7 Side Channel Analysis Attacks (Part 4)

Security for Internet-of-Things (IoT)

A Introduction

Internet-of-Things (IoT) is an ongoing technology transition with the goal of connecting the unconnected. It has the great promise of positively changing the way humans live on this planet. IoT application domains are varied and diverse such as smart healthcare, smart transportation, smart home, smart grid, etc. Three key technologies are fueling the IoT revolution. First is fast and cheap computing, second is high bandwidth communication and networking technologies, and third is intelligent algorithms. Any IoT system architecture has at least three layers (see Figure 1): edge, communication, and application. In the edge layer, edge nodes collect and filter data of interest. The collected data is transmitted to the application layer through the communication layer. Typically, the communication layer consists of established communication networks such as the Internet or Wi-Fi. The data center houses the application layer where software applications process the collected data to extract information for intelligent decision-making.

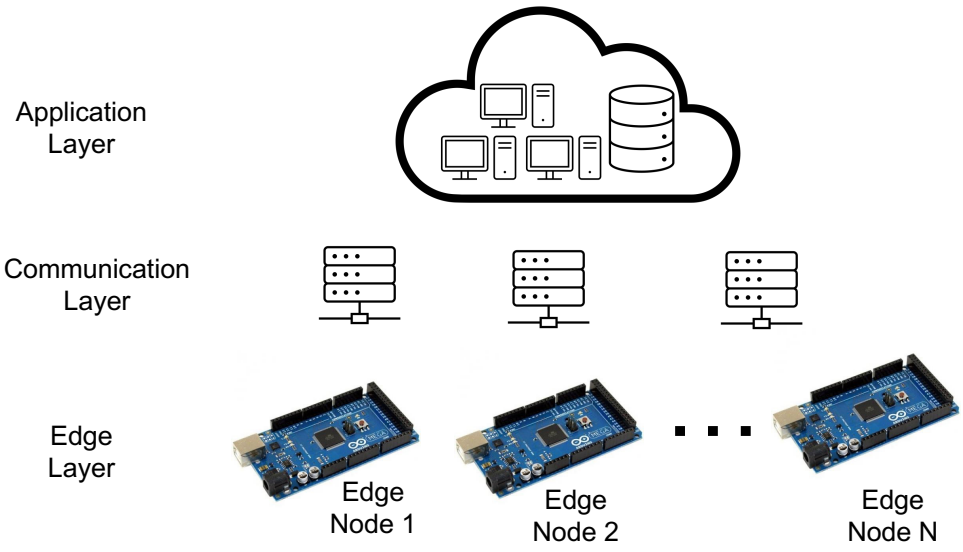


Figure 1: Illustration of an IoT system.

Take, for example, an IoT system for a smart grid. Each home on the grid will have a smart meter (an edge node) to measure the power usage and transmit that data to the Cloud. As an edge node cannot be physically protected or monitored continuously, it can be easily attacked. The attacker can get hold of the edge node and interfere with the legitimate operation. For example, they can modify the transmitted data. Typically the transmitted data is encrypted. In this lab, you will perform an attack on a public key encryption algorithm, similar to one that could be used in an edge node to establish a secure connection to the Cloud.

B Background

RSA is a public key encryption algorithm, with one public key and one private key. Mathematically, the security of RSA is derived from the computational difficulty of factoring the product of two large prime numbers. Encrypting a plaintext p requires first representing it as an integer, then computing the ciphertext as $c = p^e \bmod m$, where e is a public key exponent, and m is a public key modulo. To decrypt, the plaintext is computed as $p = c^d \bmod m$, where $d = e^{-1} \bmod \phi(m)$ and is kept secret. The value of $\phi(m)$ (Euler's totient function) depends on two secret large prime factors (p and q) of m, which is difficult to extract. In other words, if $m = p * q$ (p and q are primes), then $\phi(m) = \texttt{LCM}(p-1, q-1)$. In a naïve implementation, the secret key d may be detected using power side-channel analysis (SCA).

