Podstawy kryptografii

Kryptografia asymetryczna

Adam Olech

15 grudnia 2021

Spis treści

1 Wstęp

Informacje na temat wykonywanego ćwiczenia:

- **Język programowania**: Python
- **Interpreter**: CPython (Python 3.9.9)

Kod dostępny jest w repozytorium pod adresem `https://github.com/extinctpotato/put-crypto-lab`

1.1 Maksymalna wartość liczbowa

Wykorzystywany interpreter nie narzuca żadnych ograniczeń jeśli chodzi o maksymalną wartość typu *integer*, a więc teoretycznie możliwe jest wykorzystywanie dowolnej wielkości liczb pierwszych tak długo, jak mieszczą się w pamięci komputera.

```
> python3
Python 3.9.9 (main, Nov 20 2021, 21:30:06)
[GCC 11.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 10**100
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>>
```

Listing 1: Wyjście interpretera

2 Krótki opis RSA

RSA (Rivest-Shamir-Adleman) jest publicznym systemem kryptograficznym opracowanym w latach '70 przez badaczy na Massachusetts Institute of Technology.

W publicznym systemie kryptograficznym klucz szyfrujący jest publiczny i jest różny od klucza deszyfrującego, który jest ściśle tajny.
Użytkownik algorytmu tworzy i ujawnia klucz publiczny który bazuje na dwóch dużych liczbach pierwszych.

Bezpieczeństwo RSA polega na praktycznej trudności faktoryzacji dwóch dużych liczb pierwszych. Obecnie nie istnieje żaden znany sposób by pokonać ten system, pod warunkiem że zostanie użyty odpowiednio duży klucz.

3 Szyfrowanie przy użyciu algorytmu RSA

3.1 Generacja pary kluczy

Aby wygenerować parę kluczy (publiczny i prywatny), przygotowano następujące liczby pierwsze:

- **p**: 1013
- **q**: 2011

Wspomniane wyżej liczby zostały podane jako wejście na funkcję implementującą algorytm generujący trzy składniki pary kluczy.

```
def generate_rsa_keypair(p=None, q=None):
    # Prime numbers 'p' and 'q' must be irrecoverably
    # discarded after performing all calculations.
    p = p or randprime(0, 1000)
    q = q or randprime(0, 1000)

    # The 'n' product is used both for private and public key.
    n = p*q

    totient = (p-1)*(q-1)

    while gcd((e := randprime(0, totient)), totient) != 1:
        pass

    d = modular_multiplicative_inverse(
            a=e,
            m=totient
            )

    return e, d, n
```

Listing 2: Implementacja generatora pary kluczy

Funkcja przyjmuje na wejściu liczby p i q, choć są to parametry opcjonalne.

W przypadku wywołania funkcji bez parametrów, zostaną użyte losowe liczby pierwsze uzyskane przy pomocy funkcji *randprime* z biblioteki **Sympy**.
W wyniku działania funkcji uzyskano następujące liczby:

- **e**: 1749151
- **d**: 11236
- **n**: 2037143

Para liczb e oraz n jest kluczem publicznym, a d oraz n kluczem prywatnym.

3.1.1 Wyznaczenie wartości e

Aby wyznaczyć wartość e, należy najpierw wyznaczyć tocjent (ang. *totient*).

Można to zrobić używając funkcji Carmichaela.

$$\lambda(pq) = lcm(p-1, q-1) \tag{1}$$

Alternatywą jest użycie funkcji Eulera

$$\varphi(pq) = (p-q)(q-1) \tag{2}$$

W przypadku tej pierwszej uzyskamy najmniejszy wspólny mnożnik liczb $p-1$ oraz $q-1$,

W przypadku, w którym nie dbamy, by była to liczba najmniejsza [1], dopuszczalne jest wykorzystanie metody Eulera (która da wielokrotność wartości uzyskanej funkcją Carmichaela).

Po wyznaczeniu tocjentu, wyznaczamy e taką, która spełnia poniższe warunki:

- liczba e jest pierwsza.
- liczba e jest większa od zera i mniejsza od tocjentu.
- największy wspólny dzielnik liczby e i tocjentu nie wynosi 1.

[1] Mniejsza liczba zapewnia mniej kosztowną obliczeniowo operację szyfrowania.

3.1.2 Wyznaczenie wartości d

Aby wyznaczyć d, należy znaleźć taką liczbę, która jest modularną odwrotnością multiplikatywną reszty z dzielenia e i tocjentu.

$$d \equiv e^{-1} \pmod{\lambda(n)} \tag{3}$$

Do tego celu można zastosować rozszerzony algorym Euklidesa.

```
def xea(a, b, s1=1, s2=0, t1=0, t2=1):
    q = a // b
    r = a % b

    s3 = s1 - q*s2
    t3 = t1 - q*t2

    if r == 0:
        return b, s2, t2

    return xea(b, r, s2, s3, t2, t3)
```

Listing 3: Implementacja rozszerzonego algorytmu Euklidesa

Ostatecznie:

```
def modular_multiplicative_inverse(a, m):
    _, s2, _ = xea(a, m)
    return s2 % m
```

Listing 4: Wyliczenie modularnej odwrotności

4 Szyfrowanie wiadomości

4.1 Opis metody

Operacje szyfracji i deszyfracji kolejno odbywają się przez wyliczenie reszty z dzielenia podniesionego do potęgi liczby jawnej lub zaszyfrowanej do liczby e lub d przez liczbę d.

```
def rsa_encrypt(msg, e, n):
    return pow(msg, e) % n

def rsa_decrypt(ciphertext, d, n):
    return pow(ciphertext, d) % n
```

Listing 5: Implementacja RSA

4.2 Szyfrowanie ciągów

Z racji, że algorytm RSA działa na liczbach całkowitoliczbowych, szyfrowanie ciągów wymaga zamiany takiego ciągu na odpowiednią reprezentację liczbową.

