Assignment Name	Assignment 2: Long Numbers and Implementation Attacks
Student ID	1675746, 2107884

University of Birmingham Assessment and Feedback

Section One

Reflecting on the feedback that I have received on previous assessment; the following three issues have been identified as areas for improvement:

1	Further detail on some ideas
2	Focusing on some non-essential areas for too long
3	Conciseness of our writing

Section Two

In this assignment, I have attempted to act on previous feedback in the following ways:

1	Dividing up responsibility equally on all areas of the project
2	Time management for trickier areas
3	Cutting out as much non-essential answers as possible

Section Three

Feedback on the following aspects of this assignment (i.e. content/style/approach) would be particularly helpful to me:

1	Technical Accuracy
2	Approach effectiveness
3	Describing all wanted aspects

Hardware and Embedded Systems Security

Assignment 2: Long Numbers and Implementation Attacks

Graeme Brodie 2107884, Dee Dixon 1675746

1. Implementing Square-and-Multiply (SAM) (15%)

1.1 Average number of cycles:

e=17= 10001, t = 5

Cycles = 9220228 = 1.152536 s

1.2 Approximate cycles, e=1024:

In order to find the average number of cycles needed for the square and multiply operation, it was necessary to execute the code again with a different exponent. We selected e = 25 = 11001, as it included a single extra multiply operation. We then used simultaneous equations to find values for the average number of cycles for each operation.

e=25=11001, t is again 5, 4S and 2M

4S + M = 9220228, 4S= 9220228 - M

4S + 2M=11041488, 9220228 – M +2M = 11041488, M = 1821260 so

4S + 1821260 = 9220228, 4S = 7398968, S = 1849742

With a value of t, the number of squaring operations done is t-1. For an exponent with 50% one bits, the number of multiply operations done is (t-1)/2

If t=1024 then s=1023 and m = 512, 1535 operations in total

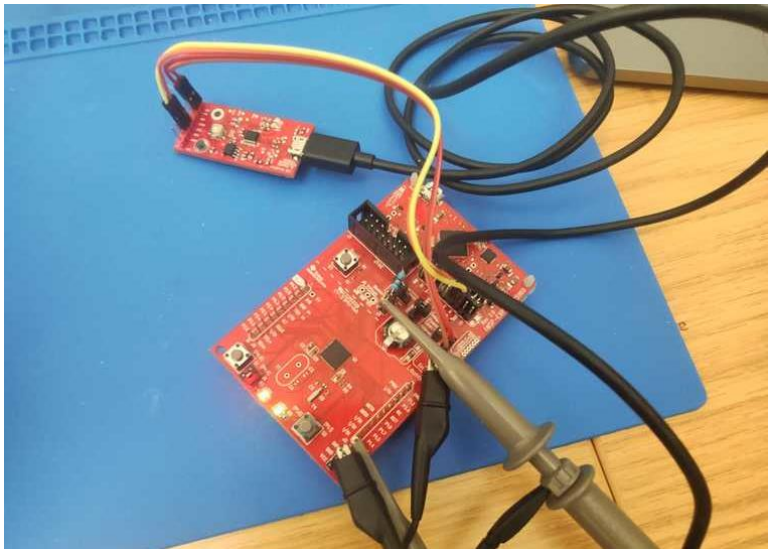
(1023 * 1849742) + (512 * 1821260) = **2,824,771,186 cycles approximately**

