Introduction to Cryptography

Alvin Lin

January 2018 - May 2018

Public Key Cryptosystems

Properties of Symmetric Key Cryptography

- The same secret key is used for encryption and decryption.
- Encryption and decryption are very similar (or even identical functions).
- Symmetric algorithms like AES or 3DES are secure, fast, and widespread but face a key distribution problem. The secret key must be transported securely because the security depends on the key.
- In a network of users, each pair of users requires an individual key. n users in a network would require $\frac{n(n-1)}{2}$, where each user stores $n-1$ keys.
- Two users can cheat each other since they have identical keys. Non-repudiation is not guaranteed by symmetric key cryptography.

Asymmetric Key Cryptography

The idea behind asymmetric key is similar to the idea of a mailbox. Anyone can drop a letter into it, but only the owner can open it. The first publication of such an algorithm was made by Whitfield Diffie and Martin Hellman in 1976. In principle, the key is split up into a public key K_{pub} and a private key K_{pr}. This allows for mechanisms such as:

- **Key Distribution**, like the Diffie-Hellman key exchange, without the need for a pre-shared secret.
- **Nonrepudiation and Digital Signatures** to prove message integrity.

- **Identification** using challenge-response protocols with digital signatures.
- **Encryption** using algorithms like RSA, though it is computationally expensive and 1000 times slower than symmetric algorithms.

In practice, hybrid systems are used, incorporating both asymmetric and symmetric algorithms. Key exchanges and digital signatures are performed using asymmetric algorithms, while data encryption is done using fast symmetric ciphers.

Public Key Algorithm Theory

Asymmetric schemes are based on some one-way function f, where computing $y = f(x)$ is computationally easy, but computing $x = f^{-1}(y)$ is computationally infeasible. One way functions are usually based on mathematically difficult problems, such as:

- **Factoring Integers**: given a composite integer n, find its prime factors.
- **Discrete Logarithm (DL)**: given a, y, m, find x such that $a^x = y \mod m$.
- **Elliptic Curves (EC)**: generalization of discrete logarithm.

Key Lengths and Security Levels

Symmetric	ECC	RSA, DL	Remark
64 bit	128 bit	$\approx$ 700 bit	short term security
80 bit	160 bit	$\approx$ 1024 bit	medium security
128 bit	256 bit	$\approx$ 3072 bit	long term security

The exact complexity of RSA and DL is difficult to estimate. The existence of quantum computers would probably be the end for ECC, RSA, and DL, but that is still decades away. Some researchers doubt that quantum computers will ever exist.

Euclidean Algorithm

Earlier, we explored the Euclidean algorithm for finding the greatest common divisor, which is based on the observation that:

$$gcd(r_0) = gcd(r_0 - r_1, r_1)$$

By reducing the problem recursive, we find that $gcd(r_i, 0) = r_i$ is the answer to the original problem. We can extend the Euclidean algorithm to find the modular inverse of $r_1 \mod r_0$. In order for the inverse to exist:

$$gcd(r_0, r_1) = 1$$

The extended Euclidean algorithm computes s, t such that:

$$gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1 = 1$$

If we take this relation mod r_0:

$$\begin{aligned} s \cdot r_0 + t \cdot r_1 = 1 s \cdot r_0 + t \cdot r_1 &\equiv 1 \mod r_0 \\ t \cdot r_1 &\equiv 1 \mod r_0 \end{aligned}$$

By definition, t is the inverse of $r_1 \mod r_0$.

Euler's Phi Function

Euler's Phi function $\phi(m)$ gives the numbers in the set of integers from 0 to $m-1$ that are relatively prime to m. Example:

$$\begin{aligned} gcd(0, 6) &= 6 \\ gcd(1, 6) &= 1 \\ gcd(2, 6) &= 2 \\ gcd(3, 6) &= 3 \\ gcd(4, 6) &= 2 \\ gcd(5, 6) &= 1 \\ \phi(6) &= 2 \end{aligned}$$

$\phi(6) = 2$ since only 1 and 5 are relatively prime to $m = 6$. If the canonical factorization (prime factorization) of m is known, where:

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_n^{e_n}$$

such that p_i are primes and e_i are positive integer exponents, then $\phi(m)$ can be calculated by:

$$\phi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1})$$

For example:

$$\begin{aligned} m &= 12 = 2^2 \cdot 3^1 \\ \phi(12) &= (2^2 - 2^1) \cdot (3^1 - 3^0) = 4 \\ m &= 899 = 29^1 \cdot 31^1 \\ \phi(899) &= (29^1 - 29^0) \cdot (31^1 - 31^0) = 38 \cdot 30 = 840 \end{aligned}$$

Finding $\phi(m)$ is computationally easy if the factorization of m is known, otherwise it becomes computationally infeasible for large numbers.

