Laboratorium II

Wstęp teoretyczny

Kryptosystem RSA jest jednym z najbardziej znanych i powszechnie stosowanych algorytmów kryptografii asymetrycznej. Jego nazwa pochodzi od inicjałów nazwisk twórców: Ronalda Rivesta, Adiego Shamira i Leonarda Adlemana, którzy opracowali go w 1977 roku. RSA opiera się na teorii liczb, w szczególności na trudności faktoryzacji dużych liczb całkowitych, co zapewnia jego bezpieczeństwo w praktycznych zastosowaniach.

Podstawą działania RSA jest wykorzystanie właściwości liczb pierwszych oraz arytmetyki modularnej. W kryptografii asymetrycznej każda ze stron komunikacji posiada dwa klucze: **klucz publiczny**, który jest udostępniany wszystkim i służy do szyfrowania wiadomości, oraz **klucz prywatny**, który jest tajny i służy do deszyfrowania wiadomości. Dzięki temu możliwe jest bezpieczne przesyłanie informacji bez konieczności uprzedniego uzgadniania wspólnego klucza.

Generowanie kluczy RSA

Wybieramy dwie duże liczby pierwsze p i q. W praktycznych zastosowaniach są to liczby o długości kilkuset lub kilku tysięcy bitów. Bezpieczeństwo RSA opiera się na tym, że choć łatwo jest pomnożyć te liczby, to bardzo trudno jest rozłożyć ich iloczyn na czynniki pierwsze. Obliczamy

$$n = p \times q$$

Moduł $ n $ będzie częścią klucza publicznego i prywatnego i będzie używany w operacjach potęgowania modularnego. Obliczamy wartość funkcji Eulera:

$$\phi(n) = (p-1)(q-1)$$

Funkcja Eulera $\phi(n)$ reprezentuje liczbę liczb naturalnych mniejszych od $ n $, które są względnie pierwsze z $ n $. Jest to kluczowy element w procesie tworzenia klucza prywatnego.

Wybieramy liczbę całkowitą e taką, że $1 < e < \phi(n)$ oraz $\gcd(e, \phi(n)) = 1$, czyli e jest względnie pierwsze z $\phi(n)$. Najczęściej wybieraną wartością jest $e = 65537$, ponieważ jest to liczba pierwsza, która zapewnia dobre właściwości bezpieczeństwa i wydajności.

Obliczamy multiplikatywną odwrotność $emodulo\phi(n)$, czyli liczbę d spełniającą: ś

$$e \times d \equiv 1 \mod \phi(n)$$

Po tych krokach mamy:

- **Klucz publiczny**: para (n, e), którą możemy udostępnić wszystkim.
- **Klucz prywatny**: para (n, d), którą musimy zachować w tajemnicy.

Proces szyfrowania wiadomości za pomocą klucza publicznego przebiega następująco:

1. **Przygotowanie wiadomości**:

   Wiadomość tekstową konwertujemy na liczbę całkowitą m, taką że $0 < m < n$. Może to wymagać zastosowania odpowiedniego kodowania (np. UTF-8) i ewentualnego podziału wiadomości na bloki, jeśli jest zbyt długa.

2. **Szyfrowanie**:

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

   Obliczamy szyfrogram c poprzez podniesienie m do potęgi e modulo n.

Proces deszyfrowania wiadomości za pomocą klucza prywatnego:

$$m \equiv c^d \mod n$$

Obliczamy oryginalną wiadomość m poprzez podniesienie c do potęgi d modulo n. Konwertujemy liczbę m z powrotem na tekst, stosując odwrotne kodowanie.

Dlaczego to działa?

Kluczowym elementem jest fakt, że operacje szyfrowania i deszyfrowania są wzajemnie odwracalne dzięki właściwościom arytmetyki modularnej i konstrukcji kluczy. Ponieważ $e \times d \equiv 1 \mod \phi(n)$, mamy:

$$(m^e)^d \equiv m^{e \times d} \equiv m^{k \times \phi(n)+1} \equiv m \times (m^{\phi(n)})^k \mod n$$

Zgodnie z **małym twierdzeniem Fermata** lub **twierdzeniem Eulera**, wiemy, że $m^{\phi(n)} \equiv 1 \mod n$ dla m względnie pierwszego z n. W ten sposób otrzymujemy m po deszyfrowaniu.

