RSA Public Key system: Modern Cryptanalysis

Senior Research from 6/5/17 - 8/18/17

Juan G. Vargas Jr.

Contents

Abstract

In today's world electronic communication helps our society maintain it's futuristic structure. Our growing reliance on computer systems' ability to communicate throughout the Internet has created what we now refer to as "big data". Modern systems have the utmost important task of protecting the integrity of that data and ensuring it can be transmitted through cyberspace. This project dives into one of the most widely used public-key system, the RSA cryptosystem, and show how and why it works. From essential number theory components, proofs, small 4-bit message example, this entry will cover the expectations of a practical encryption scheme and a scaled down "plain old" RSA program while remarking on advanced implementation techniques and RSA's computational drawback.

1 Components of a Cryptosystem

A Cryptosystem is the actual implementation of cryptographic techniques and infrastructure. Modern cryptosystems are derived from two types of schemes; symmetric and asymmetric algorithms. For the sake of conceptual understanding this paper will refer to asymmetric as "public-key", the reason will be known later. The strengths of a single algorithm are not enough to protect data in the modern era of supercomputers since each has the capacity to break a system with merely a brute force approach. Each algorithm has it's weakness, but this is why a hybrid system is used in a real-world setting. Since both symmetric and asymmetric algorithms have its benefits it is good practice to merge the two into a stronger, and more complex, cryptosystem. Even though there are different schemes, such as symmetric, asymmetric, and hybrid (a combination of both symmetric and asymmetric); all practical system will still consist of these components.

- **Plaintext**, this is the data that is transfered between nodes.
- **Ciphertext**, this is the encrypted version of the plaintext data file. The ciphertext is produced by feeding in the plaintext into an encryption algorithm.
- **Encryption algorithm**, the mathematical process that takes a plaintext and an encryption (public) key and produces a ciphertext
- **Decryption Algorithm**, also uses some mathematical process that produces a unique plaintext for any given ciphertext and decryption (private) key.

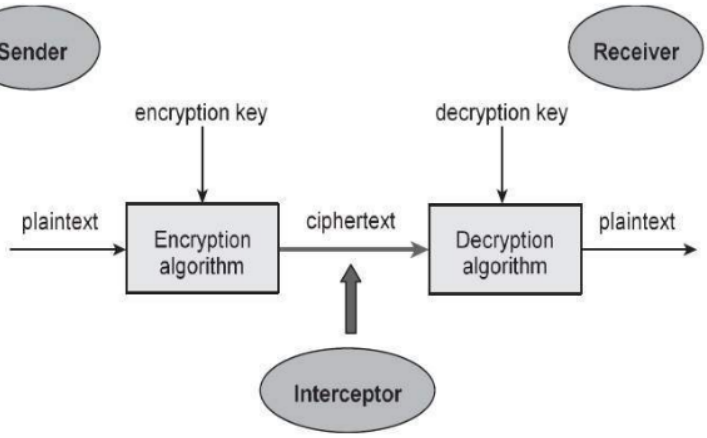

Figure 1: This is a basic cryptosystem design

Why stick to established algorithms? Certainly an unwanted attacker cannot break into a system that uses a unique approach for security, unfortunately – this is far from the truth. Just because the details of a new algorithm remains unknown to attackers, doesn't make it secure. This idea known as *security through obscurity* relies too heavily on keeping the protocols of a cryptosystem secret which can easily be reviled through reverse engineering. Algorithms the industry use now have been proven "computationally secure", meaning that even a supercomputing system will not be able to extract key values within a reasonable amount of time. Imagine using a brute force method of factoring a huge number and checking if the factored prime number are valid, this is how RSA could be broken. It would take years to use such a method thus making it computationally secure.

To avoid such a problem a common philosophy that cryptosystems follow is known as Kerckhoff's principal. *A cryptosystem should be secure even if the attacker knows all the details about the system, with the exception of the secret key. In particular, the system should be secure when the attacker knows the encryption and decryption algorithms.*

1.1 Cryptosystem Security Services

A modern cryptosystem provides four primary services, and in the case of RSA is done with padding scheme and digital signatures. Implementation on this project will not include these ideas since the project is "plain old" RSA but this concepts are important for overall understanding of a cryptosystem.

