

# 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 modulem (ang. modulus) zarówno klucza prywatnego jak i publicznego. Oznacza również ich długość (zazwyczaj podawana w bitach).
3. Oblicz funkcję Eulera - tzw. tojent:  $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$
4. Wybierz liczbę  $e$  taką że:  $1 < e < \phi(n)$  oraz  $NWD(e, \phi(n)) = 1$ .
5. Ustal liczbę  $d$  taką że:

$$de \equiv 1 \pmod{\phi(n)}. \quad (1)$$

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:

$$c \equiv m^e \pmod{n} \quad (2)$$

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

$$m \equiv c^d \pmod{n} \quad (3)$$

### 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  $2753 \times 17 \equiv 1 \pmod{3120}$ .

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):

$$m^e = 65^{17} \equiv 2790 \pmod{3233} \quad (4)$$

Oryginalną wiadomość możemy odszyfrować za pomocą wzoru (3):

$$c^d = 2790^{2753} \equiv 65 \pmod{3233} \quad (5)$$

### 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:

$$a^{p-1} \equiv 1 \pmod{p} \quad (6)$$

Chcemy pokazać, że  $m^{ed} \equiv m \pmod{pq}$  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:

$$ed - 1 = h(p-1)(q-1) \quad (7)$$

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  $m^{ed} \equiv m \pmod{p}$  rozważymy dwa przypadki.

1.  $m \equiv 0 \pmod{p}$

Wtedy:

$$m^{ed} \equiv 0 \equiv m \pmod{p} \quad (8)$$

2.  $m \not\equiv 0 \pmod{p}$

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

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

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

$$m^{ed} \equiv m \pmod{pq} \quad (10)$$

## 2. Metody użyte wewnątrz algorytmu

### 2.1. Generacja bardzo dużych liczb pierwszych

Sprawdzenie czy dana liczba jest pierwsza w przypadku liczb o rozmiarze rzędu kilkuset bitów jest operacją niezwykle złożoną obliczeniowo. W celu optymalizacji stosuje się bardziej zaawansowane testy pierwszości. Przykładem może być test *Millera-Rabina* wykorzystywany w wielu implementacjach.

Założmy że chcemy wykonać test na nieparzystej liczbie  $n$ , metoda składa się z następujących kroków:

1. Zdefiniuj parametr  $k$  określający dokładność testu (zazwyczaj przyjmuje się  $k = 100$ ).
2. Wyznacz maksymalną potęgę dwójki dzielącą  $n - 1$  i zastosuj równość  $n - 1 = 2^s \cdot d$ .
3. Powtórz  $k$  razy:
  - wybierz losowe  $a$  ze zbioru  $\{1, 2, \dots, n - 1\}$
  - jeżeli  $a^d \not\equiv 1$  i  $a^{2^{rd}} \not\equiv -1 \pmod n$  dla każdego  $r$  ze zbioru  $\{1, 2, \dots, s - 1\}$  badana liczba jest **złożona**
4. Test przeszedł pomyślnie, badana liczba jest **prawdopodobnie pierwsza**

W przypadku przedstawionej metody, prawdopodobieństwo, że liczba złożona zostanie zaklasyfikowana jako pierwsza wynosi  $4^{-k}$ . Generacja polega na wylosowaniu dowolnej liczby i sprawdzeniu czy jest pierwsza, czynność ta jest powtarzana dopóki wynik testu nie będzie pozytywny.

### 2.2. Odwrotność modulo

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

$$a \equiv x^{-1} \pmod n \quad (11)$$

Równanie to może zostać przekształcone do następującej postaci:

$$ax \equiv 1 \pmod n \quad (12)$$

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

$$ax + by = \gcd(a, b) \quad (13)$$

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

$$ax - 1 = qn \quad (14)$$

$$ax - qn = 1 \quad (15)$$

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:

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

Kolejne iteracje przebiegają następująco:

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

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  $\gcd$ ).

---

```
1: function  $\gcd(a, b)$ 
2:    $d := 0$ 
3:   while  $(a \bmod 2 == 0)$  and  $(b \bmod 2 == 0)$  do
4:      $a := a/2$ 
5:      $b := b/2$ 
6:      $d := d + 1$ 
7:   while  $a \neq b$  do
8:     if  $a \bmod 2 == 0$  then
9:        $a := a/2$ 
10:    else if  $b \bmod 2 == 0$  then
11:       $b := b/2$ 
12:    else if  $a > b$  then
13:       $a := (a - b)/2$ 
14:    else
15:       $b := (b - a)/2$ 
16:  return  $a \cdot 2^d$ 
```

---

### 2.4. Operacja szyfrowania

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

$$c \equiv b^e \pmod{m} \quad (16)$$

W celu poprawienia wydajności można do tego celu użyć metody binarnej, która została przedstawiona postaci pseudokodu (funkcja  $\text{modPow}$ ).

