Sprawozdanie
z realizacji zadania
Laboratotrium nr 1
”Wstęp do kryptoanalizy klucza publicznego”

st. kpr. pchor. Mateusz Chudzik

25 listopada 2020

Spis treści

1 Wstęp

Zadanie wykonane zostało w ramach zajęć laboratoryjnych z przedmiotu "Wstęp do kryptoanalizy klucza publicznego". Wszystkie zadania zrealizowałem według przesłanych wytycznych.

1.1 Wykonane oprogramowanie

Kody źródłowe zamierzam umieścić w prywatnym repozytorium w serwisie GitHub. `https://github.com/matchupikchu/Introduction_to_Public_Key_Cryptography`. Opisane tutaj pliki połączyłem w jeden oraz dołączyłem w mailu.

- **congruention.ipynb**, realizacja zadania 1
  Implementacja funkcji, która liczy kongruencje postaci:
  $x^a = b(modP)$
  Wymagania:
  1. Wejście: liczby całkowite a, b oraz liczba pierwsza P
  2. Wyjście: znaleziona wartość x
  3. funkcja nie może wykorzystywać funkcji wbudowanej środowiska SageMath, która realizuje taką samą funkcjonalność.
- **congruention_ solved.ipynb**, realizacja zadania 2
  W tym pliku przy wykorzystaniu funkcji congr1, która jest tożsama z funkcją **congr** napisaną w zadaniu nr 1 oraz funkcją **congr2**, która oblicza przypadki testowe, w ktorych była podana zapadka, rozwiązuje kongruencje oraz mierzę czas wykonania dla poszczególnych przypadków.
- **RSA_emulator.ipynb**, realizacja zadania 3
  W tym pliku umieściłem funkcję gen(), enc() oraz dec(), które realizują funkcjonalności opisane w tym zadaniu oraz emulator algorytmu RSA.

2 Realizacja zadania nr 1

2.1 Funkcja congr

Zrealizowałem ją w przypadku ogólnym, zakładając że użytkownik może się pomylić i niekoniecznie podać liczbę pierwszą jako P. Chcąc napisać ją w ten sposób, aby realizowała tą samą funkcjonalność, ale przy założeniu, że P jest pierwsze moglibyśmy zmienić linijkę, gdzie liczona jest wartość funkcji Eulera i moglibyśmy napisać phi = P - 1, ponieważ z własności funkcji Eulera, właśnie tyle ona wynosi dla liczb pierwszych.
Jeśli chodzi o samo działanie funkcji to po obliczeniu warości $\varphi(P)$, wyznaczam element odwrotny od a, przy wykorzystaniu funkcji **r_euclid**, ale modulo $\varphi(P)$, a następnie przy wykorzystaniu funkcji **fast_p** obliczam $b^{a^{-1}}(modP)$, dzięki czemu uzyskuje nasze poszukiwane x.

```
def congr(a, b, P):
    phi = euler_phi(P)
    aInv = (r_euclid(a, phi)[1] + phi) % phi
    x = fast_p(b, aInv, P)
    return x
```

3 Realizacja zadania nr 2

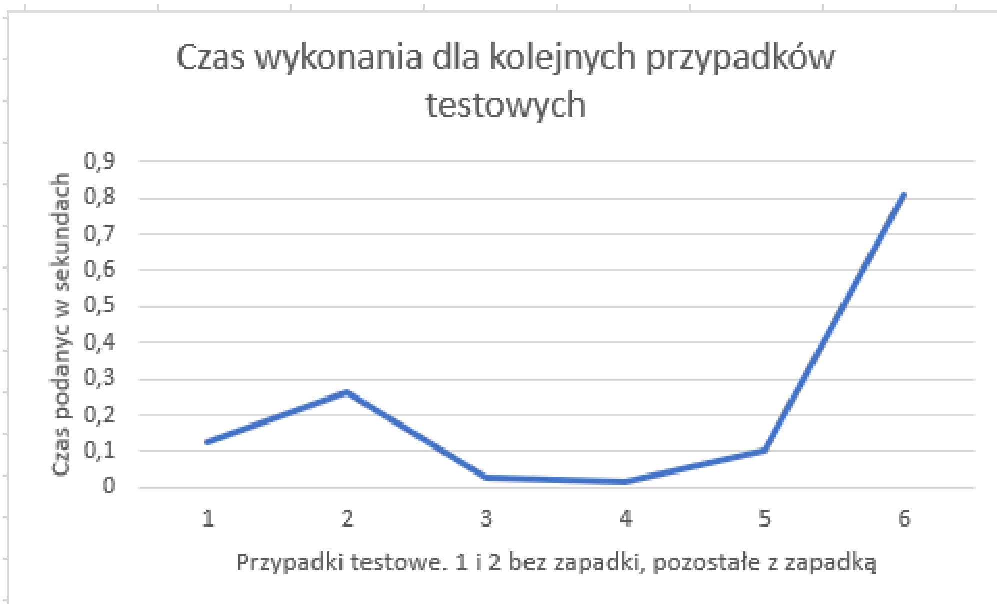
3.1 Funkcja congr1

Realizuje ona identyczną funkcjonalność co funkcja congr z zadania pierwszego, nazwę zmieniłem aby rozróżnić ją z funkcją congr2, którą za chwilę omówię.

