How RSA Works With Examples

Barry Steyn
barry.steyn@gmail.com

Published: 26 May 2012

Introduction

This is part 1 of a series of two blog posts about RSA (***part 2***[L1] will explain *why* RSA works). In this post, I am going to explain exactly how RSA public key encryption works. One of the ***3 seminal events in cryptography***[L2] of the 20th century, RSA opens the world to a host of various cryptographic protocols (like *digital signatures, cryptographic voting* etc). All discussions on this topic (including this one) are very *mathematical*, but the difference here is that I am going to go out of my way to explain each concept with a concrete example. The reader who only has a beginner level of mathematical knowledge should be able to understand exactly how RSA works after reading this post along with the examples.

PLEASE PLEASE PLEASE: Do not use these examples (specially the real world example) and implement this yourself. What we are talking about in this blog post is actually referred to by cryptographers as *plain old RSA*, and it needs to be randomly padded with ***OAEP***[L3] to make it secure. In fact, you should never ever implement any type of cryptography by yourself, rather use a library. You have been warned!

Background Mathematics

The Set Of Integers Modulo P

The set:

$$\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\} \tag{1}$$

Is called the *set of integers modulo p* (or *mod p* for short). It is a set that contains Integers from 0 up until $p - 1$.

Example: $\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Integer Remainder After Dividing

When we first learned about numbers at school, we had no notion of real numbers, only integers. Therefore we were told that 5 divided by 2 was equal to 2 remainder 1, and not $2\frac{1}{2}$. It turns out that this type of math is vital to RSA, and is one of the reasons that secures RSA. A formal way of stating a remainder after dividing by another number is an equivalence relationship:

$$\forall x, y, z, k \in \mathbb{Z}, x \equiv y \bmod z \iff x = k \cdot z + y \tag{2}$$

Equation ***2*** states that if x is equivalent to the remainder (in this case y) after dividing by an integer (in this case z), then x can be written like so: $x = k \cdot z + y$ where k is an integer.

Example: If $y = 4$ and $z = 10$, then the following values of x will satisfy the above equation: $x = 4, x = 14, x = 24, \ldots$. In fact, there are an infinite amount of values that x can take on to satisfy the above equation (that is why I used the equivalence relationship $\equiv$ instead of equals). Therefore, x can be written like so: $x = k \cdot 10 + 4$, where k can be any of the infinite amount of integers.

There are two important things to note:

1. The remainder y stays constant, whatever value x takes on to satisfy equation ***2*** .
2. Due to the above fact, $y \in \mathbb{Z}_z$ (y is in the set of integers modulo z)

Multiplicative Inverse And The Greatest Common Divisor

A multiplicative inverse for x is a number that when multiplied by x , will equal 1 . The multiplicative inverse of x is written as x^{-1} and is defined as so:

$$x \cdot x^{-1} = 1 \quad \textbf{(3)}$$

The greatest common divisor (gcd) between two numbers is the largest integer that will divide both numbers. For example, $gcd(4, 10) = 2$.

The interesting thing is that if two numbers have a gcd of 1, then the smaller of the two numbers has a multiplicative inverse in the modulo of the larger number. It is expressed in the following equation:

$$x \in \mathbb{Z}_p, x^{-1} \in \mathbb{Z}_p \iff \gcd(x, p) = 1 \quad \textbf{(4)}$$

The above just says that an inverse only exists if the greatest common divisor is 1. An example should set things straight...

Example: Lets work in the set $\mathbb{Z}_9$, then $4 \in \mathbb{Z}_9$ and $gcd(4, 9) = 1$. Therefore 4 has a multiplicative inverse (written 4^{-1}) in $\mod 9$, which is 7 . And indeed, $4 \cdot 7 = 28 = 1 \mod 9$. But not all numbers have inverses. For instance, $3 \in \mathbb{Z}_9$ but 3^{-1} does not exist! This is because $gcd(3, 9) = 3 \neq 1$.

Prime Numbers

Prime[L4] numbers are very important to the RSA algorithm. A prime is a number that can only be divided *without a remainder* by itself and 1 . For example, 5 is a prime number (any other number besides 1 and 5 will result in a remainder after division) while 10 is not a prime[1].

