Cryptography Project

Detailed Report

Name	Section	Bench No.
Essam Wisam	2	1
Mohamed Saad	2	14

Supervised by

Dr. Samir Shaheen
Eng. Sandra Wahid

May 2022

Table of Contents

Implementation Details

I. RSA Algorithm

Our implementation of the RSA algorithm is divided into two main parts. First, key generation where the user is assigned a private and public key. After providing their name, the user is free to either let their public and private keys be automatically generated; in which case we use Fermat's test to generate two equally-sized large primes (p, q) and then choose suitable e such that $gcd\big(e, \varphi(n)\big) = 1$ where $\varphi(n) = (p-1)(q-1)$ we use the standard Euclidean algorithm to calculate the greatest common divisor and use the extended version to calculate the modular inverse which is used to calculate the private key as $d = e^{-1} mod\ n$.

As for Fermat's test which we use to generate primes, the number is surely composite if Fermat's little theorem

$$a^n \equiv a\ (mod\ n)$$

does not hold, because it's only guaranteed to work for prime numbers. Note that if it holds then we can't say that the number must be prime (since some composite number can satisfy it as well) we get around the issue by applying the test many times while using different values of a. We have learnt about this prime generation method in our number theory course. Check the appendix for more and a handwritten proof of Fermat's little theorem.

The second main part of the RSA algorithm is encryption and decryption. Since it holds that

$$M^{\varphi(n)} \equiv 1\ (mod\ n)$$

By Euclid's Theorem (Provided $gcd(M,\ n) = 1$) then it also holds that

$$M^{1+k\varphi(n)} \equiv M\ (mod\ n)$$

And because we generated keys such that

$$ed \equiv M\ (mod\ \varphi(n))$$

Then it's true that

$$ed = 1 + k\varphi(n)$$

And hence

$$M^{ed} \equiv M \ (mod\ n)$$

Which provided that $M < n$ yields

$$M^{ed} = M \ mod\ n$$

Thus, we implement encryption as

$$C = M^e \ mod\ n$$

And decryption as

$$M = C^d \ mod\ n$$

Clearly both of them boil down to simply implementing modular exponentiation. For that we used the binary exponentiation method presented in our book:

$f = 1$
$for\ each\ bit\ b_k\ in\ n\ from\ left\ to\ right$:
    $f = f \times f \ mod\ n$
    $if\ b_k == 1$:
        $f = f \times a \ mod\ n$
$return\ f$

The only part that we didn't explain so far is how our implementation of RSA is not just restricted to numbers. Our approach simply envisages in converting every character in the input string to its corresponding ASCII value. In more detail, so to convert a message to a number what we do is to get the corresponding ascii code for every character in the message (8 bit for every character) then we concatenate those binary numbers into one bit binary number and that is the number corresponding to our message.

An additional note about encryption is that for the case where $M \geq n$ we will truncate the message to block of size smaller than n and we could determine the maximum size as follows:

1. Let a message m consist of k characters and each character is an 8-bit binary number, thus m as a number is less than or equals to 2^{8k}.

2. Using the above equation, the maximum message size n can hold could be deduced as follows $n \geq 2^{8x} = 256^{x}$ thus $x \leq log_{256}(n)$ thus n could decrypt $\lfloor log_{256}(n) \rfloor$ character in one go (the floor function is used as we only deal with integers).

3. We split M to m messages each one contains at most $k = \lfloor log_{256}(n) \rfloor$ characters and decrypt each one separately and send it.

The following table shows a list for some of the possible key generation input scenarios and our system's response to them:

Input	Response
Missing Name	Pop up message asking for the name.
Even e	Not Accepted regardless to p and q.
e that isn't coprime with $\varphi(n)$	Not accepted.
Any of p or q or e missing	Generate the missing values such that sign up is successful.
p and q that are not primes	Not accepted.
p and q of equal values	Not accepted since a single square root can break the cipher in this case.
Any of p or q or e not a number	The input values should be numbers.
p or q equals to 2	The value of p and q must be greater than 2 (if p or q equals to 2 the keys are easily obtained).
e equals to 1	The value of e should be greater than 1 (plain text would be send).

II. RSA World: End-to-end Encrypted Chat

We have used a text files abroach where is a main folder in which all user folders are contained and inside each user folder a text file is created that behaves like a receiving buffer that will be polled to know if a message was sent and decipher it, and the first line of that file contains the public key of the user so as to if another user wants to communicate with him/her knows the public key to use in encryption.