- **Confidentiality**, keeps information from an unauthorized users. Confidentiality is achieved with either physical securing or mathematical algorithms for encryption.

  30

- **Data Integrity**, is a service that identifies any alteration to the data. Data may be modified by unauthorized entity intentionally or accidentally. Data integrity confirms whether data is intact or manipulated since it was last created, transmitted, or stored by an authorized user. **Note:** Data integrity does not prevent the alteration of data, but states whether data has been manipulated.

- **Authentication** relays the identification of the originator of a data packet. Confirms to the receiver that the data received has been sent only by an identified and verified sender.

  - **Message Authentication** identifies the originator of the message without any regard router or system that has sent the message
  - **Entity Authentication** is assurance that data has been received from a specific entity, say a particular website.

- **Non-repudiation** ensures that an entity cannot refuse the ownership of a previous commitment or an action. Assures that the original creator of the data cannot deny the creation or transmission of the said data to a recipient or third party. This is used for dealing with situations like an online transaction. Once an order is placed electronically, a purchaser cannot deny the purchase order if Non-repudiation service was used,

1.2 Public-Key Infrastructure

In a public key cryptosystem the process requires two different keys; one used for encryption (the public key) and the other for decryption (the private key). Another important property of a public

key scheme is the guarantee that each recipient has a unique private key used for decryption. This guarantee is made when the selection of an encryption value 'e' is tested to have an inverse value, also know as the decryption value 'd'.

The big advantage in a Public-Key scheme is the ability to freely distribute public keys, as opposed to symmetric key distribution but creates problems of its own. If a public key is easily available in a public domain it is prone to interception in order to gain access to a system. In response to such an attack there must be some infrastructure to establish a trusted channel, like the security authenticators mentioned above, and some sort of key manager.

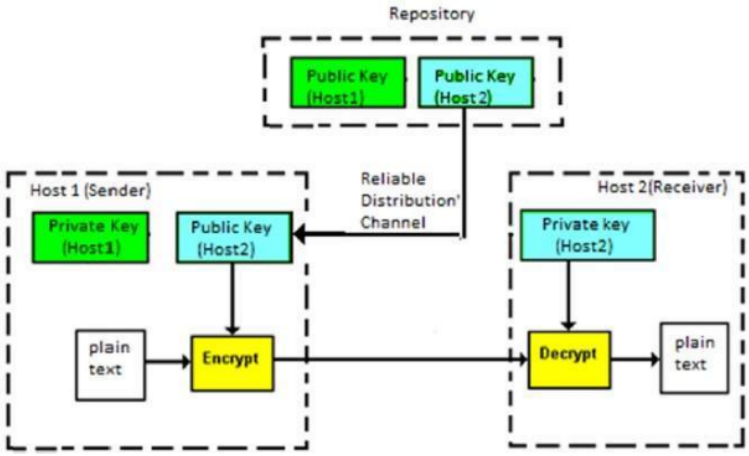

Figure 2: This is a basic Asymmetric-key cryptosystem scheme

2 Essential RSA Functions

The RSA scheme relies on a few mathematical truths; the difficulty of factoring a very large number in order to find out its prime factors used in constructing variable 'n', that prime number generation is relatively easy, and modular arithmetic works within a finite set. There are other tricks to make RSA practical in a large scale setting such as: easily simplifying very large exponent computations, and reducing the values used when solving the modulo of two variables. These theorems and algorithms are used to maintain the structure of the mathematical principals around the RSA scheme.

2.1 Modular Arithmetic

In cryptography, both symmetric and asymmetric models, are based on a finite number of elements. This doesn't mean cryptography is based off a severely limited set but simply that if we define our number set to some range we are able to use those number efficiently. Meaning, we can use modular arithmetic to perform arithmetic on this finite set.