RSA encryption and decryption involve *modular exponentiation*. Since the keys e and d will be extremely large in practice (hundreds of digits), we must use a special algorithm for computing them, especially on a small microcontroller. One technique is the *square-and-multiply* (SAM) algorithm, which takes as arguments the base b, modulus m, and exponent in binary *exp_bin* – which for each bit of the exponent, performs one or two operations. To begin, the base b is copied into a temporary variable r. If the current key bit is a 0, $r = (r * r) \bmod m$, and the loop iterates to the next key bit. If the current key bit is a 1, the same function $r = (r * r) \bmod m$ is performed, but then, it is multiplied once more by the original base value, i.e., $r = (r * b) \bmod m$:

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;
```

Hence, the number of iterations depends on the number of bits it takes to represent the exponent, and the number of operations inside the loop depends on the current key bit. We will leverage both of these factors to effectively read the secret key d right from the power trace (after processing). Note: the SAM algorithm skips the MSB '1' in the exponent for the `while` loop.

To process the signal, we can use a *template matching* approach. We will define a waveform that represents the signal we are trying to detect. Since the CPU is repeating the same instructions, we would expect the power signature to match very closely (assuming there are no countermeasures

in place). Begin by defining the portion of a trace that includes the region of interest; for example, signal s may have 10,000 samples, but the region of interest may be $s[200:500]$. This becomes the *template* signal. Sweep this template along every point in the signal under attack, subtracting the two, taking the absolute value, and adding the result. This is called the **Sum of Absolute Differences**. If the two regions match very closely, you will get an output close to 0, but if they are very different, the output will be higher.

As an example, consider the signals $s = [0, 1, 2, 0, 5, 6, 1]$ and $a = [2, 3, 1, 1, 6, 2, 3]$, where a is the signal to be attacked, and s contains your region of interest (template): $T = s[2:4] = [2, 0, 5]$. To calculate the first output, compute $a[0:2] - T = [2, 3, 1] - [2, 0, 5]$ which gives $[0, 3, -4]$. The sum of the absolute values is $y[0] = 0 + 3 + 4 = 7$. Move the window to the next indexes of a, computing $a[1:3] - T = [3, 1, 1] - [2, 0, 5] = [1, 1, -4]$, so $y[1] = 1 + 1 + 4 = 6$. Continue until we have the final output $y = [7, 6, 3, 10, 8]$. It is clear that the region in the middle of y (i.e., $y[2] = 3$) most closely matches the region of interest T identified in s, and this is given in the output y where the number is closest to 0. Note: you do not need 2 separate signals (s and a), one for both works too, as long as it contains your region of interest.

In general, you can use this method to identify regions in the signal that closely match a template, which you need to define. Identifying a suitable region of the signal that can serve as the template can be tricky - you may need to experiment with different inputs, view the resulting traces, differential traces, and so on to identify the region of interest. Selecting the width of the region (how many samples) is also important – too few and you will get too many matches; too many and you will not get a good match elsewhere due to noise or other factors. Finally, the choice of template (start location and width) will impact the threshold you will need to define, where any value below the threshold is a close match, and anything above is not. Proper selection of the template and threshold will enable successful key recovery.

Reading Check

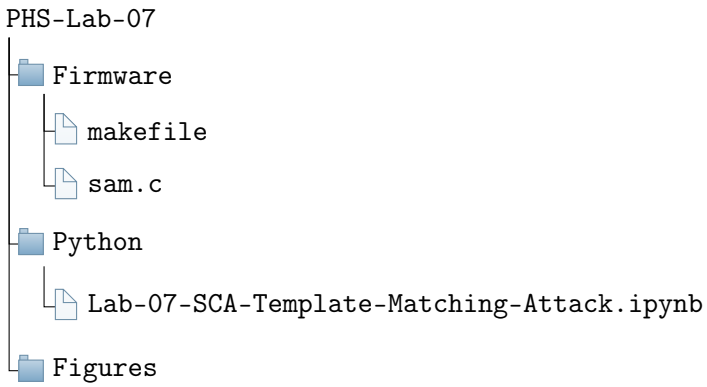
1. What vulnerability is present in the SAM algorithm as presented?
2. What is template matching? How does SAD help you identify patterns in a signal?
3. How can template matching be used to attack implementations of algorithms where random delays have been used as a countermeasure? Hint: think about how this could be used against the random delay in lab 5 (password attack).