Fermat's Little Theorem

Given a prime p and an integer a such that $a^p \equiv a \mod p$, the following is true:

$$a^{p-1} \equiv 1 \mod p$$

This is used to find the modular inverse if p is prime. We can rewrite the relation above as follows:

$$\begin{aligned} a^{p-1} &\equiv 1 \mod p \\ a \cdot a^{p-2} &\equiv 1 \mod p \end{aligned}$$

By the definition of the modular inverse:

$$\begin{aligned} a \cdot a^{-1} &\equiv 1 \mod p \\ a^{-1} &\equiv a^{p-2} \mod p \end{aligned}$$

For example, $a = 2, p = 7$:

$$\begin{aligned} 2^7 &\equiv 2 \mod 7 \\ 2^{-1} &\equiv 2^5 \mod 7 \equiv 4 \mod 7 \\ 2 \cdot 4 &\equiv 1 \mod 7 \end{aligned}$$

The RSA Cryptosystem

After Martin Hellman and Whitfield Diffie published their public key paper in 1976, Ronald Rivest, Adi Shamir, and Leonard Adleman proposed the asymmetric RSA cryptosystem in 1977. Until now, RSA was the most widely used asymmetric cryptosystem, with elliptic curve cryptography gaining popularity. It has two main applications, the transport of symmetric keys, and digital signatures. RSA operations

are done over the integer ring Z_n where $n = pq$ such that p and q are large primes. Encryption and decryption are simply exponentiations in the ring. Given the public key $(n, e) = k_{pub}$ and the private key $d = k_{pr}$:

$$y = e_{k_{pub}}(x) \equiv x^e \mod n$$
$$x = d_{k_{pr}}(y) \equiv y^d \mod n$$

where $x, y \in Z_n$. In practice, x, y, n, d are very long integer numbers, usually greater than 1024 bits. The security of the scheme relies on the fact that it is hard to derive the private exponent d given the public key (n, e).

Key Generation

Like all asymmetric schemes, RSA has set up phases during which the private and public keys are computed. To generate an rsa key pair:

1. Choose two large primes p, q
2. Compute $n = pq$
3. Compute $\phi(n) = (p-1)(q-1)$
4. Select the public exponent $e \in \{1, 2, 3, \ldots, \phi(n)-1\}$ such that $gcd(e, \phi(n)) = 1$.
5. Compute the private key d such that $de \equiv 1 \mod \phi(n)$
6. $k_{pub} = (n, e) \quad k_{pr} = d$

Choose the two large distinct primes p, q in step 1 is non trivial. Step 4, ensures that e has an inverse and thus there is always a private key d.

Implementation Aspects

The RSA cryptosystem uses only one arithmetic operation (modular exponentiation), which makes it conceptually a simple asymmetric scheme. Even though it is conceptually simple, it is orders of magnitude slower than symmetric schemes such as DES or AES due to the use of very long numbers. When implementation RSA on devices with memory constraints, arithmetic algorithms must be selected very carefully.

The Square and Multiply Algorithm

The square and multiply algorithm allows for fast exponentiation for very long numbers. This is done by converting the exponent to binary and squaring or multiplying the base depending on the current bit of the exponent. Modular reduction can be applied during each step to keep the operands small. Example for x^{26} (without modular reduction):

$$26 = (11010)_2$$

Step	Accumulated Result	Binary Exponent	Operation
1	$r = (1^2)x = x$	1	Square and Multiply
2	$r = (x^2)x = x^3$	11	Square and Multiply
3	$r = (x^3)^2 = x^6$	110	Square
4	$r = (x^6)^2x = x^{13}$	1101	Square and Multiply
5	$r = (x^{13})^2 = x^{26}$	11010	Square

This algorithm has a logarithmic complexity since its run time is proportional to the bit length of the exponent. Given an exponent with $t + 1$ bits, we need to perform t squarings and an average of $0.5t$ multiplications. Since exponents are chosen randomly, $1.5t$ is a good estimate for the average number of operations. Modular exponentiation is still computationally expensive. For devices with memory constraints, RSA can still be quite slow. Choosing a small public exponent e to perform RSA with does not weaken the security of the algorihtm, but significantly improves the speed.