---

```
1: function  $\text{modPow}(b, e, m)$ 
2:   if  $m == 1$  then
3:     return 0
4:    $res := 1$ 
5:    $b := b \bmod m$ 
6:   while  $e > 0$  do
7:     if  $e \bmod 2 == 1$  then
8:        $res := (res \cdot b) \bmod m$ 
9:      $e := e/2$ 
10:     $b := (b \cdot b) \bmod m$ 
11:  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:

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

$$m_1 = c^{d_p} \bmod p$$

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

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 poprzedniej sekcji.

### 3. Implementacja

#### 3.1. Użyty język i biblioteki

Implementacja algorytmu RSA została wykonana w języku *JAVA*, napisany program umożliwia przesyłanie zaszyfrowanych wiadomości tekstowych pomiędzy dwoma komputerami. Do obliczeń matematycznych wykorzystano bibliotekę *BigInteger*, która daje możliwość operowania na bardzo dużych liczbach całkowitych (o szerokości kilkuset bitów). Komunikacja odbywa się z wykorzystaniem protokołu *TCP/IP*. Graficzny interfejs użytkownika został wykonany w technologii *JavaFX*. Dzięki zastosowanemu podejściu, program jest uniwersalny i może zostać z łatwością uruchomiony na każdym systemie operacyjnym.

#### 3.2. Funkcjonalność

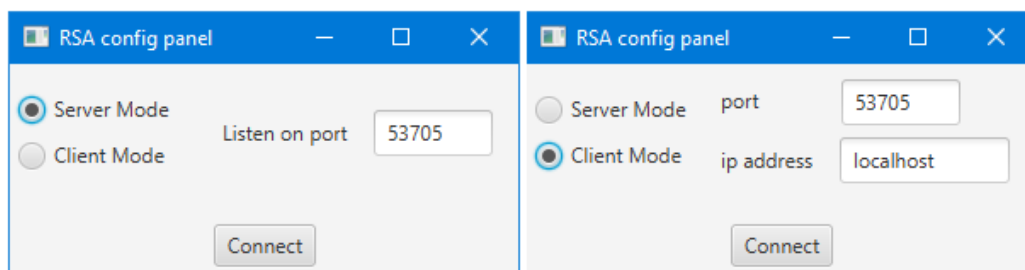
Jak zostało wspomniane w poprzednim podpunkcie, aplikacja służy do przesyłania wiadomości tekstowych. Jej działanie może zostać przetestowane poprzez uruchomienie na dwóch komputerach znajdujących się w jednej sieci lub z wykorzystaniem jednego komputera i lokalnego hosta (dokładny opis w rozdziale 3.3).

Program może zostać uruchomiony w trybie klienta lub serwera, przesyłanie wiadomości jest możliwe oczywiście w obu kierunkach. Dodatkowo w każdym kierunku komunikacji mamy możliwość określenia innej długości klucza szyfrującego. Po nawiązaniu połączenia następuje generacja klucza publicznego i prywatnego po obu stronach, następnie strony wymieniają się kluczami publicznymi. Od tego momentu zarówno serwer jak i klient mogą przysyłać szyfrowane wiadomości.