Bezpieczeństwo RSA opiera się na dwóch głównych problemach matematycznych:

1. **Trudności faktoryzacji dużych liczb całkowitych**:

   Gdy n jest iloczynem dwóch dużych liczb pierwszych, jego faktoryzacja (czyli znalezienie p i q) jest problemem trudnym obliczeniowo. Znane algorytmy faktoryzacji mają złożoność super-poliomialną, co oznacza, że czas potrzebny do faktoryzacji rośnie bardzo szybko wraz ze wzrostem długości n.

2. **Niemożności obliczenia funkcji Eulera** $\phi(n)$ bez znajomości p i q:

   Znajomość $\phi(n)$ jest kluczowa do obliczenia klucza prywatnego d. Jeśli atakujący mógłby obliczyć $\phi(n)$ bez faktoryzacji n, mógłby złamać system. Jednak obliczenie $\phi(n)$ bez znajomości czynników n jest uważane za równie trudne jak sama faktoryzacja.

Wyzwanie

Wiedząc, że klucz publiczny w kryptosystemie RSA składa się z pary liczb: `n = 140115e871b5a6f, e = 10001` odszyfruj wiadomość:

`63a584ee99130 cd21c3e55366ee d528a0b38d218b 10a9dac5fee040d`

Wszystkie wartości podane są w systemie szesnastkowym.

Do testów swojego rozwiązania możesz wykorzystać liczby pierwsze zebrane na stronie: https://t5k.org/curios/.

Łamanie kryptosystemu RSA dla małych liczb pierwszych

Zadanie 1 - przygotowanie parametrów

Napisz skrypt w Pythonie, który dla zadanych dwóch czterocyfrowych liczb pierwszych p i q:

- Oblicza moduł $n = p \times q$.
- Oblicza funkcję Eulera $\phi(n) = (p-1)(q-1)$.
- Wybiera klucz publiczny e, taki że $1 < e < \phi(n)$ i $\gcd(e, \phi(n)) = 1$.
- Oblicza klucz prywatny d, taki że $d \equiv e^{-1} \mod \phi(n)$ (inaczej $de \equiv 1 \mod \phi(n)$).

Zadanie 2 - implementacja szyfrowania

Napisz skrypt, który przyjmuje wiadomość (np. liczbową reprezentację tekstu) a następnie szyfruje ją za pomocą klucza publicznego (n, e) według wzoru $c \equiv m^e \mod n$.

Zadanie 3 - implementacja deszyfrowania

Napisz skrypt, który przyjmuje zaszyfrowaną wiadomość c, a następnie deszyfruje ją za pomocą klucza prywatnego (n, d) według wzoru $m \equiv c^d \mod n$.

Zadanie 4 - łamanie kryptosystemu RSA

Napisz skrypt, który dla danego modułu n znajduje czynniki pierwsze p i q (np. metodą prób podziału), oblicza $\phi(n)$ oraz klucz prywatny d, a następnie deszyfruje zaszyfrowaną wiadomość bez znajomości oryginalnego klucza prywatnego.

Uwagi do implementacji w Pythonie

- Potęgowanie odulo

  ```
  pow(base, exponent, modulus)
  # przykłady:
  c = pow(m, e, n)
  m = pow(c, d, n)
  ```

- Znajdowanie odwrotności

  ```
  d = pow(e, -1, phi_n)
  ```

- Największy wspólny dzielnik - funkcja a modułu `math`

  ```
  import maath
  gcd = math.gcd(e, phi_n)
  ```

- Konwersja tekstu na liczby

  ```
  m = int.from_bytes(message.encode('utf-8'), 'big')
  ```

- Dekodowanie liczb na tekst

  ```
  message = m.to_bytes((m.bit_length() + 7) // 8, 'big').decode('utf-8')
  ```