Initialization:

The script gets the username and keys as a command line arguments and check if there is no user with that name exists if so, create its folder and text file.

User selection:

A window will appear to the user that lets the user see the available user and chose one on to chat with.

Sending and Receiving:

There are two separate threads for each one the receiving one poll the text file in the user folder if any message is found it decrypts it with the user private key and show the result in the window, the sending one check if the user enters a text on the window if so, it encrypts it with the public key of the other user that was obtained from the beginning of its file.

III. RSA Encryption Efficiency Test

To test the encryption efficiency, we measure the time taken to encrypt a fixed message for different key sizes.

At first, we calculated two lists of keys one with incremental length with e having the same size as the key and the second having e constant equals to 65537, then those keys are used to encrypt a message for multiple times then get the average (we used multi-iteration method then the average for accurate time calculation).

And this is the chunk of code that we used to calculate the time:

```
lenght_of_string=2
bit_start=0
loop_size=1000

for i in range(bit_start, len(n_arr)):
    start = time.time()
    for _ in range(loop_size): encrypt(n_arr[i], e_arr[i], lenght_of_string*"a")
    end = time.time()
    time_arr.append((end-start))
```

IV. RSA Mathematical Brute-force Attack Test

In the brute-force attack we try to factorize a set of public keys ranging from 8 to 1024 bit (8, 10, 12, …, 1024) and measure the time in each case. We used a very straightforward algorithm for integer factorization that simply searched for divisors of the given integer by considering those odd ones lying only in the range up till the integer's square root (check the appendix for why looking in this range is enough.) Once a divisor is found it's guaranteed to be prime (because we are iterating from small numbers to larger ones) and dividing the public key by that should yield the other prime.

Note that however, as seen in the recursive algorithm below we keep looking for primes if after dividing the remaining number is composite and then report that the key isn't due to RSA in a wrapper function if more than two primes are found.

```
def factorize(n):
    if (n % 2) == 0:
        return [2] + factorize(n//2)

    integer = 3
    while integer <= (n**0.5):
        if n % integer == 0:
            return [integer] + factorize(n // integer)
        else:
            integer += 2                    # Since all primes are odd.
    return [n]
```

V. RSA Chosen Ciphertext Attack

In the chosen-cipher text attack we take as an input p, q and e of the victim along with a boolean "permission" that gives authority of their use to decrypt some chosen ciphertext. We also take the ciphertext that the attack is targeting (the one for which the attacks wants to know the corresponding ciphertext) and an integer r (the value multiplied by the ciphertext in the attack.)

The main modules are the Choose_Y function which returns the chosen ciphertext by the attacker based on the target ciphertext, the attack parameter r and victim's public key.

After Choosing the ciphertext we see if there is permission to decrypt it and if there is we use the result to get the corresponding message M to the target ciphertext.

```
def CCA(p, q, e, C, r, permission=True):
  _, _, _, _, n, _, d, err_msg = sign_up(str(p), str(q), str(e))
  if err_msg == '':
    if GCD(r, n) != 1:
      return "Could not establish the attack due to choice of r", -1
    Y = Choose_Y(C, r, n, e)
    if permission:
      X =  ModExp(Y, d, n)
      M = Find_M(X, r, n)
      return "", M
    else:
      return "Target denied signing.", -2
  else:
    return err_msg, -3
```

Why does the attack happen?

The attack is possible due to the fact

$$(M_1^e mod\ n)\ \times (M_2^e mod\ n)\ \ mod\ n = (M_1 \times M_2)^e\ mod\ n$$

Which follows as

$$(M_1 mod\ n)\ \times (M_2\ mod\ n)\ \ mod\ n = (M_1 \times M_2)\ mod\ n$$

And can be rephrased into

$$(E(k, M_1) \times E(k, M_2)) \bmod n = E(k, (M_1 \times M_2)^e)$$

In the attack we let M_1 be the message M we seek to read. From the communication channel we get

$$C = E(k, M_1)$$

We then let M_2 be some integer r such that $gcd(r, n) = 1$ and because we know our victim's public key we compute

$$Y = (C \times E(k, r)) \bmod n$$

And then we either assume permission from the receiver to sign that (decrypt) or access to decryption device which because

$$Y = \left(C \times E(k, r)\right) \bmod n = E(k, (M \times r)^e) = (M \times r)^e \bmod n$$

