How does RSA work, and how secure is it?

Mathematics

2585 words

gzn364

Abstract

In this essay, I investigate the mathematics behind the RSA cryptosystem - how RSA works, and how secure it is. I describe the basic operation of the RSA algorithm, and give a full proof of its ability to perform public-key encryption. Next, I detail vulnerabilities and viable attacks that have been discovered in RSA. Finally, I evaluate the security and practicality of the modern implementation of RSA, in light of future computing technologies and alternative cryptographic algorithms.

Contents

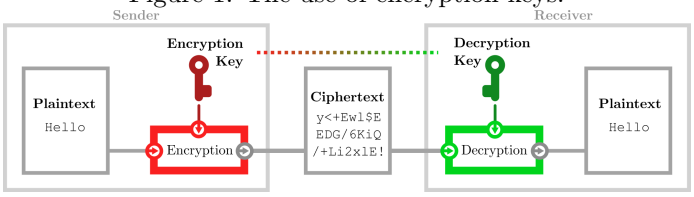
1 Introduction

Cryptography is the study of methods to communicate securely in the presence of adversarial third parties [1]. Secure communication can be achieved through encryption and decryption - pairs of algorithms that encrypt a plaintext message to unintelligible ciphertext, such that only the intended recipient can extract the original message.

Many classical methods of encryption relied upon the secrecy of the cryptographic algorithms. This is not ideal; concealing the algorithm itself from an attacker can be difficult, and provides little definitive protection from attack.

Instead, modern cryptosystems are designed to be secure even when the attacker understands the system completely. This is known as Kerckhoffs's principle, and can be achieved through the use of keys - strings of data that each provide a unique encryption and decryption procedure. Figure 1 (below) demonstrates how encryption and decryption keys control the encryption and decryption operations, respectively:

Figure 1: The use of encryption keys.



In a *symmetric* key encryption system, the encryption key and decryption key are identical. This presents the problem of key distribution: the secret key must first be agreed by both parties before encrypted communication can begin. Exchanging keys either requires a secure communication channel (which is impractical without encryption) or some kind of physical key exchange. Until the 1970s, all encryption used symmetric keys.

An *asymmetric* key (or public key) cryptosystem provides a method for secure communication without a shared key. Instead, two different keys are used for encryption and decryption. The two keys should be related such that the encryption procedure, E, and the decryption procedure, D, are inverse operations:

$$D(E(M)) = M, \tag{1}$$

$$E(D(M)) = M, \tag{2}$$

for some message, M. [2]

However, an attacker should not be able to easily compute the decryption key from the publicly distributed encryption key. Hence, anyone can send the

user an encrypted message, and only the user (as the owner of the decryption key) can decrypt it (see equation (1), above).

The RSA algorithm (named after its inventors: Rivest, Shamir, and Adleman) provides a mathematical method for public key encryption that satisfies these theoretical requirements. Its asymmetry comes from the difficulty in factoring the product of two large random prime numbers.

RSA has been widely adopted for secure online communication. It is often used to establish a shared symmetric key, in order to initiate encrypted data transfer; symmetric key encryption algorithms like AES (Advanced Encryption Standard) allow for far faster encryption.

RSA is also used for generating digital signatures. Note, that while (1) can be used to send encrypted messages, (2) allows the user to verify their identity. This is because whereas only they know the decryption procedure, anyone can perform the encryption procedure because it is publicly distributed. Thus, the user can sign a message, M, by decrypting it, and anyone can verify that signature by encrypting it, and comparing the result to the original message.

Versions of RSA are still in use today, forty years after the algorithm was invented. However, several more modern encryption methods have since been developed. In this essay, I discuss how RSA works, how secure it is, and whether it has a place in the future of cryptography.

2 The RSA Algorithm

RSA public and private keys each consist of two positive integers: an exponent (e and d for encryption and decryption, respectively) and the modulus, n. The public encryption key is therefore (e, n), and the private decryption key is (d, n). The modulus, n, appears in both the public and private keys, as it is necessary for both encryption and decryption [2].

