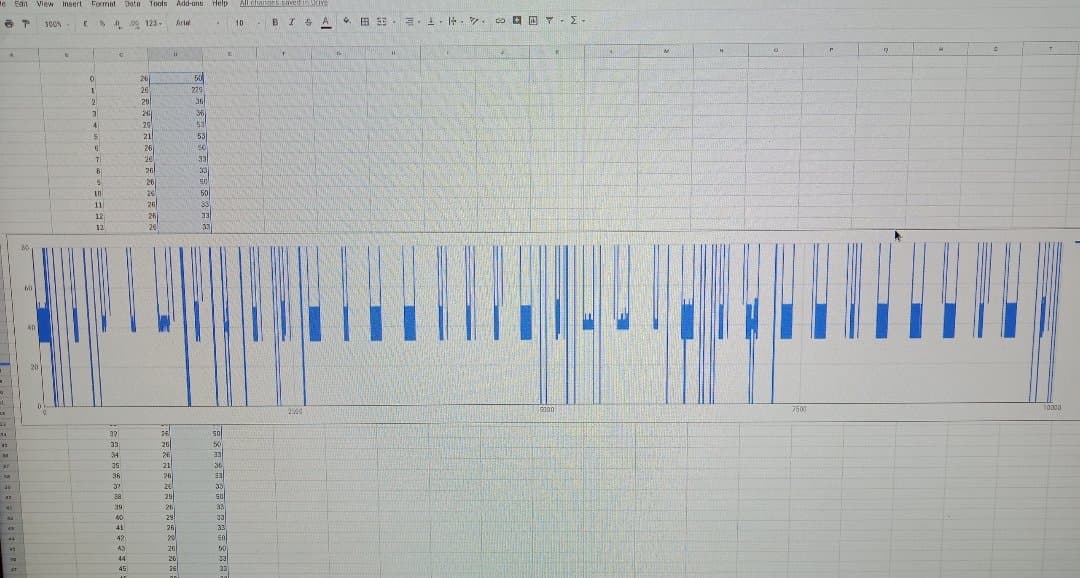
Task 1

To conduct a side-channel attack on ybn\_modexp function, I used the Mastik toolkit as per the assignment suggestion. The first step is by changing num.c to include ybn.c instead of ybn.h, as otherwise num.c will not compile as well as putting the call to ybn\_modexp in a while(1) loop to make it run continuously (this is not realistic in the sense that in a real attack, the computation that leaks the side channel data will likely not be in a while loop) while getting the familiar with the FR-trace. Then, in ybn.c I added two functions, x() and y(). Both of these functions simply iterate from 0 to 50,000 and increment a variable, doing nothing significant otherwise. Their purpose is to force the cpu to spend a quantity of time utilizing the cache for these functions, which will be captured in the trace.

I set my ybn\_modexp up in such a way that if the bit of the exponent string passed to it was set, it would run x() followed by y(), and if the bit was not set, it would run x() followed by 3x y(). The use of x() was needed each time as if only y() was used, the result was inconsistent and unclear.

The pattern this produced for modulus[5] was quite clear, with a thin segment of the y function showing if the bit was set (i.e. a 1) or a thicker segment of y if the bit was not set (i.e. a 0). In a trace of 10,000 samples, I was able to capture data on approximately 35 bits this way, with ~33 of those bits being clear enough to analyse at a glance.

From this photo there is a clear distinction between the thicker segments representing 0s, and the thinner segments representing 1s.

Because the modexp is in a while loop, the trace in this case will grab a random sample of 35 consecutive bits each time it is run, however if the trace is run multiple times, we can recover all 4096 bits of the exponent should we wish to. To find which section of the key I traced, I generated the binary representation of the exponent string and then after analyzing the graphed output, searched for the bits I was able to recover with definite clarity. For this sample pictured above, I searched for “010011101001010001000101” and was able to match this string to a section of the exponent, confirming that the bits were correct. This is again not entirely realistic as in a real attack, I would not be able to compare my obtained result to the ground truth (the hardcoded exponent), however by doing this analysis and then changing my trace to track the timings on the functions that are doing the actual operations of modexp, a similar pattern can be produced and then used in the same way to recover the bits of the exponent. The success rate of this side channel attack is slightly inconsistent due to noise being picked up in the trace, distorting the clarity on some of the bits, however with my current method I was consistently able to get an output containing 33 clearly visible bits out of the roughly 35 bits contained within one sample.