This has an important implication: For any prime number p , every number from 1 up to $p - 1$ has a $\gcd$ of 1 with p , and therefore has a multiplicative inverse in modulo p .

Euler's Totient

Euler's Totient[L6] is the number of elements that have a multiplicative inverse in a set of modulo integers. The totient is denoted using the Greek symbol phi ϕ . From ***4*** above, we can see that the totient is just the count of the number of elements that have their $\gcd$ with the modulus equal to 1. This brings us to an important equation regarding the totient and prime numbers:

$$p \in \mathbb{P}, \phi(p) = p - 1 \tag{5}$$

Example: $\phi(7) = |\{1, 2, 3, 4, 5, 6\}| = 6$ [2].

RSA

With the above background, we have enough tools to describe RSA and show how it works. RSA is actually a set of two algorithms:

1. **Key Generation**: A key generation algorithm.
2. **RSA Function Evaluation**: A function F , that takes as input a point x and a key k and produces either an encrypted result or plaintext, depending on the input and the key.

Key Generation

The key generation algorithm is the most complex part of RSA. The aim of the key generation algorithm is to generate both the *public* and the *private* RSA keys. Sounds simple enough! Unfortunately, weak key generation makes RSA very vulnerable to attack. So it has to be done correctly. Here is what has to happen in order to generate secure RSA keys:

1. **Large Prime Number Generation**: Two large prime numbers p and q need to be generated. These numbers are very large: At least 512 digits, but 1024 digits is considered safe.
2. **Modulus**: From the two large numbers, a modulus n is generated by multiplying p and q .
3. **Totient**: The totient of $n, \phi(n)$ is calculated.

4. **Public Key**: A *prime number* is calculated from the range $[3, \phi(n))$ that has a greatest common divisor of 1 with $\phi(n)$.
5. **Private Key**: Because the prime in step 4 has a gcd of 1 with $\phi(n)$, we are able to determine it's inverse with respect to $\mod \phi(n)$.

After the five steps above, we will have our keys. Lets go over each step.

Large Prime Number Generation

It is vital for RSA security that two very large prime numbers be generated that are quite far apart. Generating composite numbers, or even prime numbers that are close together makes RSA totally insecure.

How does one generate large prime numbers? The answer is to pick a large random number (a very large random number) and test for primeness. If that number fails the prime test, then add 1 and start over again until we have a number that passes a prime test. The problem is now: How do we test a number in order to determine if it is prime?

The answer: An incredibly fast prime number tester called the ***Rabin-Miller primality tester***[L8] is able to accomplish this. Give it a very large number, it is able to very quickly determine with a high probability if its input is prime. But there is a catch (and readers may have spotted the catch in the last sentence): The Rabin-Miller test is a probability test, not a definite test. Given the fact that RSA absolutely relies upon generating large prime numbers, why would anyone want to use a probabilistic test? The answer: With Rabin-Miller, we make the result as accurate as we want. In other words, Rabin-Miller is setup with parameters that produces a result that determines if a number is prime with a probability of our choosing. Normally, the test is performed by iterating 64 times and produces a result on a number that has a $\frac{1}{2^{128}}$ chance of not being prime. The probability of a number passing the Rabin-Miller test and not being prime is so low, that it is okay to use it with RSA. In fact, $\frac{1}{2^{128}}$ is such a small number that I would suspect that nobody would ever get a false positive.

So with Rabin-Miller, we generate two large prime numbers: p and q .

Modulus

Once we have our two prime numbers, we can generate a modulus very easily:

$$n = p \cdot q \tag{6}$$

RSA's main security foundation relies upon the fact that given two large prime numbers, a composite number (in this case n) can very easily be deduced by multiplying the two primes together. But, given just n , there is no known algorithm to efficiently determining n 's prime factors. In fact, it is considered a hard problem. I am going to bold this next statement for effect: **The foundation of RSA's security relies upon the fact that given a composite number, it is considered a hard problem to determine it's prime factors**.

The bold-ed statement above cannot be proved. That is why I used the term *"considered a hard problem"* and not *"is a hard problem"*. This is a little bit disturbing: Basing the security of one of the most used cryptographic atomics on something that is not provably difficult. The only solace one can take is that throughout history, numerous people have tried, but failed to find a solution to this.

