.

Step 3: Unzip PHS-Lab-07 Content-Within the "PHS-Lab-07" folder, we can upload the provided "PHS-Lab-07.zip" file. Run the unzip code located in the "RUN-TO-ZIP.ipynb" file to extract the contents of "PHS-Lab-07.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.
The directory is structured as follows:



a) Firmware: It contains makefile and sam.c which contains SAM algorithm and compiling makefile will creates sam-CWNANO.hex which is used to program the CW Nano.

b) Python: Here, we will find a Jupyter Notebook that provides a step-by-step guide for executing the attack. The notebook will walk us through the entire process.

c) Figures: To maintain a well-organized record of our results, it's recommended to save all the waveforms and plots generated within the notebook in this directory. This will help keep our data organized and readily accessible for analysis and documentation.

B. Experimental Procedure

Part A: 6A: Template Matching 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 sam-CWNANO.hex file which comes after executing the makefile in sam.c.

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.

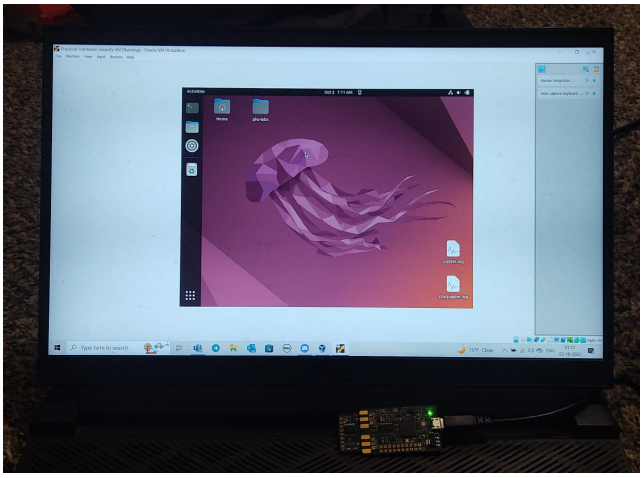


Fig. 3: Hardware and Software Setup

Step 4: To test the 'e' function we will verify the result with pow() function and the snippet code and output is shown in Fig 4.

```
# Test 'e' #1
out = run_firmware_e(62)
ans = pow(0x42,62,mod=0x17)
print("Output:", out)
print("Answer:", ans)
# do not change these lines below:
assert out == ans
print("✔ OK to continue!")
```

```
Output: 2
Answer: 2
✔ OK to continue!
```

```
# Test 'e' #2
out = run_firmware_e(65)
ans = pow(0x42,65,mod=0x17)
print("Output:", out)
print("Answer:", ans)
# do not change these lines below:
assert out == ans
print("✔ OK to continue!")
```

```
Output: 15
Answer: 15
✔ OK to continue!
```

```
# Test 'e' #3
out = run_firmware_e(999)
ans = pow(0x42,999,mod=0x17)
print("Output:", out)
print("Answer:", ans)
# do not change these lines below:
assert out == ans, "output does not match the correct answer"
print("✔ OK to continue!")
```

```
Output: 5
Answer: 5
✔ OK to continue!
```

Fig. 4: Output of Testing e function

Step 5: As we will be running the "SAM" algorithm multiple times, we have implemented a more efficient alternative within the firmware, accessible through the simpleserial command 'f.' This function is faster than the 'e' function, as it employs 32-bit arithmetic

and doesn't transmit a value over the serial interface. To carry out the attack successfully, we must generate power traces using the ChipWhisperer (CW) platform.

Step 6: we need to capture power consumption traces while sending specific key to the target device. We perform 30 trace captures in a loop and the resulting power consumption traces are recorded and stored in 'trace_array'. The code snippet to record traces is shown in Fig 5.

```
# Function: get_pwr_trace()
#   Input:  'exp_int' - integer exponent
#   Output: 'trace'   - power trace collected from the CW as a Numpy Array
def get_pwr_trace(exp_int: int,cmd) -> bytearray:
    # reset target and read start-up text
    reset_target()

    scope.arm()
    exp_bytes= int_to_bytes(exp_int)
    target.simpleserial_write(cmd,exp_bytes)

    ret=scope.capture()

    if ret:
        print("timeout")
    trace=scope.get_last_trace()


    return trace

print("✔ OK to continue!")
```

```
✔ OK to continue!
```

Fig. 5: Code snippet to collect power traces

Step 7: The 30 traces are averaged and stored in avg_traces and plotted using matplotlib. The plot is shown in the Fig 6.