Biblioteka standardowa w Pythonie pozwala na zamianę ciągu znaków na typ *int*.

```
# Declare a string.
text_str = "Hello, world!"

# Encode to ASCII byte array.
text_bytes = text_str.encode(encoding='ascii')

# Interpret as integer, byte order is little endian.
text_int = int.from_bytes(msg.encode(), 'LITTLE')
```

Listing 6: Zamiana ciągu znaków na liczbę

Istotnym ograniczeniem jest maksymalna wartość liczby jawnej – nie może ona być większa od liczby n.

Liczbę zatem należy zamienić na wiele liczb w n-liczbowym systemie liczbowym. Dzięki temu, maksymalną wartością każdej takiej liczby będzie n.

```
def int_to_base(n, b):
    if n == 0:
        return [0]

    digits = []

    while n:
        digits.append(int(n % b))
        n //= b

    return digits[::-1]
```

Listing 7: Konwersja systemu liczbowego

Przy deszyfracji, operację należy odwrócić poprzez konwersję listy liczb na system dziesiętny.

```
def base_to_int(n, b):
    r = range(0, len(n))
    i = 0

    for n_idx, b_idx in zip(r, reversed(r)):
        i += n[n_idx] * pow(b, b_idx)

    return i
```

Listing 8: Konwersja z systemu n-liczbowego na dziesiętny

4.2.1 Przykład

Zaszyfrowano 50 pierwszych znaków pewnego cytatu Arystotelesa.

```
ARISTOTLE_QUOTE = """\
We should venture on the study of every kind of animal without distaste; \
for each and all will reveal to us something natural and something beautiful.
"""
```

Listing 9: Tekst jawny

Funkcje opisane w poprzedniej sekcji zostały użyte w następujący sposób:

```
def rsa_enc_test_func(arg):
    e, d, n = generate_rsa_keypair()
    encrypted = rsa_encrypt_str(arg.plaintext, e, n)
    decrypted = rsa_decrypt_str(encrypted, len(arg.plaintext), d, n)

    l.info(f'e: {e}, d: {d}, n: {n}')
```

Listing 10: Tekst jawny

Po uruchomieniu funkcji otrzymujemy następujący rezultat:

```
e: 727, d: 1063, n: 1769
We should venture on the study of every kind of an
```

Listing 11: Wynik działania

5 Podpis cyfrowy

5.1 Zasada działania

Wiadomość jawną można zaszyfrować zarówno za pomocą klucza publicznego jak i prywatnego.

Zaszyfrowanie za pomocą klucza publicznego odbiorcy gwarantuje, że tylko odbiorca, czyli posiadacz klucza prywatnego, jest w stanie odszyfrować tę wiadomość.

Zaszyfrowanie zaś za pomocą własnego klucza prywatnego powoduje, że każdy za pomocą jawnego klucza publicznego jest w stanie odszyfrować wiadomość. Z racji, że z założenia klucz prywatny jest tajny, możliwość udowodnienia, że za jego pomocą dana wiadomość została zaszyfrowana umożliwia poświadczenie, że tak w rzeczywistości było.

5.2 Podpisanie wiadomości przy użyciu własnej implementacji RSA

Implementacja testu wykonuje następujące kroki:

- generacja dwóch par kluczy publicznych (para A i para B).
- zaszyfrowanie wiadomości kluczami prywatnymi.
- odszyfrowanie wiadomości kluczami publicznymi.
- próba odszyfrowania wiadomości zaszyfrowanej kluczem A kluczem B.
- próba odszyfrowania wiadomości zaszyfrowanej kluczem B kluczem A..

```
keypair A: (4637, 11877, 23113)
keypair B: (31627, 3883, 70097)
A: We should venture on
B: We should venture on
A (with public B): \\\\\\\\\\\\\\\\\\\
B (with public A): \\\\\\\\\\\\\\\\\\\
```

Listing 12: Wynik działania

6 Wnioski

6.1 Trudne w realizacji elementy algorytmu

Podczas implementacji algorytmu RSA, trudność może sprawić poprawne zrozumienie oraz zaimplementowanie algorytmu służącego do wyznaczenia modularnej odwrotności multiplikatywnej.

6.2 Czynniki wpływające na bezpieczeństwo

Aby zapewnić maksymalne bezpieczeństwo, należy użyć dużych liczb pierwszych p i q, których iloczyn da odpowiednio duże n.

Jak dotąd nie opracowano jeszcze żadnego algorytmu wielomianowego, który byłby w stanie dokonać faktoryzacji tych liczb. Nie udowodniono jednak, jakoby miało to być niemożliwe.

Obecnie rekomendowanym rozmiarem modułu n jest 2048 bitów. Klucze o rozmiarze poniżej 300 bitów da się złamać komputerami z półki sklepowej.

6.3 Dwa zastosowania kryptografii asymetrycznej

Kryptografia asymetryczna pozwala na zarówno bezpieczne szyfrowanie wiadomości, jak i na weryfikację pochodzenia wiadomości przy użyciu mechanizmu podpisu cyfrowego.