Totient

With the prime factors of n , the totient can be very quickly calculated:

$$\phi(n) = (p - 1) \cdot (q - 1) \tag{7}$$

This is directly from equation 5 above. It is derived like so:

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

The reason why the RSA becomes vulnerable if one can determine the prime factors of the modulus is because then one can easily determine the totient.

Public Key

Next, the *public key* is determined. Normally expressed as e , it is a prime number chosen in the range $[3, \phi(n))$. The discerning reader may think that 3 is a little small, and yes, I agree, if 3 is chosen, it could lead to security flaws. So in practice, the public key is normally set at 65537 . Note that because the public key is prime, it has a high chance of a *gcd* equal to 1 with $\phi(n)$. If this is not the case, then we must use another prime number that is *not* 65537 , but this will only occur if 65537 is a factor of $\phi(n)$, something that is quite unlikely, but must still be checked for.

An interesting observation: If in practice, the number above is set at 65537 , then it is not picked at random; surely this is a problem? Actually, no, it isn't. As the name implies, this key is public, and therefore is shared with everyone. As long as the *private key* cannot be deduced from the public key, we are happy. The reason why the public key is not randomly chosen in practice is because it is desirable not to have a large number. This is because it is more efficient to encrypt with smaller numbers than larger numbers.

The public key is actually a key pair of the exponent e and the modulus n and is present as follows

$$(e, n)$$

Private Key

Because the public key has a *gcd* of 1 with $\phi(n)$, the multiplicative inverse of the public key with respect to $\phi(n)$ can be efficiently and quickly determined using the ***Extended Euclidean Algorithm***[L9]. This multiplicative inverse is the *private key*. The common notation for expressing the private key is d . So in effect, we have the following equation (one of the most important equations in RSA):

$$e \cdot d = 1 \bmod \phi(n) \qquad \textbf{(8)}$$

Just like the public key, the private key is also a key pair of the exponent d and modulus n :

$$(d, n)$$

One of the absolute fundamental security assumptions behind RSA is that given a public key, one cannot efficiently determine the private key. I have written a follow up to this post explaining ***why RSA works***[L1], in which I discuss ***why one can't efficiently determine the private key given a public key***[L10].

RSA Function Evaluation

This is the process of transforming a plaintext message into ciphertext, or vice-versa. The RSA function, for message m and key k is evaluated as follows:

$$F(m, k) = m^k \bmod n \tag{9}$$

There are obviously two cases:

1. Encrypting with the *public key*, and then decrypting with the *private key*.
2. Encrypting with the *private key*, and then decrypting with the *public key*.

The two cases above are mirrors. I will explain the first case, the second follows from the first

Encryption: $F(m, e) = m^e \bmod n = c$, where m is the message, e is the public key and c is the cipher.
Decryption: $F(c, d) = c^d \bmod n = m$.

And there you have it: RSA!

Final Example: RSA From Scratch

This is the part that everyone has been waiting for: an example of RSA from the ground up. I am first going to give an academic example, and then a real world example.

Calculation of Modulus And Totient

Lets choose two primes: $p = 11$ and $q = 13$. Hence the modulus is $n = p \times q = 143$. The totient of n $\phi(n) = (p - 1) \cdot (q - 1) = 120$.

Key Generation

For the public key, a random prime number that has a greatest common divisor (gcd) of 1 with $\phi(n)$ and is less than $\phi(n)$ is chosen. Let's choose 7 (note: both 3 and 5 do not have a gcd of 1 with $\phi(n)$. So $e = 7$, and to determine d , the secret key, we need to find the inverse of 7 with $\phi(n)$. This can be done very easily and quickly with the *Extended Euclidean Algorithm*, and hence $d = 103$. This can be easily verified: $e \cdot d = 1 \bmod \phi(n)$ and $7 \cdot 103 = 721 = 1 \bmod 120$.

