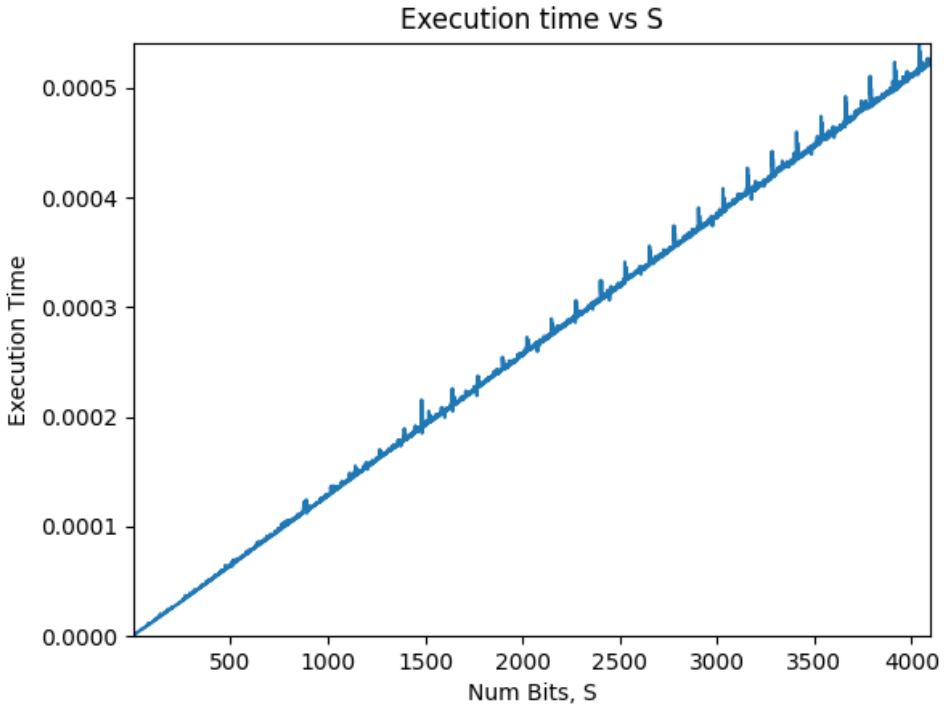
HW1 048891
Winter 2024-2025

Adam Ghadban

ID: 322661802

1)



This graph was taken by choosing the following parameters:

a = 2

e = powers of 2 (1, 2, 4, 8, etc.. until 2 ^ 4096), which means s = [1, 2, 3, …, 4097]

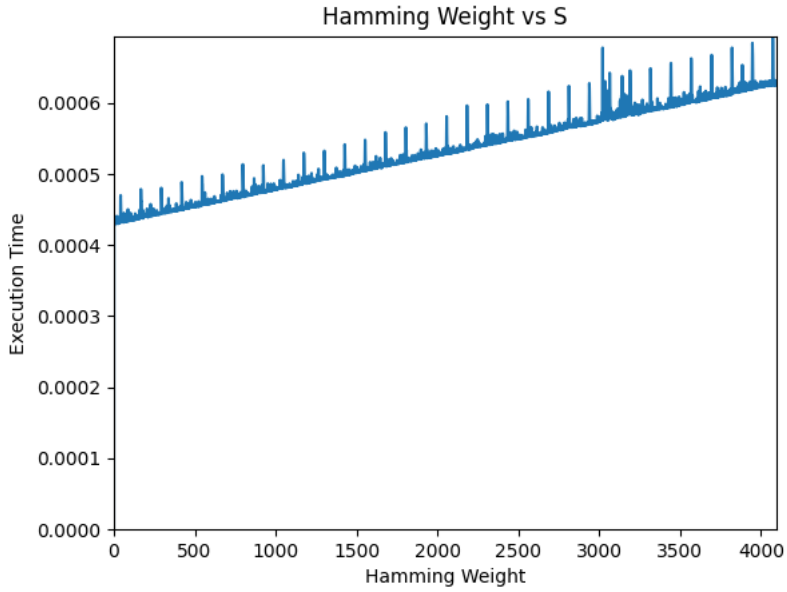
n = 20343797

The script was written using python.

Each S length was calculated 20 times then averaged over 20, to make sure we do not get spikes of computations and get as close to the average execution time as possible.

The graph state that the basic **left to write square and multiply** execution time is linear to the number of bits of e (s).

