Protokoły kryptograficzne
Zarządzanie i Wymiana Kluczy
Zadanie laboratoryjne 1a.
Zadanie laboratoryjne 1b.

8 kwietnia 2021

1 Zadania

1. Zaimplementować w środowisku *sagemath* protokół Diffiego-Hellmana dla dwóch uczestników (zadanie 1a).
2. Zaimplementować w środowisku *sagemath* protokół Diffiego-Hellmana dla trzech uczestników (zadanie 1b).

2 Środowisko *sagemath*

Strona domowa środowiska *sagemath* to https://www.sagemath.org/. Można to środowisko zainstalować lokalnie lub korzystać z wersji online (CoCalc).
Środowisko *sagemath* wykorzystuje język programowania *python* (w zależności od wersji *sagemath* wesja *pythona* to 2. albo 3.).
Dokumentacja środowiska *sagemath* jest dostępna na stronie https://www.sagemath.org/help.html.

3 Wytyczne

W ramach zadań należy zaimplementować procedury realizujące następujące elementy:

1. generacja parametrów dziedziny.
2. generacja pary kluczy prywatny/publiczny
3. realizacja kolejnych kroków protokołu.

3.1 Generacja parametrów dziedziny

Parametry wejściowe:

- b – długość charakterystyki ciała p wyrażona w bitach.

Procedura powinna wygenerować:

- charakterystykę p ciała $\mathbb{F}_p$. Liczba pierwsza p powinna mieć b bitów i być postaci:

$$p = 2n + 1,$$

  gdzie n jest również liczbą pierwszą;

- generator g podgrupy cyklicznej (w grupie $\mathbb{F}_p^*$) rzędu n. Oznacza to, że n jest najmniejszą liczbą naturalną, dla której $g^n \equiv 1 \pmod{p}$.

Procedura powinna zwrócić wartości p, n i g.

3.2 Generacja pary kluczy prywatny/publiczny

Parametry wejściowe:

- p – charakterystyka ciała $\mathbb{F}_p$;
- n – rząd podgrupy cyklicznej;
- g – generator podgrupy cyklicznej rzędu n.

Procedura powinna wygenerować:

- klucz prywatny *kpriv* będący losową liczbą całkowitą z przedziału $< 1, n-1 >$;
- klucz publiczny *kpub* spełniający

$$kpub = g^{kpriv} \pmod{p}.$$

Procedura powinna zwrócić wartości *kpriv* i *kpub*.

3.3 Realizacja kolejnych kroków protokołu

Kolejne kroki protokołu powinny być zaimplementowane jako oddzielne procedury. Parametry wejściowe i wyjściowe wynikają z opisu tych kroków. Przykładowo dla kroku 1. tj.

1. Strona A generuje losową liczbę całkowitą $x_A \in \mathbb{Z}_n^*$ oblicza

$$K_A = g^{x_A} \pmod{p}$$

i wysyła K_A do strony B.

procedura realizująca ten krok powinna przyjąć jako parametry wejściowe:

- p – charakterystyka ciała $\mathbb{F}_p$;
- n – rząd podgrupy cyklicznej;
- g – generator podgrupy cyklicznej rzędu n.

Procedura powinna zwrócić: x_A i K_A.
Uwaga: Nie ma konieczności implementacji przesyłania danych pomiędzy stronami.

3.4 Polecenia środowiska *sagemath*

Polecenia środowiska *sagemath*, które można wykorzystać przy realizacji zadania:

- $previous\_prime(a)$ i $next\_prime(a)$ – funkcje znajdowania liczb pierwszych najbliższych liczbie a;
- $is\_prime(b)$ – sprawdzenie, czy liczba b jest pierwsza;
- $F = GF(p)$ – inicjacja ciała $\mathbb{F}_p$;
- $c = F.random\_element()$ – wygenerowanie losowego elementu c z ciała F;
- $c.multiplicative\_order()$ – rząd podgrupy (multiplikatywnej) generowanej przez element c ciała F;
- c^t – podniesienie elementu c do potęgi t. Jeżeli c jest elementem ciała F, to *sagemath* automatycznie wykonuje działania w ciele (redukcja modularna);
- $F(d)$ – rzutowanie zmiennej d (d – liczba całkowita) do elementu ciała F.