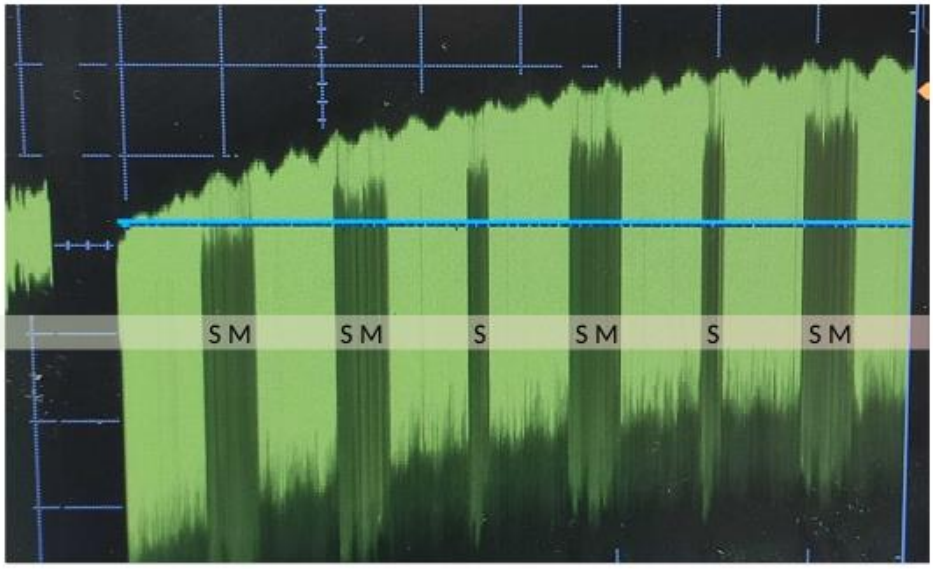
We can also take the original time taken to give an approximation of the time needed to complete these cycles

2824771186 / 9220228 = 306.37, 306.37 * 1.152536 = **353.1s**

2. Simple Power Analysis (SPA) on RSA (30%)

2.1 Oscilloscope trace of RSA operation



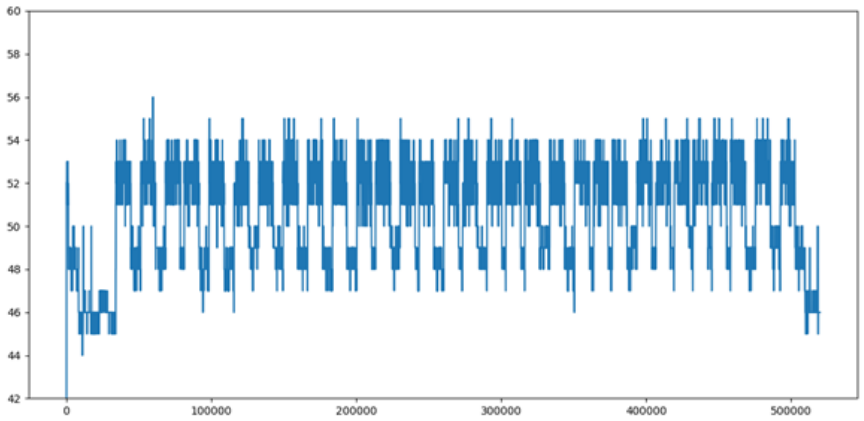


2.2 Extracting 6 bits of exponent

The first six operations are shown on the trace above. Stages corresponding to a zero bit in the exponent only include the square operation, and therefore take a shorter length of time to complete. Stages corresponding to a one bit also include a longer multiply operation, and therefore take much longer.

The sequence of operations for the first five bits is S, M, S, M, S, S, M, S. The first bit subject to computation is always a one bit, as preceding zero bits are ignored. This means the first six processed bits are 111010.

2.3 Plotting a filtered trace



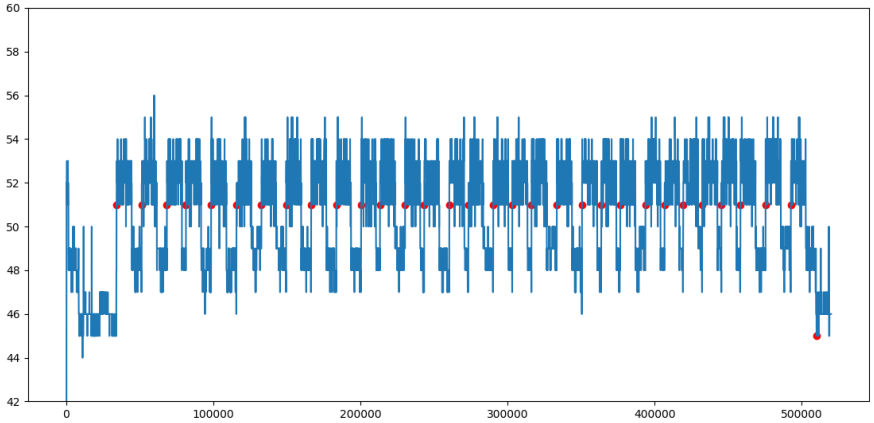
2.4 Automatic SPA

We note that power consumption increases during the long square operation and is lower for the multiply stage (including multiply operations performed as part of modular reduction). Using Python3 and numpy, the trace can be broken down into individual square-and-multiply algorithm iterations by observing the gradient of the trace as it crosses a certain value on the power axis.

