Assignment 2 - public key cryptography

Aleksander Pettersen (alekspett@gmail.com)

Abstract

A rudimentary version of the RSA-cipher was successfully implemented in C# as depicted by Stallings[1] with the exception of key size. The cipher is implemented in a 64-bit environment – making it a very vulnerable to cryptanalysis.

Introduction

The RSA, named after its creators, was one of the first public-key crypto-algorithms able to meet the security requirements set by a paper published by Diffie and Hellman[1] in the late 1970s. The RSA is a block cipher having its plaintext and ciphertext converted into a number representation ranging from zero to an upper limit set by the key parameters. The cipher is still in use today and is regarded safe as long as the key size is bigger than 768 bits[1]. This assignment sets out to implement the RSA as depicted by Stallings[1].

Design and Implementation

The RSA

The RSA is a public-key schema allowing units to communicate securely without sharing a single common key; instead secrecy is achieved by having a pair of keys[1]. The pair consists of a private key and a public key. The strength of the cipher lies in the concealment of the private key[1] and the use of prime numbers[1]. As figure 1 depicts, any message from Bob to Alice is encrypted using Alice's public key, this key is available to anyone communicating with Alice. Bob knows that Alice, and Alice only, is able decrypt the message because she is the only one possessing the private key needed.

The same message can be encrypted using Bob's private key since Alice is possessing Bob's public key; making her able to decrypt the message , but the message will be available to anyone with the knowledge of Bob's public key – reducing the confidentiality. This scenario is depicted in figure 2.

The Core Elements

Both the private key and the public key are prime numbers, allowing them to inherit the irreducible properties of primes. This property makes it easy to encrypt any given plaintext, but very hard, infeasible, to decrypt the plaintext without knowing the private key[1]. The complexity is achieved by introducing a second public factor; the product of two primes[1]. These two primes are the foundation for the RSA-algorithm and are the base of which every core element of the RSA rely on[1]. The RSA consists of four core elements: the product of the two primes, the public key, the private key and the Euler's Totient function of the two primes[1]. The last element ensures that there is not statistical coupling[1] between ciphertext, or plaintext, and the public key, or private key.

The Euler Totient Function

The Euler Totient Function, often referred to as φ, phi, is the product of two primes where each prime is decremented before the product is calculated, see figure 3. The product describes how many integers that are relative prime to the input[2]. The RSA wants this number to be as high as possible because it is indicative of how effective the cipher is at concealing its content. Given two small prime numbers, i.e. 3 and 5, the φ would be 8 and thus not providing the desired amount of confusion[1]. However if the primes are chosen to be 19069 and 21713, the φ would be 414004416, thus giving the cipher 414004416 values of which the cipher is relative prime to – making it very hard to guess the private key.

The Public Key Factor

The public key is determined solely based on the value of the Euler Totient Function, φ, thus taking advantage of the masking properties of relative primes[2]. The only formal requirements to the public key are that it is relative prime to φ and less than φ. The easiest way to enforce these requirements is to calculate the greatest common denominator of φ and a chosen number ranging from the value one through φ. The greatest common denominator is easily calculated:

```
public static long CalculateCommonDenominator(long number1, long number2)
    {
      while (number2 != 0)
      {
        var temp = number1%number2;
        number1 = number2;
        number2 = temp;
      }
      return number1;
    }
```

The greatest common denominator can also be calculated recursively, but it is faster to calculate it using the algorithm in the figure above.

The Private Key Factor

The Private Key Factor is relative prime to the Public Key Factor, and the private key factor must be relative prime to φ. This can be expressed in the following term; the remainder of the product of the private key factor and the public key factor, when divided by the φ, must be equal to the value one[1]. It can be implement by either of the following equations[2];

Private Key Factor = Public Key Factor $^{-1}$ MOD φ

Or by

Public Key Factor * Private Key Factor = 1 MOD φ

The implementation in this assignment uses the latter one for simplicity.

