Reesa

Author: Leo Honkanen

Simple RSA algorithm implementation.

No security promises.

Installation

Requires: clang>=3.0, libgmp>=5.1, python==2.7

Then run:

```
$ make
```

Tested to work on a Mac with Apple compiler toolchain.

Tests

After installation, run all tests with

```
$ python -m unittest -v tests
```

Documentation

The rest of the documentation is in Finnish, but the above should get you going.

Määrittely

Minkä ongelman ratkaiset ja millä tietorakenteella tai algoritmilla

Ratkaisen yksityisen viestinnän ongelmat lohkosalauksella.

Keskeinen algoritmi on RSA, johon liittyy avaingeneraatiossa muitakin algoritmeja kuten Erastoteneen seula ja Eulerin Φ-funktio.

Maininnan arvoisia tietorakenteita ei sisälly toteutukseen.

Miten tehokkaasti toteutuksesi tulee ongelman ratkaisemaan (aika- ja tilavaativuudet)

Salauksen ja purun tavoitetehokkuus on aikavaativuudeltaan O(e) jossa e=salausavaimen kyseinen eksponentti. Salauksessa ja purussa käytetään eri eksponentteja. Tehokkaampiakin potenssilaskun algoritmeja on, mutta aloitan yksinkertaisesta. Tilavaativuudeltaan kuitenkin vakio.

Salausavaimen generoinnissa tavoitetehokkuuteen päädytään seuraavasti:

Satunnaisluvut

n kappaletta noin 256-bittisiä satunnaisia kokonaislukuja:

Aika

n * O(256) => O(n)

Tila

O(1)

Satunnaislukuja generoidaan vain yksi kerrallaan, koska niitä tarvitaan yhtäaikaa korkeintaan 2

Alkulukujen valitseminen

joista valitaan 2 kappaletta, p ja q, kohtuullisen todennäköisiä alkulukuja:

Aika

n * O(k * log2 p * log log p * log log log p)

jossa k on alkulukutestauksien määrä ja p on kokonaisluku, p>>n

Tila

O(1)

Φ(p*q):n laskeminen

Aika

O(m log m) jossa m=p*q, m>>p (Tästä on tehokkaampiakin ratkaisuja, mutta tämä on yksinkertainen)

Tila

O(1)

Julkinen ja salainen eksponentti

Aika

O(n log n) jossa n < Φ(p*q)* => *n* < *p*q => n < m => n log n < m log m

Tila

O(1)

Näistä merkittävimmät aikavaativuudet ovat totienttifunktiolla Φ.

Tilavaativuus on puolestaan vakio.

Mitä lähdettä käytät. Mistä ohjaaja voi ottaa selvää tietorakenteestasi/algoritmistasi

Käytettävien algoritmien tietoja olen hakenut muun muassa:

http://en.wikipedia.org/wiki/RSA_(algorithm)

http://en.wikipedia.org/wiki/Euler%27s_totient_function -
http://math.stackexchange.com/questions/632837/time-complexity-of-euler-totient-function

http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Modular_integers

Testaus

Ohjelmalla on melko kattavat automaattiset testit. Ohjelman automaattinen testaus tapahtuu kahdella tasolla.

Tässä dokumentissa käsitellään testauksen periaatteita yleensä, käytettyjä testiarvoja ja niiden perusteluja on listattu varsinaisessa testausympäristössä.

Yksikkötestit

Apufunktioita jotka kukin toteuttavat jonkin kokonaisen toiminnon, yksikkötestataan. Yksikkötesteissä testataan joitakin mielenkiintoisia ominaisuuksia toiminnosta. Toisaalta testataan myös joitain virhetilanteita, joita tiedetään voivan syntyä. Kaikkia mielivaltaisia vikatilanteita ei testata, esimerkiksi C-rutiineissa odotetaan ettei niihin syötetä alustamattomia arvoja.

Integraatiotestit

Kokonaisia käyttäjälle tarjottavia kokonaisuuksia testataan integraatiotesteillä. Koska ohjelman ydintoiminnot ovat datan muunnoksia, integraatiotesteissä testataan kiertomatkoja. Näillä varmistetaan algoritmin tärkein ominaisuus eli kyky muuntaa kerran kryptattu data takaisin luettavaan muotoon.

Integraatiotesteissä testataan lisäksi ohjelman virheellisestä käytöstä aiheutuvia virheitä, jotka ohjelman pitäisi huomata ennen syvälle C-rutiineihin menemistä.

Testien tuloksia

Uutta ominaisuutta kehittäessä automaattitestit kehitettiin käsintestauksen pohjalta. Vain tiedostokäsittelyssä testit tehtiin ennen varsinaisen toiminnon loppuun kehittämistä.

Automaattitesteillä varmistetaan siis, että vanhat ominaisuudet eivät mene pahasti rikki uusia kehittäessä.

Eräs mainittavan arvoinen testitulos oli kun syötin salausalgoritmille hankalan syötteen: testaustiedoston itse. Tällä testillä paljastui vakava puute algoritmissa, joka pakotti minut toteuttamaan minimaalisen padding-menetelmän.

Toteutus

Olen tyytyväinen ohjelman tehokkuuteen. Valmiiseen ohjelmaan päätyneet tehokkuutta mitoittavat algoritmit ovat itse asiassa tehokkaampia kuin mitä määrittelyssä on annettu:

- Avaingeneraatiossa ei oikeasti käytetä totienttifunktiota, vaan se lasketaan suoraan hyödyntäen alkulukujen ominaisuuksia.
- Salauksessa ja purussa käytettävä modulaarinen eksponentiointifunktio on tehostettu O(log e)-tehokkuusluokkaan, koska O(e) olisi ollut sietämättömän hidas ohjelmassa käytetyillä sinänsä pienikokoisilla 128-bittisillä avaimilla.

Eksponentoinnin tehokkuus pseudokoodina (neliöintimenetelmä):

```
def mod_pow(base, exp, modulus):
    result = 1
    base = base mod modulus

    while exp > 0:
        if exp mod 2 == 1:
        result = (result*base) mod modulus
    exp = exp >> 1 # exp = exp / 2
    base = (base*base) mod modulus

    return result
```

exp:stä häviää yksi bitti joka silmukassa, joten silmukkaa ajetaan `ceil(log_2 exp)` kertaa.

Puutteita

Ohjelmaan en ole vielä ehtinyt toteuttamaan digitaalisia allekirjoituksia käyttäen ohjelman avainpareja.

Salaus ei ole mitenkään vahvaa. Avaimet ovat pienikokoisia, ne generoidaan pienestä ja ennakoitavasta satunnaisavaruudesta ja salausmenetelmän muissa osissa ei ole huomioitu yleisesti tunnettuja hyökkäysreittejä.