**RSA:**

The RSA algorithm was publicly described in 1977 by [Ron Rivest](http://en.wikipedia.org/wiki/Ron_Rivest), [Adi Shamir](http://en.wikipedia.org/wiki/Adi_Shamir), and [Leonard Adleman](http://en.wikipedia.org/wiki/Leonard_Adleman)

The RSA algorithm involves three steps:

[key](http://en.wikipedia.org/wiki/Key_%28cryptography%29) generation, encryption and decryption.

### Key generation

RSA involves a **public key** and a [**private key**](http://en.wikipedia.org/wiki/Private_key)**.**

The public key can be known to everyone and is used for encrypting messages.

Messages encrypted with the public key can only be decrypted using the private key.

The keys for the RSA algorithm are generated the following way:

1. Choose two distinct [prime numbers](http://en.wikipedia.org/wiki/Prime_number) *p* and *q*.
   * For security purposes, the integers *p* and *q* should be chosen at random, and should be of similar bit-length.
2. Compute *n* = *pq*.
   * *n* is used as the [modulus](http://en.wikipedia.org/wiki/Modular_arithmetic) for both the public and private keys. Its length, usually expressed in bits, is the [key length](http://en.wikipedia.org/wiki/Key_length).
3. Compute φ(*n*) = (*p* – 1)(*q* – 1), where φ is [Euler's totient function](http://en.wikipedia.org/wiki/Euler%27s_totient_function).
4. Choose an integer *e* such that 1 < *e* < φ(*n*) and [greatest common divisor](http://en.wikipedia.org/wiki/Greatest_common_divisor) gcd(*e*, φ(*n*)) = 1; i.e., *e* and φ(*n*) are [coprime](http://en.wikipedia.org/wiki/Coprime).
   * *e* is released as the public key exponent.
   * *e* having a short [bit-length](http://en.wikipedia.org/wiki/Bit-length) and small [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight) results in more efficient encryption – most commonly 216 + 1 = 65,537. However, much smaller values of *e* (such as 3) have been shown to be less secure in some settings.[[4]](http://en.wikipedia.org/wiki/RSA_%28algorithm%29#cite_note-Boneh-4)
5. Determine *d* as *d* ≡ *e*−1 (mod φ(*n*)), i.e., *d* is the [multiplicative inverse](http://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of *e* (modulo φ(*n*)).

* This is more clearly stated as solve for *d* given *de* ≡ 1 (mod φ(*n*))
* This is often computed using the [extended Euclidean algorithm](http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm).
* *d* is kept as the private key exponent.

By construction, *d*⋅*e* ≡ 1 (mod φ(*n*)).

The **public key** consists of the modulus *n* and the public (or encryption) exponent *e*.

The **private key** consists of the modulus *n* and the private (or decryption) exponent *d*, which must be kept secret. *p*, *q*, and φ(*n*) must also be kept secret because they can be used to calculate *d*.

* An alternative, used by [PKCS#1](http://en.wikipedia.org/wiki/PKCS1), is to choose *d* matching *de* ≡ 1 (mod λ) with λ = lcm(*p* − 1, *q* − 1), where lcm is the [least common multiple](http://en.wikipedia.org/wiki/Least_common_multiple). Using λ instead of φ(*n*) allows more choices for *d*. λ can also be defined using the [Carmichael function](http://en.wikipedia.org/wiki/Carmichael_function), λ(*n*).
* The [ANSI X9.31](http://en.wikipedia.org/w/index.php?title=ANSI_X9.31&action=edit&redlink=1) standard prescribes, [IEEE 1363](http://en.wikipedia.org/wiki/P1363) describes, and [PKCS#1](http://en.wikipedia.org/wiki/PKCS1) allows, that *p* and *q* match additional requirements: being [strong primes](http://en.wikipedia.org/wiki/Strong_prime), and being different enough that [Fermat factorization](http://en.wikipedia.org/wiki/Fermat_factorization) fails.

### Encryption

[Alice](http://en.wikipedia.org/wiki/Alice_and_Bob) transmits her public key (*n*, *e*) to [Bob](http://en.wikipedia.org/wiki/Alice_and_Bob) and keeps the private key secret. Bob then wishes to send message *M* to Alice.