W sekcji 1.2 został przedstawiony schemat szyfrowania pojedynczego bloku danych, w przypadku napisanej aplikacji istnieje możliwość wysyłania wiadomości tekstowych o dowolnej długości. Napisany tekst jest dzielony na równe bloki (ostatni jest oczywiście dopełniany zerami), długość takiego bloku jest na nowo losowana dla każdej kolejnej wiadomości. Podczas szyfrowania zgodnie ze wzorem (2) konieczne jest dodatkowo dopełnienie otrzymanej wartości zerowymi bitami tak, aby jej długość była równa długości klucza  $n$  (sekcja 1.1). Dzięki temu każdy zaszyfrowany blok ma identyczną długość i zaszyfrowana wiadomość może być łatwo podzielona przez odbiorcę.

W oknie aplikacji oprócz podglądu wiadomości w postaci normalnego tekstu, dostępna jest także jej zakodowana postać (w tym przypadku wiadomość jest wyświetlana jako ciąg bajtów). Dodatkowo w oknie komunikatów wyświetlane są informacje o przychodzących i wysyłanych wiadomościach oraz klucze wykorzystywane do szyfrowania.

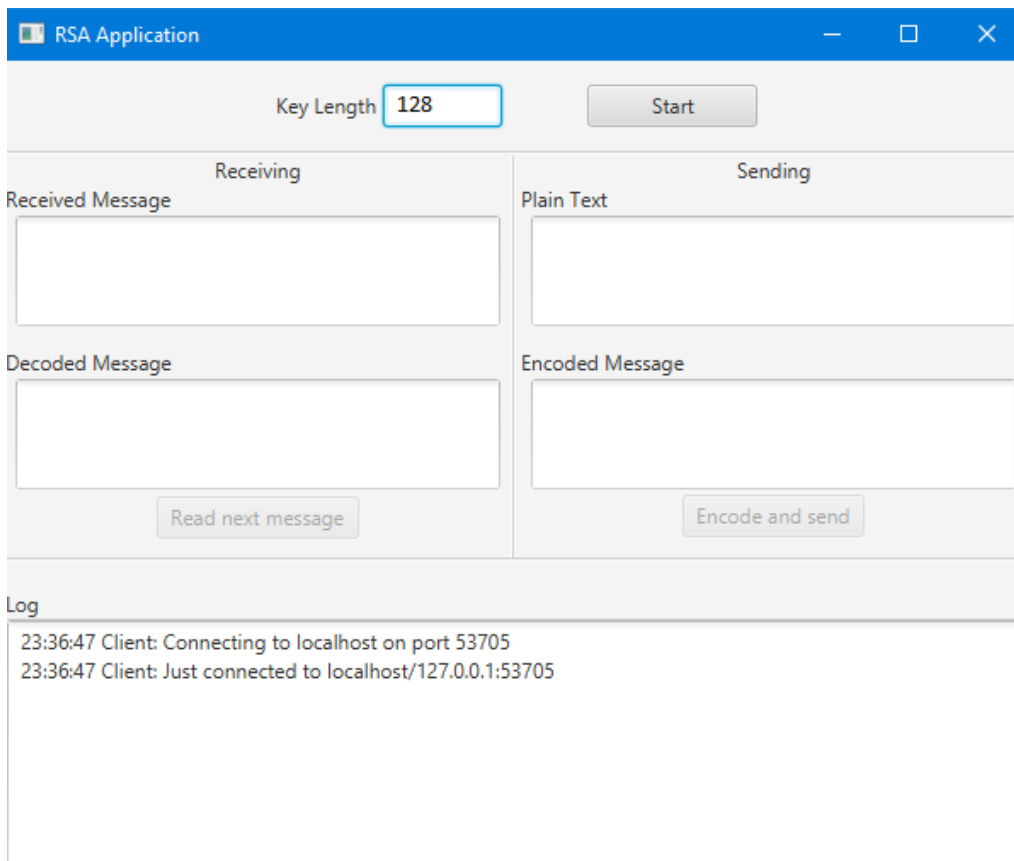
#### 3.3. Manual



Rysunek 1: Panel Konfiguracyjny w zależności od wybranego trybu

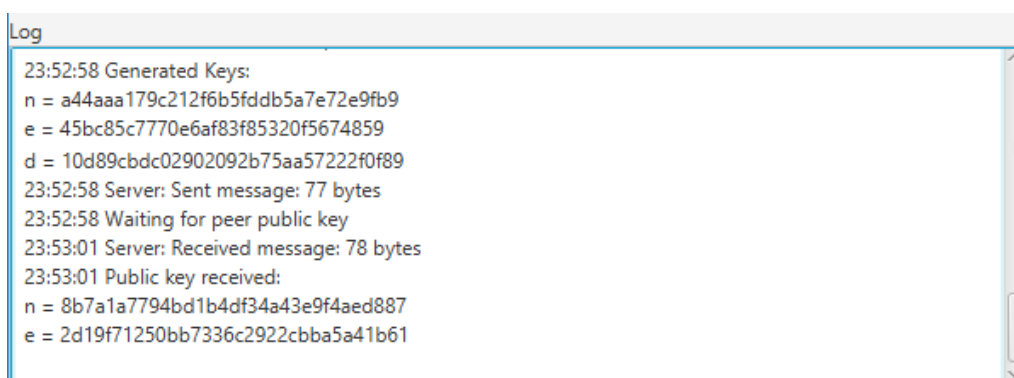
W celu przetestowania aplikacji należy postępować zgodnie z poniższą instrukcją:

1. Program należy uruchomić (plik *rsa.jar*, potrzebna *JAVA* w wersji 1.8 lub nowszej) na dwóch komputerach znajdujących się w jednej sieci lub dwa razy na jednym komputerze (okno aplikacji nie jest zbyt duże, zatem bez problemu można operować na dwóch jednocześnie).



Rysunek 2: Główne okno programu

2. Po uruchomieniu ukaże nam się ekran przedstawiony na rysunku 1. Najpierw w jednej aplikacji wybieramy *Server Mode* i naciskamy *Connect*, następnie drugą aplikację uruchamiamy w trybie *Client Mode*.
3. Po tej czynności ukaże nam się główne okno programu (rysunek 2). W polu na dole powinien pojawić się komunikat o pomyślnym nawiązaniu połączenia. W tym momencie należy w obu aplikacjach zdefiniować długość klucza i nacisnąć przycisk *Start*.
4. Gdy w oknie komunikatów obu aplikacji pojawi się informacja o wygenerowaniu kluczy oraz o otrzymaniu klucza publicznego od drugiego użytkownika (rysunek 3), możemy przejść do wysyłania i odbierania wiadomości. W celu wysłania wiadomości należy wpisać tekst w odpowiednie pole i nacisnąć przycisk *Encode and send*. Następnie można odczytać wysłaną wiadomość w drugiej aplikacji, poprzez naciśnięcie przycisku *Read next message*. Otrzymane wiadomości są przechowywane w kolejce *FIFO*.



Rysunek 3: Przykładowy log zawierający informacje o wygenerowanych kluczach