Algoritmit ja tekoäly harjoitustyö

Toteutusdokumentti

Sisältö

1 Johdanto

Ohjelma pystyy tuottamaan RSA-avainpareja[1], salaamaan viestejä ja purkamaan salauksia oikealla avaimella.
Avainparin luonti toimii seuraavalla reseptillä:

1. Etsi 2 suurta alkulukua p, q
2. Laske $n = pq$
3. Lasketaan $\lambda(n)$, missä λ on Carmichaelin funktio[2]
4. Valitaan kokonaisluku e, avoimelta väliltä $(2, \lambda(n))$, mille $gcd(e, \lambda(n)) = 1$
5. Lasketaan $d \equiv e^{-}1(mod\lambda(n))$

Viestit salaus ja purku tapahtuu seuraavasti:

1. Muutetaan viesti M kokonaisluvuksi m sovitulla metodilla.
2. Lasketaan $c \equiv m^e(mod(n))$. c on nyt salattu viesti.
3. Purku tapahtuu laskemalla $c^d \equiv (m^e)^d \equiv m(mod(n))$
4. Nyt m:n voi muuttaa takaisin tekstiksi vastaavasti pythonin sisäänrakennetuilla työkaluilla.

Yksityisellä avaimella viestin salaaminen ja julkisella purkaminen toimii täysin vastaavalla tavalla.

2 Alkukulujen generointi

Alkulukujen luomiseen käytin Miller-Rabin algoritmiä[3]. Testi tarkastaa, onko pariton luku n todennäköisesti alkuluku seuraavast:

1. Otetaan luvusta $n - 1$ kahden tekijät ulos ja kirjoitetaan $n - 1 = 2^s d$, missä s ja d ovat luonnollisia lukuja ja d pariton.
2. Toistetaan seuraava k kertaa:
   Arvotaan pariton a avoimelta väliltä $(1, n - 1)$, joka n:n kanssa jaoton.
   Tarkastetaan päteekö jokin
   - $a^d \equiv 1(mod(n))$
   - $a^{2^r d} \equiv -1(mod(n))$, jollain $0 < r < s$

   Jos mikään näistä ei päde, n on komposiitti, toisin sanoen, ei alkuluku.
   Näissä auttaa pythonin sisäänrakennettu modulaarinen exponentti -funktio `pow(base,exponent,modulus)`
3. Jos testit menivät läpi jokaisella a, meillä on todennäköisesti alkuluku.

Mitä korkeampi k, sitä suurempi tarkkuus, mutta sitä pidempään koodin ajaminen kestää. $k = 40$ antaa riittävän tarkkuuden meille. Tämän lisäksi tehostan koodia ensiksi suodattamalla pieniä alkulukuja pois. Listaan kaikki 1000 pienemmät alkuluvut ja koitan, onko n jaollinen millää niistä, ennen kuin siirryn Miller-Rabin -testiin. Arvon siksi myös a:n avoimelta väliltä (1000, n-1). Tämä nopeuttaa ja myös tarkentaa testiä. Pienet alkuluvut ovat yleisimmät jakajat, joten niillä saa hylättyä useimmat epäkelvot ehdokkaat ja koska käymme enemmän ehdokkaita läpi näin, testi myös tarkentuu.

3 Avainparin luonti

1. Etsi 2 suurta alkulukua p, q
   - Ohjelmani tuottaa 1024-bittisiä alkulukuja. Alkulukujen olisi hyvä olla kaukana toisistaan, mutta minä tyydyin varmistamaan, ettei vahingossa $p = q$
   - p ja q tulee pitää salaisina
2. Laske $n = pq$
   - n on modulus molemmille avaimille
3. Lasketaan $\lambda(n)$, missä λ on Carmichaelin funktio[2]
   - Koska $n = pq$, $\lambda(n) = lcm(\lambda(p), \lambda(q))^{(4)}$, ja koska p ja q ovat alkulukuja, $\lambda(p) = p - 1$ ja vastaavasti $\lambda(q) = q - 1$. Näin
   $$\lambda(n) = lcm(p - 1, q - 1)$$
   - $lcm(a, b)$ on lukujen a, b pienin yhteinen jakaja, joka voidaan kirjoittaa:
   $$lcm(a, b) = \frac{|ab|}{gcd(a, b)}$$
   - $gcd(a, b)$ on lukuje a ja b suurin yhteinen tekijä ja se löydetään Eukleideen algoritmillä[5].
   - $\lambda(n)$ pitää pitää salassa.
4. Valitaan kokonaisluku e, avoimelta väliltä $(2, \lambda(n))$, mille $gcd(e, \lambda(n)) = 1$
   - Suuri e tuottaa turvallisuutta, pieni nopeutta. Päädyin pitämään e:n välillä $(100, 100000)$.
   - e on julkisen avaimen exponentti.
5. Lasketaan $d \equiv e^{-}1(mod\lambda(n))$
   - Eli lasketaan $de \equiv 1(mod\lambda(n)$. Tämä onnistuu tehokkaasti hyödyntäen laajennettua Eukleideen algorimtiä[6], koska e ja $\lambda(n)$ ovat keskenään jaottomat. Algoritmi tuottaa kaksi Bézout'n lukua[7]
   - d on yksityisen avaimen eksponentti.