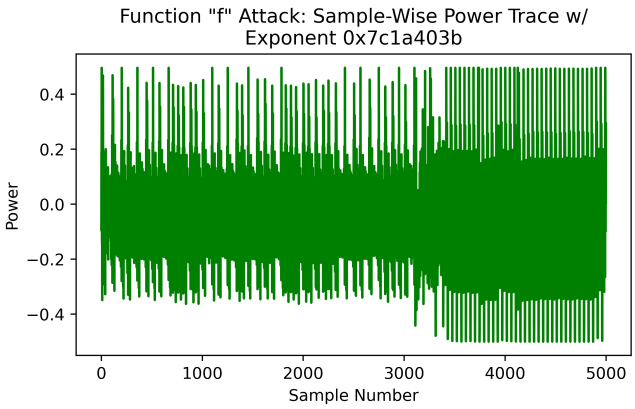


Fig. 6: Plot for Averaged traces

Step 8: Now we have find the template by observing the power trace and the spikes been observed when there is some processing. we will select the template based on that and corresponding template is shown in Fig 7.

Step 9: we need to implement template matching on the averaged trace by using obtained template T and corresponding code snippet is shown in Fig 8

Step 10: The corresponding absolute difference values and plotted using matplotlib and using the values we need to make a binary match vector(bmv) by setting a threshold and the value less than threshold depicts a match for the template. we will consider a list and append the value=1 when there is a match and 0 otherwise [5].

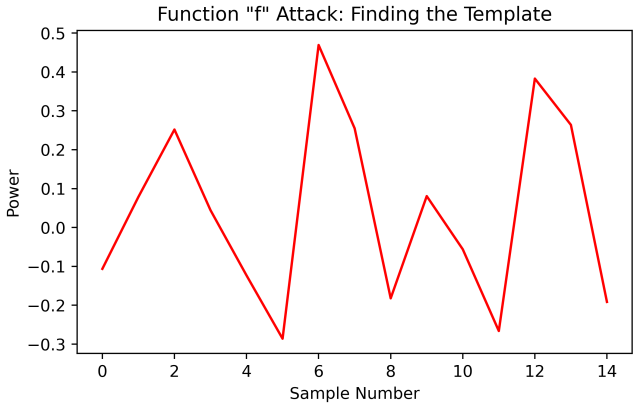


Fig. 7: Plot of Selected template

```
abs_diff_sum=[]
for i in range(len(avg_traces)-len(T)+1):
    abs_diff_sum.append(sum([abs(avg_traces[i+j] - T[j]) for j in range(len(T))]))

print(abs_diff_sum)
```

Fig. 8: Code snippet for template matching algorithm

Step 11: The indices where the match is obtained can be stored in a list which will help in finding the key. The corresponding plots of binary match vector and the aboulte sum differences is shown in the Fig 9.

Step 12: With the Binary Match Vector (BMV) at our disposal, we can pinpoint sections in the data where the key-bits under processing are either 0 or 1. The extent of separation between these matches corresponds to the duration of the intermediate operations. As per the algorithm, extended operations coincide with key-bits equal to 1 (i.e., if exp_bin[–i] == 1).

Step 13: we have stored the indices where the match occured and based on the distance between them we need to find the key. For this we are finding the differences between indices and averaging the values to find the threshold value around the average value.

Step 14: The logic is if the value is less than threshold then the distance is less in bmv which depicts that the corresponding bit is 0 and if the value is greater than threshold then the distance is more which tells the bit is 1.

Step 15: the code snippet for recovering the key and corresponding output is shown in the Fig 10

IV. Results

From the averaged trace, avg_traces[105:120] is selected as template and in the binary match vector we observed the graph as similar to barcode and the threshold distance is taken as 100 as the average value is 97.

