Chapter 1

The RSA Cipher and its Algorithmic Foundations

The most important—that is, most applied and most analyzed—asymmetric cipher is RSA, named after its inventors Ron Rivest, Adi Shamir, and Len Adleman. It uses elementary number theoretic algorithms, and its supposed security relies on the hardness of factoring large numbers into primes, although breaking RSA might be easier then factoring, see Section 2.2.

The three fundamental arithmetic algorithms for implementing RSA are

- binary power algorithm,
- Euclidean algorithm,
- chinese remainder algorithm,

the last two of them known from Part I of these lectures (in the context of linear ciphers). These algorithms are basic not only for cryptography but more generally for algorithmic algebra and number theory ("computer algebra"). Moreover they are fundamental also for numerical mathematics—for problems that don't require approximate numerical solutions in floating-point numbers but exact solutions in integers, rationals, or symbolic expressions.

1.1 Description of the RSA Cipher

Parameters

The three parameters

- $n =$ **module**,
- $e =$ **public exponent**,
- $d =$ **private exponent**,

are positive integers with

$$m^{ed} \equiv m \pmod{n} \qquad \text{for all } m \in [0 \dots n-1]. \tag{1}$$

Naive Description

The first idea is to set

$$M = C = \mathbb{Z}/n\mathbb{Z}, \quad K \subseteq [1 \dots n-1] \times [1 \dots n-1].$$

For $k = (e, d)$ we have

$$E_k : M \longrightarrow C, \qquad m \mapsto c = m^e \bmod n,$$

$$D_k : C \longrightarrow M, \qquad c \mapsto m = c^d \bmod n.$$

This description is naive for n is variable, and (necessarily, as we'll see soon) a part of the public key. In particular the sets M and C vary.

More Exact Description

We want to describe RSA in a form that fits the general definition of a cipher. To this end we note that for an l bit number n we have $2^{l-1} \leq n < 2^l$, thus fix the parameters:

- $l =$ bit length of the module (= "key length"),
- $l_1 < l$ bit length of plaintext blocks,
- $l_2 \geq l$ bit length of ciphertext blocks.

We construct a block cipher $M \longrightarrow C$ over the alphabet $\Sigma = \mathbb{F}_2$ with

$$M = \mathbb{F}_2^{l_1} \subseteq \mathbb{F}_2^{l_2} = C.$$

The key $k = (n, e, d) \in \mathbb{N}^3$ is chosen with ($2^{l-1} \leq n < 2^l$ or equivalently:)

$$\ell(n) := \lfloor \log_2 n \rfloor + 1 = l, \quad 1 \leq e \leq n-1, \quad 1 \leq d \leq n-1,$$

such that equation (1) holds. The symbol $\ell(n)$ denotes the number of bits, that is, the length of the binary representation of n.

To encrypt a plaintext block m of length l_1 by E_k we interpret it as the binary representation of an integer. The result c, a non-negative integer $< n$, has a binary representation by l_2 bits—completed with leading zeroes if necessary, or better yet, with random leading bits.

To decipher the ciphertext block c we interpret it as a non-negative integer $c < n$ and transform it into $m = c^d \bmod n$.

Really Exact Description

See PKCS = 'Public Key Cryptography Standard' #1:
`https://tools.ietf.org/html/rfc8017`.

Questions to Address

- How to find suitable parameters n, d, e such that (1) holds?
- How to efficiently implement the procedures for encryption and decryption?
- How to assess the security?

Speed

Note that encryption and decryption are significantly slower than for common symmetric ciphers. (Estimates range up to a factor of roughly 10^4.)

1.2 The Binary Power Algorithm

The procedure for raising powers in a quite efficient way has a natural description in the abstract framework of a multiplicative semigroup H. The task is: Compute the power x^n for $x \in H$ and a positive integer n—the product of n factors x—by *as few multiplications as possible*. The naive direct method,

$$x^n = x \cdot (x \cdots x)\,,$$

involves $n-1$ multiplications. The expense is proportional with n, hence grows *exponentially* with the number $\ell(n)$ of bits (or decimal places) of n. A much better idea is the **binary power algorithm**. In the case of an additively written operation (strictly speaking for the semigroup $H = \mathbb{N}$) it is known also as Russian peasant multiplication, and was known in ancient Egypt as early as 1800 B. C., in ancient India earlier than 200 B. C.

The specification starts from the binary representation of the exponent n,

$$n = b_k 2^k + \cdots + b_0 2^0 \quad \text{with } b_i \in \{0,1\},\ b_k = 1,$$

thus $k = \lfloor \log_2 n \rfloor = \ell(n) - 1$. Then

$$x^n = (x^{2^k})^{b_k} \cdots (x^2)^{b_1} \cdot x^{b_0}.$$

This suggests the following procedure: Compute $x, x^2, x^4, \ldots, x^{2^k}$ in order by squaring k times (and keeping the intermediate results), and then multiply the x^{2^i} that have $b_i = 1$. The number of factors is $\nu(n)$, the number of 1's in the binary representation. In particular $\nu(n) \le \ell(n)$. This makes a total of $\ell(n) + \nu(n) - 2$ multiplications.