The Chinese Remainder Theorem (CRT)

Sun Tzu, the author of *The Art of War*, is said to have created the Chinese remainder theorem as a method of counting the size of the ancient Chinese armies. Let $n_1, \ldots, n_i, \ldots, n_r$ be integers greater than 1. If the n_i are pairwise coprime, and if $a_1, \ldots, a_r$ are integers such that $0 \leq a_i \leq n_i$ for every i, then the Chinese Remainder Theorem states that the system

$$\begin{aligned} x &\equiv a_1 \mod n_1 \\ x &\equiv a_2 \mod n_2 \\ &\cdots \\ x &\equiv a_r \mod n_r \end{aligned}$$

has a unique solution x such that $0 \leq x < N$. Additionally, let $N = n_1 \cdot n_2 \cdot n_3 \cdot \cdots \cdot n_r$. Then

$$x \equiv (a_1 b_1 N_1 + a_2 b_2 N_2 + \cdots + a_r b_r N_r) \mod N$$

where $N_i = \frac{N}{n_i}$ and b_i is determined from

$$b_i N_i \equiv 1 \mod n_i$$

for i from 1 to r.

Example

$$\begin{aligned}
x &\equiv 2 \mod 3 \\
x &\equiv 3 \mod 5 \\
x &\equiv 2 \mod 7 \\
N &= n_1 n_2 n_3 = 105 \\
N_1 &= 35 \quad N_2 = 21 \quad N_3 = 15 \\
35b_1 &\equiv 1 \mod 3 \quad 2b_1 \equiv 1 \mod 3 \\
21b_2 &\equiv 1 \mod 5 \quad 1b_2 \equiv 1 \mod 5 \\
15b_3 &\equiv 1 \mod 7 \quad 1b_3 \equiv 1 \mod 7 \\
b_1 &= 2 \quad b_2 = 1 \quad b_3 = 1 \\
x &\equiv (2 \cdot 2 \cdot 35 + 3 \cdot 1 \cdot 21 + 2 \cdot 1 \cdot 15) \mod 105 \\
&\equiv 233 \mod 105 \\
&\equiv 23
\end{aligned}$$

Example

$$\begin{aligned}
x &\equiv 0 \mod 3 \\
x &\equiv 3 \mod 4 \\
x &\equiv 4 \mod 5 \\
N &= 3 \cdot 4 \cdot 5 = 60 \\
N_1 &= 20 \quad N_2 = 15 \quad N_3 = 12 \\
20b_1 &\equiv 1 \mod 3 \quad 2b_1 \equiv 1 \mod 3 \\
15b_2 &\equiv 1 \mod 4 \quad 3b_2 \equiv 1 \mod 4 \\
12b_3 &\equiv 1 \mod 5 \quad 2b_3 \equiv 1 \mod 5 \\
b_1 &= 2 \quad b_2 = 3 \quad b_3 = 3 \\
x &\equiv (0 \cdot 2 \cdot 20 + 3 \cdot 3 \cdot 15 + 4 \cdot 3 \cdot 12) \mod 70 \\
&\equiv 39
\end{aligned}$$

Application to RSA

With regard to its application to public key cryptography, choosing a small private key d for RSA results in security weaknesses. d must have at least $0.3t$ bits, where t is the bit length of the modulus n. However, the Chinese Remainder Theorem can be used to somewhat accelerate exponentiation with the private key d. We can replace the computation of

$$x^{d \mod \phi(n)} \mod n$$

by two computations:

$$x^{d \mod (p-1)} \mod p$$
$$x^{d \mod (q-1)} \mod q$$

where q and p are "small" compared to n. This involves three distinct steps.

1. Transformation of the operand into the CRT domain
2. Modular exponentiation in the CRT domain
3. Inverse transformation into the problem domain

For example, consider the problem $90^{547} \mod 899$. First we need to transform it into the CRT domain. Transformation into the CRT domain requires knowledge of p and q. These are only known to the owner of the private key, hence it cannot be applied to speed up encryption. The transformation computes (x_p, x_q) which is the representation of x in the CRT domain.

$$\begin{aligned} d &= 547 \\ n &= 899 = 29 \cdot 31 \\ x_p &= x \mod p = 90 \mod 29 = 3 \\ x_q &= x \mod q = 90 \mod 31 = 28 \end{aligned}$$

