Lab 07: Side Channel Analysis Attacks (Part 4) Security of Internet-of-Things (IoT)

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

I. Introduction

Internet-of-Things (IoT) helps connect various devices that are used daily and see applications in domains such as smart healthcare, smart transportation, smart home, smart grid, and so on [1]. This technological transition is beneficial in improving the human lives on this planet [1]. The three key components of this IoT revolution is cheap computing, high bandwidth communications in networking technologies, and finally, intelligent algorithms. However, malicious adversaries have also evolved with the advancements in IoT technologies. For example, the attackers could exploit the vulnerable edge nodes (which cannot be physically protected or continuously monitored) in IoT devices to disrupt legitimate operations [1]. Such as attacks on public key algorithms.

Public-key cryptography consists of a public key and a private key pair. RSA is a widely used public-key algorithm whose security depends upon the computational difficulty to factor the product of two large prime numbers [1]. It utilizes modular exponentiation for encryption and decryption operations using a technique called square-and-multiply (SAM) which takes the base, modulus, and exponent (in binary) as arguments [1]. Therefore, the total number of iterations depends upon the bit length of the exponent and the number of operations per iteration depends upon the value of the current bit (1 or 0) [1].

In this work, the authors mount a security attack on the SAM algorithm by observing and analyzing the operations within a loop to identify whether the current bit being processed is a 1 (which takes two modular operations) or 0 (which takes one modular operation). We gather the power traces from the ChipWhisperer (CW) running the SAM algorithm for integer values (32-bit numbers). The signals are analyzed using template matching where a template signal is swept across the signal under attack and then performing Sum of Absolute Differences (SAD) on the resulting trace [1]. This helps in identifying the regions that are similar to the template using a threshold value, i.e., it is a match if the value falls below the threshold otherwise it is not [1]. Finally, by plotting the graph of the obtained SAD vector, we can determine whether the processed bit is a 1 or a 0.

II. Reading Check

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

Answer: The square and multiply (SAM) algorithm is used to perform modular exponentiation of large prime numbers for a given base, modulus and binary exponent [1]. The algorithm performs one or two operations based on the value of the current bit being processed. If the current bit is 0, the algorithm performs a single multiplication operation. If the current bit is 1, then the algorithm performs 2 operations - a squaring operation followed by another multiplication with the base value [1]. Therefore, the number of iterations depends upon the bits in the exponent, and the operations inside a loop are based on the value of the current bit [1]. Thus, the power consumed by the algorithm differs because of these operations, and observing these differences in power traces can reveal the secret key.

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

Answer: Template matching is when for a given template signal, we try to find similar signals in a trace that we want to attack. This is useful because the power traces from a CPU performing the same operations should match very closely [1]. The Sum of Absolute Differences is calculated for the resulting trace obtained after the template is swept across the signal being attacked. The SAD approach helps us identify patterns in a signal by highlighting the regions in the attacked signal that closely resembles the template, i.e., for the regions that match closely, the output is close to 0 and vice versa [1].

Question 3: How can template matching be used to attack implementations of algorithms where random delays have been used as a countermeasure? Hint: think about how this could be used against the random delay in lab 5 (password attack).

Answer: Template matching could thwart security countermeasures such as random delays in the SAM algorithm. This is because the template matching focuses on the power values whenever a spike is observed in a small region when an operation is performed. Since the delays only add more time before an operation is performed, the added delays will not affect the power consumed by an operation. [**?**].