The Encryption

The RSA encrypts plaintext by assigning each letter a given number value. In order to avoid repetition of the key in the ciphertext, thus opening for a cryptanalysis attack, it is required that the sum of the plaintext is smaller than the product of the primes[1]. This objective has not been accomplished in the implementation as it only supports primes less than 2^{64}-1. The RSA encrypts the plaintext in blocks of letters[1], but due to the restriction in prime number length, the application will encrypt each letter separately. The process of encryption follows a strict algorithm[1];

$$\text{Ciphertext} = \text{PublicKey}(\text{Public Key Factor}, \text{Plaintext})$$
$$\rightarrow \text{Ciphertext} = \text{Plaintext}^{\text{Public Key Factor}} \text{ MOD Product of Primes}$$

However, doing $\text{Plaintext}^{\text{Public Key Factor}}$ and afterwards doing MOD product of primes will cause major performance issues[1]. This is solved by implementing an algorithm where the remainder of the product is used as base for each round of multiplication.
The figure below shows how this is implemented in Microsoft's C#.

```
public static long ModPower(long number, long power, long mod)
    {
        var tempNumber = number;
        for (int i = 1; i < power; i++)
        {
            number = number*tempNumber;
            number = number%mod;
        }
        return number;
    }
```

The Decryption

Since the RSA is not a symmetric cipher, it does not require the decryption process to be a reverse algorithm of the encryption, thus opening for the use of a different key; the private key. The private key consists of the Private Key Factor and the product of primes. The decryption process is similar to the encryption process – with one exception; the ciphertext is raised to the power of the prime key factor[1].

$$\text{Plaintext} = \text{PrivateKey}(\text{Private Key Factor}, \text{Ciphertext})$$
$$\rightarrow \text{plaintext} = \text{Ciphertext}^{\text{Private Key Factor}} \text{ MOD Product of Primes}$$

There is no way of separating the ciphertext values as it is pure binary without any break character. The solution is to add padding to the binaries during the encryption process[1] and for this implementation a padding of 16 was chosen. This allows the program to work with UTF8 values ranging from one through 65536, but the program has not implemented any support for characters beyond the normalized Latin alphabet.

Figure 4 shows a simple example in courtesy of Stallings[1].

The Pseudorandom Number Generator

Stallings[2] provides a chapter on Number Theory and generation of pseudo random number in his works, thus making it relevant to implement a pseudorandom *prime* number generator for use in the program. Stallings defines three criteria for a generator to be pseudorandom;

- The function should be a full-period generating function. That is, the function should generate all the numbers between 0 and modulo before repeating[2].
- The generated sequence should appear random[2].
- The function should implement efficiently with 32-bit arithmetic[2].

The pseudorandom number generator implemented builds on the theory behind the Linear

Congruential[2] Generator, explained by Stallings. This generator meets the given requirements and relies on the simple equation;

$$\boxed{X_{n+1} = aX_n \bmod m}$$

Where X_{n+1} is the next number in the sequence, X_n is the number at index n, a is the multiplying factor, and m is the modulo. The value a has been chosen by IBM[2] to be 7^5, or 16807, because of the low amount of 1s in the binary representation of the number – thus making it very speedy when implemented in a binary environment fulfilling the third requirement. The modulus is set to 2^{31} -1so that the generator fulfills the first requirement and at the same time fulfilling the second requirement by generating numbers over a wide spectrum. The initial value of X_n is based on a seed input.

The Really Big Number

Seeing the limitations of using a number system based on 64 bits, there was made an attempt to implement a number system where the size of the number is limited by 2^{64} -1 number of decimals. The number system consists of a list structure where each index in the list represents a 10 base number, where the largest base is at the beginning of the list. There are methods for addition, subtraction, multiplication, remainder and ModPow, and they all work to a certain degree. The code was verified by unit tests during development, but the code failed during testing when calculating really large primes. At one point in time, during subtraction, the code was unable to deduce the next 10 base value when lending is required, causing some indexes in the list to be larger than the value nine, and deducing the value at index zero when the given index value is zero, thus resulting in a negative number or a list whereas some of the index values are greater than nine. There is also a performance issue when working with big numbers as it only achieves O(N) run time at best, due to the amount of lookups and comparisons in the respective arithmetic methods - thus making it useless in a production setting.