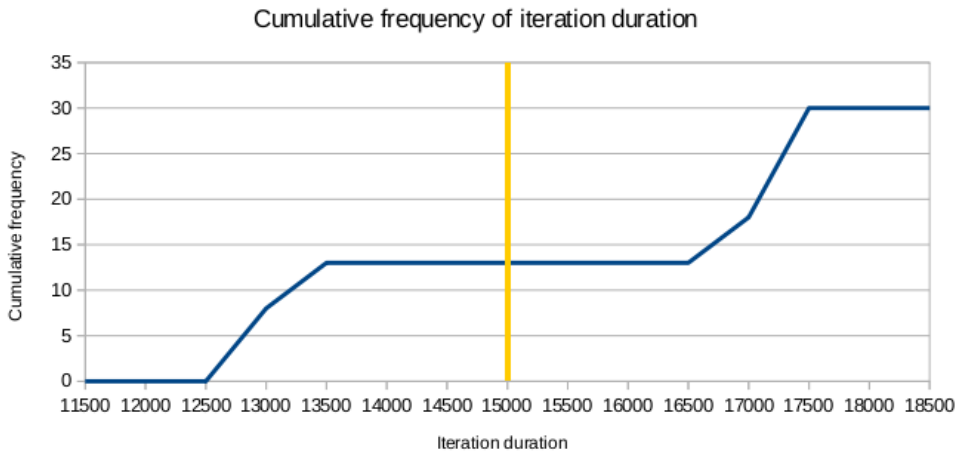
We select 50 as the baseline value on the power axis. This power value is only ever recorded between operations - never during either square or multiply operations. For the purpose of enumerating iterations, we define the demarcation between iterations as the point when the trace crosses this line with a positive gradient.

We make two additional allowances. One is for the start of the trace, where a positive gradient across the baseline marks the start of a new iteration but not the end of a previous one. The other is for the end of the trace, when the final iteration is followed by a decrease in power to 45 and below, not an increase through 50.

The iterations can then be excised from the full trace as follows, with each red circle (excluding the last) marking the start of a new iteration.



By measuring the time taken for each iteration we can discern whether the iteration contains both square and multiply operations, or just a square operation. As expected, there were two distinct populations of iterations in the time domain; we defined those iterations with a length of < 15000 as square iterations, and those >=15000 as square-and-multiply iterations:



By classifying each iteration as it is extracted, our program reconstructs the key:

11101111 11101010 10011001 00000111 = 0xefea9907

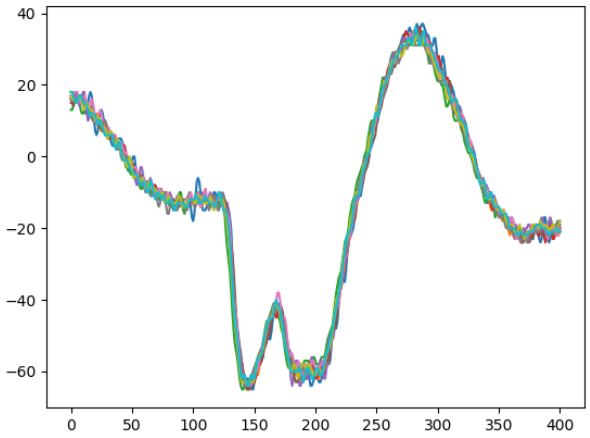
3. Differential Power Analysis (DPA) on AES (35%)