III. Methods

For this experiment we gather power traces of the operations of the SAM algorithm 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, capturing the power traces, as well as data analysis and plotting graphs.
2) The "Firmware" directory holds the C file containing two versions of the SAM algorithm - for 64-bit number and for 32-bit numbers. The attack is mounted on the 32-bit integers. 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
│  ├─ sam-CWNANO.hex
│  └─ sam.c
├─ Python
│  └─ Lab-05-SCA-Statistical-Analysis.ipynb
└─ Figures
   ├─ avg_trace.pdf
   ├─ sad_output.pdf
   ├─ template.pdf
   └─ bmv.jpg
```

B. Hardware Setup

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

C. Experimental Procedure

Part A: Firmware script

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

- The "sam.c" files contains the SAM algorithm functions for 64-bit and 32-bit numbers.
- These functions are called in the main using simpleserial parameters 'e' (for 64-bit arithmetic) and 'f' (for 32-bit arithmetic).

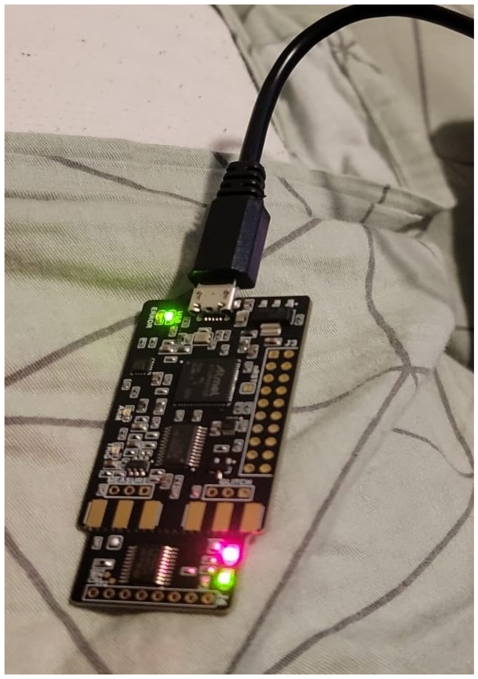


Fig. 1: CW nano board Connected to PC

Part B: Functionality of SAM algorithm using function 'e'

Obtaining the data is done by:

- Python file ""Lab-07-SCA-Statistical-Analysis.ipynb" imports chipwhisperer, matplotlib.pyplot, numpy, and scipy.stats.
- The "sam.c" code is compiled in the python file and the CW is then programmed with the C code using the "sam-CWNANO.hex" file.
- A reset target function is defined to reset the target on the board.
- The functionality of the SAM algorithm is confirmed by comparing the output of the function 'e' (64-bit SAM) to the result of Python's built-in pow() for the base = 0x42, mod = 0x17, and three test exponent values.

Part C: Attacking function the SAM Algorithm

- A function is defined to capture about 20-30 power traces of the board during its processing of SAM algorithm in function 'f' for a given exponent value.
- The captured power traces are then averaged sample-wise for improved signal quality. The output is then plotted, a portion of which with a power spike (processing bit value 1) is identified and saved as a template signal trace. (The template is also plotted.)
- The defined template is then used to perform SAD across the sample-wise average power trace vector. The resulting output is plotted.