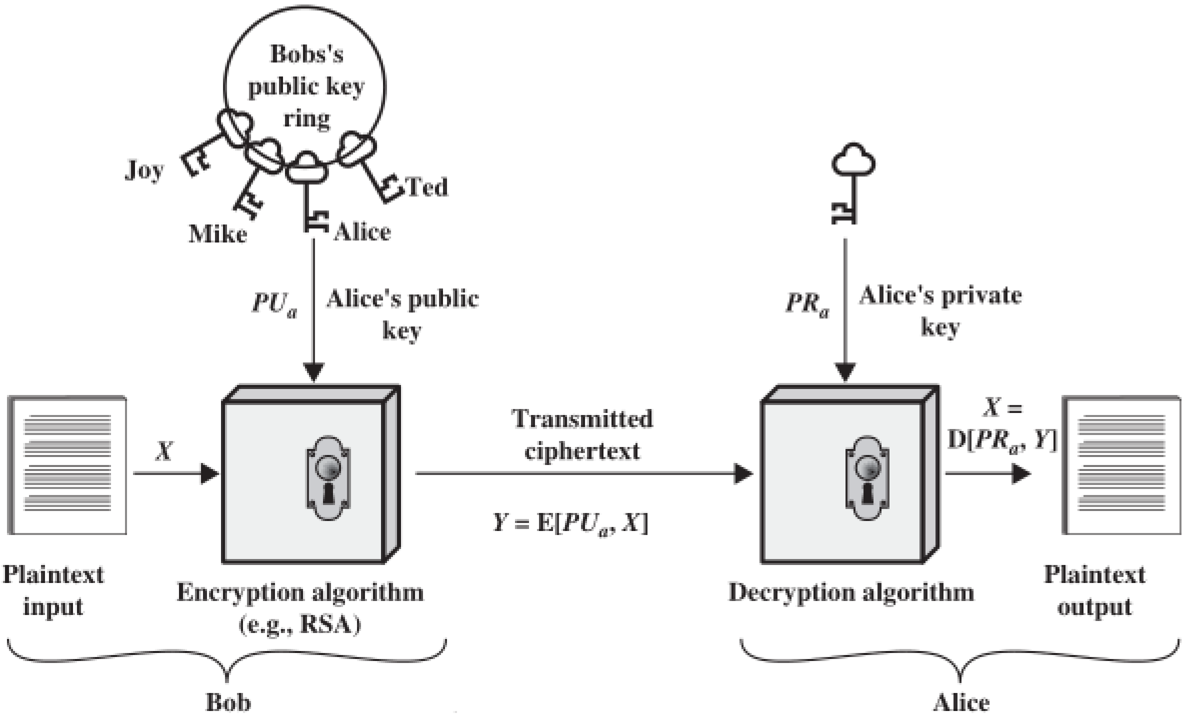


Figure 1- Encryption with Public Key[1]