2)



Here we also chose the following parameters:

a = 2

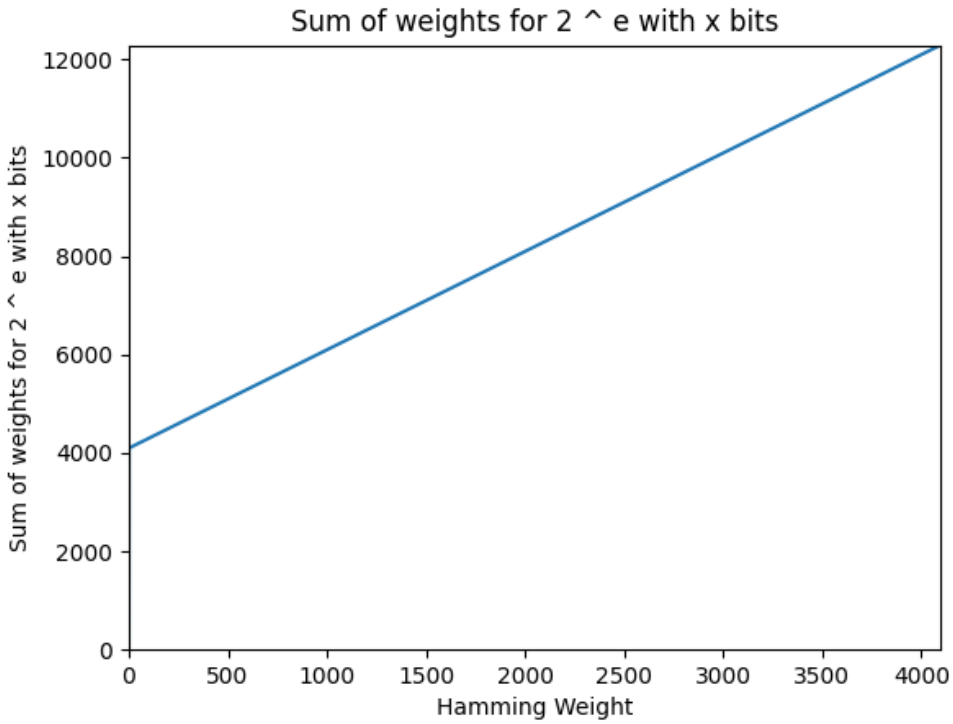
e = [000..0, 1000000.., 110000.., 111000.., .., 11111...110, 1111...111]

n = 20343797

We can see a linear behavior, with the exceptions of spikes due to memory and caching (most likely).

This means that with every increasing hamming weight (another bit equal to 1), we add more operations. This shows that we can use side-channel to attack this algorithm to figure out the exponent.

3)