To encrypt a message, it must first be represented as an integer, M, such that $0 \leq M < n$. Then, M can be encrypted by raising it to the e^{th} power modulo n, resulting in ciphertext, C, where

$$C \equiv M^e \mod n.$$

The decryption procedure is the same, but for the use of d as the exponent. C is raised to the d^{th} power modulo n, to recover the message, M,

$$M \equiv C^d \mod n.$$

Each user, U, publicly distributes their unique encryption key (e_U, n_U), whilst keeping their decryption key (d_U, n_U) strictly secret.

2.1 Choosing Keys

The key components, e, d, n must be chosen carefully, to allow for proper encryption and decryption. This should be done as follows.

Firstly, two large prime numbers, p and q, are chosen at random. A larger integer, n, is then computed as the product of p and q:

$$n = p \cdot q.$$

Although n will be made public, its factors (p and q) will be kept secret, thus hiding any easy method for finding d from e.

Next, randomly select the decryption exponent[1], d, such that d is a large random integer that is coprime to $(p-1)(q-1)$:

$$\gcd(d, (p-1)(q-1)) = 1.$$

Finally, compute e as the multiplicative inverse of d modulo $(p-1)(q-1)$. That is, e, where

$$e \cdot d \equiv 1 \mod (p-1)(q-1).$$

See Appendix B for an example of an algorithm for calculating e.

2.2 Euler's Totient

Euler's totient, $\phi(N)$, is defined as the number of positive integers less than some positive integer, N, that are coprime to N. For example, the positive integers less than 8, with numbers coprime to 8 in bold, are: ($\underline{\mathbf{1}}, 2, \underline{\mathbf{3}}, 4, \underline{\mathbf{5}}, 6, \underline{\mathbf{7}}$). Therefore, $\phi(8) = 4$.

Euler's totient is fundamental to RSA. Euler's totient *theorem* - that for any two coprime integers, a and N, a raised to the power of $\phi(N)$ is congruent to 1 modulo N - is particularly useful:

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

It implies that for any positive integer message, M, that is coprime to the modulus, n, M raised to the power of $\phi(n)$ is congruent to 1 modulo n:

[1] It does not matter in which order e and d are computed; they are the inverse of each other and are therefore interchangeable as encryption or decryption exponent.

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

Now, some prime number, P, has no factors. Therefore, all positive integers less than P are coprime to P. Hence, the totient of P is equal to one less than P:

$$\phi(P) = P - 1, \quad \text{where } P \text{ is prime.}$$

Also, $\phi(n)$ is a multiplicative function; that is, for any two coprime positive integers, a and b, the totient of their product is equal to the product of their totients:

$$\phi(ab) = \phi(a) \cdot \phi(b).$$

Hence, the totient of the modulus can be calculated as the product of the totients of its two prime factors

$$\begin{aligned} \phi(n) &= \phi(pq) \\ &= \phi(p) \cdot \phi(q) \\ &= (p-1)(q-1). \end{aligned}$$

Now, if d is chosen such that it is coprime to $\phi(n)$, then there exists a multiplicative inverse of d, modulo $\phi(n)$; that number is e:

$$\begin{aligned} e \cdot d &\equiv 1 \mod \phi(n) \\ &\boxed{\equiv 1 \mod (p-1)(q-1).} \end{aligned}$$

2.3 Proof of Correctness

The RSA encryption and decryption processes detailed above, E and D, can be proven to always hold true for (1) and (2), if the keys are chosen appropriately [2][4]. That is, E and D are *inverse permutations* (every possible message or ciphertext is both the ciphertext for one other message, and itself a possible message with a corresponding ciphertext), where

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

If e is the multiplicative inverse of d, modulo $\phi(n)$, then the following is true:

$$\begin{aligned} & ed = 1 + k \cdot \phi(n). \\ \implies M^{ed} &\equiv M^{1+k\cdot\phi(n)} \mod n \\ &= M \cdot (M^{\phi(n)})^k \mod n \\ &= M \cdot 1^k \mod n && \text{iff } M \text{ is coprime to } n. \\ &\boxed{= M \mod n.} \end{aligned}$$

If M is coprime to n, then $M^{ed} \equiv M \mod n$.