For example, consider the set of 7 numbers:

$$\{0, 1, 2, 3, 4, 5, 6\}$$

Perform any arithmetic as long as the results are less than 7 such as:

$$4 + 1 = 5$$
$$3 \text{ x } 2 = 6$$

Notice that 5 + 6 is greater than 7. What if we divide the result of 5 + 6 by the number 7 and only consider the remainder as our answer? That would insure that all arithmetic within the set has a result that is also contained within the original number set. Since 5 + 6= 11 and 11 / 7 = 4

$$5 + 6 \equiv 4 \, mod \, 7$$

This finding yields the following definition.

Modulo definition:
Let a,b,n $\in \mathbb{Z}$ (Where $\mathbb{Z}$ is a set of all integers) and n > 0.

$$a \equiv b (mod \, n)$$

If n divides a - b, n is refereed as the modulus and b is called the remainder

The modulo relation can be rewritten as:

$$a = k \cdot n + b \qquad for \; 0 < b < n$$

By definition a ≡ b(mod n) is a congruence relationship and a congruence relationship satisfies all the condition of an equivalence relation so their properties also apply.

Reflexivity: a ≡ a(mod n)
Symmetry: a ≡ b(mod n) if and only if b ≡ a(mod n)
Transitivity: If a ≡ b(mod n) and b ≡ c(mod n), then a≡ c(mod n)

2.2 Extracting Modular Inverses

Finding the greatest common divisor is an import concept in RSA. One common use for the gcd is to find if a number pair is co-prime, which is when two numbers have no common factors other than 1. What the gcd provides is the product of all common primes of two numbers, meaning:

$$84 = 2 \cdot 2 \cdot 3 \cdot 7$$

$$30 = 2 \cdot 3 \cdot 5$$

$$gcd(84, 30) = 2 \cdot 3 = 6$$

Factorization works for smaller numbers but for larger number a more robust algorithm is needed. This is where Euclidean algorithm excels, by reducing the gcd of two given numbers to two smaller numbers this reduces the amount of computation needed. The Euclidean algorithm isn't the focus in RSA, instead it is the Extended Euclidean algorithm (EEA) that is vital. EEA takes two numbers and along with computing the GCD it also finds the numbers modular inverse. The EEA does this by working through this linear combination form when $r_0 > r_1$:

$$gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$

Where s and t are coefficients, these values are the modular inverses for the given number pair r_0 and r_1. EEA executes similarly to Euclidean algorithm but also expresses the current remainder r_i for each iteration as the aforementioned mentioned linear combination:

$$r_i = s_i \cdot r_0 + t_i \cdot r_1$$

The last iteration yields:

$$r_l = gcd(r_0, r_1) = s_l \cdot r_0 + t_l \cdot r_1 = sr_0 + tr_1$$

Typically this visual aid used for calculating, we'll use $gcd(r_0 = 20, r_1 = 3)$ as an example:

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i]r_0 + [t_i]r_1$
2	$20 = 6 \cdot 3 + 2$	2 = [1] · 20 + [-6] ·3
3	$3 = 1 \cdot 2 + 1$	1 = [-1]·20 + [7]·3
4	$2 = 2 \cdot 1 + 0$	complete

This example is concluded with s = -1 and t = 7, which are the inverses of r_0 and r_1 respectively. In RSA the EEA is used to compete decryption value 'd' given the parameters gcd($\phi(n)$,e) = 1. The implementation used for the EEA goes as follows:

Extended Euclidean Algorithm: (EEA)
Input: positive integers r_0 and r_1 with $r_0 > r_1$
Output: gcd(r_0, r_1), as well as s and t such that gcd(r_0, r_1) = s · r_0 + t · r_1
Pseudo code:
$s_0 = 1$ $t_0 = 0$
$s_1 = 0$ $t_1 = 1$
i = 1
Implementation:
DO
    i = i + 1
    $r_i = r_{i-2}$
    $q_{i-1} = (r_{i-2} - r_i) / r_{i-1}$
    $s_i = s_{i-2}$
    $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
WHILE
RETURN
    gcd(r_0, r_1) = r_{i-1}
    s = s_{i-1}
    t = t_{i-1}