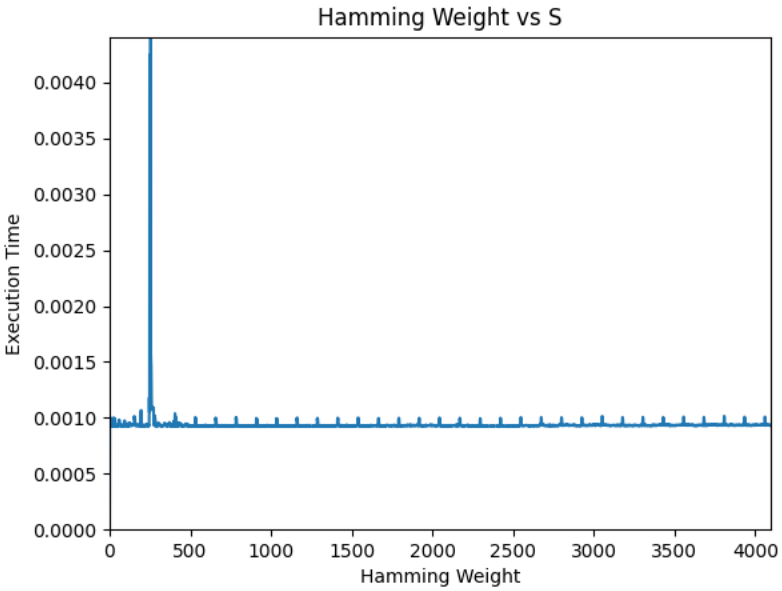


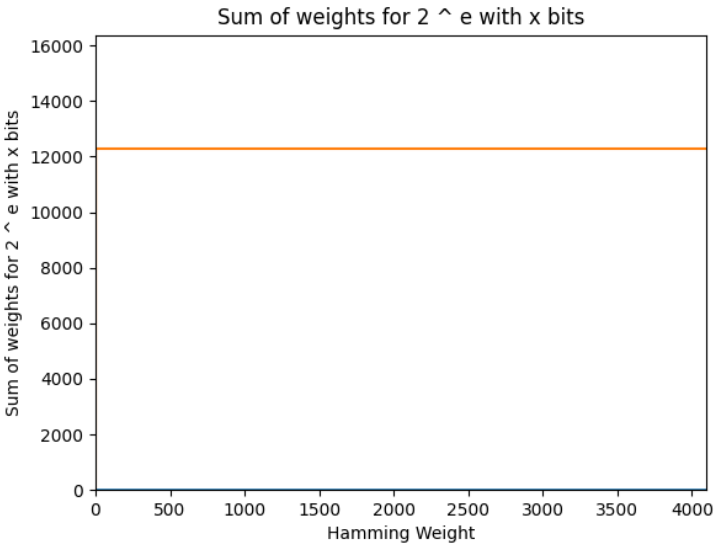
We set a weight of 1 for the exponentiation, and a weight of 2 for the multiplication, and the graph shows the sum of the weights for every operation of 2 ^ e, where e is an exponent with 4096 bits with a certain hamming weight.

For an exponent with hamming weight = 0, we expect a weight of 4096, and with each big going from 0 to 1, we add another 2, since we want to square, but also want to multiply, which results overall in a gradient = 2, and an end weight sum of 4096 * 3 = 12288.

This suggests that the number of operations differ per exponent, as we learned in class.

4)





After implementing an SPA protected algorithm, we can see that regardless of the hamming weight of the key (exponent), we get the same execution time (apart from spikes due to caching most likely).

Moreover, since we do a dummy operation, there is no difference of the sum of the weights, as we always do multiplication and exponentiation in every iteration.

Algorithm is protected against SPA.

5) We will assume full precision of fault injections, including time and space, meaning that we can choose which iteration and which code section to inject fault to.

By assuming this, we can inject faults into a (assume a single iteration fault), and check if the value changed then the bit was 1, otherwise, it was 0.

Repeat this until we can compute the key with exhaustive search.

Note that we kept the left to right implementation as follows:

```python
def dummy_l2r_square_and_multiply(a, e, n, w_s = 0, w_m = 0):
    s = len(e) # num of bits
    b = 1
    weights = []
    for i in range(0, s):
        b = (b * b) % n # square
        # k is already a left to right representation
        weights.append(w_s)
        if e[i] == 1: # conditional multiply
            b = (b * a) % n
        else:
            _ = (b * 1) % n
        weights.append(w_m)
    return weights
```

The function with fault injection capabilities i

```python
def inject_dummy_l2r_square_and_multiply(a, e, n, inject_fault_at_i = -1):
    s = len(e) # num of bits
    b = 1
    for i in range(0, s):
        b = (b * b) % n # square
        # k is already a left to right representation
        if e[i] == 1: # conditional multiply
            if inject_fault_at_i == i:
                b = (b * (a - 1)) % n
            else:
                b = (b * a) % n
        else:
            _ = (b * 1) % n
    return b
```

First, generate a random key with 4096 bits and save it as a secret key.

Second, the adversary will activate the function with a chosen a, and the function will return $a^e mod\ n$, save the result.

Third, for every bit i, inject a fault into a, and check if the result changed. If yes, then the bit (s – i) is 1, otherwise it is 0.

6) First, we must define the fault model:

   a. the adversary knows the algorithm (Kerckhoff's principle)

   b. the adversary can manipulate code, such as intermediate values (R0 and R1).

   The attack will be as follows:

   for every bit i, inject faults into R0 and R1 and set them to 1 if the bit was 1 (the faults will be in the part of the code that branches to bit == 1).