3.1 First 10 traces and first byte of first 10 inputs

We again use Python3 in conjunction with pyplot and numpy to read and display the data.

```
$ python3 traces_task3_part1.py
Specify number of traces to load: 10
Loaded sbox:
63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16

Loaded 10 traces.
First byte of input for loaded traces:
d8, e8, 42, 3d, 96, e5, cb, 0b, 1f, 33,
```



3.2 S-Box function

We store the S-Box in a linear hexdump format in the file sbox-concat.txt. This is loaded into a Python list when the script executes:

```
sbox = []

sbox_f = open("sbox-concat.txt", "r")
sbox_linear = sbox_f.readline()
sbox_f.close()

for i in range(0, 256):
  sbox.append(int(sbox_linear[i*2:i*2 +2], 16))
```

This list is then passed into the sbox function, along with the input and key candidate first bytes. The input byte and key candidate byte are XORed, and used as an index into our S-Box list:

```
def apply_sbox(sbox, input_byte, key_byte):
  return sbox[input_byte ^ key_byte]
```

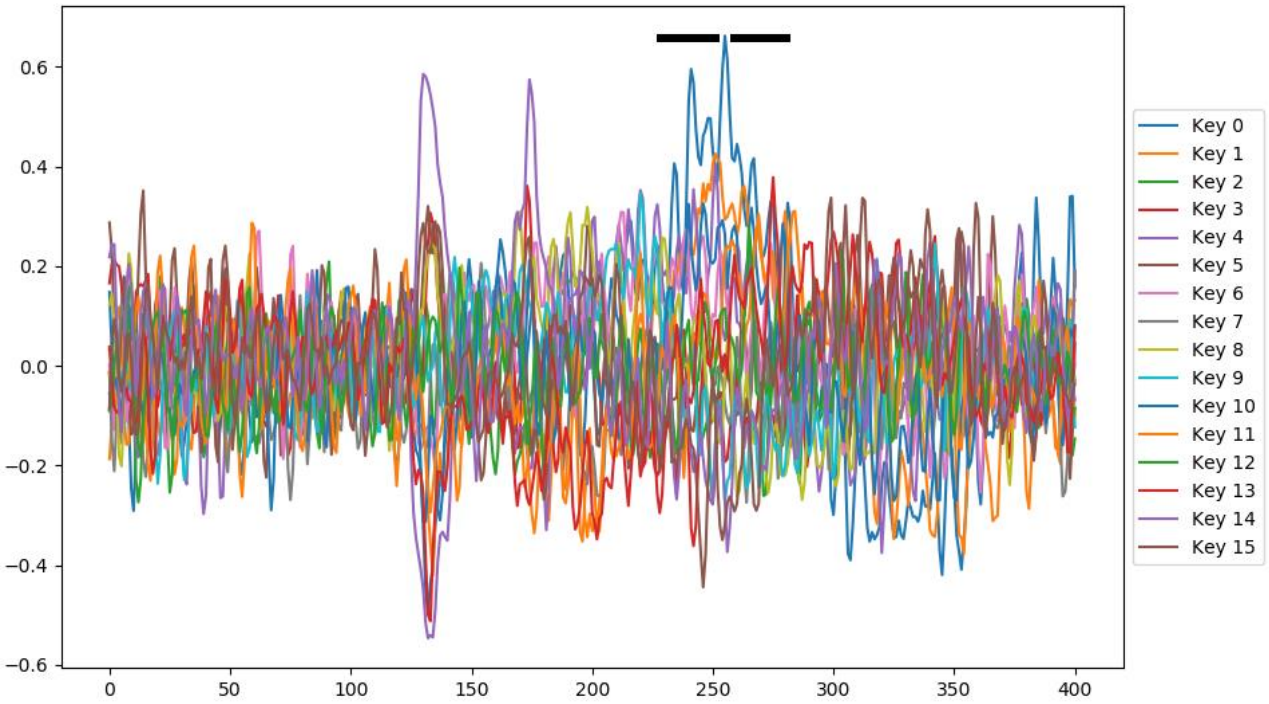
3.3 MSB extraction function

```
def msb(input_byte):
   return (input_byte & 0b10000000) != 0
```