We have shown:

Proposition 1 *Let H be a semigroup. Then for all $x \in H$ and $n \in \mathbb{N}$ we can compute x^n by at most $2 \cdot \lfloor \log_2 n \rfloor$ multiplications.*

This expense is only *linear* in the bit length of n. Of course to assess the complete expense we have to account for the cost of multiplication in the semigroup H.

Here is a description as pseudocode:

Procedure BinPot

Input parameters:

x = base

[locally used for storage of the iteratively computed squares]

n = exponent

Output parameters:

y = result x^n

[locally used for accumulation of the partial product]

Instructions:

$y := 1$.

while $n > 0$:

if n is odd: $y := yx$.

$x := x^2$.

$n := \lfloor n/2 \rfloor$.

Remarks

1. The algorithm is almost optimal, but not completey. The theory of "addition chains" in number theory yields an asymptotic behaviour of $\log_2 n$ for the average minimum number of multiplications, roughly half the value from Proposition 1.

2. That the numbers of involved multiplications differ depending on the exponent is the starting point of *timing* and *power attacks* invented by Paul Kocher. Imagine a device, say a smart card, that computes powers with a secret exponent. Then the different timings or power consumptions reveal information about the exponent.

1.3 The Carmichael Function

We assume $n \geq 2$.

The Carmichael function is defined as the exponent of the multiplicative group $\mathbb{M}_n = (\mathbb{Z}/n\mathbb{Z})^\times$:

$$\lambda(n) := \exp(\mathbb{M}_n) = \min\{s \geq 1 \mid a^s \equiv 1 \pmod{n} \quad \text{for all } a \in \mathbb{M}_n\};$$

in other words, $\lambda(n)$ is the maximum of the orders of the elements of $\mathbb{M}_n$.

Remarks

1. Euler's theorem may be expressed as $\lambda(n)|\varphi(n)$ ("exponent divides order"). A common way of expressing it is

   $$a^{\varphi(n)} \equiv 1 \pmod{n} \quad \text{for all } a \in \mathbb{Z} \text{ with } \gcd(a, n) = 1.$$

   Both versions follow immediately from the definition.

2. If p is prime, then $\mathbb{M}_p$ is cyclic—see Proposition 2 below—, hence

   $$\lambda(p) = \varphi(p) = p - 1.$$

By the chinese remainder theorem we have $\mathbb{M}_{mn} \cong \mathbb{M}_m \times \mathbb{M}_n$, hence by Lemma 22 of Appendix A.10:

Corollary 1 *For coprime $m, n \in \mathbb{N}_2$*

$$\lambda(mn) = \operatorname{lcm}(\lambda(m), \lambda(n)).$$

Corollary 2 *If $n = p_1^{e_1} \cdots p_r^{e_r}$ is the prime decomposition of $n \in \mathbb{N}_2$, then*

$$\lambda(n) = \operatorname{lcm}(\lambda(p_1^{e_1}), \ldots, \lambda(p_r^{e_r})).$$

Remarks

3. The Carmichael function for powers of 2 (proof as **exercise** or in Appendix A.1):

   $$\lambda(2) = 1, \quad \lambda(4) = 2, \quad \lambda(2^e) = 2^{e-2} \quad \text{for } e \geq 3.$$

4. The Carmichael function for powers of odd primes (proof as **exercise** or in Appendix A.3):

   $$\lambda(p^e) = \varphi(p^e) = p^{e-1} \cdot (p - 1) \quad \text{for } p \text{ prime} \geq 3.$$

To prove the statement in Remark 2 we have to show that the multiplicative group mod p is indeed cyclic. We prove a somewhat more general standard result from algebra:

Proposition 2 *Let K be a field and $G \leq K^\times$ be a finite subgroup of order $\#G = n$. Then G is cyclic and consists exactly of the n-th roots of unity in K.*

Proof. For $a \in G$ we have $a^n = 1$, hence G is contained in the set of zeroes of the polynomial $T^n - 1 \in K[T]$. Thus K has exactly n different n-th roots of unity, and G contains all of them.

Now let m be the exponent of G, in particular $m \leq n$. Lemma 24 of Appendix A.10 yields that all $a \in G$ are even m-th roots of unity. Hence $n \leq m$, so $n = m$, and G has an element of order n. $\diamond$

1.4 Suitable Parameters for RSA

Proposition 3 *Let $n \geq 3$ be an integer. The following statements are equivalent:*

(i) *n is squarefree.*

(ii) *There exists an $r \geq 2$ with $a^r \equiv a \pmod{n}$ for all $a \in \mathbb{Z}$.*

(iii) **[RSA equation]** *For every $d \in \mathbb{N}$ and $e \in \mathbb{N}$ with $de \equiv 1 \pmod{\lambda(n)}$ we have $a^{de} \equiv a \pmod{n}$ for all $a \in \mathbb{Z}$.*

(iv) *For each $k \in \mathbb{N}$ we have $a^{k \cdot \lambda(n)+1} \equiv a \pmod{n}$ for all $a \in \mathbb{Z}$.*

Proof. "(iv) $\Longrightarrow$ (iii)": Since $de \equiv 1 \pmod{\lambda(n)}$, we have $de = k \cdot \lambda(n) + 1$ for some k. Hence $a^{de} \equiv a \pmod{n}$ for all $a \in \mathbb{Z}$.