3.2 Funkcja congr2

Napisałem ją do obliczenia przypadków z zapadką. Była mi do tego potrzebna kolejna funkcja, ponieważ na wejściu powinna być jeszcze podana zapadka, a pisanie oddzielnie kodu dla każdgo przypadku wydało mi się niepotrzebne. Cały kod rozwiązujący podane kongruencje umieściłem w pliku dołączonym do maila.
Od funkcji congr czy congr1 różni się tym, że wartość funkcji Eulera obliczam na podstawie podanej wartości zapadki (p) oraz wartości q, które jest równe N / p. Wiedząc, że N = p * q oraz p i q są liczbami pierwszymi możemy policzyć szybko φ(N), które jest równe w tym przypadku (p - 1) * (q - 1), co wynika z własności funkcji Eulera. Reszta działania się nie różni od funkcji congr i congr1.

Z wykresu możemy wnioskować, że kongruencja jest rozwiązywana w czasie wykładniczym. Poza tym jasno widać, że odzyskanie zapadki znacznie rozszerza dla nas możliwość rozwiązania kongruencji względem długości modułu N. Po



Rysunek 1: Wykres zależności czasu dla kolejnych przypadków po uruchomieniu

uruchomieniu skryptu napisanego w ramach tego zadania oprócz czasów obliczeń poszczególnych przypadków ustaliłem, że długości modułu w kolejnych przypadkach testowych wynosi:

1. 106 bitów
2. 136 bitów
3. 260 bitów
4. 268 bitów
5. 517 bitów
6. 774 bitów

4 Realizacja zadania nr 3

4.1 Funkcja gen()

Funkcja **gen()**, na podstawie wejściowej wartości m, która oznacza długośc modułu w bitach, generuje parę kluczy publiczny i prywatny.
Poniżej umieszczam pseudokod

Algorithm 1 gen()

Input: $m : intm$
Output: $(n, e), (n, d)$

$m1 = m//2$
$p = randprime(2^{(}m1 - 1), (2^{m}1) - 1)$
$q = randprime(2^{(}m1 - 1), (2^{m}1) - 1)$
while p == q **do**
    $q = randprime(2^{(}m1 - 1), (2^{m}1) - 1)$
end while
$n = p * q$
$phin = (p - 1) * (q - 1)$
while True **do**
    $e = randint(3, phin - 1)$
    $eukl = xgcd(e, phin)$
    if eukl[0] == 1 **then**
        $d = eukl[1]$
        $break$
    end if
end while
return $(n, e), (n, d)$

Funkcja generuje odpowiednie losowe liczby pierwsze p i q, różne od siebie nawzajem. Następnie obliczany jest ich iloczyn oraz wartość funkcj $\varphi(n)$, która na podstawie własności funkcji Eulera jest równa (p - 1) * (q - 1). Następnie w pętli losowana jest wartość e z przedziału 3 do $\varphi(n)$ - 1. Jeśli NWD(e, $\varphi(n)$) jest równy 1, to obliczana jest wartość elemntu odwrotnego do e mod($\varphi(N)$) (oznaczony w pseudokodzie jako d) oraz pętla jest przerywana, w innym wypadku pętla działa, aż do wylosowania odpowiedniego e.

4.2 Funkcja enc()

Funkcja **enc()** jej zadaniem jest szyfrowanie według algorytmu RSA. Na wejściu dostaje klucz publiczny oraz tekst jawny, a na wyjściu powinniśmy otrzymać szyfrogram.
Poniżej zamieszczam pseudokod.

Algorithm 2 enc()

Input: $(n, e), m$
Output: c

```
c = pow(m, e, n)
return c
```

Nie jest to skomplikowana funkcja, jej zadaniem jest tylko podniesienie tekstu jawnego do potęgi e, modulo n. Oczywiście, gdybyśmy działali np. na Stringu funkcja wygladałaby inaczej, jednak w celach edukacyjnych, dla zbadania złożoności czasowej algorytmu RSA, uważam że wystarczająca jest obsługa liczb całkowitych.

4.3 Funkcja dec()

Funkcja **dec()** jej zadaniem jest deszyfrowanie według algorytmu RSA. Na wejściu dostaje klucz prywatny oraz tekst zaszyfrowany, a na wyjściu powinniśmy otrzymać tekst jawny.
Poniżej zamieszczam pseudokod.

