RSA Cryptosystem

December 1, 2019

Robert Vincent Caldwell

In 1977, at the Massachusetts Institute of Technology, Ron Rivest, Adi Shamir, and Leonard Adleman formulated the algorithm now known as RSA – each initial representing their surnames in the same order they appeared in their published paper. During the development process, Rivest and Shamir would develop encryption schemes and Adleman would crack them. Eventually, Rivest had a stroke of inspiration and worked through the night to produce a one-way function that was different from the Diffie-Hellman protocol and exponentiation (Weakdh.org, 2019).

The subsequent protocol conforms to what Rivest developed after returning home from a Passover seder:

Alice and Bob want to send secret messages to each other. However, they also know that Eve is lingering in the shadows, waiting to intercept their communications. So cleverly, Alice tells Bob that they should use a public-key to encrypt messages – that anyone, including Eve, can have – and, a private-key that is kept secret and only used to decrypt the messages they receive from each other. These will be their asymmetric keys.

Unbeknownst to Eve, Alice chooses two relatively large prime numbers p and q. Then, Alice multiplies p and q together in order to produce the composite modulus, n. Ideally, the value of n should be between 1024 – 4096-bits – with the preference being above the threshold of 2048-bits. As of 2015, this should be 600-digits or more and produced by the combination of 300-digit p and q values (Rosen, 2019).

Alice sends Bob the composite modulus value n. Bob can now produce a public-key. He begins by choosing an encryption key, e, such that $\phi(n) = \phi(pq) = (p-1)(q-1)$ and the greatest common divisor of e and $\phi(n)$ equals 1, or $gcd(e, \phi(n)) = 1$.

Once Bob has his public-key pair e and n, he may then produce his private-key pair d and n; where d$\equiv$1(mod(p-1)) and $e \cdot d \equiv$1(mod(q-1)). Bob sends Alice his public-key (e, n) while keeping his private-key pair (d, n) secret.

Once Alice obtains Bob's public-key, Alice may proceed to encrypt messages send them to Bob without worrying whether Eve intercepts them or not. This is because the RSA cryptosystem complies with Kerckhoffs's axiom. The axiom, stated by the 19th-century Netherlands born cryptographer Auguste Kerckhoffs, states that a cryptosystem should remain secure even if everything about the system becomes public knowledge, with exception to the key value(s) used

(Sciencedirect.com, 2019).

Alice encrypts a plaintext message into cyphertext by computing c = m·e(mod n); where c is the resulting cyphertext, m is the plaintext message, and e, n is the public-key values. Alice then sends the cyphertext message, c, to Bob, who deciphers the ciphertext message as m = c·d(mod n).

Example of key generation, written in Python:

[1]:
```
'''
Generate Public and Private-Keys
'''
def keys(p, q, e):
    n, d = (p * q), modinv(e, find_modulus(p, q))

    return e, d, n
```

In general, RSA is relatively slow compared to other encryption methods and is therefore primarily used for encrypting key values in symmetric cryptosystems. RSA is an asymmetric encryption method. Asymmetric encryption uses two different values for encrypting and decrypting data. Whereas, symmetric key encryption, also known as private-key encryption, uses the same key to encrypt and decrypt the data. Overall, private-key systems make the process of encryption faster. However, notably, the encryption itself is less secure.

To begin developing an RSA cryptosystem we must begin with the foundational concept, modular arithmetic. Modular arithmetic plays a central role at the heart of many cryptographic ciphers (Rosen, 2019). While the idea may sound daunting, it is quite simple. To illustrate just how simple, we may look no further than an analog clock.

By counting the hours starting at midnight we get 12:00 am, 1:00 am, 2:00 am, ... , 10:00 am, 11:00 am, and when we reach noon the time "switches" from am to pm; then, the counting starts again, 12:00 pm, 1:00 pm, 2:00 pm, and so forth. Therefore, performing addition and multiplication on the set of integers, $\mathbb{Z}_m$, from 0 to m-1 and reducing each sum or product (mod m) is called modular arithmetic.

For example, in $\mathbb{Z}_5$ - where $\mathbb{Z}_5$ means (mod 5), $4+3=2$, $(4+3)(\text{mod } 5) = 2 \rightarrow (7)(\text{mod } 5) = 2$; 5 goes into 7 one time with a remainder of 2.

Generalizing the explanation, we can prove:

Given d, d$\neq$ 0, and $a \geq$0, we compute numbers q_n and r_n such that satisfy $a = q_n d + r_n$; where $r_n \geq 0$. q_n is the provisional quotient and r_n is the corresponding provisional remainder.