C Hardware and Software Tools

For this lab, you will need the ChipWhisperer Nano or equivalent board, as well as a ChipWhisperer VM set up as in the previous labs.

D Getting Started

Download and open the PHS-Lab-07 directory, which contains 3 folders: Firmware, Python, and Figures.



The **Python** directory contains a Jupyter Notebook which walks you through the attack.

The **Firmware** directory contains the script needed to program your CW's target: "sam.c" and "makefile". The makefile is needed to compile the "sam.c" firmware and generate the "sam-CWNANO.hex" file. The "sam.c" is the firmware that contain the SAM algorithm as described before. This firmware has two functions, accessible with simpleserial command 'e' and 'f'. Both perform the same modular exponentiation using the SAM algorithm. The 'e' function uses 64-bit arithmetic (uint64_t datatypes) while the 'f' function uses 32-bit arithmetic (uint32_t datatypes). Hence, be cautious with data overflows.

The **Figures** directory is where you should save all your waveforms generated from the notebook.

E Lab Assignment

1) Set up your notebook to connect to the CW Nano and program the provided firmware files.
2) Begin by testing the 'e' function (i.e., 64-bit SAM) by comparing its output (simpleserial_read) to the Python's built-in pow() function. Note that pow() has an optional 3rd argument for the modulus. The base and modulus in the "sam.c" file are hardcoded as 0x42 and 0x17, respectively. Construct a 16-element bytearray and fill in the first few bytes with hex values, e.g. exponent = bytearray([0]*16), exponent[0] = 0x1F, exponent[1] = 0xB2, exponent[2] = 0x42, etc. Limit the exponent to 64-bits. Confirm functionality with 3 test vectors, and include these outputs in your report.
3) The 'f' function is easier to attack because it uses integer (i.e., 32-bits) arithmetic and does not return a value over serial. Acquire 20-30 power traces from the 'f' function. Note the SAM is a fairly long-running operation, and the duration depends on the highest index of a '1' bit, e.g. 0x80000001 will take longer (more iterations) to run than 0x00000001. Therefore, you may need to adjust the number of ADC samples to capture the full operation in the power side channel. Average (sample-wise) the power traces together to improve signal quality. Plot the output.
4) Using any of the techniques you have previously worked with, identify a portion of the signal which represents the processing of a 1. Define a region around this sample point which will serve as the template, for example, starting at sample 500, and lasting 75 samples. Plot your template.
5) Implement SAD template matching to compare this defined template against the average trace from step 3. Plot the output. An example is shown in Fig. 2.

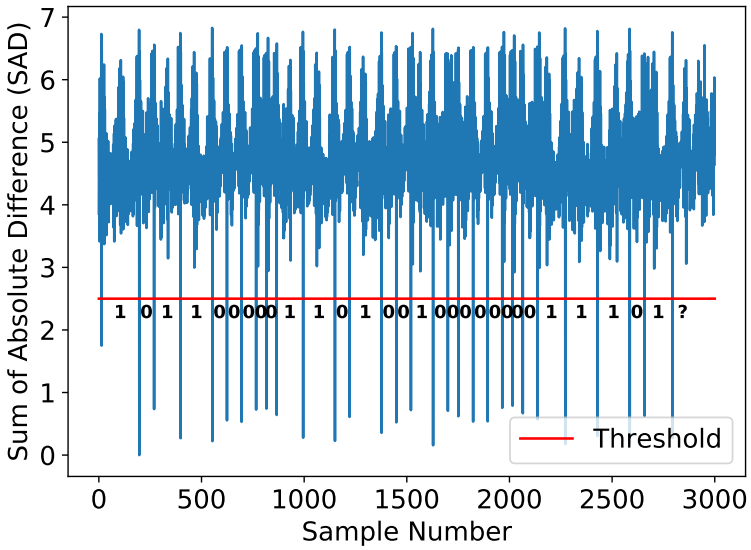


Figure 2: Example output from Step 5. In this case, a threshold of 2.5 allowed for key recovery.

6) Identify a threshold which will serve as a cutoff as in Fig. 2. Any sample which falls below the threshold will indicate a match. Use this threshold to produce a binary match vector (BMV) with a 1 in every position where there was a match to the template. Plot the resulting output.

7) Using the BMV, you should identify regions where the key-bits being processed are 0 or 1. The greater the spacing between the matches, the longer the intervening operations took. Per the algorithm above, long-running operations occur when the key-bit is a 1 ($\texttt{if}(exp\_bin[i] == 1)$). You should now be able to visually inspect this resulting signal and read off the value of the key (exponent) directly from this plot. About how many samples were there in "1" regions? "0" regions?

   Print your recovered key found, the actual key, and the number of matching (correct) bits. Remember that SAM algorithm skips the MSB '1' in the exponent. An example output is shown in Fig. 3.

8) **Graduate Only:** Reading off the key manually (visually) from the plot can be time consuming and prone to errors. Automate the process of analyzing the BMV from step 7 and print the resulting key.

