Student Name: Yi-Ting, Hsieh(a1691807)

- **Task 1**

We use Flush and Reload to perform side-channel attack by using Mastik.
1.) Download Mastik and install it with some useful libraries.

Local Machine (Dell Latitude e7250):

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  2
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               61
Model name:          Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
Stepping:            4
CPU MHz:             2981.254
CPU max MHz:         3200.0000
CPU min MHz:         500.0000
BogoMIPS:            5190.40
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            4096K
NUMA node0 CPU(s):   0-3
```

Operating System:

```
Linux ArchLinux 4.18.16-arch1-1-ARCH #1 SMP PREEMPT Sat Oct 20 22:06:45 UTC
2018 x86_64 GNU/Linux
```

2.) Environment setting for Mastik. Implement a simple FR-victim program, provided from the SP-workshop, and run FR-trace to perform attack:

command:

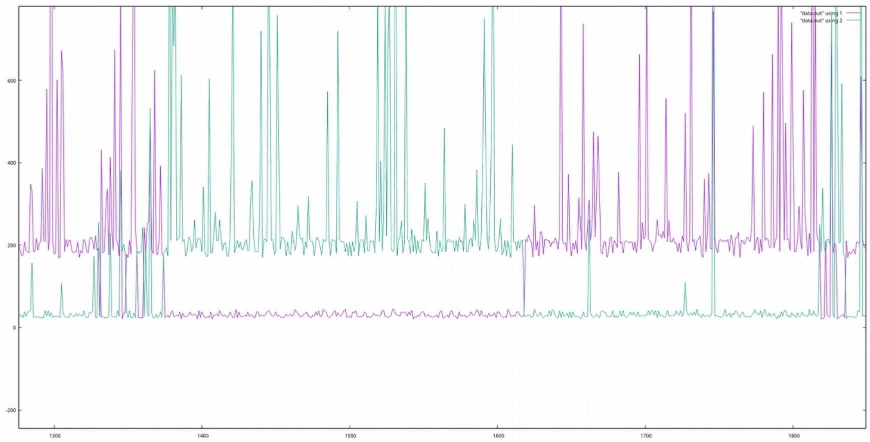
$ ./FR-threshold

output:

	:	Mem	Cache
Minimum	:	122	63
Bottom decile	:	184	69
Median	:	208	80
Top decile	:	375	83
Maximum	:	65535	421

command: (in Mastik/demo folder)

$ ./FR-trace -f ./FR-victim -m x -m y -c 10000 -s 10000



If we use gnuplot to draw our data, we will get the above diagram. We can easily see the pattern 1/1/3/3 as expected.

3.) Compile our num.c with provided ybn.

Command line(in mastik/demo folder):

$ gcc -g -std=gnu99 -I../src -L../src/ -o num num.c ybn.c -g -lmastik -dwarf -lelf -lbfd

or just call: $ make (in mastik/demo folder)

Then, we can perform Flush and Reload attack on our num.

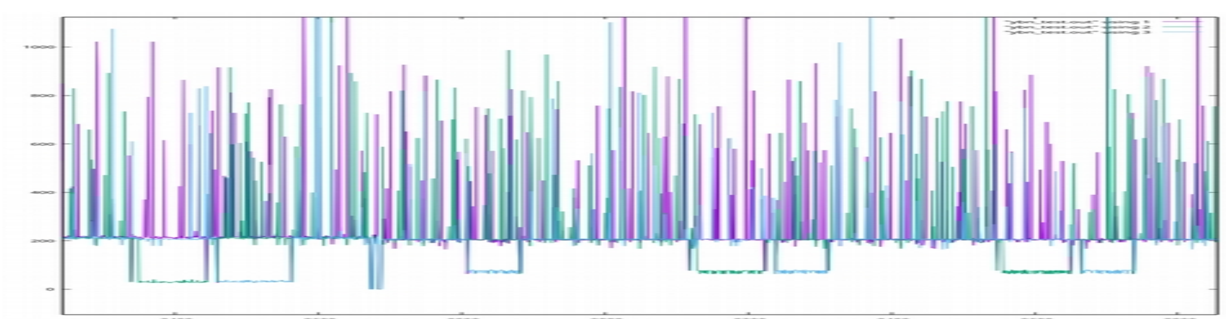
Command:

$ ./FR-trace -f ./num -m ybn_sqr -m ybn_mul -c some_number -s some_number

In order to know what pattern we expect to see, we insert a wait function(it's just a for loop and do nothing) in our ybn_mul, and we can call the following command to collect the data:

$ ./FR-trace -f ./num -m ybn_num -wait -m ybn_modexp -c 15000 -s 6000

If we draw the data, we'll get:



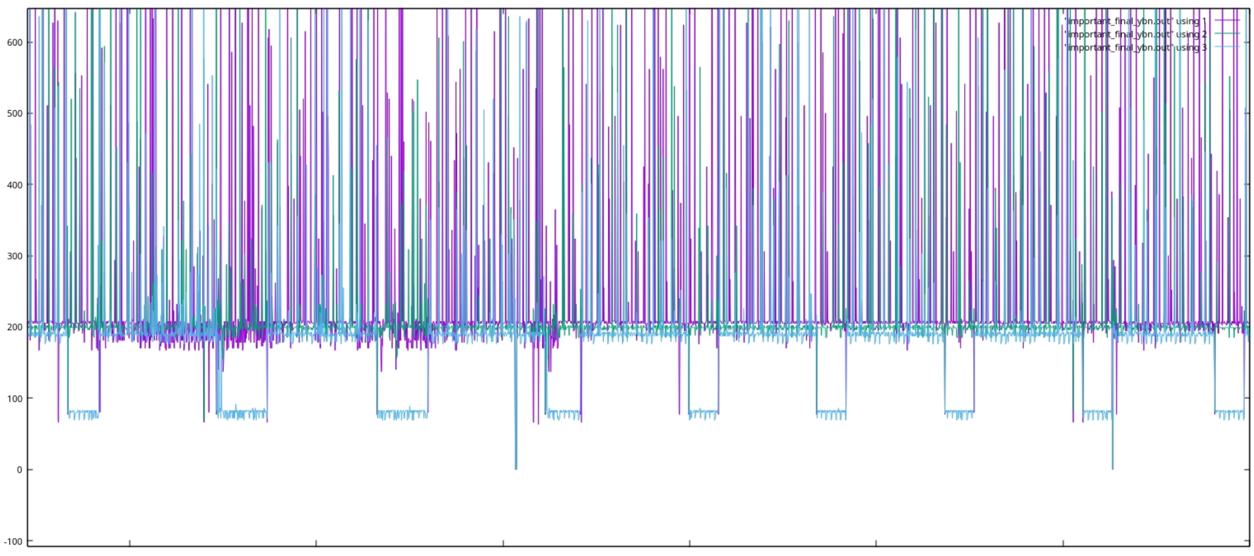
The green line is wait function and the blue line is modexp. We can clearly see the pattern if the program enters if statement in ybn_modexp and call ybn_mul, it will calls wait function in ybn_mul. We get the following exponent bit sequence: "*0011011001110110010110101010001110*". If we search it in our exponent bit, we can get "*00110110011101100101101010100000111*" is the most likely to be the sequence. We can learn that it has 33 bits over 35 bits correct, which is quite good.

4.) Then, let’s run the attack again on our original ybn.c with num.c. However, we dont have wait function to highlight some interval for us to identify whether the program enters ybn_mul or not. We have to find out the right parameters for -s, read the diagram and identify when the program enters the ybn_mul.

We use the following command:

$ ./FR-trace -f ./num -m ybn_mul -m ybn_sqr -m ybn_modexp -c 10000 -s 5400