Przykład użycia:

```
sage: # ustalenie wartości b
sage: b = 10
sage: b
10
sage: # znalezienie liczby pierwszej mniejszej od 2^b
sage: p = previous_prime(2^b)
sage: p
1021
sage: # sprawdzenie pierwszości p
sage: is_prime(p)
True
sage: # zainicjowanie ciała GF(p)
sage: F = GF(p)
sage: F
Finite Field of size 1021
sage: # wygenerowanie losowego elementu ciała
sage: c = F.random_element()
sage: c
280
sage: # obliczenie rzędu podgrupy generowanej przez c
sage: c.multiplicative_order()
102
sage: # sprawdzenie, czy c^102 daje 1
sage: c^102
1
sage: # zainicjowanie liczby całkowitej o wartości 2000
sage: d = 2000
sage: # wypisanie typu dla d -- liczba całkowita
sage: type(d)
<type 'sage.rings.integer.Integer'>
sage: # rzutowanie d do ciała F
```

```
sage: e = F(d)
sage: e
979
sage: # wypisanie typu dla e -- element ciała skończonego
sage: type(e)
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: # przykład deklaracji procedury (dla n!)
sage: def silnia(n):
....:     if (n==0):
....:         wyn = 1
....:     else:
....:         wyn = n*silnia(n-1)
....:     return wyn
sage: silnia(10)
3628800
# przykład deklaracji procedury zwracającej dwa wyniki: w1 = a mod b i w2 = a div b
sage: def divrem(a, b):
....:     w1 = mod(a,b)
....:     w2 = a // b
....:     return w1, w2
....:
sage: r, q = divrem(20,3)
sage: r, q
(2, 6)
```

4 Rozliczenie zadań

W celu rozliczenia zadań należy przesłać w ramach narzędzia *MicrosoftTeams* jeden plik tekstowy (dla każdego zadania) zawierający:

- kody zaimplementowanych procedur;
- sekwencję wywołań procedur (z wypisaniem wartości wyjściowych):
  - generacji parametrów dziedziny (wartość b np. 512);
  - realizacji kolejnych kroków protokołu;

Na początku pliku w komentarzu proszę umieścić własne dane (imię, nazwisko, grupa szkoleniowa).

Zaimplementowane procedury jak i wykonywane kroki muszą byś opatrzone komentarzami (w kodzie źródłowym). Na ekranie muszą być wyświetlane informacje o wykonywanych krokach protokołu, wyznaczanych i przesyłanych wartościach. Jako wzór w rozdziale 5. przedstawione jest przykładowe zadanie z rozwiązaniem.

Wymagania dotyczące przesyłania rozwiązań

- Realizacja protokołu Diffiego-Hellmana dla dwóch uczestników rozliczana jest w ramach „Zadania 1a".
- Realizacja protokołu Diffiego-Hellmana dla trzech uczestników rozliczana jest w ramach „Zadania 1b".

5 Przykładowe zadanie z rozwiązaniem

5.1 Zadanie (przykładowe)

Zaimplementować poniżej opisany protokół wykorzystujący algorytm RSA, tj:

1. Wygenarować klucze dla systemu RSA.
2. Zaimplementować protokół **TEST**

5.1.1 Generacja kluczy RSA

Należy zaimplementować procedurę generującą parę kluczy $kpriv = (n, d)$ i $kpub = (n, e)$ na potrzeby algorytmu RSA. Parametrem wejściowym procedury jest pożądana długość (w bitach) modułu n. Procedura powinna zwrócić $kpriv$ i $kpub$.

5.1.2 Protokół TEST

Protokół **TEST**:

1. Stona A generuje losową wartość $rand$ z zakresu od 2 do $n-2$. Następnie A szyfruje $rand$ z wykorzystaniem algorytmu RSA i klucza publicznego $kpubB = (n, e)$ strony B:

   $$c = rand^e \pmod{n}$$

   i wysyła c do strony B.

2. Strona B odszyfrowuje $rand$ korzystając ze swojego klucza $kprivB = (n, d)$:

   $$rand = c^d \pmod{n}$$

   oblicza

   $$res = (rand - 1) \pmod{n}$$

   i odsyła res do strony A.

3. Strona A sprawdza, czy $res = (rand - 1) \pmod{n}$.

5.2 Implementacja generacji kluczy RSA

```
# Procedura generacji kluczy RSA
# wejście:
#   m - długość modułu (w bitach)
# wyjście:
#   n, e, d - elementy klucza publicznego i prywatnego
def genrsakeys(m):
    m1 = m // 2
    pom1 = randint(2,2^(m1-1))
    pom2 = 2^m1-pom1;
    p = next_prime(pom1)
    pom1 = randint(2,2^(m1-1))
    pom2 = 2^m1+pom1;
    q = next_prime(pom2)
    n = p*q
    phin = (p-1)*(q-1)
```

```
    while True:
        e = randint(3,phin-1)
        if gcd(e,phin)==1:
            d = inverse_mod(e,phin)
            break
    return n, e, d
```

5.3 Implementacja kolejnych kroków protokołu

5.3.1 Krok 1.

```
# Krok 1
# wejście:
#   n, e - klucz publiczny B
# wyjście:
#   rand - wygenerowana wartość losowa
#   c - wartość do wysłania
def krok1(n,e):
    rand = randint(2,n-2)
    c = power_mod(rand,e,n)
    return rand, c
```

5.3.2 Krok 2.

```
# Krok 2
# wejście:
#   n, d - klucz prywatny B
#   c - wartość otrzymana od A
# wyjście:
#   res - wartość do wysłania
def krok2(n,d,c):
    rand = power_mod(c,d,n)
    res = mod(rand-1,n)
    return res
```

