Kriptografski sistem RSA

1. Splošna predstavitev problema

Beseda kriptografija je skovanka, ki izhaja iz dveh grških besed, kryptos in gráph. Prva pomeni skrito ali skrivno, druga pa pisavo. Kljub grškemu izvoru besed, pa v stari Grčiji ni bilo prevelike potrebe po šifriranju, saj je bilo zaradi velike nepismenosti že samo branje dovolj zahtevna naloga, da je večino sporočil ostalo tajnih. Kljub temu pa naj bi kriptografijo prvi uporabljali Špartanci, čeprav ne v smislu, kot ga danes razumemo pod tem pojmom. Prvi, ki je dokazano šifriral svoja sporočila, je bil rimski poveljnik in kasnejši cesar Gaj Julij Cezar, ki je za komunikacijo s svojimi generali uporabljal zamenjavo črk in cifer s črkami, ki so bile za vnaprej določeno razdaljo nižje v abecedi. Prejemnik je lahko sporočilo prebral le, če je poznal to razdaljo. Ta način kodiranja je dobil ime Cezarjeva šifra, podatek, ki ga potrebujemo, da lahko sporočilo beremo, pa ključ. Tak način kodiranja se je nato ohranil skozi stoletja. Tako pošiljatelj kot sprejemnik sta imela enak skrivni ključ, ki pa sta si ga morala na nek varen način izmenjati, po navadi preko kurirja ali osebno. Z nastankom računalnika in računalniških mrež je postal tak način kodiranja neučinkovit.

Dandanes se za šifriranje pogosto uporablja sistem z asimetričnim ključem, kjer ključ za enkripcijo ni več enak ključu za dekripcijo. Vsak uporabnik ima tako par ključev, javni in zasebni ključ. Medtem, ko je javni ključ prosto dostopen, pa ima zasebni ključ zgolj uporabnik sam in z njim dekodira sporočila, ki so bila namenjena njemu. Eden prvih tovrstnih algoritmov, ki je bil sposoben tako generiranja elektronskih podpisov, kot tudi kodiranja sporočil, je algoritem RSA. Algoritem je dobil svoje ime po priimkih avtorjev, ki so ga leta 1978 prvi javno predstavili (Ron Rivest, Adi Shamir in Leonard Adleman) in je uporaben za digitalno poslovanje (digitalno podpisovanje, izkazovanje istovetnosti, komunikacija).

Za šifriranje in branje sporočil potrebujemo torej dva ključa, javnega in zasebnega, ki ju je potrebno zgenerirati tako, da lahko sporočilo zakodirano z javnim ključem, preberemo pa samo s privatnim ključem. V nadaljevanju bomo pogledali, kako ju generiramo in uporabimo.

2. Pomoč pri implementaciji

Za uspešno šifriranje in branje sporočil moramo najprej generirati oba ključa, nato pa le ta uporabiti za kodiranje in dekodiranje sporočil. Vse to pa je mogoče le, če imamo ustrezne ključe, zato bomo najprej pogledati, kako jih dobimo.

Algoritem za generiranje ključev

Pri generiranju ustreznih ključev je izrednega pomena generator praštevil, saj jih rabimo v več korakih algoritma. Ta je naslednji:

1. Izberemo dve približno enako veliki, vendar različni praštevili p in q ($p \neq q$). V ta namen lahko uporabimo metodo Miller-Rabin.

2. Izračunamo produkt $n = pq$ ter Eulerjevo funkcijo tega produkta, $\varphi(n)$. Ker je n produkt praštevil, funkcijo $\varphi(n)$ izračunamo kot naslednji produkt:

   $\varphi(n) = (p-1)(q-1)$.

3. Izberemo naključno manjše liho število e, tako da velja $1 < e < \varphi(n)$ in $\gcd(e, \varphi(n)) = 1$.
4. Izračunamo skriti eksponent d kot multiplikativni inverz števila e po modulu $\varphi(n)$, kar pomeni, da je potrebno izpolniti enačbo $e\,d \equiv 1 \pmod{\varphi(n)}$.
5. Javni ključ P predstavlja naslednji par $P = (e, n)$, skrivni ključ S pa par $S = (d, n)$.

