WIKIPEDIA

RSA algorithm

RSA (Rivest–Shamir–Adleman) is an algorithm used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. Asymmetric means that there are two different keys. This is also called public key cryptography, because one of the keys can be given to anyone. The other key must be kept private. The algorithm is based on the fact that finding the factors of a large composite number is difficult: when the factors are prime numbers, the problem is called prime factorization. It is also a key pair (public and private key) generator.

RSA involves a public key and private key. The public key can be known to everyone- it is used to encrypt messages. Messages encrypted using the public key can only be decrypted with the private key. The private key needs to be kept secret. Calculating the private key from the public key is very difficult.

Contents

Concepts used

RSA use a number of concepts from cryptography:

- A one-way function that is easy to compute; finding a function that reverses it, or computing this function is very difficult.
- RSA uses a concept called discrete logarithm. This works much like the normal logarithm: The difference is that only whole numbers are used, and in general, a modulus operation is involved. As an example $A^X=b$, modulo n. The discrete logarithm is about finding the smallest x that satisfies the equation, when a b and n are provided

Generating keys

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

1. Choose two different large random prime numbers p and q. This should be kept secret.

2. Calculate $n = pq$.
   - n is the modulus for the public key and the private keys.
3. Calculate the totient: $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$, and e is co-prime to $\phi(n)$.
   i.e.: e and $\phi(n)$ share no factors other than 1: $\gcd(e, \phi(n)) = 1$; See greatest common divisor.
   - e is released as the public key exponent.
5. Compute d to satisfy the congruence relation $de \equiv 1 \pmod{\phi(n)}$.
   i.e.: $de = 1 + x \cdot \phi(n)$ for some integer x. (Simply to say: Calculate $d = (1 + x \cdot \phi(n))/e$ to be an integer)
   - d is kept as the private key exponent.

Notes on the above steps:

- Step 1: Numbers can be probabilistically tested for primality.
- Step 3: changed in PKCS #1 v2.0 to $\lambda(n) = \operatorname{lcm}(p-1, q-1)$ instead of $\phi(n) = (p-1)(q-1)$.
- Step 4: A popular choice for the public exponents is $e = 2^{16} + 1 = 65537$. Some applications choose smaller values such as $e = 3$, 5, or 35 instead. This is done to make encryption and signature verification faster on small devices like smart cards but small public exponents may lead to greater security risks.
- Steps 4 and 5 can be performed with the extended Euclidean algorithm; see modular arithmetic.

The **public key** is made of the modulus n and the public (or encryption) exponent e.
The **personal key** is made of p,q and the private (or decryption) exponent d which must be kept secret.

- For efficiency a different form of the **private key** can be stored:
  - p and q: the primes from the key generation;
  - $d \bmod (p-1)$ and $d \bmod (q-1)$: often called dmp1 and dmq1;
  - $q^{-1} \bmod p$: often called iqmp.
- All parts of the private key must be kept secret in this form. p and q are sensitive since they are the factors of n, and allow computation of d given e. If p and q are not stored in this form of the private key then they are securely deleted along with other intermediate values from key generation.
- Although this form allows faster decryption and signing by using the Chinese Remainder Theorem (CRT) it is considerably less secure since it enables side channel attacks. This is a particular problem if implemented on smart cards, which benefit most from the improved efficiency.
  (Start with $y \equiv x^e \pmod{n}$, and let the card decrypt that. So it computes $(y^d \bmod p)$ or $(y^d \bmod q)$, whose results give some value z. Now, induce an error in one of the computations. Then $\gcd(z - x, n)$ will reveal p or q.)

Encrypting message

Alice gives her public key (n, e) to Bob and keeps her private key secret. Bob wants to send message M to Alice.

First he turns M into a number m smaller than n by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c corresponding to:

$$c = m^e \mod n$$

This can be done quickly using the method of exponentiation by squaring. Bob then sends c to Alice.

Decrypting message

Alice can recover m from c by using her private key d in the following procedure:

Given m, she can recover the original distinct prime numbers, applying the Chinese remainder theorem to these two congruences yields

$$m^{ed} \equiv m \bmod pq.$$

Thus,

$$c^d \equiv m \bmod n.$$

Therefore:

$$m = c^d \bmod n$$

A working example

Here is an example of RSA encryption and decryption. The prime numbers used here are too small to let us securely encrypt anything. You can use OpenSSL to generate and examine a real keypair.