After doing this “highlighted” side channel attack (using wait functions), I removed them again from the program and by changing the slot size parameter to 30,000, I was able to find a different pattern that was able to similarly distinguish between bits that were 1 or 0. using the command :  
  
./FR-trace -f ./num -m ybn\_modexp -m ybn\_sqr -m ybn\_mul -c 10000 -s 30000 > output.txt

I was able to track when modexp was being executed, and by using that as a reference point could then look at how often mul was getting picked up each time. If mul was appeared on the graph twice in quick succession, the bit was a 1 (as mul was actually running), and if mul only appeared once, the bit was 0 (as mul was cached but not executed).

After extracting a selection of bits I was once again able to confirm that my analysis was correct, as after removing the while loop as well I was able to capture a section near the beginning of the exponent.

To recover the whole exponent in a single trace is possible, however I did not attempt this as google sheets has a limit on how many samples it can display (which is not big enough to encapsulate the whole exponent). IF the exponent can be traced multiple times, it would still be possible to recover the whole exponent by delaying the trace slightly each time to capture a different section, and then by finding areas of overlap between the recovered sections reconstruct the exponent.

Task 2

In the second task we are required to attack ybn\_modexp when using a progressively smaller modulus as argument. The effect this has is simple; the smaller the modulus, the shorter the function will take to complete and thus the more similar a 1 and a 0 bit will look in the trace result. This was immediately clear, as going from modulus[5] to modulus[4] increased the number of bits displayed by almost double (as the modulus is now only ~ half as long), but at the same time lowered the number of clear bits from 33 out of 35 on average down to roughly 70 out of 80.

Going down even further in modulus size by using modulus[3], the number of bits displayed increased further up to ~120, but the number of clear bits dropped to roughly 60, then for modulus [2], the number of bits captured in the trace increased again, but only approximately one third of the bits were distinguishable as definitely a 1 or definitely a 0. For modulus[1], it followed the same trend where the sample was able to capture a lot of bits being processed by modexp, however due to the shorter time taken per bit all distinguishing features of the bits has been lost.

Finally for modulus[0] none of the bits were identifiable, as at this point we were barely able to discern that a bit had been processed by modexp, and nothing more.

As far as viable success rates go, I would say personally that the smallest attackable value was modulus[3], as even at this value and any of the ones below the result was akin to simply guessing the bits and having a 50% chance of being correct. With modulus [3], I was able to recover half of the bits of the exponent with definite clarity, leaving me only having to guess the remaining half in which case we would be able to recover ~75% of the bits from a sample if it captured all 4096 bits in 1 go (as simply guessing 50% of the bits would still result in 25% of the total bits being correct).

Part 3

To modify ybn.c to perform modular exponentiation in a way that guards against side channel attacks, I first created a ybn to store a second result. Then to eliminate the need of an if statement checking if the exponent bit is a 1 or a 0, I used a variable *condition* to contain the result of that if statement, updating it each iteration of the inner for loop that processed each exponent bit.

Then, on each iteration I evaluate ybn\_sqr, ybn\_div, ybn\_mul and then ybn\_div again (keeping track of their results as I do the computations) before using my condition variable to select what value gets stored in the returned result. If the assumption that ybn\_sqr, ybn\_div and ybn\_mul are all evaluated in constant time is valid, than the whole exponentiation process will be completed using constant time methods (the process of copying result2 into the returned result1 is also constant time).

To test that this way of performing modular exponentiation is indeed not susceptible to timing based side channel attacks, I ran FR-trace on the modified version and found that in contrast to the unmodified version, function calls to ybn\_modexp were indeed indistinguishable from each other; a set bit looked identical in the trace as an unset bit, making it impossible to determine if a bit was set or not.

It was still possible to determine when a bit was being processed, meaning that the total length of the exponent is still able to be determined. This is not a security risk as such however, since knowing that the exponent is 4096 bits long will be of little help for actually recovering what the exponent was without any other data to work with.

Finally, this way of rewriting modexp to evaluate in constant time, while protecting against timing based attacks, is still vulnerable. If an attack was able to disassemble ybn.c, find the modexp function, realise that there existed a relationship between the condition variable and the value of the bit of the exponent being processed, they could run a trace to see the activity on the if statement, and narrow their search space down to two possibilities, either the function jumps when the bit is 1, or the function jumps if the bit is 0. This way if an attacker knows the pattern of jumps made by the if statement, as well as the length of the key, it would be possible to gather enough information to make even an incomplete exponent able to be recovered (so long as enough of the exponent was able to be recovered).