Praštevila v prvem koraku izračunamo s pomočjo generatorja praštevil, ki je bil predmet prve vaje. Ker morata biti števili približno enako veliki, to dosežemo tako, da jih omejimo s številom bitov.

Generator naključnih števil bomo uporabili tudi v tretjem koraku. Potrebno je preveriti le to, če je največji skupni delitelj števila e in $\varphi(n)$ večji kot 1. Če to velja, potem je potrebno generirati novo število e, sicer pa se lahko premaknemo na korak 4.

Izračun multiplikativnega inverza je poseben primer modulske linearne enačbe oblike, $a\,x \equiv b \pmod{n}$. Ta tip enačb rešujemo s pomočjo posebne funkcije `MODULAR_ LINEAR_EQUATION_SOLVER(a, b, n)`, ki je predstavljena v izpisu 1.

```
function MODULAR_LINEAR_EQUATION_SOLVER(a, b, n)
 begin
      EXTENDED_EUCLID(a, n, d, x, y);

      if d deljivo z b then
        begin
         x0 := x*(b/d) mod n;
         for i := 0 to d-1 do
           print(x0 + i*(n/d) mod n);
        end
      else
        print(Rešitev ne obstaja);
end
```

Izpis 1: Iskanje rešitve modulske linearne enačbe

Iz izpisa 1 lahko vidimo, da podobno kot pri sistemu linearnih enačb, tudi tukaj ni nujno, da rešitev obstaja. Toda, če upoštevamo, da v kolikor računamo multiplikativni inverz, v našem primeru števil e in $\varphi(n)$, je vrednost spremenljivke b enaka ena. Ker pa sta števili e in $\varphi(n)$ tuji, nam tudi funkcija `EXTENDED_EUCLID` vrne kot največji skupni delitelj, d, vrednost ena. Zaradi tega je pogoj v pogojnem **if** stavku izpolnjen in rešitev obstaja, še več, rešitev je natanko ena.

Da lahko funkcijo `MODULAR_LINEAR_EQUATION_SOLVER` implementiramo, potrebujemo še funkcijo `EXTENDED_EUCLID`. Ta nam poišče največji skupni delitelj števil a in b, to je vrednost d, ter vednosti x in y, ki izpolnita naslednjo enačbo: $d = a\,x + b\,y$. Psevdokod funkcije `EXTENDED_EUCLID` je prikazan v izpisu 2.

```
function EXTENDED_EUCLID(a, b, d, x, y)
 begin
      if b = 0 then
        begin
          d := a;
          x := 1;
          y := 0;
        end
      else
        begin
          EXTENDED_EUCLID(b, a mod b, n_d, n_x, n_y);
          d := n_d;
          x := n_y;
          y := n_x - (a/b)*n_y;
        end
end
```

Izpis 2: Pseudokod funkcije `EXTENDED_EUCLID`

Funkcija `EXTENDED_EUCLID` lahko vrne v parametru x tudi negativno vrednost, zaradi tega je potrebno opozoriti, da je ostanek pri deljenju, ki ga uporabljamo v funkcijah, vezan na naravna števila, ki pa negativnih vrednosti ne poznajo, medtem ko je funkcija, ki jo imamo vgrajena v programskih jezikih vezana na cela števila. To pomeni, da v kolikor računamo $a \bmod b$ in je a negativno število, je potrebno kot rezultat vrniti vsoto $a+b$. Ker je v našem primeru skriti eksponent d naravno število, je potrebno to upoštevati pri izračunu.

Kodiranje sporočil

Kodiranje sporočil poteka po naslednjem algoritmu:

1. javni ključ naslovnika sporočila je $P=(e,n)$,
2. sporočilo predstavimo kot naravno število M,
3. tajnopis C izračunamo po naslednji enačbi: $C=M^e \pmod n$,
4. tajnopis C pošljemo naslovniku.