1. Choose two random prime numbers p and q:

   $p = 61$ and $q = 53$;

2. Compute $n = pq$:

   $n = 61 \times 53 = 3233$;

3. Compute the totient $\phi(n) = (p - 1)(q - 1)$:

   $\phi(n) = (61 - 1)(53 - 1) = 3120$;

4. Choose $e > 1$ coprime to 3120:

   $e = 17$;

5. Choose d to satisfy $ed \equiv 1 \bmod \phi(n)$:

   $d = 2753$, with $17 \times 2753 = 46801 = 1 + 15 \times 3120$.

The **public key** is ($n = 3233$, $e = 17$). For a padded message m the encryption function $c = m^e \bmod n$ becomes:

$$c = m^{17} \bmod 3233$$

The **private key** is ($n = 3233$, $d = 2753$). The decryption function $m = c^d \bmod n$ becomes:

$$m = c^{2753} \bmod 3233$$

For example, to encrypt $m = 123$, we calculate

$$c = 123^{17} \bmod 3233 = 855$$

To decrypt $c = 855$, we calculate

$$m = 855^{2753} \bmod 3233 = 123$$

Both of these calculations can be computed fast and easily using the square-and-multiply algorithm for modular exponentiation.

Deriving RSA equation from Euler's theorem

RSA can easily be derived using Euler's theorem and Euler's totient function.

Starting with Euler's theorem,

$$m^{\phi(n)} \equiv 1 \pmod{n}$$

we must show that decrypting an encrypted message, with the correct key, will give the original message. Given a padded message *m*, the ciphertext *c*, is calculated by

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

Substituting into the decryption algorithm, we have

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}$$

We want to show this value is also congruent to *m*. The values of *e* and *d* were chosen to satify,

$$ed \equiv 1 \pmod{\phi(n)}$$

Which is to say, there exists some integer *k*, such that

$$ed = k \times \phi(n) + 1$$

Now we substitute into the encrypted then decrypted message,

$$\begin{aligned} m^{ed} &\equiv m^{k\phi(n)+1} \\ &\equiv m \times m^{k\phi(n)} \\ &\equiv m \times \left(m^{\phi(n)}\right)^k \pmod{n} \end{aligned}$$

We apply Euler's theorem, and achive the result.

$$m \times (1)^k \equiv m \pmod{n}$$

Padding schemes

When used in practice, RSA must be combined with some form of padding scheme, so that no values of *M* result in insecure ciphertexts. RSA used without padding may have some problems:

- The values m = 0 or m = 1 always produce ciphertexts equal to 0 or 1 respectively, due to the properties of exponentiation.
- When encrypting with small encryption exponents (e.g., e = 3) and small values of the m, the (non-modular) result of m^e may be strictly less than the modulus n. In this case, ciphertexts may be easily decrypted by taking the eth root of the ciphertext with no regard to the modulus.
- RSA encryption is a deterministic encryption algorithm. It has no random component. Therefore, an attacker can successfully launch a chosen plaintext attack against the cryptosystem. They can make a dictionary by encrypting likely plaintexts under the public key, and storing the resulting ciphertexts. The attacker can then observe the communication channel. As soon as they see ciphertexts that match the ones in their dictionary, the attackers can then use this dictionary in order to learn the content of the message.

In practice, the first two problems can arise when short ASCII messages are sent. In such messages, *m* might be the concatenation of one or more ASCII-encoded character(s). A message consisting of a single ASCII NUL character (whose numeric value is 0) would be encoded as *m* = 0, which produces a ciphertext of 0 no matter which values of *e* and *N* are used. Likewise, a single ASCII SOH (whose numeric value is 1) would always produce a ciphertext of 1. For systems which conventionally use small values of *e*, such as 3, all single character ASCII messages encoded using this scheme would be insecure, since the largest *m* would have a value of 255, and 255^3 is less than any reasonable modulus. Such plaintexts could be recovered by simply taking the cube root of the ciphertext.

To avoid these problems, practical RSA implementations typically embed some form of structured, randomized padding into the value *m* before encrypting it. This padding ensures that *m* does not fall into the range of insecure plaintexts, and that a given message, once padded, will encrypt to one of a large number of different possible ciphertexts. The latter property can increase the cost of a dictionary attack beyond the capabilities of a reasonable attacker.