Yields

$$X = (M \times r)^{ed} \bmod n = M \times r \bmod n$$

And since we guaranteed earlier that $gcd(r, n) = 1$, the attack is concluded since

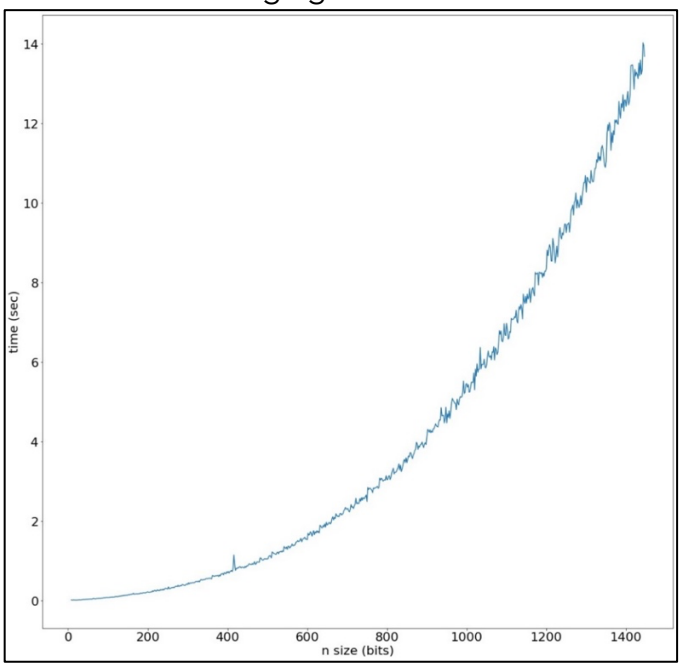
$$M = (r^{-1} \times X) \bmod n$$

This is, in-depth is all the math that makes the chosen-ciphertext attack possible. Note that after all, it's a chosen-ciphertext attack and some sort of access to the decryption device is needed. The common scheme for this is to ask the receiver to sign the chosen ciphertext (which is equivalent to decrypting it.)

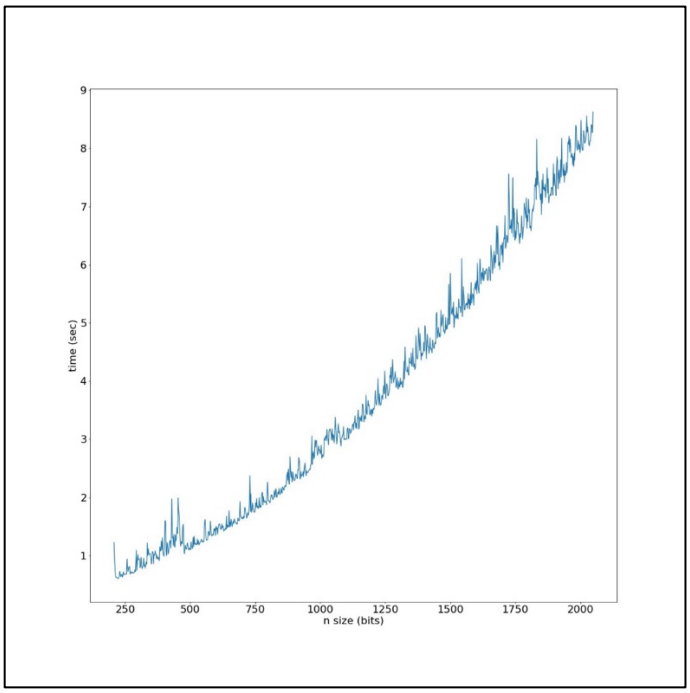
Analysis Results & Conclusions

I. RSA Encryption Efficiency

The first with changing e:



Then we used e with fixed value of 65537 and those are the results:



Results:

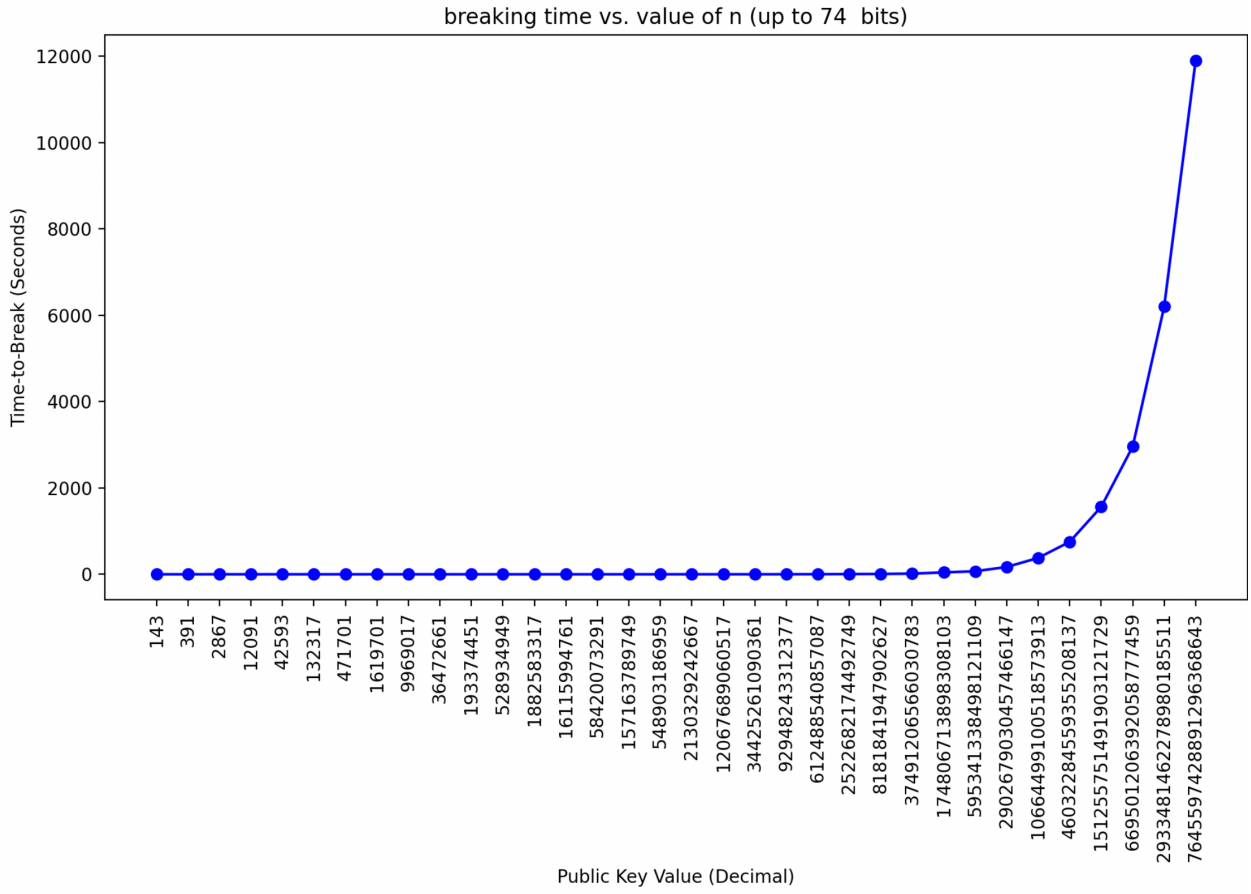
As we can see the encryption time goes in an exponential trend with the increase in size of n but as with smaller e the time is less the exponential is more nuanced in the first one.

As of time most of it were in calculating the key array which took around 47-min the keys tried are from 8-bit to 2048-bit.

Then the encryption time which took just a few minutes.

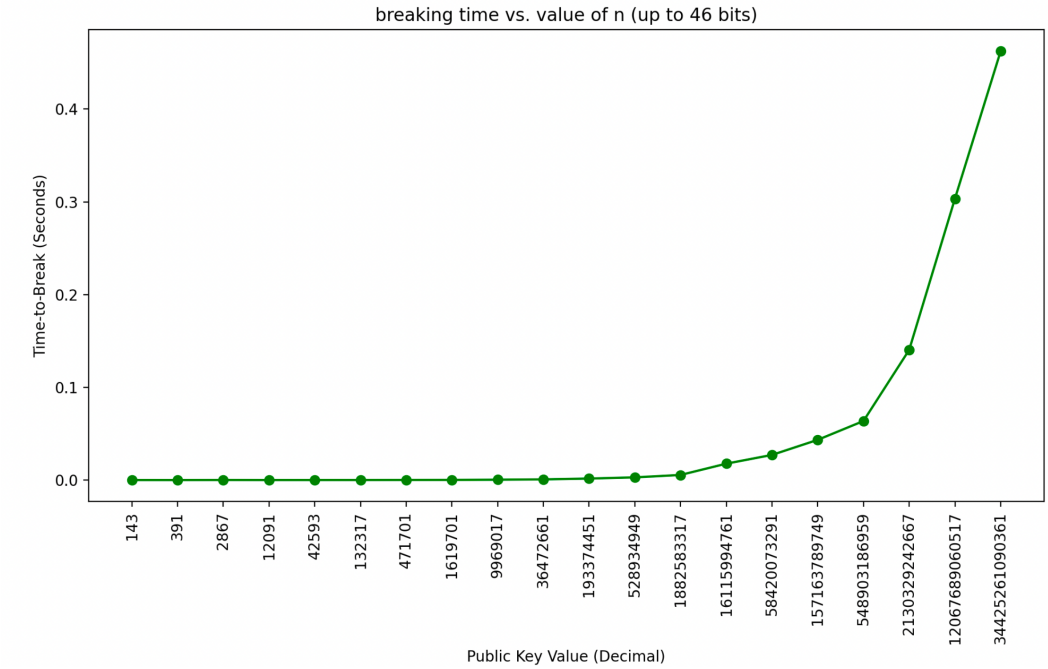
The computer used for this test was an instance in the cloud from **PaperSpace** that had an **8-Core CPU** and **30GB of RAM**.