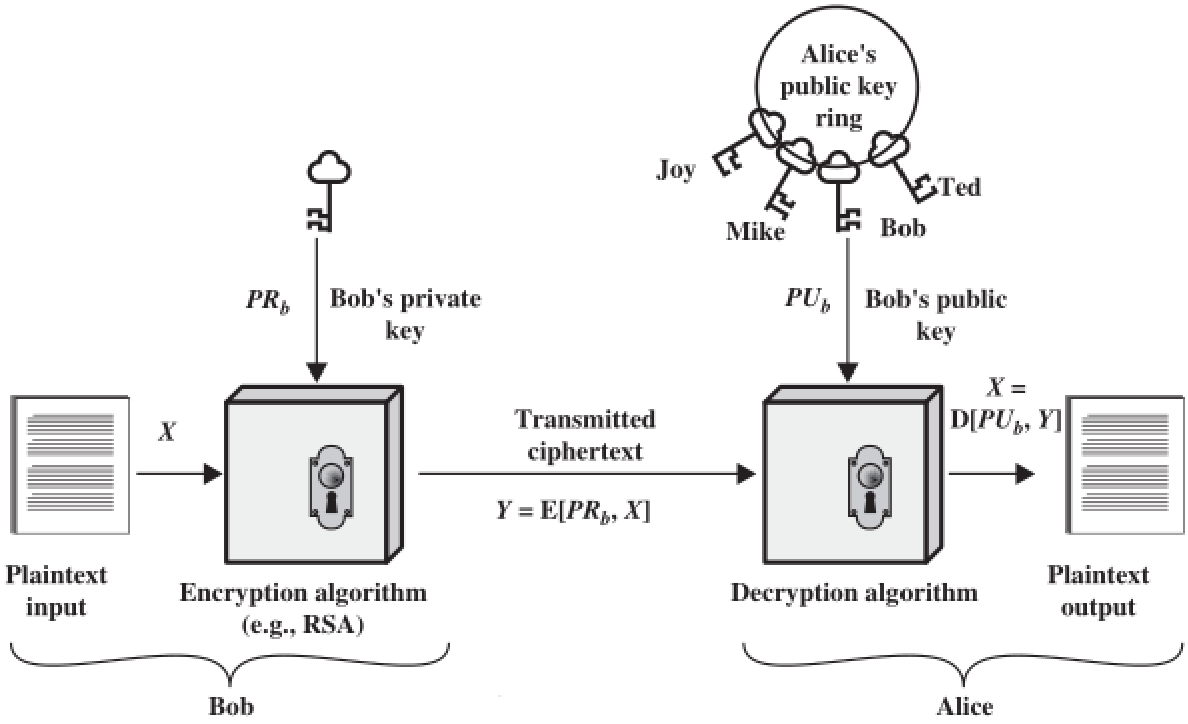


Figure 2 - Encryption with Private Key[1]

$$\phi(n) = \phi(pq) = \phi(p) \times \phi(q) = (p - 1) \times (q - 1)$$

Figure 3- Euler's Totient Function[1]

