Name: Peter Atef Fathi	
Section: 1	B.N: 19

Brute-Force Attack

The idea of the attack:

1- try the values from [n/e: n] to find the private key this is the first loop and that because of our observations that the private key has more probility to be greater than the public key.

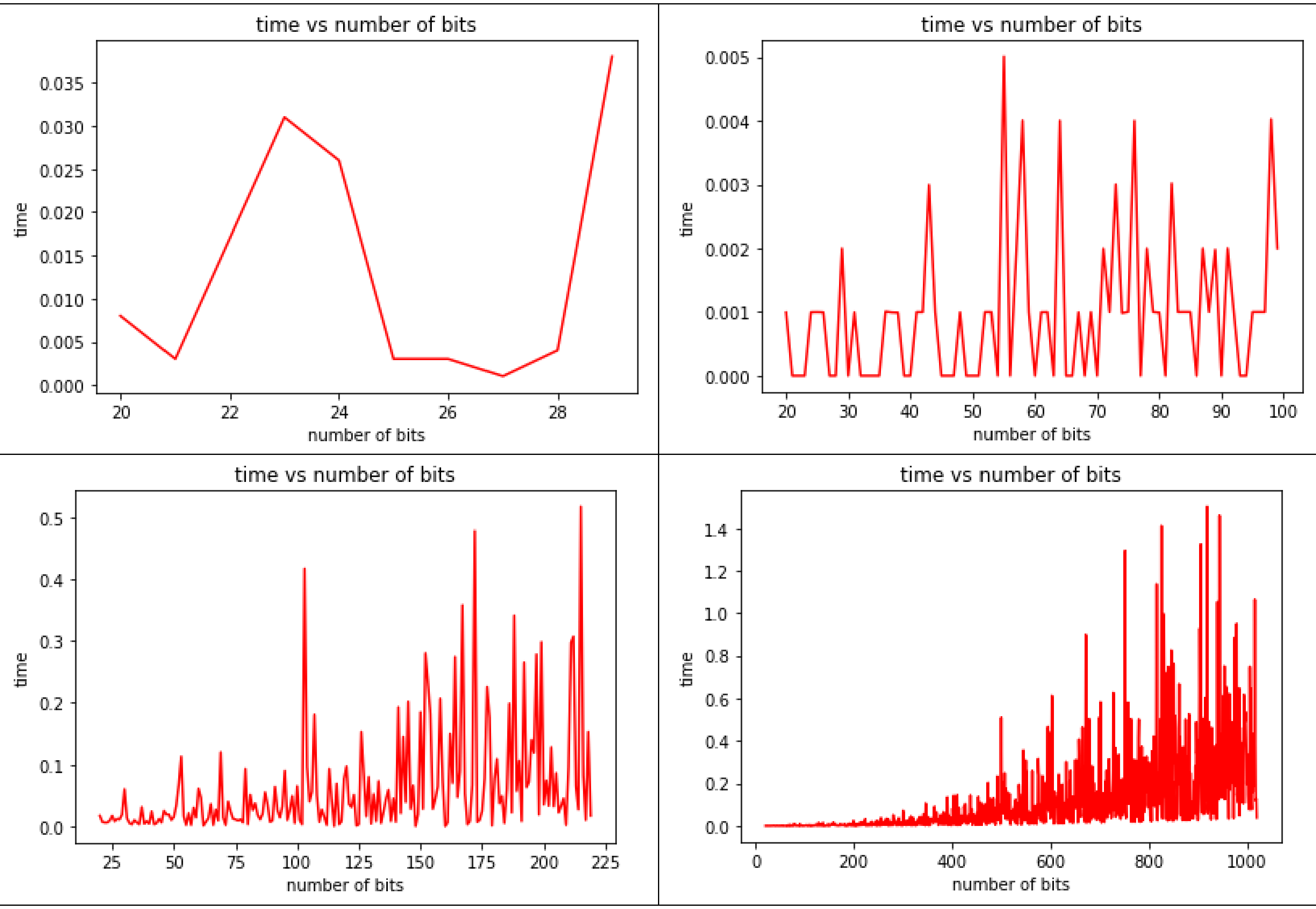
2- if the private key is not found in the first loop then try the values from [1: n/e] to find the private key this is the second loop.

3- if the private key is not found in the second loop then the private key is not found and the attack fails

Code of the attack:

```
# c is the ciphertext
# p is the plaintext
# start is the start of the range of the private key(= n/e)
# end is the end of the range of the private key(= n)
# this function performs the brute-force attack using the public key
def attack(c: list[int], p: str, n: int,start:int,end:int)-> int :
    # the range of the key is from 1 to n
    for d in range(start, end):
        # if the plaintext is equal to the ciphertext
        x = decryption((d, n), c)
        if x.__contains__(p):#contains 34an al padding
            # return the private key
            return d
    # if the private key is not found after the previous loop
    # then the private key is less than the public key
    for d in range(1,start):
        # if the plaintext is equal to the ciphertext
        x = decryption((d, n), c)
        if x.__contains__(p):#contains 34an al padding
            # return the private key
            return d
    # if the private key is not found
    return -1
```

Observations:



Conclusion:

- The previous two graphs show that if the number of bits used to generate the keys increases, the time needed to attack increases.
- The previous graphs for Plaintext = “hi” to minimize the calculations.