2.3 Number Theory Components

A common requirement for the RSA infrastructure is certain variables must be **Relatively prime**. Numbers are relatively prime when they have no common factors other than 1, in other words you cannot evenly divide both by some common value other than 1. This idea is used along with Fermat's Little Theorem in order to simplify the computation of exponents in modular arithmetic and is stated as:

Theorem 1. ***Fermat's Little Theorem***
Let a be an integer and p be a prime, then:

$$a^{p-1} \equiv 1(modp)$$

To see why Fermat's theorem works assume 'a' is some integer and 'p' is a prime number that is not divisible by 'a'. Let set R ={1,2,3,...(p-1)} express every integer congruent to positive nonzero integers {1,2,3,...(p-1)} (mod p) . By multiplying all elements in R by a, R·a = {1a,2a,3a,...(p-1)}, we see that this is merely a rearrangement of original set R. This is made clear with an example:

Let a = 3 and p=7

$$1 \cdot 3 \equiv 3 \text{ (mod 7)} = 3$$
$$2 \cdot 3 \equiv 6 \text{ (mod 7)} = 6$$
$$3 \cdot 3 \equiv 2 \text{ (mod 7)} = 2$$
$$4 \cdot 3 \equiv 5 \text{ (mod 7)} = 5$$
$$5 \cdot 3 \equiv 1 \text{ (mod 7)} = 1$$
$$6 \cdot 3 \equiv 4 \text{ (mod 7)} = 4$$

A generalization can be made of this observation as:

$$\text{R·a} = \{\text{a·i(mod p)} \mid \text{i} \in \text{R}\}$$

To further prove this observation there are two cases to address.
Case 1: We can say that none of the elements in R are congruent to 0. Suppose:

$$a \cdot i \equiv 0(modp)$$

Then thats to say p/(r·a) but is impossible since p $\nmid$ a and r $<$ p

Case 2: All elements are distinct since there are no two sequences congruent to each other
$\forall$ i, j $\in$ R: a· i, a·j So, ai $\neq$ aj(mod p) when

$$0 < i < p$$
$$0 < j < p$$

By assuming that:

$$ai - aj = a(i - j)$$

We know that p $\nmid$ a but does p divide i-j? By referring to the inequalities and multiplying the second inequality by -1 and adding the two together we get:

$$0 < i < p$$
$$-p < -j < 0$$
$$-p < i - j < p$$

Since both i and j are distinct then i -j $\neq$ 0 then p $\nmid$ i -j. Which solidifies the original assertion, proving our generalization holds true for all cases. Now that we have shown that both R and a·R are equivalent but rearranged sets we can say:

$$\text{a}\cdot\ 2\text{a}\cdot\ 3\text{a}\cdot\ldots(\text{p-1})\text{a} \equiv 1\cdot\ 2\cdot\ 3\ldots(\text{p-1}) \pmod{\text{p}}$$
$$a^{p-1}\ (\text{p-1})! \equiv (\text{p-1})! \pmod{\text{p}}$$
$$a^{p-1} \equiv 1 \pmod{\text{p}}$$

Thus, proving Fermat's Little Theorem.

Theorem 2. ***Euler's Theorem***
Let a and n be integers with gcd(a,n) = 1, then:

$$a^{\phi(n)} \equiv 1(mod n)$$

A generalization of Fermat's Little theorem to any integer moduli that are not exclusively primes is Euler's theorem. If the stated 'n' variable is prime, it holds that:

$$\phi(n) = (n^1 - n^0) = p - 1$$

When 'n' is NOT a prime number but 'p' and 'q' are prime:

$$\phi(n) \equiv \phi(p) \cdot \phi(q) \equiv (p-1)(q-1)$$

So it would become apparent that:

$$a^{\phi(p)} = a^{p-1} \equiv 1(mod\, p)$$

2.4 Fast Exponentiation

The computational load of the RSA scheme is intensive. For example: assuming practical application of RSA with a 2048-bit modulus, an encryption value 'e' and decryption value 'd' can be within the range of 1024 to 3072 (generally e is the smaller value of the two). Just with those values alone it is easy to see how any data message to be encrypted/decrypted will quickly become a computational nightmare for a single exchange. Straightforward exponentiation for 2^{26} would look something like this:

$$x \xrightarrow{\text{SQ}} x^2 \xrightarrow{\text{MUL}} x^3 \xrightarrow{\text{MUL}} x^4 \xrightarrow{\text{MUL}} x^5 \ldots . x^{26}$$