He first turns *M* into an integer *m*, such that 0 ≤ *m* < *n* by using an agreed-upon reversible protocol known as a [padding scheme](http://en.wikipedia.org/wiki/RSA_%28algorithm%29#Padding_schemes). He then computes the ciphertext *c* corresponding to

 c \equiv m^e \pmod{n} .

This can be done quickly using the method of [exponentiation by squaring](http://en.wikipedia.org/wiki/Exponentiation_by_squaring). Bob then transmits *c* to Alice.

### Decryption

Alice can recover *m* from *c* by using her private key exponent *d* via computing

 m \equiv c^d \pmod{n} .

Given *m*, she can recover the original message *M* by reversing the padding scheme.

(In practice, there are more efficient methods of calculating *cd* using the precomputed values below.)

### Using the Chinese remainder algorithm

For efficiency many popular crypto libraries (like OpenSSL, Java and .NET) use the following optimization for decryption and signing based on the [Chinese remainder theorem](http://en.wikipedia.org/wiki/Chinese_remainder_theorem). The following values are precomputed and stored as part of the private key:

* pand q: the primes from the key generation,
* d_P = d\text{ (mod }p-1\text{)},
* d_Q = d\text{ (mod }q-1\text{)}and
* q_\text{inv} = q^{-1}\text{ (mod }p\text{)}.

These values allow the recipient to compute the exponentiation *m* = *cd* (mod *pq*) more efficiently as follows:

* m_1 = c^{d_P}\text{ (mod }p\text{)}
* m_2 = c^{d_Q}\text{ (mod }q\text{)}
* h = q_\text{inv}*(m_1-m_2)\text{ (mod }p\text{)}(if m_1 < m_2then some libraries compute *h* as q_\text{inv} \times (m_1+p-m_2)\text{ (mod }p\text{)})
* m = m_2 + (h*q)\,

This is more efficient than computing *m* ≡ *cd* (mod *pq*) even though two modular exponentiations have to be computed. The reason is that these two modular exponentiations both use a smaller exponent and a smaller modulus.

### A working example

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but one can also [use OpenSSL to generate and examine a real keypair](http://en.wikibooks.org/wiki/Transwiki:Generate_a_keypair_using_OpenSSL).

1. Choose two distinct prime numbers, such as

p = 61and q = 53.

1. Compute *n* = *pq* giving

n = 61 \times 53 = 3233.

1. Compute the [totient](http://en.wikipedia.org/wiki/Totient) of the product as φ(*n*) = (*p* − 1)(*q* − 1) giving

\varphi(3233) = (61 - 1)(53 - 1) = 3120.

1. Choose any number 1 < *e* < 3120 that is [coprime](http://en.wikipedia.org/wiki/Coprime) to 3120. Choosing a prime number for *e* leaves us only to check that *e* is not a divisor of 3120.

Let e = 17.

1. Compute *d*, the [modular multiplicative inverse](http://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of *e* (mod φ(*n*)) yielding

d = 2753.

The **public key** is (*n* = 3233, *e* = 17). For a padded [plaintext](http://en.wikipedia.org/wiki/Plaintext) message *m*, the encryption function is *m*17 (mod 3233).

The **private key** is (*n* = 3233, *d* = 2753). For an encrypted [ciphertext](http://en.wikipedia.org/wiki/Ciphertext) *c*, the decryption function is *c*2753 (mod 3233).

For instance, in order to encrypt *m* = 65, we calculate

c \equiv 65^{17} \equiv 2790 \pmod{3233} .

To decrypt *c* = 2790, we calculate

m \equiv 2790^{2753} \equiv 65 \pmod{3233}.

Both of these calculations can be computed efficiently using the [square-and-multiply algorithm](http://en.wikipedia.org/wiki/Square-and-multiply_algorithm) for [modular exponentiation](http://en.wikipedia.org/wiki/Modular_exponentiation). In real life situations the primes selected would be much larger; in our example it would be relatively trivial to factor *n*, 3233, obtained from the freely available public key back to the primes *p* and *q*. Given *e*, also from the public key, we could then compute *d* and so acquire the private key.

Practical implementations use the [Chinese remainder theorem](http://en.wikipedia.org/wiki/Chinese_remainder_theorem) to speed up the calculation using modulus of factors (mod *pq* using mod *p* and mod *q*).