We define q_0=0 and r_0=a. Then, $a = q_0 d + r = 0(d) + a$. Now, assuming q_n and r_n satisfies $a = q_n d + r_n$, we define q_{n+1} and r_{n+1}:

case 1: $0 \leq r_n < d$. In that case, r_n is the remainder and q_n is the quotient. There is no q_{n+1} or r_{n+1} in this case.

case 2: $r_n \leq d$. In that case, r_n is too large. So, we can move a d: $a = q_n d + r_n = (q_n + 1)d + (r_n - d)$.

Therefore, we define $q_{n+1} = q_n + 1$ and $r_{n+1} = r_n - d$. Then, $a = q_{n+1}d + r_{n+1}$. Since $r_n \geq d$, $r_{n+1} \geq 0$. If a<0, then we need to add d to r_n in each step and decrease q_n by 1. Since the initial provisional remainder is negative, this operation is done while $r_n < 0$ (Rosen, 2019).

However, of the mathematics necessary to develop the RSA cryptosystem, the Extended Euclidean Algorithm is arguably the most important concept. The Extended Euclidean Algorithm also serves as a foundation for every other required operation (Rosen, 2019).

Essentially, the Euclidean algorithm is the continuous application of the division algorithm for integers and is particularly useful when a and b are coprime. For example, if a and b are arbitrary non-negative integers, then there exist unique non-negative integers q and r such that:

$$A = qb + r; \text{ where } 0 \leq r < b \text{ and } a \neq b$$

So, if we repeatedly divide the divisor by the remainder until the remainder reaches 0, the last non-zero remainder is the Greatest Common Divisor.

As such, finding the $gcd(52, 28)$ using the Euclidean Algorithm:

$$52 = 28\textit{1+24 } \textit{28} = \textit{24}1 + 424 = 4 \cdot 6 + 0$$

Thus, the $gcd(52, 28) = 4$, and the linear representation, known as the Bézout's identity, is $52x + 28y = 0$; where x = 7 and y = -13.

Example of an Extended Euclidean Algorithm, written in Python:

[2]:
```
'''
Extended Euclid's algorithm for determining the greatest common divisor for␣
 ↪large numbers
'''
def egcd(a, b):
    x, y, u, v = 0, 1, 1, 0
    while a != 0:
        q, r = b // a, b % a
        m, n = x - u * q, y - v * q
        b, a, x, y, u, v = a, r, u, v, m, n
    gcd = b
    return gcd, x, y
```

Furthermore, the integer x from $ax + by = gcd(a, b)$ is the modular multiplicative inverse. To find the modular multiplicative inverse, we simply need to reverse the steps of the Extended Euclidean Algorithm and recursively work backwards (Rosen, 2019).

French mathematician Étienne Bézout's identity, known as Bézout's lemma, can be used to prove the existence of the modular multiplicative inverse used in the completion of the RSA cryptosystem (Rosen, 2019). The theorem states, for any arbitrary non-zero integers a and b, the greatest common divisor can be represented by $d = gcd(a, b)$. Then, there exists integers x and y, such that:

$$\text{ax + by = d}$$

Thus, we can prove relatively prime integers, $gcd(a, b) = 1$ in the form $ax + by = 1$ for integers a, b, x, and y. In order to find the values for x and y, we can apply the Euclidean algorithm to calculate $gcd(a, b)$. For example, if we wish to find $gcd(2056, 511) = 1$, then:

$$\begin{gathered} 2056 = 511 \cdot 4 + 12 \\ 511 = 12 \cdot 42 + 7 \\ 12 = 7 \cdot 1 + 5 \\ 7 = 5 \cdot 1 + 2 \\ 5 = 2 \cdot 2 + 1 \end{gathered}$$

Using back substitution gives us:

$$\begin{gathered} 5 - 2 \cdot 2 \\ 5 - (7 - 5 \cdot 1) \cdot 2 \\ 5 \cdot 3 - 7 \cdot 2 \\ (12 - 7 \cdot 1) \cdot 3 - 7 \cdot 2 \\ 12 \cdot 3 - 7 \cdot 5 \\ 12 \cdot 3 - (511 - 12 \cdot 42) \cdot 5 \\ 12 \cdot 213 - 511 \cdot 5 \\ (2056 - 511 \cdot 4) \cdot 213 - 511 \cdot 5 \\ 2056 \cdot 213 - 511 \cdot 857 \end{gathered}$$

Therefore, a = 2056, b = 511, x = 213, y = -857.

Accordingly, based on the previous example, we can prove that if a and b are integers such that $gcd(a, b) = 1$, then there exists a modular multiplicative inverse x such that $ax \equiv 1 (\text{mod b})$. So, since $gcd(2056, 511) = 1$, then:

$1 \equiv 2056x + 511y \equiv 2056x (\text{mod } 511)$; where the modular multiplicative inverse is equal to 213.