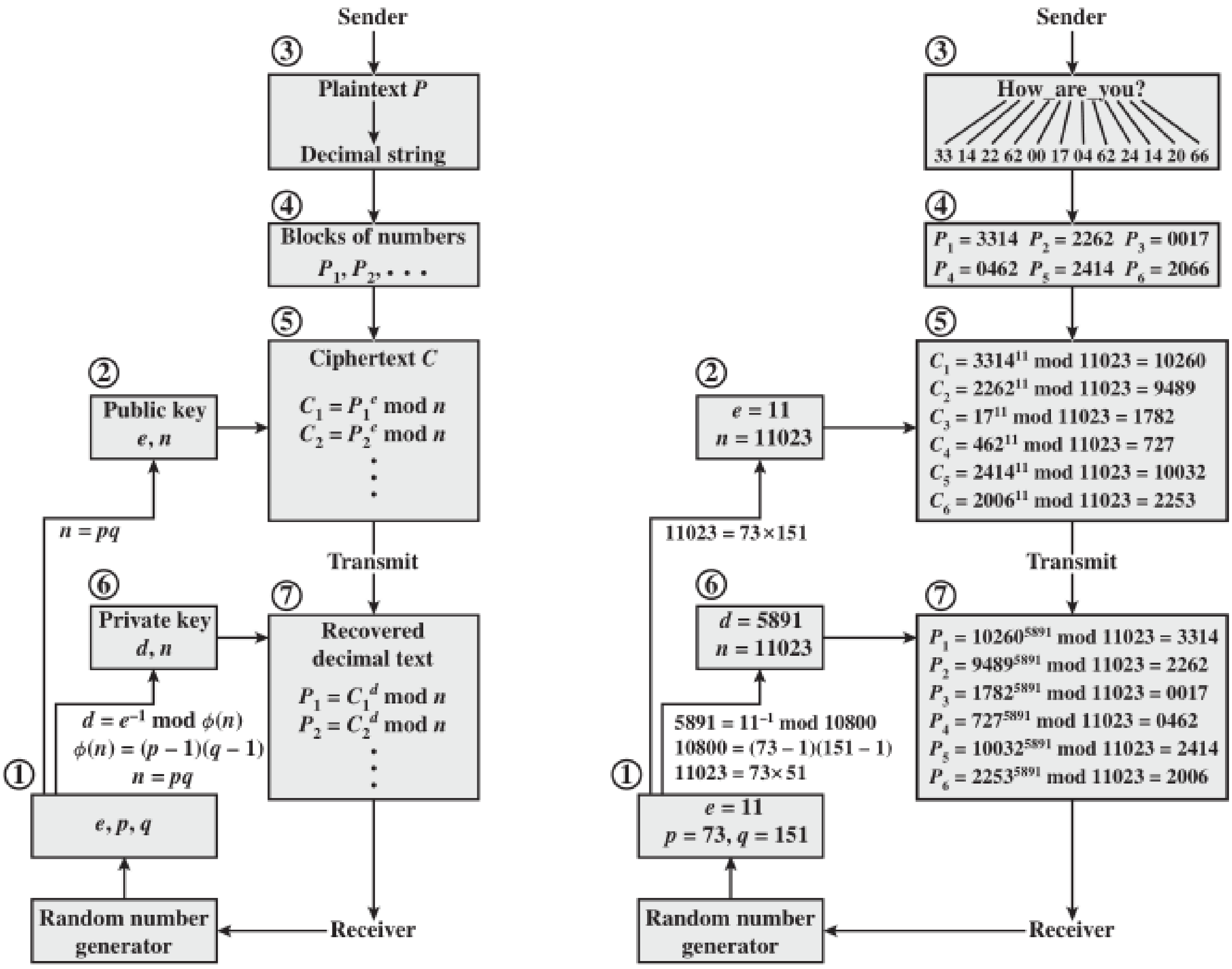
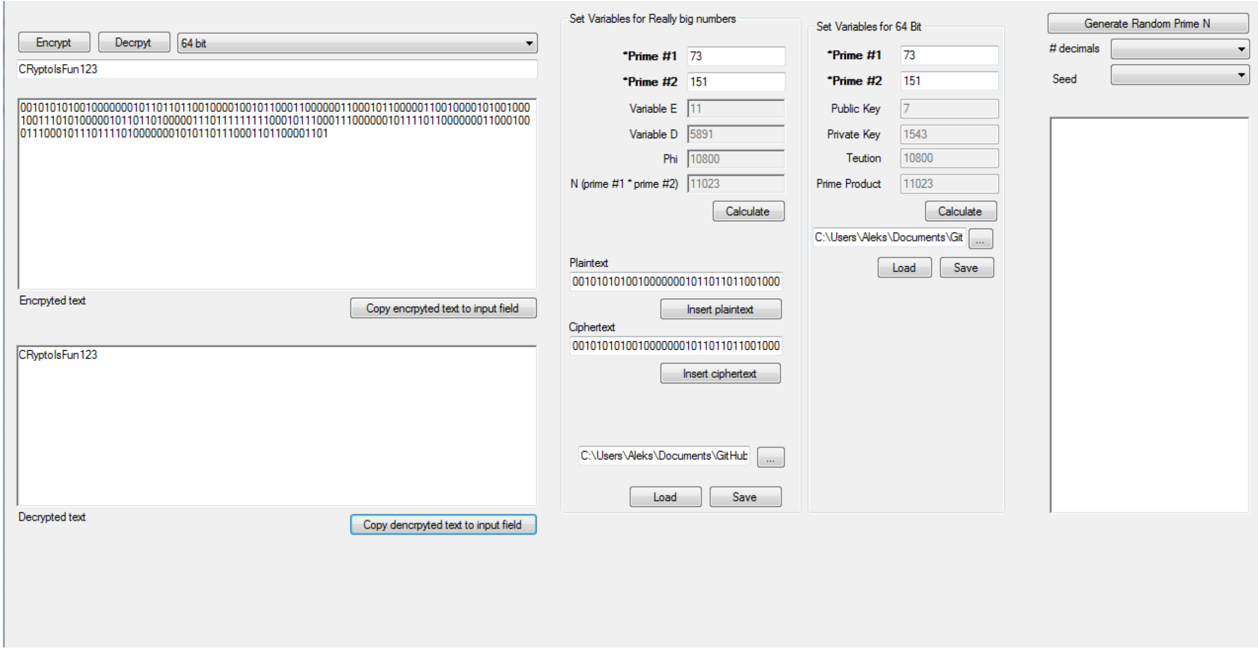