- Using a threshold value, the resulting SAD output is then iterated over to determine whether a power spike was a match. If a sample falls below the threshold it

is counted as a match otherwise it is not. Using this logic, a binary match vector (BMV) is created.
- Analyzing the plot of the BMV allows one to determine the key. We take this further however by implementing code in python that puts the position of each match from the SAD algorithm in a list, and then each of these items in the list is analyzed for their distance from each other (resulting in a 1 or 0 for the key.)
- The actual key, the recovered key, and the number of matching bits are returned.
- The board is disconnected.

IV. Results

The results of our work are shown in Figures 2-7 respectively. In figure 2, we show a sample-wise averaged power trace of the SAM Algorithm. Note that within the first 1500 samples, there are multiple spikes on the top of the graph. This information was used to help determine where/how to extract the template needed (shown in Figure 3). We also include, in figure 7, a demonstration that the SAM algorithm functionality matches with the power traces in python.

Figure 3 shows the template we extracted and used for our attack. This template window had to be fine-tuned until accurately captured succinctly the salient data needed for the template matching procedure.

In figure 4, we show the SAD algorithm output after applying the template matching to the power trace. Note that within the first 1300 samples, the SAD values are at or near 0 for salient points in the algorithm. These data points help to clearly show what we need to focus on for our binary vector matching and ultimately the key-breaking attack.

In figure 5 we show the Binary Matching Vector output, that allowed us to complete our the attack. Small distances between spikes indicate a binary value of 0 whereas larger distances indicate a 1. Additionally, if one zooms in closely on the graph, the 1s also correlate with a double power spike (correlating to the thicker lines for 1). Our experimentation and testing showed that when a 1 occurs in the key, there is a duration of about **140 cycles** or more than occur. On the other hand, when a 0 occurs, it ends up being between **30-80 cycles** that occur. The recovery key obtained from the binary matching vector was **001110001000000**. We defaulted the automatic key generation to output a 0 for the least significant bit.