3.4 DPA of 1300 traces using key candidates with first bytes 0-15

Our DPA suggests that the key of value 0x00 is correct.

```
Maximum absolute value was 0.6620911131952134 for key 0
```



4. Pen-and-Paper Problems (20%)

4.1 Fault Injection (FI) on CRT-RSA (10%)

n = 91, e = 5
Correct signature s = 48, plaintext x = 55. Faulty signature s0 = 35, plaintext x = 55

Bellcore:

q= gcd(s – s0, n) = gcd(13, 91)

Euclid: a=13, b =91 r =0

First Round: a =91, b =13, r =13

Second Round: a = 13 b=0, r=0

Output = 13, q = 13

If n =91 = p • q = p • 13. p = 91/13 = 7

Answers: p = 7, q = 13

Lenstra:

$y = s0^e \bmod n$, if e = 5 = 101, then t=3

$y = 35^5 \bmod 91$, using SaM

x = 35, y = 35, then $y = 35^2 \bmod 91 = 350 . 3 + 175 = 1050 + 175 = 140 + 175 = 315 \bmod 91$

y = 42 mod 91, because e1 is not 1, we skip multiply and proceed to e0

x = 35, y = 42, then $y = 42^2 \bmod 91 = 420 . 4 + 84 = 56 . 4 + 84 = 224 + 84 = 308 \bmod 91$

y= 35 mod 91, e0 is 1 so multiply is invoked

y = 35 • 35, which was done earlier and outputs y = 42 mod 91, y= 42

q = gcd(y – x, n) = gcd(42 - 55, 91) = gcd(-13, 91)

Euclid: a=-13, b =91

First Round: a =13, b =78, r =78

Second Round: a = 0 b=13, r=13

Third Round: a = 13 b=0, r=0

Output = 13, q = 13

If n =91 = p • q = p • 13, p = 91 / 13 = 7

Answers: p=7, q= 13

Advantage of Lenstra:

When using Bellcore, its ability to function relies on the being able to enter the correct and faulty signatures, this causes issues when you can't have the same value twice due to randomised padding, signature schemes or other common implementations. With Lenstra only the faulty signature and the plaintext x are needed to solve for p and q.

4.2 Countermeasures (10%)

4.2.1 Detection and algorithmic differences

Detection-based countermeasures generally work on the hardware level, including observing external factors of the device (temperature, light etc) or by running the software several times (either in parallel or sequentially) and checking that the result is the same, this can in most cases be a costly endeavour, either in terms of time or economically. Algorithmic countermeasures, on the other hand, generally occur within the software, running extra checks on signatures or adding randomness into the code execution times to obscure the information an attacker could receive.

4.2.2. RSA-CRT power glitch fault

a) When creating the signature that has had a power fault injected, you could run the code twice, either in parallel or sequentially, and check that the signatures are the same.

b) Algorithmically once the signature is calculated, it can be verified by decrypting the result using the public key given and comparing hashes.

4.2.3 SPA attack Prevention

The attack in Task 2 could be prevented by adding noise to the power output, there are power increases when initialising a SaM instead of just a Square and so adding extra processing no matter what operation is being completed so they are all approximately the same amplitude would stop attackers being able to pinpoint the operations.

4.2.4 Time Randomization

Many of the options used within side-channel attacks use the time taken for a piece of code to execute different functions as a key factor in being able to uncover the needed value. Likewise, fault injection attacks rely on an attacker being able to precisely identify when a specific operation is taking place by examining a side-channel factor. Randomizing these times makes it harder to distinguish between a specific function being called and any other activity the code is implementing, and makes precise timing information from one trace invalid in another.

In addition, side-channel attacks like DPA that use multiple traces rely on synchronisation between sample points in each trace. Randomly introduced delays disrupt this alignment; while this can be compensated for, it makes DPA much more difficult.

4.2.5 Amplitude noise

The addition of amplitude noise can be counteracted by using more traces in DPA. The number of traces needed for a given level of noise can be estimated statistically by calculating the correlation coefficient between the attacker's calculated theoretical data and the noisy traces. For an adversary capable of collecting many traces, the addition of amplitude noise is not sufficient to prevent a successful attack.