Sender
3
Plaintext P
Decimal string
4
Blocks of numbers
$P_1, P_2, \ldots$
5
Ciphertext C
$C_1 = P_1^e \bmod n$
$C_2 = P_2^e \bmod n$
2
Public key
e, n
$n = pq$
Transmit
6
Private key
d, n
7
Recovered decimal text
$P_1 = C_1^d \bmod n$
$P_2 = C_2^d \bmod n$
$d = e^{-1} \bmod \phi(n)$
$\phi(n) = (p-1)(q-1)$
$n = pq$
1
e, p, q
Random number generator
Receiver
Sender
3
How_are_you?
33 14 22 62 00 17 04 62 24 14 20 66
4
$P_1 = 3314$ $P_2 = 2262$ $P_3 = 0017$
$P_4 = 0462$ $P_5 = 2414$ $P_6 = 2066$
5
$C_1 = 3314^{11} \bmod 11023 = 10260$
$C_2 = 2262^{11} \bmod 11023 = 9489$
$C_3 = 17^{11} \bmod 11023 = 1782$
$C_4 = 462^{11} \bmod 11023 = 727$
$C_5 = 2414^{11} \bmod 11023 = 10032$
$C_6 = 2006^{11} \bmod 11023 = 2253$
2
$e = 11$
$n = 11023$
$11023 = 73 \times 151$
Transmit
6
$d = 5891$
$n = 11023$
7
$P_1 = 10260^{5891} \bmod 11023 = 3314$
$P_2 = 9489^{5891} \bmod 11023 = 2262$
$P_3 = 1782^{5891} \bmod 11023 = 0017$
$P_4 = 727^{5891} \bmod 11023 = 0462$
$P_5 = 10032^{5891} \bmod 11023 = 2414$
$P_6 = 2253^{5891} \bmod 11023 = 2006$
$5891 = 11^{-1} \bmod 10800$
$10800 = (73-1)(151-1)$
$11023 = 73 \times 51$
1
$e = 11$
$p = 73, q = 151$
Random number generator
Receiver

Figure 4 - RSA Example[1]

Test Results

The program works as expected with the 64-bit implementation, but does not calculate correctly when using the Really Big Number-implementation. The performance of the 64-bit implementation is good and could be used as a program in a human environment, but it is too slow to be used in a production low-level environment. This might be because the program does not manage to utilize more than 25% of the CPU power on my computer, mainly bottlenecked by the list stack in Microsoft’s C#.