Fermat factoring algorithm Attack

The idea of the attack:

The algorithm is based upon the being able to factor the difference of 2 squares. X^2 – y^2 = (x + y)(x − y)

If n = x^2 – y^2 , then n factors: n = (x + y)(x − y). But, every positive odd integer can be written as the difference of two squares. In particular for the integers that we use of RSA modulo n = pq,

$$n = pq = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

Let k be the smallest positive integer so that k^2 > n , and consider k^2 − n . If this is a square, we can factor n: if k^2 − n = h^2 , then n = (k + h)(k − h). If it is not a square, increase the term on the left by one and consider (k+1)^2 – n. If this is a square, n factors. If (k +1)^2 − n is not a square, consider (k+2)^2 − n . Etc. Eventually, we will find an h so that (k + h)^2 − n factors.

That is so because $\left(\frac{n+1}{2}\right)^2 - n = (\frac{n-1}{2})^2$

In this case, n factors as n = n ×1. k ≤ k + h ≤ (n+1)/2

Here is an example. n = 6699557. (n^1/2) ≈ 2588.35 ;so, k = 2589.

K^2 – n^2 = 25892 − 6699557 = 582 . So,

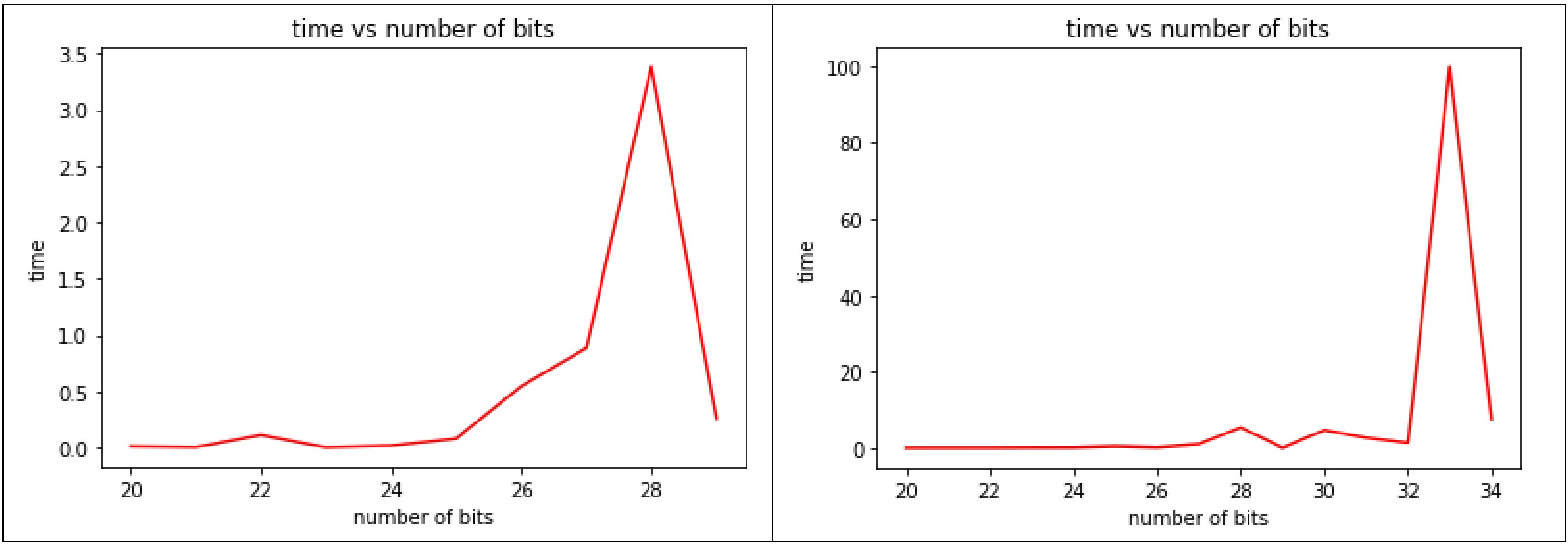
6688557 = 25892 − 582 = (2589 + 58)(2589 − 58) =2647 × 2531 (p*q)

Fermat's factorization algorithm works well if the factors are roughly the same size.