However, the proof is not yet complete , as M is not *always* coprime to n. Specifically, M is not coprime to n when M is either of the prime factors of n, p and q:

$$\begin{aligned} M &\in \{p, q\}. \\ \gcd(M, n) &\neq 1. \\ M^{\phi(n)} &\neq 1 \mod n. \end{aligned}$$

The proof be completed using special case of the Chinese Remainder Theorem (for a proof of this case, see Appendix A), that:

$$\begin{aligned} M^{ed} &\equiv M \mod q, \\ M^{ed} &\equiv M \mod p, \\ \implies M^{ed} &\equiv M \mod n. \end{aligned}$$

If $M = p$, then M can be written as:

$$\begin{aligned} M &\equiv 0 \mod p. \\ \implies M^{ed} &\equiv 0 \mod p, \\ \boxed{M^{ed} &\equiv M \mod p.} \end{aligned}$$

Also, $M = p \implies M \neq q$. Hence, gcd $(m, q) = 1$, so,

$$\begin{aligned} M^{\phi(q)} &\equiv 1 \mod q, \\ M^{q-1} &\equiv 1 \mod q. \end{aligned}$$

Now, given $ed \equiv 1 \mod \phi(n)$, then $ed - 1 = k(p-1)(q-1)$:

$$\begin{aligned} ed &\equiv 1 \mod \phi(n). \\ ed &= k \cdot \phi(n) + 1 \\ &= k(p-1)(q-1) + 1, \\ ed - 1 &= k(p-1)(q-1). \end{aligned}$$

Hence,

$$\begin{aligned} M^{ed} &\equiv M \cdot M^{ed-1} \mod q \\ &= M \cdot M^{k(p-1)(q-1)} \\ &= M \cdot (M^{q-1})^{k(p-1)} \\ &= M \cdot (1)^{k(p-1)} \\ \boxed{&= M \mod q.} \end{aligned}$$

If $M = p$, then:

$$\begin{aligned} M^{ed} &\equiv M \mod p, \\ M^{ed} &\equiv M \mod q, \\ \implies \boxed{M^{ed} &\equiv M \mod n.} \end{aligned}$$

If $M = q$, then the above argument holds true, by substituting p for q. Therefore,

$$\begin{aligned} &\boxed{M^{ed} \equiv M \mod n,} \\ &\forall M \in \mathbb{Z}, \quad 0 \leq M < n. \end{aligned}$$

Hence, E and D, are inverse permutations, and satisfy (1) and (2).

3 RSA in Practice

What I describe in Section 2 is known as *"text-book"* RSA - as opposed to deployed RSA. Deployed versions of RSA are practical implementations of the algorithm, which must mitigate several vulnerabilities in its basic form (see Section 4 - Attacks on RSA), while increasing its efficiency.

For example, the public encryption exponent, e, need not be a large random number, as it is distributed publicly, and therefore need not be resistant to guessing. Therefore, it is common for e to be chosen from a list of small, suitable primes: $3, 5, 17, 257, 65537$. The other key values, n, p, q, d, are then randomly generated based on this value of e.

Using a small value for e drastically improves the speed with which encryption, $M^e \mod n$, can be calculated. It does not, however, compromise security. On the other hand, using a small decryption exponent does compromise security.[3]

The original RSA paper [2] recommends several optimisations, including using the Euclidean algorithm for determining e as the modular multiplicative inverse of d modulo n. I developed an efficient algorithm for calculating the modular inverses, for my Python program (see Appendices B and C).

4 Attacks on RSA

There are no known attacks on RSA that can *"break"* the algorithm (they provide no definitive method for decrypting any given RSA ciphertext). However, RSA has several inherent vulnerabilities which may be exploited by an attacker if not dealt with sufficiently when implementing the algorithm.[3].

4.1 Computing $\phi(n)$, without factorising n

If an attacker discovered a method for computing $\phi(n)$, they would be able to compute the decryption exponent, d, for any public key value; RSA would be broken[2].

However, computing $\phi(n)$ enables easy factorisation of n. Therefore, this method of attack should be no easier than factorising n. The modulus, n, can be factored given $\phi(n)$ as follows.