64 bit

CRyptoIsFun123

```
001010101001000000001011011011001000010010110001100000011000101100000110010000101001000
10011101010000001011011010000001110111111111000101110001110000000101111011000000011000100
01110001011101111010000000010101101110001101100001101
```

Encrpyted text

CRyptoIsFun123

Decrypted text

Set Variables for Really big numbers

*Prime #1	73
*Prime #2	151
Variable E	11
Variable D	5891
Phi	10800
N (prime #1 * prime #2)	11023

Plaintext: 001010101001000000001011011011001000

Ciphertext: 001010101001000000001011011011001000

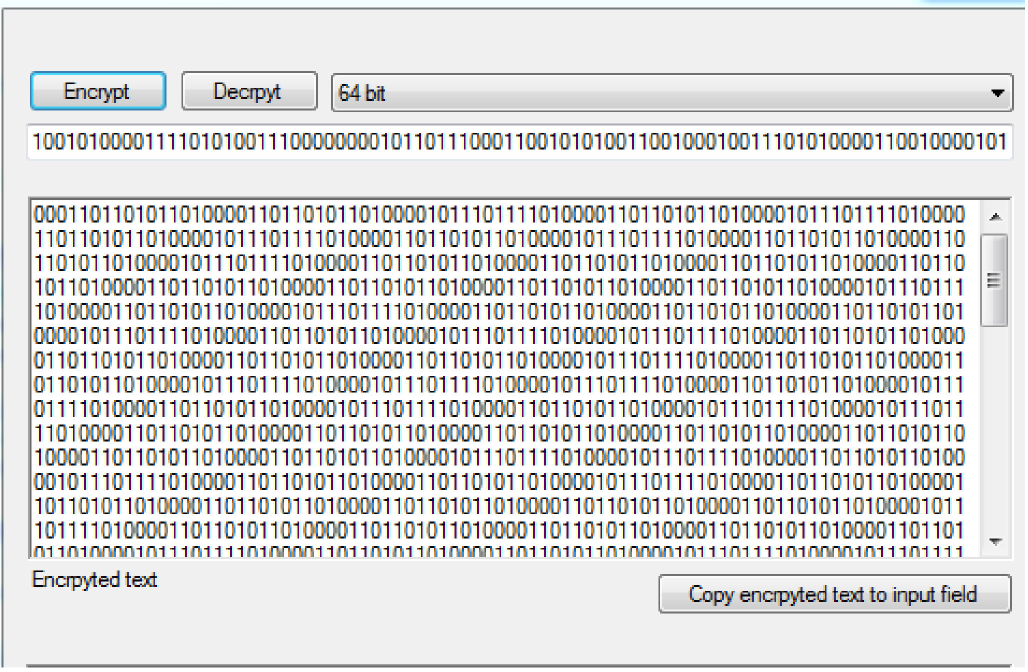
C:\Users\Aleks\Documents\GitHub

Set Variables for 64 Bit

*Prime #1	73
*Prime #2	151
Public Key	7
Private Key	1543
Teution	10800
Prime Product	11023

C:\Users\Aleks\Documents\Git

The screenshot below depicts the padding problem as there is a clear horizontal asymptotic indicator of 0’s between each letter.



64 bit

100101000011110101001110000000010110111000110010101001100100010011101010000110010000101

```
00011011010110100001101101011010000101110111101000011011010110100001011101111010000
11011010110100001011101111010000110110101101000010111011110100001101101011010000110
11010110100001011101111010000110110101101000011011010110100001101101011010000110110
10110100001101101011010000110110101101000011011010110100001101101011010000101110111
10100001101101011010000101110111101000011011010110100001101101011010000110110101101
00001011101111010000110110101101000010111011110100001011101111010000110110101101000
01101101011010000110110101101000011011010110100001011101111010000110110101101000011
01101011010000101110111101000010111011110100001011101111010000110110101101000010111
01111010000110110101101000010111011110100001101101011010000101110111101000010111011
11010000110110101101000011011010110100001101101011010000110110101101000011011010110
10000110110101101000011011010110100001011101111010000101110111101000011011010110100
00101110111101000011011010110100001101101011010000101110111101000011011010110100001
10110101101000011011010110100001101101011010000110110101101000011011010110100001011
10111101000011011010110100001101101011010000110110101101000011011010110100001101101
01101000010111011110100001101101011010000110110101101000010111011110100001011101111
```

