Kryptografia - projekt
Algorytm RSA

Piotr Janus, Kamil Piszczek

1. Opis metody

Algorytm RSA jest jednym z pierwszych praktycznych kryptosystemów korzystających z koncepcji klucza publicznego i jest on powszechnie stosowany do zabezpieczenia transmisji danych. W tym kryptosystemie, klucz szyfrujący jest publiczny i różni się od klucza deszyfrującego, który jest tajny. Ta asymetria kluczy wynika z faktu, że niezwykle trudno jest rozłożyć na czynniki iloczyn dwóch bardzo dużych liczb pierwszych.

Nazwa algorytmu RSA pochodzi od pierwszych liter nazwisk jego twórców - Ron Rivesta, Adi Shamira oraz Leonarda Adlemana, którzy po raz pierwszy opisali go w 1977r.

Algorytm RSA jest stosunkowo wolnym algorytm i z tego powodu jest on rzadziej używany do bezpośrednio szyfrowania danych użytkownika. W większości przypadków, przy pomocy RSA przekazywane są klucze do symetrycznych algorytmów szyfrujących, które potem mogą być wykorzystywane do stosunkowo bezpiecznej wymiany danych z dużo wyższą prędkością.

1.1. Generacja kluczy

Zarówno klucz prywatny jak i publiczny, które wykorzystywane są w algorytmie RSA tworzone są według poniższej procedury:

1. Wybierz dwie różne liczby pierwszy p oraz q.
2. Oblicz $n = pq$. n jest modułem (ang. modulus) zarówno klucza prywatnego jak i publicznego. Oznacza również ich długość (zazwyczaj podawana w bitach).
3. Oblicz funkcję Eulera - tzw. tocjent: $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$
4. Wybierz liczbę e taką że: $1 < e < \phi(n)$ oraz $NWD(e, phi(n)) = 1$.
5. Ustal liczbę d taką że:
$$de \equiv 1 \mod \phi(n). \tag{1}$$

Klucz publiczny definiowany jest jako para liczb (n, e), gdzie n to moduł klucza, natomiast e to jego wykładnik.

Klucz prywatny definiowany jest analogicznie jako para liczb (n, d), gdzie n to moduł klucza, natomiast d to jego wykładnik.

1.2. Szyfrowanie i deszyfrowanie

Załóżmy, że chcemy zaszyfrować wiadomość M. Na początku dzielimy ją na mniejsze m bloki według łatwego do odwrócenia schematu. Następnie obliczamy wartość zaszyfrowanej wiadomości c według wzoru:

$$c \equiv m^e \mod n \tag{2}$$

Załóżmy, że chcemy odszyfrować szyfrogram c i odzyskać wiadomość m. Obliczamy wartość odszyfrowanego fragmentu wiadomości według wzoru:

$$m \equiv c^d \mod n \tag{3}$$

1.3. Prosty przykład

Postępujemy według przepisu 1.1:

1. $p = 61$ oraz $q = 53$.
2. $n = 61 \times 53 = 3233$.
3. $\phi(3233) = (61 - 1)(53 - 1) = 3120$.
4. $e = 17$, ponieważ $1 < 17 < 3120$ oraz $NWD(3120, 17) = 1$.
5. $d = 2753$, ponieważ $2753 \times 17 \equiv 1 \mod 3120$.

Publiczny klucz to para: $(n, e) = (3233, 17)$. Prywatny klucz to para: $(n, e) = (3233, 2753)$.
Szyfrogram dla wiadomości $m = 65$ obliczamy z podanego wzoru (2):

$$m^e = 65^{17} \equiv 2790 \mod 3233 \tag{4}$$

Oryginalną wiadomość możemy odszyfrować za pomocą wzoru (3):

$$c^d = 2790^{2753} \equiv 65 \mod 3233 \tag{5}$$

1.4. Dowód poprawności

Dowód poprawności algorytmu RSA opiera się na małym twierdzeniu Fermata. Jeżeli p jest liczbą pierwszą i a nie jest podzielne przez p to wtedy:

$$a^{p-1} \equiv 1 \mod p \tag{6}$$

Chcemy pokazać, że $m^{ed} \equiv m \mod pq$ dla każdej liczby całkowitej m. Zgodnie z wcześniejszymi założeniami p oraz q są różnymi liczbami pierwszymi, natomiast e oraz d to liczby całkowite spełniające warunek (1). Ponieważ: $\phi(pq) = (p - 1)(q - 1)$, to na mocy (1) możemy stwierdzić, że dla każdego dodatniego i całkowitego h zachodzi równanie:

$$ed - 1 = h(p - 1)(q - 1) \tag{7}$$

Do zweryfikowania, czy liczby takie jak m czy m^{ed} przystają do siebie modulo pq wystarczy sprawdzić, czy przystają one modulo p oraz modulo q oddzielnie. Wynika to, z chińskiego twierdzenia o reszcie.

Do pokazania, że $m^{ed} \equiv m \mod p$ rozważymy dwa przypadki.