Firstly, $p + q$ can be computed in terms of $\phi(n)$ and n.

$$\begin{aligned}\phi(n) &= (p-1)(q-1)\\ &= pq - p - q + 1\\ &= n - (p+q) + 1.\\ p + q &= n - \phi(n) + 1.\end{aligned}$$

Now, $p - q$ can be found as the square root of $(p+q)^2 - 4n$:

$$\begin{aligned}(p-q)^2 &= p^2 - 2pq + q^2.\\ (p+q)^2 &= p^2 + 2pq + q^2.\\ \implies (p-q)^2 &= (p+q)^2 - 4pq.\\ p - q &= \sqrt{(p+q)^2 - 4pq}.\end{aligned}$$

Finally, q can be computed as half the difference between $p + q$ and $p - q$:

$$\text{Let,} \quad a = p + q, \quad b = p - q.$$
$$\text{Then,}$$

$$\begin{aligned} a - b &= p + q - (p - q) \\ &= 2q. \end{aligned}$$

$$q = \frac{a - b}{2}.$$

$$\boxed{p = \frac{n}{q}.}$$

Hence, n can be factorised given $\phi(n)$ and n. Therefore, any attack by computing $\phi(n)$ should be no easier than factorising n.

4.2 Common modulus attack

When implementing RSA, it might be tempting to use a fixed modulus value, n, such that multiple people use the same modulus, with different keys. Some central authority would generate n, and then assign public and private keys to each user, generated from that n value.[3]

This would seem to work, if each user keeps their assigned private key secret. However, any user assigned keys generated from n could thereby factorise n, enabling them to determine the private key of any other user using the same modulus, n.

Hence, RSA becomes insecure if the same modulus is (knowingly) used by multiple users.

4.3 Blinding attack

The RSA cryptosystem can be used for signing data, as well as for encryption. However, this mechanism is susceptible to attack if not implemented correctly. One example of an attack on the signing mechanism is a *blinding* attack.[3]

Bob uses public key (e, n) and private key (d, n). Marvin, an adversary, wishes to obtain Bob's signature for some message, M, where

$$\text{Bob's signature,} \quad S \equiv M^d \mod n.$$

Bob refuses to sign M, as this would be sufficient proof that Bob sent the message (he did not).

Marvin now chooses a random integer r, $0 \leq r < n$. He generates a new message, M', where

$$M' \equiv r^e \cdot M \mod n.$$

Bob may now be convinced to sign message M', as it appears to be innocent. He signs it with his corresponding signature, S',

$$S' \equiv (M')^d \mod n.$$

Unfortunately for Bob, Marvin can now obtain Bob's signature, S, for message M:

$$\begin{aligned}
\frac{S'}{r} &\equiv \frac{1}{r}(M')^d \mod n \\
&= \frac{1}{r}(r^e \cdot M)^d \mod n \\
&= \frac{r^{ed}}{r} \cdot M^d \mod n \\
&\equiv \frac{r}{r} \cdot M^d \mod n \qquad \textbf{Note,} \quad r^{ed} \equiv r \mod n. \\
&= M^d \mod n \\
&\boxed{\equiv S \mod n.}
\end{aligned}$$

This technique enables Marvin to obtain Bob's signature for a message, M, of his choosing, by asking Bob to sign a random 'blinded' message, M'. Note that Bob has no information about message M.

Blinding attacks are easily averted, by applying a hash function, H, to M prior to signing. Therefore Marvin would only be able to obtain a signature for $H(M')$, which could not be used (as above) to derive a signature for M.

Interestingly, blinding can be a useful property of RSA signing. It enables Bob to sign the message M without having to see M itself. This can be useful for implementing anonymous digital cash system.

5 Conclusion

The mathematics behind the RSA cryptosystem is beautifully simple. Encryption and decryption are achieved through the same basic function: raising the message (or ciphertext) to the power of the relevant key, and reducing the result modulo n. Further, it is incredible that such fundamental mathematics has found widespread application in modern technology.

In this essay, I have described that mathematics, to answer how *and* why RSA works. I have also probed the security of the system.