II. RSA Mathematical Brute-force Attack Analysis

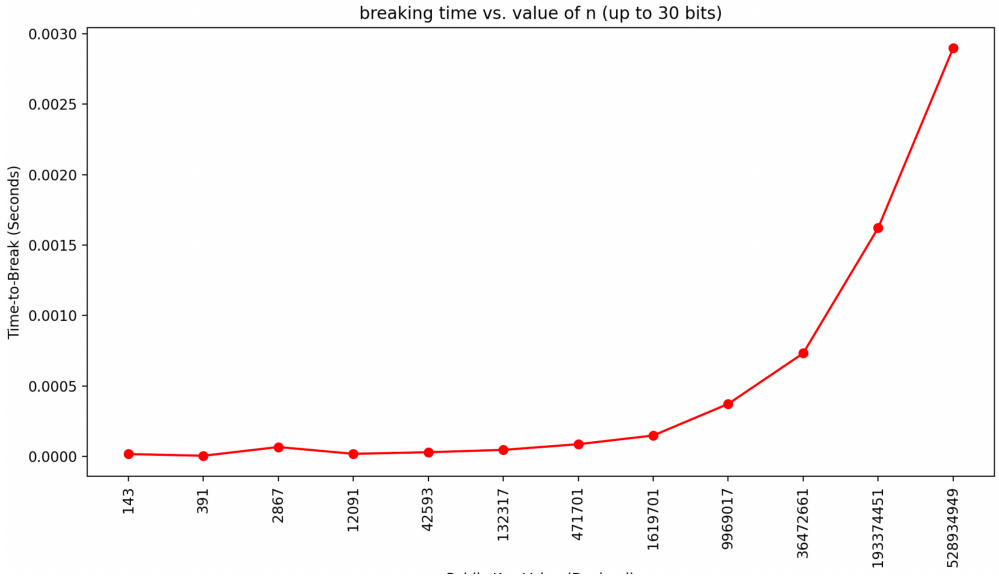


In the mathematical brute-force attack (explained earlier.) We started by setting up an array of public keys of length 8, 12, …, 1024 we then ran the factoring algorithm on the whole array and reported the time results in a text file. We kept trying to break the keys for more than 450 minutes (7.5 hours) in this period we were able to break keys up to 74 bits (and it took as much as 3.33 hours to break the 74-bit key).

Notice that although it seems that the time for breaking smaller is the same (the flat region on the left.), it's not actually flat but it's just that the scale on y is large due to the time taken to break the very large keys. In particular, this is what we get if stop when the key-length hits 46 bits.



And this is what we get



If we stop when the key length hits 30 bits. The trend is overall clearly exponential which is indeed the known time complexity of prime factorization. Notice that we were easily

able to break RSA for 74 bits which means that unlike symmetric encryption we can't use such small key-lengths which justifies why we resorted to using 256-bit RSA in our algorithm's implementation (which as far as we believe has not been broken by a normal computer so far.)

The computer used for this test was the Macbook M1 Pro which has specifications

Processor & GPU	Apple M1 Chip (2.1 – 3.2 GHz)
Processor Cores	8 Cores (4 performance cores and 4 efficiency cores)
Memory	LPDDR4X 8 GB RAM

Further Notes

No further notes, thank you for reading.