Encrpyted text

Discussion

When encrypting the phrase “cryptoisfun123” with the keys used in the Test Results-part, one receives a binary;

```
000010001101001000011110101001110000100101100011000000110001011000001100
010000101001000100111010100000011010010000001110111111111100010001110100
000000101111011000000011000100011100010111011110100000001010110111000110
1100001101
```

Encrypting “cryptoisfun321”;

```
000010001101001000011110101001110000100101100011000000110001011000001100
010000101001000100111010100000011010010000001110111111111100010001110100
00000010111101100000001100010001110001101100001101000000101011011100010
11101111010
```

When encrypting “cryptoisfun321”, the use of single letter encryption blocks becomes obvious. The similarity is very unfortunate as it makes the cipher vulnerable to attacks when even a single phrase in a sentence is repeated during transmission. This would be less of a problem if the algorithm implemented a block size determined by the base of the key size, i.e. block size of 50 bytes when the key is 1024 bits, thus making it nearly infeasible to crack by pure cryptanalysis. This attack method is if often referred to as “chosen ciphertext attacks”[1]. The use of even-number padding reduces confidentiality as the cipher lacks the typical permutation operations often found in block ciphers. One could use asymmetric padding, or odd-number padding, to cope with this aspect, but it would require more overhead in the message text[1].

The cipher is vulnerable to brute force attacks just as any other cipher[1], but the RSA has overcome this by implementing a 1024-bit key, or even 2048-bit key. However, this enhanced protection against brute force attacks comes at the cost of speed and efficiency. My implementation is limited to the 64-bit space, thus making it very easy to brute force as there are at maximum 2^{64}-1 possible keys. With a modern CPU this kind of brute force attack would consume less time than brewing a proper can of coffee.

There was made an attempt at implementing a number system allowing for numbers as large as $10^{18446744073709551616}$ -1, but the run time when using a 10 base number system implemented on a x86-CPU was merely too time consuming, especially when there was a bug in the subtraction method – impossible to debug when reaching numbers with 309 decimals or larger.

The RSA is also subject to mathematical attacks in the form of prime number deduction. The cipher broadcasts the public key, which contains the product of the two primes. If the initial two primes are not large enough[1], in the 10^{75} to 10^{100} order, the process of factorizing the prime product will only take about 8000 MIPS-years when the key size is 512 bit[1], whereas 250 MIPS-years is equivalent to a 1 GHz Intel Pentium processor[1]. Given the vast amount of computational power available in datacenters today, 8000 MIPS-years is nothing.

Conclusion

The objective of this assignment was to implement a rudimentary RSA cipher with a simple user interface. The implementation encrypts and decrypts data as specified by the standards of RSA, with the exception of block size and key size. These factors make the cipher very vulnerable to cryptanalysis, and it is very much possible to improve the security of the implementation by introducing a number system capable of handling numbers larger than 64-bit. It is somewhat sad that the “Really Big Number”-work did not pull through as it was my solution to the problem. The implementation of “Really Big Number” took over 70 hours to implement compared to around 2 hours for implementing the 64-bit version, thus making it a tolling task. A different way to improve the cipher is to move away from a number system with base 10 to a number system with base 2, making it faster to perform arithmetic operations, as well as easier to implement in a closed hardware circuit.

The implementations in this assignment are not fast or secure enough to be used in any production environment.

Works Cited

1. Stallings, William. Cryptography And Network Security - Principles And Practice, 5th edition. Essex: Pearson Education Limited, 2011: Chapter 9.
2. Stallings, William. Cryptography And Network Security - Principles And Practice, 5th edition. Essex: Pearson Education Limited, 2011: Chapter 7.

All illustrations are in courtesy of Stallings, William. Cryptography And Network Security - Principles And Practice, 5th edition.