The RSA cryptosystem has not been 'broken' - no method has been found to compute d, the e^{th} root of C modulo n, given (e, n, C). I have studied several methods of attacking RSA, and although all of them could allow an attacker to decrypt messages encrypted with RSA if the attacks are not mitigated in the deployed algorithm, none of them could not be easily solved by slight modification to the algorithm.

That RSA has survived for so long is testament to its security. Although I have not covered all possible attacks, all of the sources I found indicated that RSA was still completely secure.

That said, that RSA cannot be broken does not mean it necessarily should be used. Cryptographic technology has advanced significantly since the 1970s; new types of asymmetric key cryptography are available. Elliptic curve cryptography, for example, provides a much higher level of security than RSA for the same key sizes - a typical ECC key is just 255 bits long.

References

[1] Fred C. Piper and Sean Murphy. *Cryptography.* A Very Short Introduction. Oxford University Press, 2002. ISBN: 978-0-19-280315-3.

[2] R. L. Rivest, A. Shamir and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a606588.pdf (visited on 03/12/2019).

[3] D. Boneh. "Twenty Years of Attacks on the RSA Cryptosystem". In: *Notices of the AMS* 46.2 (1999), pp. 203–213. URL: https://www.ams.org/notices/199902/boneh.pdf (visited on 10/12/2019).

[4] P. Meelu and S. Malik. "RSA and its Correctness through Modular Arithmetic". In: *AIP Conference Proceedings* 1324.1 (2010), pp. 463–466. URL: https://doi.org/10.1063/1.3526259 (visited on 24/12/2019).

[5] Burt Kaliski. "The Mathematics of the RSA Public-Key Cryptosystem". In: *Math Awareness Month, AMS* (2006). URL: http://www.mathaware.org/mam/06/Kaliski.pdf (visited on 14/11/2019).

[6] David Ireland. *RSA Algorithm.* DI Management Services, Australia. 2018. URL: https://www.di-mgt.com.au/rsa_alg.html (visited on 02/12/2019).

Appendices

A Proof of Chinese Remainder Theorem Case

In proving the correctness of the RSA algorithm in the case that M is not coprime to p and q, the following case of the Chinese Remainder theorem was used:

$$\begin{aligned} x &\equiv y \mod p, \\ x &\equiv y \mod q, \\ \implies x &\equiv y \mod pq. \end{aligned}$$

This can be proved as follows [4]:

$$\begin{aligned} x &\equiv y \mod p. \\ \implies x &= y + kp. \\ x - y &= kp. \\ \implies p &\mid x - y. \end{aligned}$$

Similarly,

$$q \mid x - y.$$

Now, because p and q are coprime,

$$\begin{aligned} pq &\mid x - y. \\ x - y &= t(pq). \\ \boxed{\implies x &\equiv y \mod pq.} \end{aligned}$$

B Euclidean Algorithm

The decryption exponent, d is found as the modular multiplicative inverse of e,

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

and can be calculated using the Extended Euclidean algorithm.

For example, if $e = 157$, and $\varphi(n) = 776$:

$$\begin{aligned} 776 &= 4(157) + 148 \\ 157 &= 1(148) + 9 \\ 148 &= 16(9) + 4 \\ 9 &= 2(4) + 1 \end{aligned}$$

Now, substituting back,

$$\begin{aligned}
1 &= 1(9) - 2(4) \\
1 &= 1(9) - 2(148 - 16(9)) \\
1 &= -2(148) + 33(9) \\
1 &= -2(148) + 33(157 - 1(148)) \\
1 &= 33(157) + -35(148) \\
1 &= 33(157) + -35(776 - 4(157)) \\
1 &= -35(776) + 173(157)) \\
1 &= 173e - 35\varphi(n)
\end{aligned}$$

So, $m = -35$, $d = 173$.

This algorithm involves using algebra to simplify the expression at each step; e.g. expanding brackets and collecting like terms. This algebra is easy for a human, but hard to program such that a computer could execute it.

Instead of programming the computer to do the same as a human would, I created a recursive Python function that achieved the same result (see Appendix C.3: `number.euclidean`). The following is an explanation of that function.