The values *dp*, *dq* and *q*inv, which are part of the private key are computed as follows:

* d_p = d\text{ (mod }(p-1)\text{)} = 2753 \text{ (mod } (61-1)\text{)} = 53
* d_q = d\text{ (mod }(q-1)\text{)} = 2753 \text{ (mod } (53-1)\text{)} = 49
* q_\text{inv} = q^{-1} \text{ (mod } p\text{)} = 53^{-1} \text{ (mod } 61\text{)} = 38(Hence: q_\text{inv} \times q \text{ (mod } p\text{)} = 38 \times 53 \text{ (mod } 61\text{)} = 1)

Here is how *dp*, *dq* and *q*inv are used for efficient decryption. (Encryption is efficient by choice of public exponent *e*)

* m_1 = c^{d_p} \text{ (mod } p\text{)} = 2790^{53} \text{ (mod } 61\text{)} = 4
* m_2 = c^{d_q} \text{ (mod } q\text{)} = 2790^{49} \text{ (mod } 53\text{)} = 12
* h = (q_{Inv} \times (m_1 - m_2)) \text{ (mod } p\text{)} = (38 \times -8) \text{ (mod } 61\text{)} = 1
* m = m_2 + h \times q = 12 + 1 \times 53 = 65(same as above but computed more efficiently)

**ECC:**

**Elliptic curve cryptography (ECC)** is an approach to [public-key cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography) based on the algebraic structure of [elliptic curves](http://en.wikipedia.org/wiki/Elliptic_curve) over [finite fields](http://en.wikipedia.org/wiki/Finite_field).

Ell[iptic Curve Cryptography](http://www.certicom.com/index.php/ecc) (ECC) was discovered in 1985 by Victor Miller (IBM) and Neil Koblitz (University of Washington) as an alternative mechanism for implementing [public-key cryptography](https://www.rsa.com/rsalabs/node.asp?id=2165).

I  assume that those who are going through this article will have a basic understanding of [cryptography](https://en.wikipedia.org/wiki/Cryptography) ( terms like encryption and decryption ) .

The equation of an elliptic curve is given as,

[http://bithin.files.wordpress.com/2012/02/eccequation.png?w=500](http://bithin.files.wordpress.com/2012/02/eccequation.png)

Few terms that will be used,

**E -> Elliptic Curve**

**P -> Point on the curve**

**n -> Maximum limit ( This should be a prime number )**

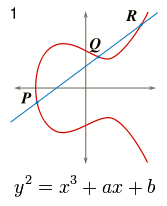
[](http://bithin.files.wordpress.com/2012/02/ell-curve.png)

Fig 3

The fig 3 show are simple elliptic curve.

### ****Key Generation****

Key generation is an important part where we have to generate both public key and private key. The sender will be encrypting the message with receiver’s public key and the receiver will decrypt its private key.

Now, we have to select a number **‘d’** within the range of **‘n’**.

Using the following equation we can generate the public key

### ****Q = d \* P****

**d** = The random number that we have selected within the range of ( **1 to n-1** ). **P** is the point on the curve.

**‘Q’ is the public key** and **‘d’ is the private key.**

### Encryption

Let ‘m’ be the message that we are sending. We have to represent this message on the curve. This have in-depth implementation details. All the advance research on ECC is done by a company called [certicom](http://www.certicom.com/index.php/ecc).

Conside ‘m’ has the point ‘M’ on the curve ‘E’. Randomly select ‘k’ from [1 - (n-1)].

Two cipher texts will be generated let it be **C1** and **C2**.

### ****C1 = k\*P****

### C2 = M + k\*Q

C1 and C2 will be send.

### ****Decryption****

We have to get back the message ‘m’ that was send to us,

### M = C2 – d \* C1

M is the original message that we have send.

### ****Proof****

How does we get back the message,

M = C2 – d \* C1

‘M’ can be represented as ‘C2 – d \* C1′

C2 – d \* C1 = (M + k \* Q) – d \* ( k \* P )          ( C2 = M + k \* Q and C1 = k \* P )

=  M + k  \* d \* P – d \* k \*P          ( canceling out k \* d \* P )

= M  ( Original Message )