The next step is exponentiation. Given d_p and d_q such that $d_p \equiv d \mod (p-1)$ and $d_q \equiv d \mod (q-1)$, one exponentiation in the problem domain requires two exponentiations in the CRT domain.

$$\begin{aligned} d_p &= 547 \mod 28 = 15 \\ d_q &= 547 \mod 30 = 7 \\ y_p &\equiv x_p^{d_p} \mod p = 3^{15} \mod 29 = 26 \\ y_q &\equiv x_q^{d_q} \mod q = 28^7 \mod 31 = 14 \end{aligned}$$

In practice, p and q are chosen to have half the bit length of n. Inverse transformation requires modular inversion twice, which is computationally expensive.

$$\begin{aligned} c_p &= q^{-1} \mod p = 31^{-1} \mod 29 = 15 \\ c_q &= p^{-1} \mod q = 29^{-1} \mod 31 = 15 \end{aligned}$$

This is assembled with y_p, y_q into the final result $y \mod n$ in the problem domain.

$$\begin{aligned} y &\equiv \Big[(q \cdot c_p) \cdot y_p + (p \cdot c_q) \cdot y_q\Big] \mod n \\ y &\equiv \Big[(31 \cdot 13) \cdot 26 + (29 \cdot 15) \cdot 14\Big] \mod 899 = 200 \end{aligned}$$

The primes p and q typically change infrequently. Therefore, the cost of inversion can be neglected because $q \cdot c_p$ and $p \cdot c_q$ can be precomputed and stored. We ignore the transformation and inverse transformation steps since their costs can be neglected under reasonable assumptions. If we assume that n has $t + 1$ bits, both p and q are about $\frac{t}{2}$ bits long. The complexity is determined by the two exponentiations in the CRT domain. The operands are only $\frac{t}{2}$ bits long. For the exponentiations, we use the square-and-multiply algorithm. This gives a total complexity of $1.5t$ as well, but since the operands have half the bit length of the problem exponent, each operation is 4 times faster.

Finding Large Primes

Generating keys for RSA requires finding two large primes p and q such that $n = pq$ is sufficiently large. The size of p and q is typically half the size of the desired size of n. To find primes, random integers are generated and tested for primality. The random number generator should be non-predictable otherwise an attacker could guess the factorization of n.

Factoring p and q to test for primality is typically not feasible. However, we are not interested in the factorization, we only want to know whether p and q are composite. Primality test are typically probabilistic. They are not 100% accurate but their output is correct with very high probability. A probabilistic test has two outputs:

- The input is composite. This is always true if it is returned.
- The input is a prime. This is only true with a certain probability.

Fermat Primality Test

The basic idea is that Fermat's Little Theorem holds for all primes. If a number p' is found for which $a^{p'-1} \not\equiv 1 \mod p'$, it is not a prime. For certain numbers like the Carmichael numbers, this test returns a false positive when the numbers are composite. The Fermat-Primality test is usually implemented using a candidate number p' and a security parameter s.

```
for i = 1 to s:
  choose a random a ∈ {2, 3, ..., p' − 2}
  if a^(p'−1) ≢ 1 mod p' then:
    return p' is composite
  return p' is likely a prime
```

We call a a **Fermat witness** if it shows that p' is composite or a **Fermat liar** if it makes the algorithm give a false result.

Miller-Rabin Primality Test

Given the decomposition of an odd prime candidate p' as

$$p' - 1 = 2^u \cdot r$$

where r is odd, if we can find an integer a such that

$$a^r \not\equiv 1 \mod p'$$

$$a^{r \cdot 2j} \not\equiv p' - 1 \mod p'$$

for all $j \in \{0, 1, \ldots, u-1\}$, then p' is composite. Otherwise, it is probably a prime. The Miller-Rabin primality test is also implemented using a candidate number p' and a security parameter s such that $p' - 1 = 2^u \cdot r$.

```
for i = 1 to s:
  choose a random a ∈ {2, 3, ..., p' − 2}
  z ≡ a^r mod p'
  if z ≠ 1 and z ≠ p' − 1 then:
    for j = 1 to u − 1:
      z ≡ z^2 mod p'
      if z = 1 then:
        return p' is composite
    if z = p' − 1 then:
      return p' is composite
return p' is likely a prime
```

You can find all my notes at `http://omgimanerd.tech/notes`. If you have any questions, comments, or concerns, please contact me at alvin@omgimanerd.tech