So what can be done to drastically decrease the computational load? Rather than approaching the problem above where we merely squaring the starting x then multiply by x until we have x multiplying itself 28 times the Square-and-Multiply Algorithm distributes the squaring computation amongst this string of multiplication as follows:

$$x \xrightarrow{\text{SQ}} x^2 \xrightarrow{\text{MUL}} x^3 \xrightarrow{\text{SQ}} x^6 \xrightarrow{\text{SQ}} x^12 \xrightarrow{\text{MUL}} x^{13} \xrightarrow{\text{MUL}} x^{26}$$

in only six operations we can solved for x^{26} as opposed to 26 operations. This is done by the Square-and-Multiply Algorithm as stated.

Square-and-Multiply algorithm
Input:
base element 'x'
exponent 'e'
Output:
returns x · y where y is the result of binary exponentiation
Pseudo Code:
```
if e is zero
    y = 1
while e is greater than 1
    if e is even then
        x = x · x
        e = e/2
    else          y = x · y
        x = x · x
        e = (e -1)/2
return x · y
```

2.5 Primality Tests

Primes are a core component of the RSA algorithm but there is a problem with using random primes, how do you find a huge random prime and how would you even know if it is an actual prime rather than some composite number? There are primality test algorithms but they do not provide certain results, merely that a number is probably prime or not a prime. Lucky the nature of these primality tests is that you can run them multiple times in order to increase the odds of said number being an actual prime. In this "plain old" RSA implementation we will use the Miller-Robin primality test.

Miller-Rabin Primality test
Input:
Prime number candidate 'p' when p-1 = 2^s·r where r is odd
Output:
Candidate 'p' tested with specific iterations,if 'p' is composite returns true, false otherwise
Pseudo Code:
```
s = p -1 ,t = 0
While s is even:
    Halve s and increment t by 1
Loop for however many times, depending on prime number size
   Select a random integer from 2 to p-1
   Set v = a^s(mod p)
   If v!= 1
       Index i is set to 0
       While v!=(p-1):
           if i == t-1: return false
           else: i incremented
               v = v^2 (mod p)
   return True
```

Proof of the Miller-Rabin test goes as follows:

The more values in 'a' we test, the better the accuracy. It can be shown that for any odd composite n, at least 3/4 of the values 'a' are witnesses for the compositeness of number 'n'. If 'n' is composite then the Miller–Rabin primality test states that n is probably prime with a probability at most 4^{-k}

According to FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION(FIPS): Digital Signature Standard (DSS), of page 80 table C-2 there is a standard number of test each prime must iterate through in order to be used in practical settings and we shall use as well. The number of tests on a number 'n' are as follows:

Parameters	M-R test only
p and q: 512 bits	For p and q: 7 times
p and q: 1024 bits	For p and q: 4 times
p and q: 1536 bits	For p and q: 7 times

There are other strategies used in this implementation in order to increase the lightly hood of a number being know as a prime. This is important in RSA since a non-prime number will not ensure proper encryption/decryption since the theorems used will not hold with a non-prime number.

3 The RSA Cryptosystem

3.1 Key Generation

There are two keys that must be generated; the public and private key. RSA Key Generation is a critical part of the RSA algorithm because the values chosen at this level determines how secure the algorithm is. For example, when selecting encryption exponent 'e' there must be a unique inverse, so any value will not work. The encryption value must also be a reasonably big number even if the set of possible integers include {1,2,...$\phi(n)-1$} Values chosen at this point must to checked in order to ensure that the algorithm is in a sable form.