By definition, a multiplicative inverse exists if and only if a and b are relatively prime. In number theory, two integers a and b are said to be relatively prime or coprime if the only positive integer that divides both of them is 1 (Rosen, 2019). Consequently, any prime number that divides one does not divide the other; this principal is derived from the fundamental theorem of arithmetic – otherwise known as the unique factorization theorem. From which, the theorem states that every integer greater than 1 either is prime or is the product of a unique combination of prime numbers. This is then equivalent to their greatest common divisor being 1.

Example of an Modular Multiplicative Inverse, written in Python:

[3]:
```
'''
Finds the modular inverse -
This is specifically used to find the decryption key: d = (phi*i + 1)/e
```

```
'''
def modinv(e, n):
    gcd, x, y = egcd(e, n)
    if gcd != 1:
        return None  # The modular inverse does not exist in this case
    else:
        return x % n # Else, the modular inverse exists and is returned
```

Besides being used to compute the Greatest Common Divisor of some arbitrary integers a, b such that $ax + by = gcd(a, b)$, the Extended Euclidean Algorithm is also the reciprocal of modular exponentiation (Rosen, 2019).

Fast modular exponentiation is necessary in order to use large prime numbers and execute primality tests. Without fast modular exponentiation, computing a^n(mod m) for large integers n becomes quickly impractical, if not impossible. Executing the RSA encryption, computing large integers n becomes increasingly important. Otherwise, generating public and private-key values turns out to be an unrealistic endeavor. Therefore, to compute a^n(mod m) we can use $n = 2^k$. For example, to quickly calculate a^{341}(mod 7) we can substitute 341 with $2^8, 2^6, 2^4, 2^2, 2^0$, or:

$$[((2^8)(\text{mod } 7)) + ((2^6)(\text{mod } 7)) + ((2^4)(\text{mod} 7)) + ((2^2)(\text{mod } 7)) + ((2^0)(\text{mod } 7))](\text{mod } 7) = 5$$

One of the last operations used in developing the RSA encryption is Euler's totient function. The function counts the total number of non-negative integers up to a given integer n that are relatively prime, or coprime, to n. Generally, the function is denoted using $\phi(n)$ – colloquially called Euler's phi function. This function is specifically used to compute the decryption key value (Rosen, 2019).

Calculating $\phi(n) = (p-1)(q-1)$, such that $n - \phi(n) + 1 = p + q$. For example, to calculate $\phi(52)$ using a pencil-and-paper technique, we first use the fundamental theorem of arithmetic to factor 52 into $13 \cdot 2 \cdot 2 = 13^1 \cdot 2^2$. We can then list out all the integers between 1 and 52, removing the integers whose unique prime factorization includes either a 2 or 13. This will leave a set containing the values $\{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 41, 43, 45, 47, 49, 51\}$. Thus, $\phi(52) = 24$.

Overall, the strength of the encryption depends on both keeping the $\phi(n)$ value secret and sufficiently large enough to make factorization and solving the discrete logarithm problem impractical. Currently, that means that the key value should be typically between 1024 to 2048-bits long. This translates into prime integers of length 309 to 617-digits. However, ideally, the key value will be between 2048 and 4096-bits long – or 617 to 1234-digits.

Example of an Euler's Totient Function, written in Python:

[4]:
```
'''
Finding the modulus value
'''
def find_modulus(p, q):
    mod_n = (p-1)*(q-1)
    return mod_n
```

In the process of developing an RSA encryption scheme, I first sat down and wrote out the necessary steps in pseudocode. After the pseudocode was finished, I began coding each step in Python, attempting to use the least amount of code and the most efficiency possible. While I was in the process of coding the encryption block, I attempted to locate examples of how other people had finished the same project – spurned on by curiosity regarding different coding patterns. Unfortunately, all the examples I was able to find encrypted each character. The issue with encrypting each character is that the resulting system is nothing more than a fancy substitution cipher. As a substitution cipher, the system becomes vulnerable to several cryptanalysis techniques, including frequency analysis.

So, to create a proper block encryption scheme, I first had to convert each character into its ASCII numerical representation. Then, group each pair of ASCII values into blocks no larger than the composite modular n value, determined when generating our public and private-keys. After these blocks of values were produced, each block is then encrypted using the public-key, e value. If any of the encrypted blocks then contain an uneven number of digits, zeros are added to pad the beginning of each block until all blocks contain the same number of digits. Finally, all the blocks are combined into a single block element. The Python code below illustrates each step and displays the working encryption process.