5.3.3 Krok 3.

```
# Krok 3
# wejście:
#   n - moduł z klucza publicznego B
#   rand - wartość wygenerowana w kroku 1
#   res - wartość otrzymana od B
# wyjście:
#   informacjs "OK", jeśli res=(rand-1) mod n, "WRONG" w p.p
def krok3(n, rand, res):
    if res==mod(rand-1,n):
        print("    OK")
    else:
        print("    WRONG")
```

5.4 Postać pliku do wysłania

Nazwa pliku: NAZWISKO_labX.txt, gdzie X jest numerem kolejnych zajęć laboratoryjnych.
Jako parametry wejściowe można przyjąć dowolne ale sensowne wartości. W przykładzie przyjęto $m = 40$.
Uwaga:
W preambule należy umieścić wpis:

```
from __future__ import print_function
```

Natomiast do wypisywania tesktu na ekranie należy wykorzystać funkcję:

```
print(...)
```

Przykładowo:

```
print("Klucz prywatny: n, d =", n, d)
```

Proszę **nie używać**:

```
print "Klucz prywatny: n, d =", n, d
```

gdyż jest to poprawna forma dla pythona w wersji 2. Opisane powyżej rozwiązanie działa poprawnie dla pythona w wersji 2. i 3.

Zawartość pliku do wysłania:

```
# Imię Nazwisko grupa
from __future__ import print_function

# Procedura generacji kluczy RSA
# wejście:
#   m - długość modułu (w bitach)
# wyjście:
#   n, e, d - elementy klucza publicznego i prywatnego
def genrsakeys(m):
    m1 = m // 2
    pom1 = randint(2,2^(m1-1))
    pom2 = 2^m1-pom1;
    p = next_prime(pom1)
    pom1 = randint(2,2^(m1-1))
    pom2 = 2^m1+pom1;
    q = next_prime(pom2)
    n = p*q
    phin = (p-1)*(q-1)
    while True:
        e = randint(3,phin-1)
        if gcd(e,phin)==1:
            d = inverse_mod(e,phin)
            break
    return n, e, d

# Krok 1
# wejście:
```

```
#   n, e - klucz publiczny B
# wyjście:
#   rand - wygenerowana wartość losowa
#   c - wartość do wysłania
def krok1(n,e):
    rand = randint(2,n-2)
    c = power_mod(rand,e,n)
    return rand, c

# Krok 2
# wejście:
#   n, d - klucz prywatny B
#   c - wartość otrzymana od A
# wyjście:
#   res - wartość do wysłania
def krok2(n,d,c):
    rand = power_mod(c,d,n)
    res = mod(rand-1,n)
    return res

# Krok 3
# wejście:
#   n - moduł z klucza publicznego B
#   rand - wartość wygenerowana w kroku 1
#   res - wartość otrzymana od B
# wyjście:
#   informacjs "OK", jeśli res=(rand-1) mod n, "WRONG" w p.p
def krok3(n, rand, res):
    if res==mod(rand-1,n):
        print("    OK")
    else:
        print("    WRONG")

# Generacja kluczy
print("Generacja kluczy RSA dla B")
n, e, d = genrsakeys(40)
# wypisanie kluczy
# prywatny
print("  Klucz prywatny: n, d =", n, d)
# publiczny
print("  Klucz publiczny: n, e =", n, e)
# Realizacja protokołu
print("\n\nRealizacja protokołu")
# Krok 1 (strona A)
print("Krok 1 (A)")
print("  Obliczone wartości:")
rand, c = krok1(n,e)
# wypisanie wyników
# wygenerowana warość rand
print("    rand =", rand)
```

```
# wyznaczona wartość c
print("    c =", c)
# wyznaczona wartość c do wysłania
print("  Wysyłam do B c =", c)
# Krok 2 (strona B):
print("Krok 2 (B)")
print("  Obliczone wartości:")
res = krok2(n, d, c)
# wypisanie wyników
# wyznaczona wartość res
print("    res =", res)
# wyznaczona wartość res do wysłania
print("  Wysyłam do A res =", res)
# Krok 3 (strona A)
print("Krok 3 (A)")
print("  Obliczone wartości:")
krok3(n, rand, res)
```

5.5 Sprawdzenie poprawności

Przed wysłaniem można/należy sprawdzić wynik wykonania. Dla tego przykładu otrzymujemy:

```
Generacja kluczy RSA dla B
  Klucz prywatny: n, d = 605597530189 468511589485
  Klucz publiczny: n, e = 605597530189 470480402053


Realizacja protokołu
Krok 1 (A)
  Obliczone wartości:
    rand = 94405968384
    c = 399347920905
  Wysyłam do B c = 399347920905
Krok 2 (B)
  Obliczone wartości:
    res = 94405968383
  Wysyłam do A res = 94405968383
Krok 3 (A)
  Obliczone wartości:
    OK
```