Finally, in figure 6 we show our code for the attack. We implemented the code within the binary matching code. We stored the sample position of every power spike in a list and then iterated through this list, analyzing the distance between spikes (correlating to the processing of the physical hardware) to find the binary key.

V. Discussion

These results in Section IV show that despite using proper encryption, hardware can still be attacked without side channel attack mitigation. While the RSA encryption made utilizing an attack more difficult than it would've been without it, the power leaks from the power side channel allowed us to see into what should've been a black box hardware implementation and attack the devices security.

Additionally, the results demonstrate the wide variety of attacks that are possible with side channel leakage. In previous work, we demonstrated how side channel attacks could be used to obtain a password in code. Here, we show that it can be used as well to break encryption. This work reiterates the importance of implementing security measures against side channel attacks in writing secure code.

In this experiment, the template matching procedure was difficult. Obtaining a proper template for an accurate attack requires a thorough analysis of the power trace and additional fine tuning of the template. If we were to do this experiment next time, we would utilize less samples (around 2500) initially. This may have led to us finding a better template match faster. Of course, it could also be due to a limitation in our display of the graphs and our display hardware that hindered us in obtaining the template faster. Besides experimenting with template matching, the rest of the experiment went smoothly. We were able to plot power traces, analyze them, implement the binary matching vector and attack the FPGA without much issue.

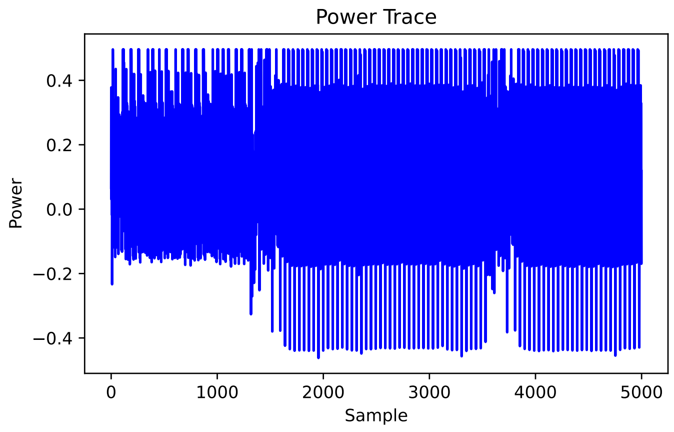


Fig. 2: Power Trace of SAM Algorithm

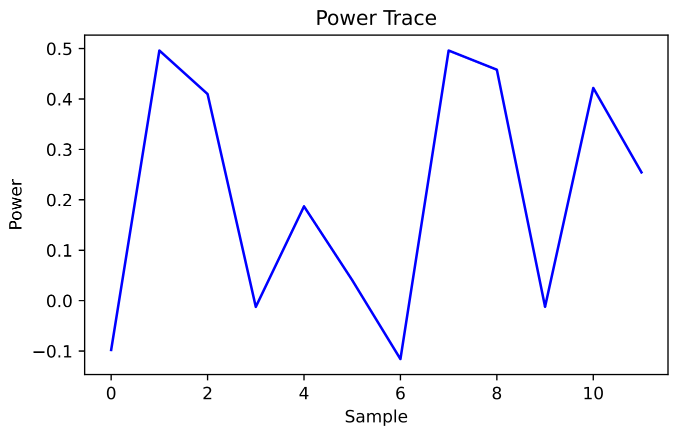


Fig. 3: Template for Template Matching

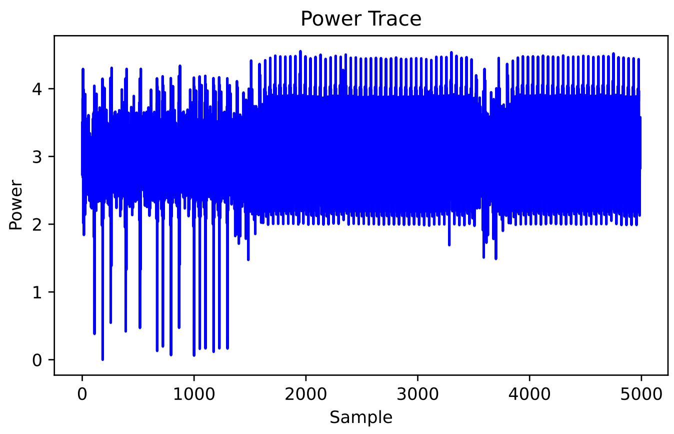


Fig. 4: SAD Algorithm Output

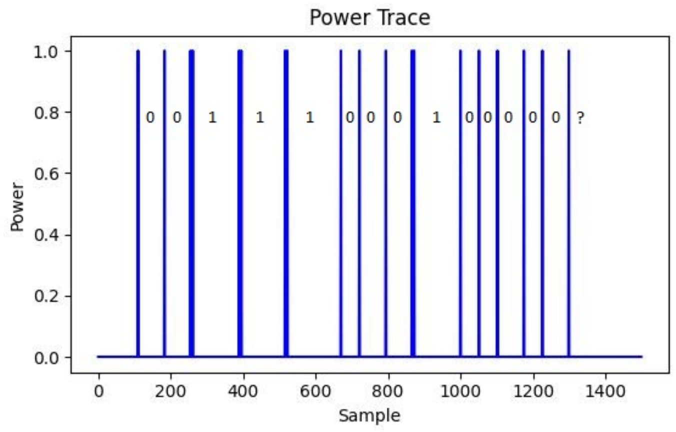
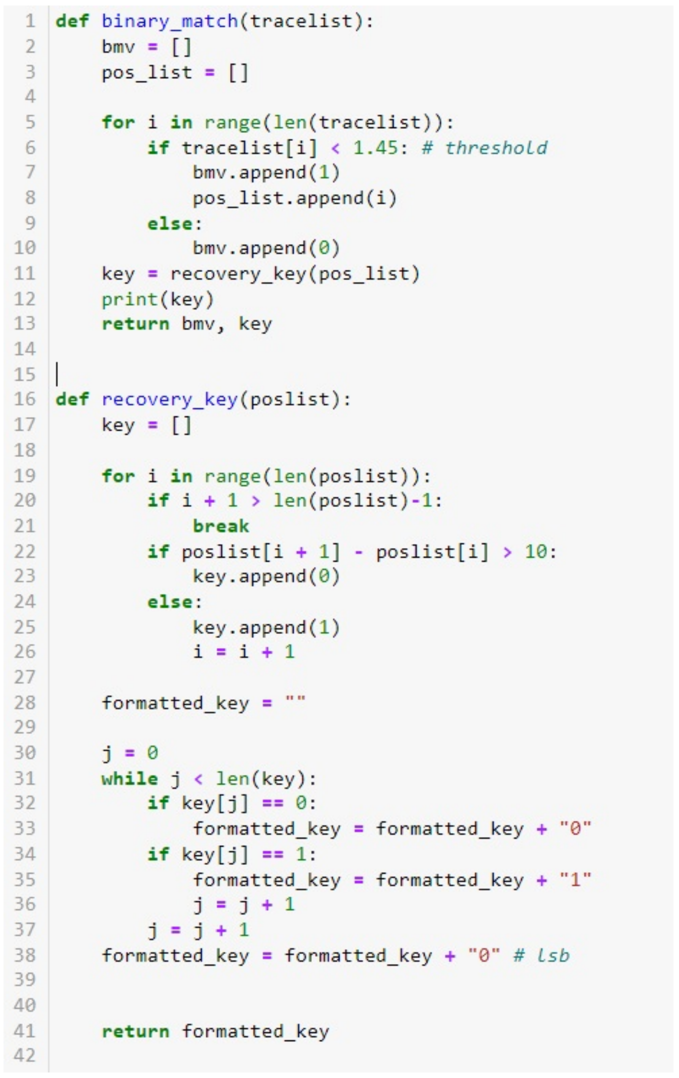
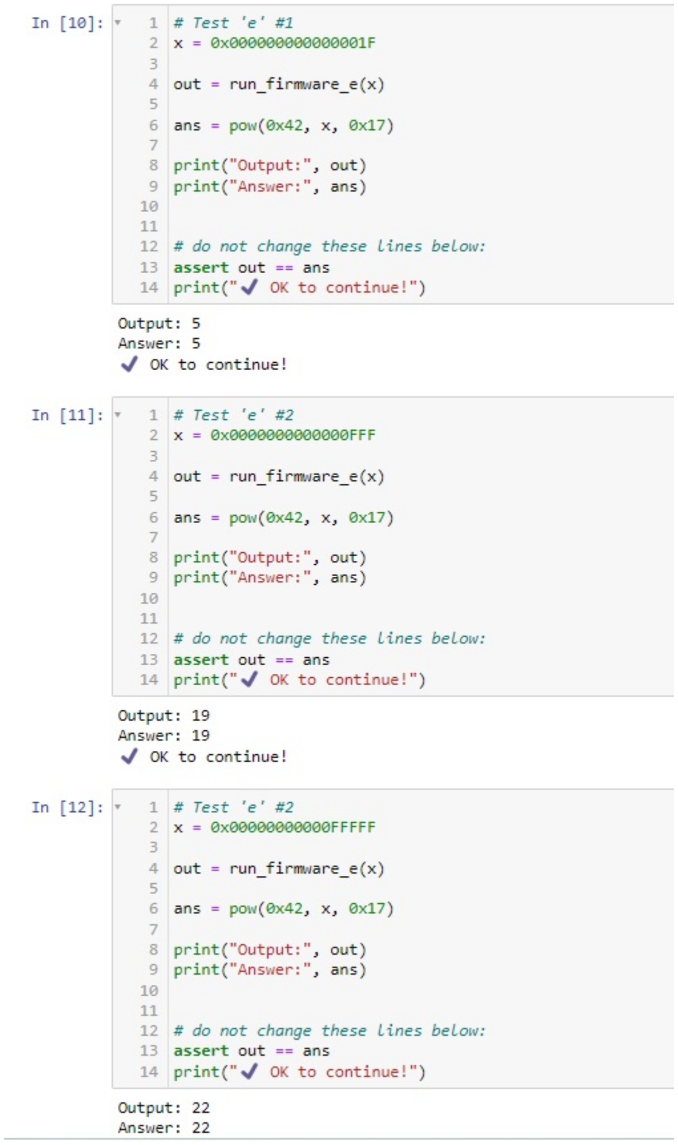


Fig. 5: Binary Matching Vector Output