Example of how RSA block encryption works, written in Python:

[5]:
```
'''
Encryption
'''
def encrypt(keypack, plaintext):
    key, n = keypack

    cipher = [(ord(char)) for char in plaintext]
    # print('Step 1: ', cipher) # Converts each character into its ASCii value

    # Encrypts each block by converting ASCii values based on a^b mod m
    cipher = [str(pow(int(char), key, n)) for char in cipher]

    # The for loop will add leading zeros and buffers as necessary
    for idx in range(len(cipher)):
        cipher[idx] = str(cipher[idx])
        while len(str(cipher[idx])) < (len(str(n))):
            cipher[idx] = '0%s' % cipher[idx]
        if len(cipher) % 2 == 1:
            cipher.append('    ')

    # # print('Step 2:', cipher) # Combines blocks into largest size block␣
↪possible but no larger than n; where n=p*q
    ciphertext = [i + j for i, j in zip(cipher[::2], cipher[1::2])]

    # print('Step 3:', ciphertext) # Combines pairs of n blocks
    ciphertext = ''.join(ciphertext)
```

```
    return ciphertext
```

Once a single encrypted block element is obtained, we will need to decrypt the said block to read the plaintext message contained within. The process of decryption is very similar to encryption, just reversed. However, the major roadblock experienced when creating the decryption was in properly breaking up the encrypted block element. After attempting several alternatives, I was eventually successful. The solution was in separating the code-block into groups of n and then quickly dropping the leading zeros by converting each block from string-values into integer-values and back into strings. This technique produced the same blocks found in step 2 of the above encryption code-block. Therefore, all that remained was to apply the decryption key to each block, separate the blocks into pairs of 2, and then convert the ordinal ASCII value back into its assigned plaintext value. This process is illustrated below:

Example of how RSA block decryption works, written in Python:

[6]:
```
'''
Decryption
'''
def decrypt(keypack, ciphertext):
    key, n = keypack

    # Quick drop of all leading zeros from each of the blocks by converting each␣
 ↪string block into an integer
    # and then back into string values
    decipher = str(int(''.join(ciphertext)))

    # Separates the encrypted string into individual list elements, and then␣
 ↪combines into blocks of size n
    decipher = [str(''.join(idx).strip()) for idx in ciphertext]
    while '' in decipher:
        decipher.remove('')
    decipher = [(ciphertext[i:i + (len(str(n)))]) for i in range(0,␣
 ↪len(ciphertext), (len(str(n))))]
    if decipher[-1].startswith(' '):
        decipher.pop(-1)

    # Decrypts each block by converting ASCii values based on (a^b)^c mod m
    decipher = [str(pow(int(char), key, n)) for char in decipher]

    # Converts each list element from a string to an ASCii integer and then to␣
 ↪its character value
    plaintext = [chr(int(char)) for char in decipher]

    return ''.join(plaintext)
```

Bringing all of the concepts together, below is an example of RSA encryption:

Choose a prime 'p' value: 43

Choose a prime 'q' value: 59

public-key [Use this key for encryption]: (71 , 2537)

private-key [Use this key for decryption]: (995 , 2537)

Encrypted:

0867092220611994187302700270206109222311139620610922209207850922
1897379077523111897039018732311027802930785205420612258187320610
3162311206119942311076909222092203018730278231124340908034422261
1862176231108900724089007240890089007240890231108900724089007240
8900890072407242311089007240890089008900890089008900724

Decrypted: Professor Moriarty, the "Napoleon of crime" LIVES! 01010010 01010011 01000001

Chances are, if you have logged into a server or recently visited a website that performs credit card transactions, you have used RSA encryption in one form or another. However, most likely, this involved your computer using the servers public-key to encrypt the data using a faster symmetric-key cryptosystem which is not only a more efficient but preferred as society expects data to be delivered faster and more secure than it was delivered yesterday. Rivest, Shamir, and Adleman's contribution to cryptography and information security was a significant contribution paving the way for security in the 21st-century. With the advent of quantum computing, RSA encryption may become obsolete. However, until that happens, RSA is here to stay.

Bibliography and References:

Rosen, K. (2019). Discrete mathematics and its applications. 8th ed. New York City: McGraw-Hill, pp.251 - 324.

Sciencedirect.com. (2019). Kerckhoffs Principle - an overview. [online] Available at: https://www.sciencedirect.com/topics/computer-science/kerckhoffs-principle [Accessed 1 Dec. 2019].

Weakdh.org. (2019). Diffie-Hellman key exchange. [online] Available at: https://weakdh.org/ [Accessed 1 Dec. 2019].

Caldwell, R. (2019). Ganymede | Jupyter Cryptex. [online] GitHub. Available at: https://github.com/v-ca/Ganymede [Accessed 1 Dec. 2019].