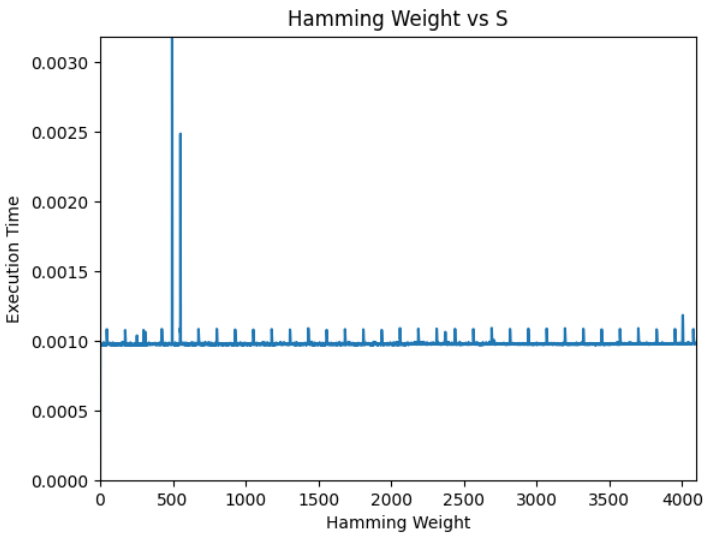
   Original algorithm:

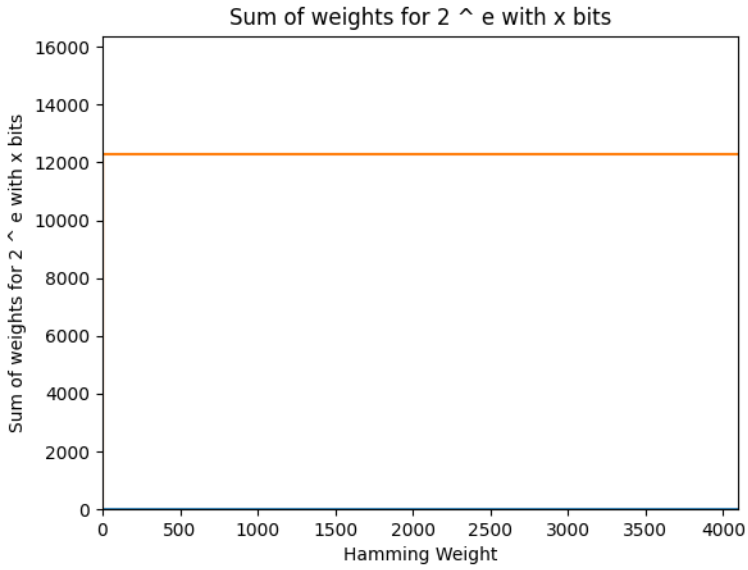
```python
def montgomery_power_ladder(a, e, n, w_s = 0, w_m = 0):
    R0 = 1  # Represents x^(2^i) mod n
    R1 = a % n  # Represents x^(2^i + 1) mod n

    weights = []
    # Process the exponent bits from MSB to LSB
    for bit in e:
        if bit == 1:
            R0 = (R0 * R1) % n
            weights.append(w_m)
            R1 = (R1 * R1) % n
            weights.append(w_s)
        else:
            R1 = (R0 * R1) % n
            weights.append(w_m)
            R0 = (R0 * R0) % n
            weights.append(w_s)

    return R0
```

Fault algorithm:

```python
def inject_montgomery_power_ladder(a, e, n, inject_fault_at_i = -1):
    R0 = 1  # Represents x^(2^i) mod n
    R1 = a % n  # Represents x^(2^i + 1) mod n

    # Process the exponent bits from MSB to LSB
    for i in range(len(e)):
        if e[i] == 1:
            if inject_fault_at_i == i:
                R0 = 1
                R1 = 1
            else:
                R0 = (R0 * R1) % n
                R1 = (R1 * R1) % n
        else:
            R1 = (R0 * R1) % n
            R0 = (R0 * R0) % n
    return R0
```

Now in order for the attack to succeed, we need to choose an a (a base) that f(a) != 1 to prevent false positives.





```python
def part5(part):
    e = 1
    while e == 1:
        e = generate_random_binary_list()
    orig_value = inject_dummy_l2r_square_and_multiply(2, e, n)
    key_bits = []
    for i in range(0, num_bits + 1):
        if part == 4:
            observed_val = inject_dummy_l2r_square_and_multiply(2, e, n, inject_fault_at_i=i)
            if observed_val == orig_value:
                key_bits.append(0)
            else:
                key_bits.append(1)
        if part == 6:
            observed_val = inject_montgomery_power_ladder(2, e, n, inject_fault_at_i=i)
            if observed_val == orig_value:
                key_bits.append(0)
            else:
                key_bits.append(1)

    observed_key = int(''.join(map(str, key_bits)), 2)
    original_key = int(''.join(map(str, e)), 2)
    assert observed_key == original_key, "OBSERVED KEY IS NOT EQUAL TO ORIGINAL KEY"
```

This is the code to find the correct key.