Standards such as PKCS have been carefully designed to securely pad messages prior to RSA encryption. Because these schemes pad the plaintext *m* with some number of additional bits, the size of the un-padded message *M* must be somewhat smaller. RSA padding schemes must be carefully designed so as to prevent sophisticated attacks. This may be made easier by a predictable message

structure. Early versions of the PKCS standard used ad-hoc (https://simple.wiktionary.org/wiki/ad-hoc) constructions, which were later found vulnerable to a practical adaptive chosen ciphertext attack. Modern constructions use secure techniques such as Optimal Asymmetric Encryption Padding (OAEP) to protect messages while preventing these attacks. The PKCS standard also has processing schemes designed to provide additional security for RSA signatures, e.g., the Probabilistic Signature Scheme for RSA (RSA-PSS).

Signing messages

Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice but Bob has no way of verifying that the message was actually from Alice since anyone can use Bob's public key to send him encrypted messages. So, in order to verify the origin of a message, RSA can also be used to sign a message.

Suppose Alice wishes to send a signed message to Bob. She produces a hash value of the message, raises it to the power of d mod n (just like when decrypting a message), and attaches it as a "signature" to the message. When Bob receives the signed message, he raises the signature to the power of e mod n (just like encrypting a message), and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's secret key, and that the message has not been tampered with since.

Note that secure padding schemes such as RSA-PSS are as essential for the security of message signing as they are for message encryption, and that the same key should never be used for both encryption and signing purposes.

References

Other websites

- The Original RSA Patent as filed with the U.S. Patent Office by Rivest; Ronald L. (Belmont, MA), Shamir; Adi (Cambridge, MA), Adleman; Leonard M. (Arlington, MA), December 14, 1977, **U.S. Patent 4,405,829 (http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=4405829)**.
- PKCS #1: RSA Cryptography Standard (http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-rsa-cryptography-standard.htm) (RSA Laboratories website)
  - The PKCS #1 standard "provides recommendations for the implementation of public-key cryptography based on the **RSA** algorithm, covering the following aspects: cryptographic primitives; encryption schemes; signature schemes with appendix; ASN.1 syntax for representing keys and for identifying the schemes".
- Explanation of RSA using colored lamps (https://www.youtube.com/watch?v=vgTtHV04xRI) at YouTube
- Thorough walk through of RSA (http://www.di-mgt.com.au/rsa_alg.html)
- Prime Number Hide-And-Seek: How the RSA Cipher Works (http://www.muppetlabs.com/~breadbox/txt/rsa.html)
- Onur Aciicmez, Cetin Kaya Koc, Jean-Pierre Seifert: On the Power of Simple Branch Prediction Analysis (http://eprint.iacr.org/2006/351)
- A New Vulnerability In RSA Cryptography, CAcert NEWS Blog (http://blog.cacert.org/2006/11/193.html)

- Example of an RSA implementation with PKCS#1 padding (GPL source code) (http://polarssl.org/source_code)
- Kocher's article about timing attacks (http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf)
- An animated explanation of RSA with its mathematical background by CrypTool (https://www.cryptool.org/images/ct1/presentations/RSA/RSA-Flash-en/player.html) Archived (https://web.archive.org/web/20180928124928/https://www.cryptool.org/images/ct1/presentations/RSA/RSA-Flash-en/player.html) 2018-09-28 at the Wayback Machine
- An interactive walkthrough going through all stages to make small example RSA keys (https://rsa.gnos.space/) Archived (https://web.archive.org/web/20200707214322/https://rsa.gnos.space/) 2020-07-07 at the Wayback Machine
- Hacking Secret Ciphers with Python (http://inventwithpython.com/hacking), Chapter 24, Public Key Cryptography and the RSA Cipher (http://inventwithpython.com/hacking/chapter24.html)
- Grime, James. "RSA Encryption" (https://web.archive.org/web/20181006042830/http://www.numberphile.com/videos/RSA.html). Numberphile. Brady Haran. Archived from the original (http://www.numberphile.com/videos/RSA.html) on 2018-10-06. Retrieved 2019-06-20.
- How RSA Key used for Encryption in real world (http://www.gnudeveloper.com/groups/cyber_security/Cryptography_RSA_Key_Exchange_works_in_realtime_using_Keytool_openSSL%20.html)
- Prime Numbers, Factorization, and their Relationship with Encryption (https://www.putorius.net/factor-prime-numbers-encryption.html)