Algorithm 3 dec()

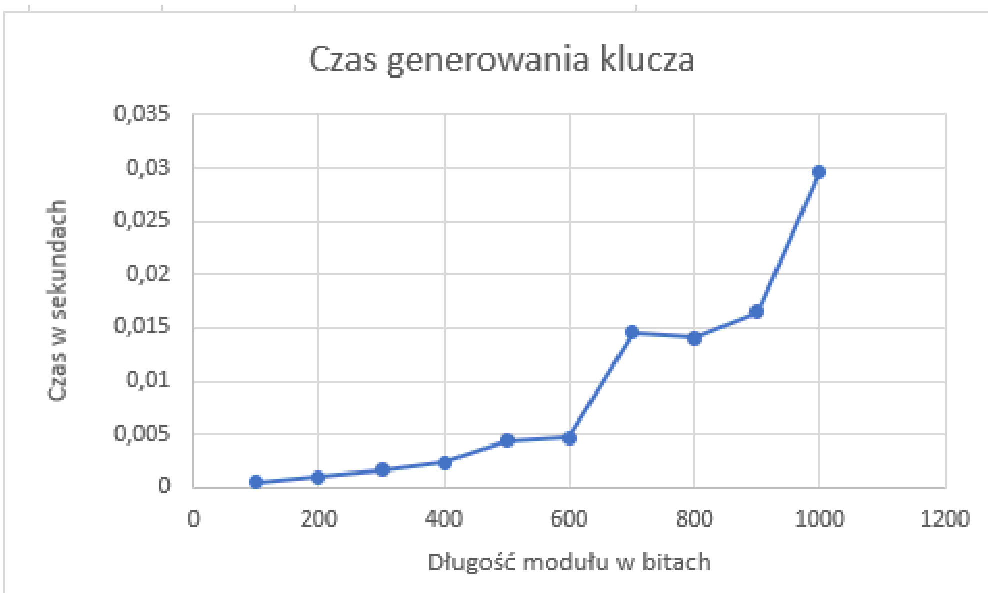
Input: $(n, d), c$
Output: m

```
m = pow(c, d, n)
return m
```

Tak samo jak funkcja **enc()** funkcja **dec()** realizuje prostą funkcjonalność podnoszenia modularnego. Tak na prawdę moglibyśmy zaimplementować funkcje **dec** poprzez odpowiednie użycie funkcji enc().

5 Czasy działania poszczególnych funkcji dla długości modułu od 100 do 1000 bitów

5.1 Analiza czasu generowana klucza, funkcja gen()



Rysunek 2: Wykres zależności czasu generowania klucza od długości modułu

Z wykresu możemy odczytać, że zależność czasu generowania kluczy od długości modułu w bitach jest wykładnicza. Co nie nie jest dobrą własnością, ponieważ opóźnia to działanie całego algorytmu. Na końcu dokumentu przeanalizuję zależności pomiędzy operacjami generowania kluczy, szyfrowania i deszyfrowania.

5.2 Analiza czasu szyfrowania, funkcja enc()

Rysunek 3: Wykres zależności czasu szyfrowania od długości modułu

Czas szyfrowana w zależności od długości modułu w bitach również wygląda na rosnący wykładniczo, choć nie tak szybko jak w przypadku generowania kluczy oraz te wartości czasowe są mniejsze dla poszczególnych długości modułu.

5.3 Analiza czasu deszyfrowania, funkcja dec()

Rysunek 4: Wykres zależności czasu deszyfrowania od długości modułu

Czas deszyfrowania wygląda jakby był w zależnoście wielomianowej od długości modułu w bitach. Co niewątpliwie jest warte uwagi, deszyfrowanie w przypadku, gdy uzyskamy w jakiś sposób klucz prywatny trwa rząd wielkości mniej niż generowanie kluczy.

6 Podsumowanie

Z umieszczonych wykresów oraz moich rozważań podczas ich analizy można dojść do wniosku, że szyfr RSA ma kilka kluczowych wad związanych z jego czasem dzialania.
Największym mankamentem tego algorytmu jest jego powolne generowanie kluczy, co napewno nie sprawdziłoby się jeśli chcielibyśmy szyfrować szybko jakiś pakiet danych, chyba że korzystalibyśmy z bazy danych wygenerowanych kluczy, z drugiej strony takie rozwiązanie generuje zagrożenia ze strony ataków na bazę danych kluczy. Poza tym drugie zadanie, gdzie obliczaliśmy kongruencję pokazało, że jeśli przeciwnik zdobyłby choć część klucza, to i tak bardzo mocno przyśpieszy to deszyfrowanie.