1. $m \equiv 0 \mod p$

   Wtedy:

   $$m^{ed} \equiv 0 \equiv m \mod p \tag{8}$$

2. $m \not\equiv 0 \mod p$

   Przekształcamy przy pomocy zależności (7) oraz twierdzenia Fermata (6):

$$m^{ed} = m^{ed-1}m = m^{h(p-1)(q-1)}m = (m^{p-1})^{h(q-1)}m \equiv 1^{h(q-1)}m \equiv m \mod p \tag{9}$$

Analogiczne rozumowanie można przeprowadzić dla czynnika q. Te dwie zależności dowodzą poprawności algorytmu RSA:

$$m^{ed} \equiv m \mod pq \tag{10}$$

1.5. Próby złamania

2. Metody użyte wewnątrz algorytmu

2.1. Generacja bardzo dużych liczb pierwszych

2.2. Odwrotność modulo

Podczas operacji generowania kluczy (sekcja 1.1) konieczne jest wyznaczenie liczby całkowitej x spełniającej następujący warunek:

$$a \equiv x^{-1} \mod n \tag{11}$$

Równanie to może zostać przekształcone do następującej postaci:

$$ax \equiv 1 \mod n \tag{12}$$

Problem ten może zostać sprowadzony do *Rozszerzonego Algorytmu Euklidesa*, który wyznacza liczby x i y z następującego równania:

$$ax + by = gcd(a, b) \tag{13}$$

Równanie (12) może zostać doprowadzone do powyższej postaci poprzez następujące przekształcenia:

$$ax - 1 = qn \tag{14}$$

$$ax - qn = 1 \tag{15}$$

Powyższe równanie ma postać zgodną ze wzorem (13), w tym przypadku dane jest a oraz n, natomiast niewiadomymi są x i q. Należy zwrócić uwagę, że do rozwiązania problemu (11) konieczna jest tylko wartość x.

Pierwszym krokiem *Rozszerzonego Algorytmu Euklidesa* jest zdefiniowanie następujących oznaczeń:

$$\begin{array}{ll} r_0 = a & r_1 = b \\ s_0 = 1 & s_1 = 0 \\ t_0 = 0 & t_1 = 1 \end{array}$$

Kolejne iteracje przebiegają następująco:

$$\begin{array}{c} r_{i+1} = r_{i-1} - q_i r_i = b \\ s_{i+1} = s_{i-1} - q_i s_i = 0 \\ t_{i+1} = t_{i-1} - q_i t_i = 1 \end{array}$$

Gdzie q_i definiujemy jako iloraz liczb r_{i-1} oraz r_i. Warunkiem stopu jest $r_{k+1} = 0$, w takim przypadku szukane wartości x i y to odpowiednio s_k oraz t_k.

2.3. Największy wspólny dzielnik dla dużych liczb

Największy wspólny dzielnik liczb a i b oznaczany jako $gcd(a, b)$ może zostać wyznaczony między innymi algorytmem binarnym. Został on wybrany ze względu na łatwość implementacji i wydajność, w przeciwieństwie do innych metod (np. metody Euklidesa) wymaga jedynie dzielenia przez 2. Algorytm został przedstawiony w postaci pseudokodu (funkcja *gcd*).

2.4. Operacja szyfrowania

Zgodnie z opisem w sekcji 1.2 szyfrowanie polega na przeprowadzeniu operacji:

$$c \equiv b^e \mod m \tag{16}$$

W celu poprawienia wydajności można do tego celu użyć metody binarnej, która została przedstawiona postaci pseudokodu (funkcja *modPow*).

```
function gcd(a, b)
    d := 0
    while (a mod 2 == 0) and (b mod 2 == 0) do
        a := a/2
        b := b/2
        d := d + 1
    while a ≠ b do
        if a mod 2 == 0 then
            a := a/2
        else if b mod 2 == 0 then
            b := b/2
        else if a > b then
            a := (a − b)/2
        else
            b := (b − a)/2
    return a · 2^d
```

```
function modPow(b, e, m)
    if m == 1 then
        return 0
    res := 1
    b := b mod m
    while e > 0 do
        if e mod 2 == 1 then
            res := (res · b) mod m
        e := e/2
        b := (b · b) mod m
    return res
```

2.5. Operacja deszyfrowania

Podobnie jak operacja szyfrowania, także operacja deszyfrowania w przypadku dużych liczb może być bardzo złożona obliczeniowo. Jedną z możliwości jest wykorzystanie metody analogicznej jak przy szyfrowaniu. Warto zwrócić uwagę, że w przypadku deszyfrowania znamy liczby p i q które zostały wykorzystane do generacji kluczy (sekcja 1.1). Obliczenia mogą zostać dodatkowo przyspieszone poprzez wykorzystanie *Chińskiego twierdzenia o resztach*, przyjmując oznaczenia jak w sekcji 1.2 możemy odczytać wiadomość m stosując następujące przekształcenia:

$$
\begin{aligned}
&d_p = d \mod (p-1)\\
&d_q = d \mod (q-1)\\
&q_{inv} = q^{-1} \mod p
\end{aligned}
$$

$$
\begin{aligned}
&m_1 = c^{d_p} \mod p\\
&m_2 = c^{d_q} \mod q\\
&h = (q_{inv} \cdot (m_1 - m_2)) \mod p\\
&m = m_2 + h \cdot q
\end{aligned}
$$

Do obliczenia wartości q_{inv} możemy użyć metody opisanej w sekcji 2.2, natomiast wartości m_1 oraz m_2 mogą zostać wyznaczone przy pomocy metody binarnej opisanej w poprzednie sekcji.

3. Implementacja

3.1. Użyty język i biblioteki

3.2. Funkcjonalność

3.3. Interfejs użytkownika