Unfortunately, the pattern of ybn_sqr function is not that obvious on my computer(I’ve tried it hundred of times with different -s parameter on different machines too, I still couldn’t get the good diagram for ybn_sqr vs ybn_mul). In order to identify the program enters the ybn_mul, we trace ybn_modexp to help us identify different interval. Here is the diagram we got:



The purple line is ybn_mul, green line is ybn_sqr and blue line is ybn_modexp.
If we see there’s a thick purple line in between two ybn_modexp, we can say the program has entered the ybn_mul in such interval and the exponent bit is 1. Otherwise, the exponent bit is 0. By doing this, we can get the exponent bits sequence from the attack, “*00110110011101100101101010*”, which is also in the real exponent bits sequence. If we can run the program multiple times, or we can increase the fetching sample size, we are possible to get all the exponent bits in such side-channel attack. However, sometimes the program may have some noise or random pattern that may affect the correctness of our data.

- **Task 2**

1.) Edit num.c to change the value of variable num in main() to 4, 3, 2, 1, and 0.
We can inspect the source code and learn that this would change the size of our modulus. Then, we can perform the same side-channel attack as described in Task 1. By comparing the accuracy of the result, we can analyze which modulus is attackable.

2.)
when num = 5, using modulus[5]:
we’ve already discussed this in Task-1.

when num = 4, using modulus[4]:
If we use the same parameters for -c and -s arguments, we will notice that it almost double the number of bits been displayed. However, it also lowers down the accuracy to roughly 1 out of 14. It is quite good and attackable.

when num = 3, using modulus[3]:
Moving down to even further by using modulus[3]. We can easily notice that the diagram becomes more dense and the accuracy drops to around 1 out of 11. However, we also notice there’re more bits untraceable(which means the access time of that sample is 0, we have no choice but guess the bit). **Overall, we can still find out the pattern and perform the attack in this stage.**

when num = 2, using modulus[2]:
We capture more bits in the trace again, while the data is quite hard to distinguish whether the bit is 0 or 1. Only around 30% of the bits we can tell is either 1 or 0.

when num = 1, using modulus[1]:
The diagram becomes really dense, we can’t even tell what is going on.

when num = 0, using modulus[0]:
None of the bits are identifiable. Also, the ybn_mul seems always in the cache(the access time is always less than 100 cycles).

3.) Based on the result of our attack, we can say the smallest attackable value is when num is equal to 3(when the modulus[3]). However, there’re more bits untraceable compared to modulus[4] and modulus[5]. It still has very good chance to predict the correct exponent bits.

- **Task 3**

1.) In our ybn_modexp(), we already know we can use side-channel attack on ybn_mul. Therefore, we can modify the order of our if-statement to avoid such attack by doing constant-time programming.

In our original ybn_modexp, it was:

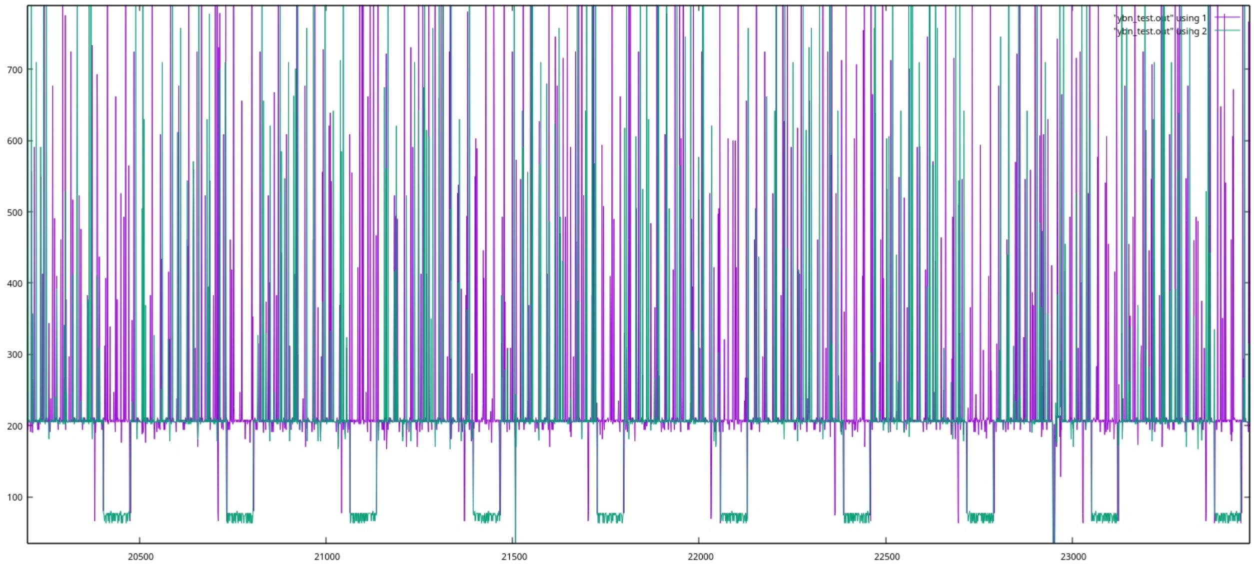
```
ybn_modexp(result, base, exponent, modulus) {
      …
      for ():
            for():
                  result := (result * result) mod modulus