"(iii) $\Longrightarrow$ (ii)": Since $n \geq 3$, we have $\lambda(n) \geq 2$. Choosing an arbitrary d with $\gcd(d, \lambda(n)) = 1$ and a corresponding e by congruence division $\mathrm{mod}\lambda(n)$ we get (ii) with $r = de$.

"(ii) $\Longrightarrow$ (i)": Assume there is a prime p with $p^2|n$. Then by (ii) we have $p^r \equiv p \pmod{p^2}$. But because of $r \geq 2$ we have $p^r \equiv 0 \pmod{p^2}$, contradiction.

"(i) $\Longrightarrow$ (iv)": By the chinese remainder theorem we only have to show that $a^{k \cdot \lambda(n)+1} \equiv a \pmod{p}$ for all prime divisors $p|n$.

Case 1: $p|a$. Then $a \equiv 0 \equiv a^{k \cdot \lambda(n)+1} \pmod{p}$.

Case 2: $p \nmid a$. Because of $p - 1|\lambda(n)$, we have $a^{\lambda(n)} \equiv 1 \pmod{p}$, hence $a^{k \cdot \lambda(n)+1} \equiv a \cdot (a^{\lambda(n)})^k \equiv a \pmod{p}$. $\diamond$

Corollary 1 *The RSA procedures work for a module n if and only if n is squarefree.*

To find suitable exponents d and e we have to know $\lambda(n)$ or, better yet (and necessarily as it will turn out) the prime decomposition of n. Then the procedure of key generation suggests itself:

1. Choose different primes $p_1, \ldots, p_r$ and form the module $n := p_1 \cdots p_r$.
2. Compute $\lambda(n) = \mathrm{lcm}(p_1 - 1, \ldots, p_r - 1)$ using the Euclidean algorithm.
3. Choose the public exponent $e \in \mathbb{N}_2$, coprime with $\lambda(n)$.
4. Compute the private exponent d with $de \equiv 1 \pmod{\lambda(n)}$ by congruence division.

Then take the pair (n, e) as public key, and the exponent d as private key.

Corollary 2 *Who knows the prime decomposition of n can compute the private key d from the public key (n, e).*

Practical Considerations

1. The usual choice is $r = 2$. Then the module has only two prime factors p and q that, as a compensation, are very large. Factoring this kind of integers $n = pq$ seems especially hard. It is crucial that the primes are chosen completely at random. Then an attacker has no hint for a guess.

2. For e we may choose a prime with $e \nmid \lambda(n)$, or a "small" integer say $e = 3$—more on the dangers of this choice later. A common standard choice is the prime $e = 2^{16} + 1$, provided $e \nmid \lambda(n)$. The binary representation of this integer contains only two 1's, making the binary power algorithm for enryption very fast. (For digital signature this is the verification of the signature.) However this choice of e doesn't make decryption (or generating a digital signature) more efficient.

3. After generating the keys we don't need p, q, and $\lambda(n)$ anymore, so we could destroy them.

   However: Since d is a "random" integer in the interval $[1 \ldots n]$ taking d-th powers is costly even with the binary power algorithm. It becomes somewhat faster when the owner of the private key computes $c^d \bmod p$ and $\bmod q$—using integers of about half the size—and then composes the result $\bmod n$ with the chinese remainder theorem. This procedure yields a small advantage in speed for decryption (or generating a digital signature).

4. Instead of $\lambda(n)$ we could use its multiple $\varphi(n) = (p-1)(q-1)$ for calculating the exponent.

   Advantage: We save (one) lcm computation.

   Drawback: In general we get a larger exponent d, slowing down each single decryption.

5. Notwithstanding Corollary 1 the RSA procedure works in a certain sense even if the module n is not squarefree. Decryption using the chinese remainder theorem is slightly more complex, involving an additional "Hensel lift." However decryption breaks down for plaintexts a that are multiples of a prime p with $p^2 | n$. Note that this effect is compatible with Corollary 1!

   The danger of hitting a plaintext divided by a multiple prime factor of n by chance is negligeable but grows with the number of prime factors. Even for a squarefree module n a plaintext divided by a prime factor would immediately yield a factorization of n, and hence reveal the private key.

Attention

The cryptanalytic approaches of the following chapter result in a set of side conditions that should be strictly respected when generating RSA keys.

Exercises

1. Let p and q be two different odd primes, and $n = p^2q$. Characterize the plaintexts $a \in \mathbb{Z}/n\mathbb{Z}$ that satisfy the RSA equation $a^{de} \equiv a \pmod{n}$. Generalize the result to arbitrary n.

2. Show that an integer $d \in \mathbb{N}$ is coprime with $\lambda(n)$ if and only if d is coprime with $\varphi(n)$.