Ker gre v primeru izračuna tajnopisa C za izračun potence velikih števil, je to težko izračunati zaradi velikosti tipov števil, ki jih imamo na voljo. Zaradi tega za to uporabimo algoritem `MODULAR_EXPONENTATION`, ki je prikazan v izpisu 3.

```
function MODULAR_EXPONENTATION(a, b, n)
 begin

    d := 1;
    Razbij b v dvojiško predstavitev [bj, bj-1,...,b0]

    for i:=j downto 0 do
      begin
        d := d*d mod n;
        if bi = 1 then
          d := d*a mod n;
      end
    return d;
end
```

Izpis 3: Pseudokod funkcije `MODULAR_EXPONENTATION`

Dekodiranje sporočil

Dekodiranje sporočila poteka po naslednjem postopku:

1. skrivni ključ naslovnika je $S=(d,n)$,
2. naravno število M se izračuna z enačbo $M=C^d \pmod n$,
3. vrednost M pretvori se v tekstovno sporočilo.

3. Zahteve naloge

Implementirati je potrebno aplikacijo, ki omogoča tvorbo para javnega in zasebnega ključa. V datoteko *privkey.txt* se naj shrani zasebni ključ in v datoteko *pubkey.txt* se naj shrani javni ključ. Uporabnik naj ima možnost določiti število bitov praštevil p in q, ki sta osnova za generiranje ključa. V osnovni različici naj omogoča do 15 bitov za praštevili p in q.

Aplikacija naj na podlagi javnega ključa naslovnika omogoča šifriranje poljubne datoteke, kar vključuje tudi fotografije itd. Kodirano sporočilo se naj shrani v novo datoteko (npr. *enc.bin*). Aplikacija naj na podlagi zasebnega ključa naslovnika omogoča dekodiranje datoteke (*enc.bin*). Rezultat se naj zapiše v novo datoteko (npr. *msg.txt ali msg.jpg*).

Datoteke se naj kodira po skupinah bitov (največje možne vrednosti sporočila M morajo biti manjše od produkta $n=pq$). Če je npr. n=35 (5 x 7), naj aplikacija datoteko bere oziroma kodira po skupinah po 5 bitov, oziroma po $\lfloor \log_2(pq) \rfloor$ bitov. Upoštevajte, da lahko zakodirano sporočilo zahteva en bit več kot prej. Vsebine zakodiranega sporočila pa se ne sme izgubiti. Primer: za zapis C=34 pri n=35 potrebujemo 6 bitov. Torej pri zapisovanju zakodiranega sporočila in pri dekodiranju zakodiranega sporočila predpostavite en bit več oziroma $\lceil \log_2(pq) \rceil$ bitov.

Pri tej nalogi nas še zanima hitrost tvorbe ključa in kodiranja oziroma dekodiranja glede na velikost ključev. Najprej narišite graf časovne zahtevnosti tvorbe ključev glede na število bitov števila n (število bitov praštevil p in q naj gre od 3 do 15). Nato še izmerite čas kodiranja in dekodiranja poljubnega števila glede na število bitov števila n in narišite graf. Da bo možno iz grafa pridobiti ustrezne zaključke, pri vsakem n-ju izvedite več ponovitev, vsakič z drugačnim številom.

Pri tako majhnih ključih, kot jih omogoča osnovna različica aplikacije, je možno iz javnega ključa zelo hitro uganiti privatni ključ (https://en.wikipedia.org/wiki/RSA_Factoring_Challenge). Zato se v praksi za kodiranje uporabljajo bistveno večji ključi (1024-bitni in več). Aplikacijo razširite za delovanje s poljubno velikimi ključi. V ta namen je potrebno aplikacijo razširiti, da deluje z večjimi številskimi tipi. Vse obravnavane algoritme v tej in prejšnji nalogi implementirajte s knjižnico za predstavitev števil z večjim številom bitov (BigInt), npr.:

- https://www.boost.org/doc/libs/1_71_0/libs/multiprecision/doc/html/boost_multiprecision/tut/ints/cpp_int.html
- https://gmplib.org/,
- https://www.boost.org/doc/libs/1_71_0/libs/multiprecision/doc/html/boost_multiprecision/intro.html.