RSA Key generation
Output: public key = k_{pub} = (n,e) and private key = $k_{private}$ = (n,d)

1. Choose two large prime numbers p and q
2. Compute n = p · q
3. Compute Φ(n) = (p - 1)(q - 1)
4. Select the public exponent e ∈ 1,2, such that
$$gcd(e, \Phi(n)) = 1$$
5. Compute the private key d such that
$$d \cdot e \equiv 1 mod \Phi(n)$$

Note: The condition that gcd(e, Φ(n)) = 1 ensures that the inverse of e exists modulo Φ(n), so that there is always a private key d.

In accordance with Step 1, prime number selection for 'p' and 'q' should be about 512 bits, or 155 decimal digits, in size for practical application. This is because the RSA modulus value n (computed in Step 2) should be at least 1024 bits in order to meet practical standards. RSA becomes easy to break when an outside user is able to find one of the prime factors used for $\phi(n)$ so searching for a good prime number of about 512 bits and ensuring those primes are not within the same range as each other adds to the attackers complexity of trying to find the prime factors. It is also worth noting that both these values, p and q, must be co-prime of each other. By the very nature of primes, they have no other factors other than themselves and the number 1 so this should not be a concern unless the prime number selection algorithm does not guarantee that the numbers chosen are prime, which is a real problem for algorithms like the Rabin-Miller primality algorithm.

3.2 Encryption and Decryption

Before encryption can begin on the plaintext there are some matters to address. The plaintext value may represent any data–number, letters, or symbols– so how would we use this data found in 'x'? First we must convert the value 'x' into a single number representation, whether it be a decimal or hexadecimal representation is a design choice. For this example the plaintext data value will be converted into a single decimal value. To do so let's assume the plaintext is storing the word 'attack', now the algorithm must convert each individual element into their corresponding 4-bit ACSII values but expressed in binary. This practice is typically called *binary decomposition*.

'attack'

↓

0097 0116 0116 0097 0099 0107

↓

01100001 01110100 01110100 01100001 01100011 01101011

↓

$(011000010111010001110100011000010110001101101011)_2$

↓

$(107152796640107)_{10}$

With key generation already settled the encryption and decryption stages are fairly straightforward. Using the new plaintext value 'x' and the encryption exponent 'e' the encryption algorithm works as follows.

RSA Encryption: Given the public key (n,e) = k_{public} and the plaintext 'x':

$$y = x^e (mod\, n)$$

The conclusion of the encryption process generates the ciphertext from the plaintext and reassigns this new value to 'x'. The now modified 'x' value is now referred to as the ciphertext. Using the ciphertext on the decryption process will extract the original plaintext using a similar approach to encryption.

RSA Decryption: Given the private key (n,d) = $k_{private}$ and the ciphertext 'y':

$$x = y^d (mod\, n)$$

Although the data value 'x' has undergone encryption and decryption, this does not conclude the process. Since the original plaintext was converted into a singular decimal value it is necessary

to recompose the original method. First the value 'x' must be converted into a binary string. That binary string is then separated into 8 bit strands, where each strand represents an ACSII value, those ACSII values are converted back into text and now the original message has been recovered.

$$(107152796640107)_{10}$$
$$\downarrow$$
$$(01100001011101000111010001100001011000110110101 1)_2$$
$$\downarrow$$
$$01100001 \quad 01110100 \quad 01110100 \quad 01100001 \quad 01100011 \quad 01101011$$
$$\downarrow$$
$$0097 \ 0116 \quad 0116 \quad 0097 \quad 0099 \quad 0107$$
$$\downarrow$$
$$\text{'attack'}$$

In real applications *x,y,n* and *d* would be around 1024 bits long. This makes the application of the Square-and-Multiply algorithm, and the Chinese Remainder Algorithm critical for successful RSA behavior. Even from our simple example it is plain to see how big the numbers can become, the Square-and-Multiply algorithm ensures that each multiplication done onto the data value is optimal. The Chinese Remainder Algorithm helps bring down the cost of computation by reducing the now exponentiated 'x' value mod 'n'. By identifying that the current larger values in x (mod n) can be reduced to the smaller but equivalent x_2 (mod n_2) values, the encryption and decryption computational load is reduced significantly.

3.3 RSA Proof of Correctness

In order to define RSA as a valid cryptosystem algorithm, we prove it's generality for any message x that we wish to encrypt and decrypt.