Code of the attack:

```
# this function performs the fermat factoring algorithm to find the factors of n
def fermatFactoringAlgo(n: int):
    # find the square root of n
    k = math.ceil(math.sqrt(n))
    # find the square of k
    h_square = k * k - n
    # find the square root of h_square
    h = int(math.sqrt(h_square))
    # while the square of h is not equal to h_square
    while h * h != h_square:
        # increase a by 1
        k = k + 1
        # find the square of k
        h_square = k * k - n
        # find the square root of h_square
        h = int(math.sqrt(h_square))
    # return the factors
    return k - h, k + h
```

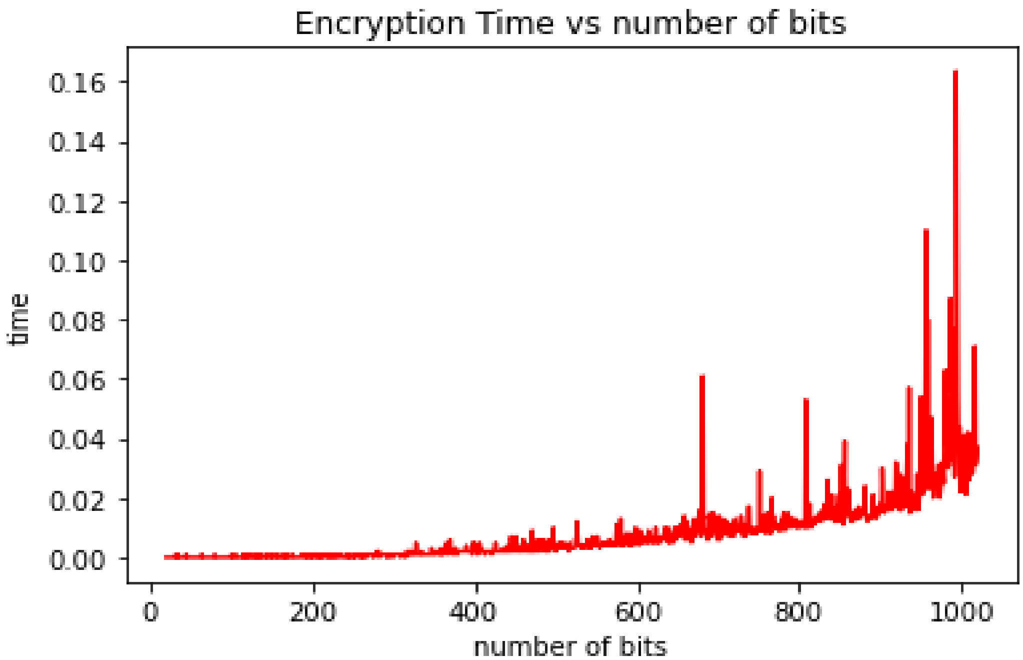
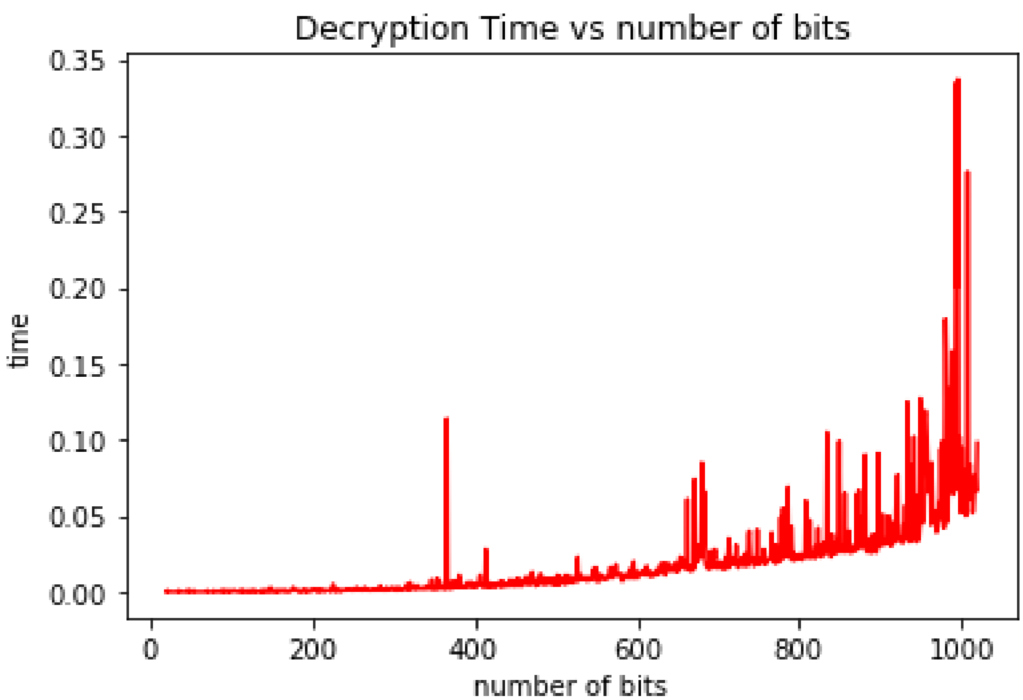
Observations:



Conclusion:

- The previous two graphs show that if the number of bits used to generate the keys increases, the time needed to attack increases.
- *<u>Note that</u>*: this algorithm doesn't depend on the plaintext or ciphertext.
- It's noticed that the brute-force attack with the previous technique is faster than Fermat Attack.

Analysis about different key sizes (number of bits of n) and how it affects the speed of encryption/decryption:

	It shows that if the number of bits used to generate the keys increases, the time needed to encrypt the plaintext increases.
	It shows that if the number of bits used to generate the keys increases, the time needed to decrypt the ciphertext increases.