Each step before substitution can be written as:

$$\begin{gathered}
a = mb + c, \\
\text{or,} \\
a \equiv c \mod b.
\end{gathered}$$

For example, at the first iteration, a and b are the two input values:

$$\begin{aligned}
ed &= 1 \mod \varphi(n). \\
\varphi(n) &= m(e) + c. \qquad \text{(where, } m = d\text{)}
\end{aligned}$$

Specifically, the input must be $a = \varphi(n)$ and $b = e$, as a must be bigger than b. Now, m and c can be calculated:

```
m = a // b
c = a - m * b
```

Here, `a//b` denotes a divided by b and rounded *down* to the next greatest integer.

Consecutive iterations of the Euclidean algorithm can be denoted:

$$\begin{aligned} a &= m_1 b + c \\ b &= m_2 c + d \\ c &= m_3 d + e \\ &\vdots \end{aligned}$$

This arrangement demonstrates the essence of the algorithm:

- A number a is divided by a smaller number b, and the remainder is calculated, c.
- For the next iteration, the process is repeated but with $a = b$ and $b = c$, to find a new remainder, d.
- This process is repeated until a remainder of 1 is reached ($c = 1$).

The algorithm so far can be written as the following recursive function:

```
def euclidean(a, b):

    m = a // b
    c = a - (m * b)

    if c == 1:
        return

    else:
        euclidean(b,c)
        return
```

' This function does not return anything yet, it just creates the equation list before *substitution* - the second part of the Extended Euclidean algorithm. Substitution is done once a remainder of 1 is found. Lets say this remainder is found after four iterations. Then,

$$\begin{aligned} c &= m_3 d + e \\ d &= m_4 e + 1 \end{aligned}$$

Now,

$$\begin{aligned} 1 &= d - m_4 e \\ &= d - m_4 (c - m_3 d) \\ &\vdots \end{aligned}$$

Each step in the substitution can therefore be represented as:

$$1 = Q(x_{n+1}) + P(x_n),$$

where x_n is the next value to be substituted, x_{n+1} the value to be substituted after 1 iteration, x_{n+2} the value to be substituent after 2 iterations, etc.

For example, in the example above, e is substituted first, then d, then c. Therefore, $x_n = e$, $x_{n+1} = d$, and $x_{n+2} = c$.

If, $1 = Q(x_{n+1}) + P(x_n)$, is the result from the previous iteration of substitution, then the equation,

$$x_{n+2} = m(x_{n+1}) + xn,$$

was derived from the current iteration. Written in terms of a, b, and c, as it is in the program:

$$1 = Q(b) + P(c).$$
$$a = mb + c.$$

Clearly, then, it is not necessary to pass the values of b and c as parameters through the iterations as these values cascade through the recursive structure, and so are known inherently. Instead, the coefficients, Q and P, should be returned. When the remainder of one is found, and the recursion tree is terminated:

$$c = a - mb, \qquad (\text{c} = 1)$$
$$1 = a - mb.$$
$$1 = Q(a) + P(b). \qquad (\text{where } Q = 1,\ P = -m)$$

Then,

$$\begin{aligned} 1 &= Q(b) + P(c) \\ &= Q(b) + P(a - mb) \\ &= Q(b) + P(a) - Pm(b) \\ &\boxed{= P(a) + (Q - Pm)(b)} \end{aligned}$$

$$= Q'(a) + P'(b).$$

Where,

$$Q' = P$$
$$P' = Q - Pm$$

In this way, the algorithm can calculated consecutive pairs of coefficients without needing the ability to adaptively re-arrange the expression (like a human).

Instead, once the 1-remainder is found, the coefficients for $1 = Q(a) + P(b)$ can be solved for all iterations, all the way up to the first, hence solving the original equation. For the final function, implementing this method, see Appendix C.3: `number.euclidean`.

C Python program

To help me understand how RSA works in practice, I wrote the following script based on what I had learned. The program is written in Python 3.