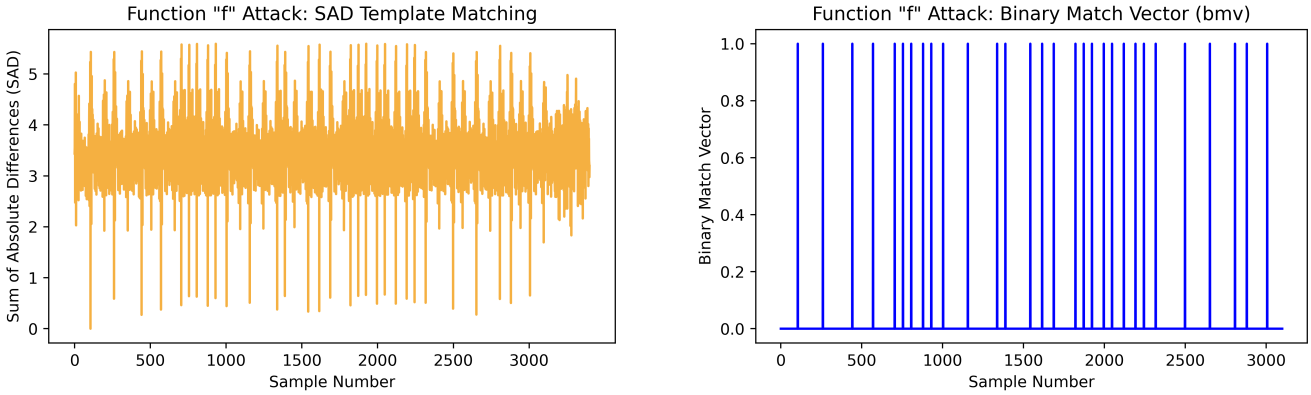


Fig. 9: Plot for SAD Template matching and Binary Match Vector

```python
# we have stored the indices where the match oc
print(ind)
avg_diff=[]
for i in range(1,len(ind)):
    avg_diff.append(ind[i]-ind[i-1])

print(avg_diff)
total_avg=sum(avg_diff)/len(ind)
print(total_avg)

r_key=""
for i in avg_diff:
    if i>100:
        r_key+="1"
    else:
        r_key+="0"
print(r_key)
```

```python
# Expect LSB missing

recovered_key = hex(int(r_key,2))
actual_key = key

print(f"Recovered Key:\t ",r_key,sep="",end="?")
print()
print("Actual Key:\t", format(bin(actual_key)[2:]))
#print(f"Number of Correct Bits: {correct_bits}/32")
```

```
Recovered Key:   11111000001101001000000011101?
Actual Key:      11111000001101001000000011011
```

Fig. 10: Code Snippet for automated key extraction and corresponding key

By visually inspecting the BMV, we detected '1' regions that corresponded to bits being processed in the SAM algorithm. The number of samples in '1' regions provided insight into the key bits being '1.' From this analysis, the secret key (exponent) was manually extracted, and the number of matching bits was compared to the actual key to assess the accuracy of the recovery.

we automated the process of key recovery from the BMV. A program or script was created to analyze the BMV and generate the resulting key, further streamlining the recovery process. The key obtained was "11111000001101001000000011101?" since there is no next match to compare to.

V. Discussion

The results of this laboratory experiment shed light on the vulnerability of RSA encryption to side-channel attacks and the effectiveness of template matching in recovering secret keys from power traces. The primary objective was to explore the vulnerability of RSA, particularly in a small microcontroller environment, where computational resources are constrained. The square-and-multiply (SAM) algorithm was employed for modular exponentiation, and side-channel analysis techniques were used to recover a crucial secret key, 'd,' from the power trace. The results demonstrated the feasibility of this attack, emphasizing the importance of robust countermeasures for IoT systems.

The outcome of the experiment aligned with expectations, revealing the susceptibility of RSA in a small microcontroller setting to side-channel attacks. However, challenges were encountered while defining the template and selecting an appropriate threshold for template matching. The success of the template matching technique heavily relies on the precise identification of the region of interest in the power trace. The presence of random delays in power consumption patterns can significantly impact the effectiveness of this process. To enhance future attempts, it would be advisable to explore more advanced template matching algorithms and to consider machine learning approaches for adapting to dynamic patterns introduced by countermeasures like random delays. This experiment underscores the

significance of secure algorithm design and robust countermeasures in safeguarding IoT systems from potential side-channel attacks. Future endeavors in this domain should focus on innovative methods to mitigate the impact of countermeasures on template matching and improve the accuracy of key recovery processes.

VI. Conclusion

In conclusion, this laboratory experiment delved into the realm of side-channel attacks and their implications for Internet-of-Things (IoT) security. It illuminated the vulnerability of RSA encryption, particularly in constrained microcontroller environments where side-channel information can be exploited for secret key recovery. The key takeaways from this lab are multifaceted. Firstly, it underscores the significance of considering security aspects in IoT system design, especially for edge nodes that serve as the initial data collection points. Vulnerabilities in these nodes can have far-reaching consequences for the overall security of IoT applications. Secondly, the experiment showcased the effectiveness of template matching and the Sum of Absolute Differences (SAD) technique in extracting sensitive information from power traces. This highlights the need for robust countermeasures that can thwart such side-channel attacks.

From this lab, we have learned that the security of IoT systems is a multifaceted challenge that extends beyond encryption algorithms. Future explorations in this area should focus on innovative countermeasures that can effectively mask side-channel information, making template matching attacks less feasible. Additionally, machine learning approaches for adapting to dynamic power consumption patterns introduced by countermeasures could be an exciting avenue to explore. In the ever-evolving landscape of IoT technology, addressing security concerns comprehensively is vital to ensuring the confidentiality and integrity of sensitive data. Therefore, further research into more advanced countermeasures and side-channel analysis techniques remains a promising direction for future investigations.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Experiment 7: Side Channel Analysis Attacks/Introduction," in *Practical Hardware Security Course Manual*. University of South Florida, Nov 2023.

[2] ——, "Experiment 7: Side Channel Analysis Attacks/Background," in *Practical Hardware Security Course Manual*. University of South Florida, Nov 2023.

[3] ——, "Experiment 7: Side Channel Analysis Attacks/Background-1," in *Practical Hardware Security Course Manual*. University of South Florida, Nov 2023.

[4] ——, "Lecture Note 8:Side Channel Analysis/Cryptology SCA Attacks/SAD Example ," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Nov 2023.

[5] ——, "Lecture Note 8:Side Channel Analysis/Cryptology SCA Attacks/RSA Example," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Nov 2023.