6. Tämän jälkeen kirjaan kaikki salassa pidettävät osat nolliksi, ettei niistä vahingossakaan jää mitään muistijälkiä.

Julkisen eksponentti, e, ja modulus, n, muodostavat julkisen avaimen ja "dekryptori", d toimii moduluksen kanssa yksityisenä avaimena.

4 Salaus ja purku

Salaan viestin seuraavasti:

1. Muunnan salattavan viestin, M, `str`-olion `.encode`-metodilla biteiksi muuttujaan `string_to_bytes`.
2. Muunnan bitit kokonaisluvuksi, m, `int`-olion metodilla `from_bites(bites, format)`
3. Lasken modulaarisen potenssin $c \equiv m^e \ mod(n)$ pythonin `pow()`-funktiolla
4. Tallennan salatun viestin, c, tiedostoon.

Purkaminen:

1. Ladataan tallennettu kokonaisluku tiedostosta.
2. Lasketaan $c^d \equiv (m^e)^d \equiv m \ mod(n)$.
3. Muunnetaan kokonaisluku, m, takaisin biteiksi `int`-olion metodilla `to_bites(number, format)`.
4. Bitit käännetään takaisin tekstiksi `str`-olion `.decode`-metodilla.

5 Ohjelman toiminta ja tehokkuus

Miller-Rabin toimii ajassa $O(k(log(n))^3)$, missä n on testattava kokonaisluku ja k, kuinka monella eri luvullu testataan lukua, n. Se on ainoita aikaa selkeästi vieviä prosesseja ohjelmassa. Testeissä tuotan 40 isoa alkulukua ja testeihin kuluu noin 30 sekuntia.

Ohjelma ajaa yksinkertaisen tekstipohjaisen käyttöliittymän jossa on neljä vaihtoehtoa:

1. Tuota avainpari:
   - Ohjelma kysyy avainparille nimeä, esim. `avain`
   - Ohjelma tuottaa yllä kuvatulla tavalla avain parin ja tallentaa tiedostot `avain_public.txt` ja `avain_private_txt`. Ensimmäisessä on modulo ja julkinen eksponentti, jälkeimmäisessä modula ja "yksityinen dekryptori".
2. Salaa viesti:
   - Ohjelma pyytää syötteeksi salattavan viestin, millä avaimella salaus halutaan tehdä ja onko avain julkinen vai yksityinen.

- Jos haluttu avain löytyy, pyydetään viestin tallennusta varten tiedostonimi ja viesti tallennetaan salattuna tiedostoon

3. Salauksen purku:
   - Ohjelma pyytää syötteeksi purettavan viestin tiedostonimen, millä avaimella salaus halutaan tehdä ja onko avain julkinen vai yksityinen.
   - Jos avain löytyy, koitetaan purkaa salaus. Toistaiseksi aina kun olen koittanut purkaa väärällä avaimella, ohjelma on tuottanut virheen, joka kaataisi ohjelman. Tämä on korvattu nyt tulosteella. Teknisesti on myös mahdollista, että väärä avain saa tuotettua tekstiksi kelpaavan bittijonon, mutta sen ei pitäisi muistuttaa salattua viestiä juurikaan
4. Lopeta:
   - Lopettaa ohjelman

6 Kielimallien käytöstä

Miller-Rabin testin rakentamisessa yritin hyödyntää chatGPT:n apuja, kun en tahtonut saada sitä toimimaan. Lopulta kuitenkin wikipedian pseudokoodin uudelleen ja uudelleen puhtaaksi kirjoittaminen tuotti onnistumisen. Muuten Wikipedian lähteet olivat riittävät. Omia sotkuja tuli välillä puitua stackoverflowssa tai vastaavassa.

7 Lähteet

Laitoin avainsanoihin hyperlinkkejä lähteisiin, mutta nähtävästi github pdf-lukija ei tunnista näitä. Tässä vielä lähteet listattuna

1. `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`
2. `https://en.wikipedia.org/wiki/Carmichael_function`
3. `https://en.wikipedia.org/wiki/Miller\%E2\%80\%93Rabin_primality_test`
4. `https://fi.wikipedia.org/wiki/Pienin_yhteinen_jaettava`
5. `https://en.wikipedia.org/wiki/Euclidean_algorithm`
6. `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`
7. `https://fi.wikipedia.org/wiki/B\%C3\%A9zout\%E2\%80\%99n_lemma`