C.1 Main Program

```
#rsa.py
import math, random, number

def genD(e,t):
    '''
    Generates decryption exponent, d
    from encryption exponent, e, and t = phi(n)
    '''
    #   k * phi(n) + 1 = e * d
    #   e * d = 1 mod phi(n)
    #  [d = 1/e mod phi(n) ; dodgy notation, 1/e fraction]
    #   d = inverse(e) mod phi(n)

    return number.mod_inv(e,t) % t

def genE(t):
    '''
    Randomly selects encryption exponent, e, such that:
        e coprime with phi(n)
        1 < e < phi(n)
    '''
    while True:
        c = random.randint(2,t)
        if number.coPrime(c,t):
            return c

def genKeys(l, u):
    '''
    Generate RSA keys, such that p,q within bounds l,u (inclusive)
    '''
    p, q = RandomPrime(l,u), RandomPrime(l,u)
    n = p * q
    t = number.totient(p,q) # t = phi(n), Euler Totient Function
    e = genE(t)
    d = genD(e,t)
    return (e,d,n)
```

```
37
38 def RandomPrime (lower, upper):
39     '''
40     Generates random prime number between bounds (inclusive)
41     '''
42     #efficient up to around 10 decimal places
43     if upper-lower <= 1:
44         return None
45
46     while True:
47         c = random.randint(lower,upper) #candidate
48
49         if (number.isPrime(c)):
50             return c
51
52 def encrypt (m, e, n):
53     return (m ** e) % n
54
55 def decrypt (c, d, n):
56     return (c ** d) % n
57
58 if __name__ == '__main__':
59
60     #Alice = Reciever, Bob = Sender
61     '''ENCRYPTION'''
62
63     #Alice generates keys
64     e,d,n = genKeys(100,9999)
65     print('public : ',e,n)
66     print('private: ',d,n)
67
68     #Bob; public key = (e,n)
69     m = 126 #message Bob wants to send Alice. Encrypts using Alice's
       public key
70     c = encrypt(m, e, n) #ciphertext, Bob transmits
71
72     print('\nmessage =', m)
73     print('ciphertext =', c)
74
75     #Alice; private key = (d,n)
76     p = decrypt(c, d, n) #Alice recieves ciphertext and decrypts using
       her private key
77     print('plaintext  =', p)
78
79
80     '''SIGNING'''
81     #Alice: uses same keys (e,d,n)
82     data = 986 #Alice wishes to sign data to verify that she sent it
83     s = decrypt(data, d, n) #signature; Alice signs using her private
       key, (d, n)
84     #signiture, s, is transmitted along with data
85
86     print('\ndata =', data)
87     print('signature =', s)
88
89     #Bob recieves data & signature
```

```
    v = encrypt(s, e, n) #Bob encrypts s, to p (using Alice's public
    key)
    #if (v==data), Bob knows only Alice could have sent the data, and
    that it has not been altered
    print('verification =', v)
```

C.2 Number Theory Module

I separated any functions for number theory - including the modular multiplicative inverse calculation - into a separate script.

```
#number.py
import math

def mod_inv(a, b):
    """
    Returns x, solution to
    [ax = 1 mod b]
    , using Euclidean algorithm
    """
    # => kb + ax = 1

    k,x = euclidean(b,a)

    return x

def euclidean(a, b):
    '''
    Euclidean algorithm for finding modular inverse in RSA

    a > b
    a coprime b

    See euclidean.py for further explanation
    '''
    #   a = c mod b
    #   a = mb + c

    m = a // b
    c = a - m * b

    if (c == 1):

        #   a = mb + 1
        #   1 = a - mb
        #   1 = Q(a) + P(b)
        #   return (Q,P)

        return (1, -m)

    else:
        Q,P = euclidean (b, c)

        return (P, Q - P * m)

def totient(p,q):
    '''
```

```
    computes Euler totient function, phi(n)
    where n = pq, p & q prime
    '''

    return (p-1) * (q-1)

def isPrime(n):
    '''
    Checks primality exhaustively
    '''

    if (n % 2) == 0:
        return False

    target = int(math.sqrt(n)) + 1
    for i in range(3, target):
        if (n % i) == 0:
            return False

    return True

def coPrime(a,b):
    return math.gcd(a,b) == 1

if __name__ == '__main__':
    print(euclidean(776,157))
    print(euclidean(157,73))
```

C.3 Example Output

```
public :  5133463 22004467
private:  4659367 22004467

message = 126
ciphertext = 14174587
plaintext  = 126

data = 986
signature = 13743726
verification = 986
```