```
def binary_match(tracelist):
    bmv = []
    pos_list = []

    for i in range(len(tracelist)):
        if tracelist[i] < 1.45: # threshold
            bmv.append(1)
            pos_list.append(i)
        else:
            bmv.append(0)
    key = recovery_key(pos_list)
    print(key)
    return bmv, key


def recovery_key(poslist):
    key = []

    for i in range(len(poslist)):
        if i + 1 > len(poslist)-1:
            break
        if poslist[i + 1] - poslist[i] > 10:
            key.append(0)
        else:
            key.append(1)
            i = i + 1

    formatted_key = ""

    j = 0
    while j < len(key):
        if key[j] == 0:
            formatted_key = formatted_key + "0"
        if key[j] == 1:
            formatted_key = formatted_key + "1"
            j = j + 1
        j = j + 1
    formatted_key = formatted_key + "0" # lsb


    return formatted_key

```

Fig. 6: Key Generation Code



```
In [10]:
# Test 'e' #1
x = 0x000000000000001F

out = run_firmware_e(x)

ans = pow(0x42, x, 0x17)

print("Output:", out)
print("Answer:", ans)


# do not change these lines below:
assert out == ans
print("✓ OK to continue!")
```

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

```
In [11]:
# Test 'e' #2
x = 0x0000000000000FFF

out = run_firmware_e(x)

ans = pow(0x42, x, 0x17)

print("Output:", out)
print("Answer:", ans)


# do not change these lines below:
assert out == ans
print("✓ OK to continue!")
```

```
Output: 19
Answer: 19
✓ OK to continue!
```

```
In [12]:
# Test 'e' #2
x = 0x00000000000FFFFF

out = run_firmware_e(x)

ans = pow(0x42, x, 0x17)

print("Output:", out)
print("Answer:", ans)


# do not change these lines below:
assert out == ans
print("✓ OK to continue!")
```

```
Output: 22
Answer: 22
```

Fig. 7: Python POW vs SAM Algorithm

VI. Conclusion

In this experiment, we utilize SCA to perform an attack on an RSA encrypted hardware implementation. We obtained and analyzed the power side channel of the FPGA board, and from this determine a template that captures some particular computation on the device. Utilizing this template, we perform SAD template matching, identify a salient cutoff threshold, and then produce a binary match vector with which we obtain the encryption key.

This experiment is a demonstration of the power of side channel attacks. Despite utilizing RSA encryption, side channel analysis still allowed for a devastating attack on the hardware. We simultaneously learned how to perform a side channel attack on an RSA encrypted device. For future work, we could investigate how a machine learning algorithm could potentially allow for a more automated attack of this sort.

References

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Experiment 7: Side Channel Analysis Attacks (Part 4)," in *Practical Hardware Security Course Manual*. University of South Florida, Aug 2022.