Encryption/Decryption

Lets choose our plaintext message, m to be 9 :

Encryption:

$$m^e \bmod n = 9^7 \bmod 143 = 48 = c$$

Decryption:

$$c^d \bmod n = 48^{103} \bmod 143 = 9 = m$$

A Real World Example

Now for a real world example, lets encrypt the message "attack at dawn". The first thing that must be done is to convert the message into a numeric format. Each letter is represented by an ascii character, therefore it can be accomplished quite easily. I am not going to dive into converting strings to numbers or vice-versa, but just to note that it can be done very easily. How I will do it here is to convert the string to a bit array, and then the bit array to a large number. This can very easily be reversed to get back the original string given the large number. Using this method, "attack at dawn" becomes

19766202164023008896244827187751 50 (for those interested, ***here***[L11] is the code that I used to make this conversion).

Key Generation

Now to pick two large primes, p and q . These numbers must be random and not too close to each other. Here are the numbers that I generated: using Rabin-Miller primality tests:

p

12131072439211271897323671531612440428472427633701410925634549
31230196437304208561932419736532241686654101705736136521417171
17137979742993348710628298035 41

q

12027524255478748885956220793734512128733387803682075433653899
98395517985098879789986914690080913161115334681705083209602216
01463663463918124709871054152 33

With these two large numbers, we can calculate n and $\phi(n)$

n

14590676800758332323018693934907063529240187237535716439958187
10198734387990053589383695714026701498021218180862924674228281
57022922076746906543401224889672472407926969987100581290103199
31785875366371086235765651050788371429711563734278891146353510
27120327651665184117268598379886721118372050855263466187400 53

$\phi(n)$

14590676800758332323018693934907063529240187237535716439958187
10198734387990053589383695714026701498021218180862924674228281
57022922076746906543401224889648313811232279966317301397777852
36530154784827347887129722205858745715289160645926971811926897
11635550708026439995295496441168119475165139381842966835212 80

e - the public key

65537 has a gcd of 1 with $\phi(n)$, so lets use it as the public key. To calculate the private key, use extended euclidean algorithm to find the multiplicative inverse with respect to $\phi(n)$.

d - the private key

894894250092744443682285459217730939196695860658842574454978544564876748396298183909349419732628796167979706089172836798754993315741611138540888132754881105882471930775825272784379065040156806234235500672400424666656542323835029222154936232894721388664458187891279461234078077257026266440910365023725451397 13

Encryption/Decryption

Encryption: $19766202164023008896244827187751 50^{e} \bmod n$

350521113386730266902124239370533285118807608115799816206428023466858106231098502359430490809733862411137840407947041939782153784997654130836464387847409523069325349451950801838615742252262188798272324539128205968864403775360824656817500744174591514854074458625110234722355608230534977915189288202722577877 86

Decryption:

350521113386730266902124239370533285118807608115799816206428023466858106231098502359430490809733862411137840407947041939782153784997654130836464387847409523069325349451950801838615742252262188798272324539128205968864403775360824656817500744174591514854074458625110234722355608230534977915189288202722577877 86 $^{d} \bmod n$

19766202164023008896244827187751 50 (which is our plaintext "attack at dawn")

This real world example shows how large the numbers are that is used in the real world.

Conclusion

RSA is the single most useful tool for building cryptographic protocols (in my humble opinion). In this post, I have shown *how* RSA works, I will ***follow this up***[L1] with another post explaining *why* it works.

Links

L1. http://doctrina.org/Why-RSA-Works-Three-Fundamental-Questions-Answered.html
L2. http://doctrina.org/The-3-Seminal-Events-In-Cryptography.html
L3. http://en.wikipedia.org/wiki/OAEP
L4. http://en.wikipedia.org/wiki/Prime_number
L5. http://en.wikipedia.org/wiki/Composite_number
L6. http://en.wikipedia.org/wiki/Euler%27s_totient_function
L7. http://en.wikipedia.org/wiki/Cardinality
L8. http://en.wikipedia.org/wiki/Rabin-Miller
L9. http://en.wikipedia.org/wiki/Extended_euclidean_algorithm
L10. http://doctrina.org/Why-RSA-Works-Three-Fundamental-Questions-Answered.html#wruiwrtt
L11. https://gist.github.com/4184435#file_convert_text_to_decimal.py

---

1. A ***Composite***[L5] number is the formal name given to a number that is not prime.
2. In set theory, anything between |{...}| just means the amount of elements in {...} - called ***cardinality***[L7] for those who are interested