Theorem 3. *For all integers x, when $e > 0$*

$$x^{ed} \equiv x(mod\, n)$$

Suppose that gcd(x,n) = 1 and start with the construction rule for public and private keys:

$$e \cdot d \equiv 1(mod \phi(n))$$

Using an alternative representation of the modulo operator yields:

$$e \cdot d \equiv 1 + t \cdot \phi(n)$$

Coefficient t is found using the EEA when gcd($\phi(n)$, e) = s ·ϕ(n) + t ·e Then comes the application of this equivalence relation. We also can assume that when e·d is applied as an exponent to x the outcome is x^1 since encryption and decryption leaves the original message x.

$$x^{ed} \equiv x^{1+\phi(n)t} \equiv x^1 \cdot (x^{\phi(n)})^t \equiv (x^{\phi(n)})^t \cdot x(mod\, n)$$

Recalling Euler's Theorem 2 we can manipulate the equation due to the modulo's symmetric property to $1 \equiv \phi(n)\, mod\, n$. We now can make a generalization:

$$(\phi(n))^t mod\, n \equiv (1)^t \equiv 1$$

With this generalization we now have:

$$(x^{\phi(n)})^t \cdot x \equiv 1 \cdot x(mod\, n)$$

Thus proving that:

$$(x^e)^d \equiv x(mod\, n)$$

3.4 RSA Example

Before any process can begin the key pairs must be generated. This process yield us with values for: *p, q, n, ϕ(n), and e.*

NOTE: In order to remain within a reasonable bit range, this example will merely encrypt the plaintext value x = '7'. The selection of prime numbers p and q must be relatively prime (as shown in RSA proof).

$$p = 3, q = 11$$

Yielding the following values for n and ϕ(n). In order to maintain the algorithm we check that value 'x' is within the set { 0,1,...,(n-1)}, these values check out. If n-1 is less than the value 'x' choose greater primes.

$$n = p * q = 33$$

$$\phi(n) = (p-1)(q-1) = 20$$

Next is the selection of encryption value 'e' and selection must ensure that the inverse of value 'e' exists (e value requirements are listed in 2.1) The inverse of e is given by the Extended Euclidean Algorithm when given parameters gcd(ϕ(n),e) = s·ϕ(n) + t·e

i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = [s_i]r_0 + [t_i]r_1$
2	$20 = 6 \cdot 3 + 2$	$2 = [1] \cdot 20 + [-6] \cdot 3$
3	$3 = 1 \cdot 2 + 1$	$1 = [-1] \cdot 20 + [7] \cdot 3$
4	$2 = 2 \cdot 1 + 0$	complete

$$e = 3, d = 7$$

This ends the key generation phase as all required values are defined within appropriate boundaries. Next the plaintext data is prepared for encryption and decryption using the methods used in 3.1 Key Generation, but for simplicity we shall assume plaintext is still '7'.

Not we set up for encryption:

$$x^e(modn) = 7^3(mod\, 33) = 343(mod\, 33) = 13$$

The plaintext is now encrypted as '13', which is now assigned to 'y' value or the ciphertext. To retrieve the original message we must undergo decryption on the ciphertext.

$$x = y^d(mod\, n) = 13^7(mod\, 33) = 62,748,517(mod\, 33) = 7$$

Thus retrieving our original encrypted value.

4 Project Overview

The goal of this analysis is to mimic the behavior of an RSA cryptosystem on a small scale, meaning, there will be no large scale key management system in place, nor padding protocols. The RSA scheme is heavily dependent on modular inverses and exponentiation, so research among these topics was required. Number theory concepts that were stated in the text (Understanding Cryptography) are discussed and practiced with instructor. Concepts were further dissected and discussed like proving Fermat's Little Theorem was broken down to modular properties and integer ring ideas. Research and programming "plain old RSA" in Python. Outside of this course there will be time for further implement ion on a standard RSA cryptosystem and simulate an active attack on such system.

References

- Christof Paar: Understanding Cryptography A Textbook for Students and Practitioners

- Digital Signature standard
  http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf#page=80