```
1 print("Recovered Key (hex) = 0x%04x"%recovered_key)
2 print("Actual Key (hex)    = 0x%04x"%actual_key)
3 print("Recovered Key (bin) =", format(recovered_key, '032b'))
4 print("Actual Key (bin)    =", format(actual_key, '032b'))
5 print("Number of matching bits =", num_same, "/ 32")
```

```
Recovered Key (hex) = 0x6c1a403a
Actual Key (hex)    = 0x6c1a403b
Recovered Key (bin) = 01101100000110100100000000111010
Actual Key (bin)    = 01101100000110100100000000111011
Number of matching bits = 31 / 32
```

Figure 3: Automated key recovery from Fig. 2. Total number of bits recovered is (32 - 1) = 31. Notice that the LSB is incorrect (this is expected).

Tips/Hints: There are a number of parameters that have to be just right in order to succeed in this attack. Here are some tips/hints for successfully extracting the key/exponent:

1. Identifying a good template is crucial to success. The start location will depend on your particular hardware, so yours may differ from your partner's.
2. If the SAD output does not look right, try adjusting the start location and/or width of the template.
3. If there are very few instances where the match signal is close to 0, or too many instances where it is close to 0, your template is probably starting at the wrong place. If you are close, and have roughly the right number of matches, try finer adjustments of the start location and adjustments to the width. Use the example in the lecture slides as a guide.
4. When selecting a match threshold, think about the number of bits you would expect the exponent to be and use that information to guide you. If your template is correct, your threshold will intersect the match signal once for every bit of the exponent (minus 1 bit). For a 32-bit exponent, the threshold should intersect the match signal 31 times.
5. Template matching will not be able to get the LSB of the exponent (if the LSB is a 1), since there is no next match to compare to. If you entered an exponent with an LSB of 1, don't worry if your recovered key is off by 1 bit.

F Badges and Achievements

This experiment involves the use of the **HDL**, **FPGA**, and **Serial Communication**. By successfully completing the lab you can unlock the following achievements:

- HDL1: Verilog HDL
- FPGA3: FPGA Programming and Interfacing
- COM3: Advanced Serial UART

Depending on the demonstrated skill, each achievement will be further divided into bronze, silver, and gold levels.

G Deliverables

Submit your report along with all code. Follow all report guidelines as detailed in the Appendix. The following result(s) must be included in the report:

- Answers to the reading check questions.
- Example outputs from 3 test vectors demonstrating correct functionality of the modular exponentiation algorithm ('e') as verified by pow().
- Plot of the average power trace from step 3
- Plot of the '1' template from step 4
- Plot of the SAD algorithm output from step 5.
- Plot of the binary match vector (BMV) from step 6, with annotations indicating regions of '1' and regions of '0'.
- Report the approximate duration of the 0 vs 1 regions of the BMV and the complete recovered key from step 7.
- (Graduate) screenshot or pseudo-code demonstrating the automated key recovery algorithm from step 8.