                  if (exp→ybn_data[i] & m):
                        result := (result * base) mod modulus

      return result;
}
```

By applying constant-time programming:

```
ybn_modexp(result, base, exponent, modulus) {
      …
      res2 = ybn_alloc();
      for ():
            for():
                  int condition = exp→ybn_data[i] & m:
                  result := (result * result) mod modulus

                  res2 := (result * base) mod modulus

                  if (condition):
                        result := res2 (copy function → only require constant time)
                  (else: result := result)

      return result;
}
```

When the program is running, it will always call ybn_mul, ybn_div and store it in a ybn pointer, called res2. Then the program inspects the condition and decides whether it has to update result. Because the copy function would only take constant time compared to other function calls(ybn_sqr, ybn_div and ybn_mul), this approach can avoid such side-channel attack. If the spy try to trace ybn_mul, he will only see the program calls ybn_mul in every iteration in the nested for-loop, which is the same as tracing the ybn_sqr. In addition, if the spy tries to trace the copy function, since it only has constant time complexity(maybe it only requires a couple cycles), the spy is very less likely to catch all the copy function call and get the correct bit of exponent. Therefore, the ybn_modexp becomes unbreakable. Though, the spy may still have chance to know how many bits in the exponent. The spy wont be able to find out the correct bits of our exponent.

p.s: adding the diagram:



As you can see, the program calls ybn_mul in every iteration in the for-loop. There’s no way for the spy to attack ybn_mul for gaining information of exponent bits.