Dokumentácia k projektu do predmetu KRY
Implementácia a prelomenie RSA

*Matúš Liščinský*
*xlisci02@stud.fit.vutbr.cz*

# Úvod

Úlohou tohto projektu bolo zoznámiť sa s algoritmom RSA (Rivest–Shamir–Adleman) pre asymetrické šifrovanie a v jazyku C/C++ implementovať generovanie parametrov RSA, t.j. súkromného a verejného kľúča, šifrovanie, dešifrovanie a prelomenie šifry faktorizáciou verejného modulu. Pre implementáciu sú použité i vybrané funkcie z knižnice GMP (GNU Multiprecision Library[1]) umožňujúcej výpočty s ľubovoľnou aritmetickou presnosťou.

# Generovanie RSA parametrov

Na vstupe pre generovanie kľúčov sa očakáva číslo $B$ reprezentujúce veľkosť verejného modulu v bitoch, následne sa hľadajú také prvočísla $p$ a $q$, pre ktoré platí, že $p \neq q$, $bitcount(p) = \lfloor B/2 \rfloor$ a $bitcount(q) = B - \lfloor B/2 \rfloor$, kde funkcia $bitcount(x)$ vyjadruje počet bitov potrebných pre uloženie čísla $x$ a v implementácii odpovedá funkcii `mpz_sizeinbase(x,2)`. Podmienka $p \neq q$ musí byť splnená z bezpečnostných dôvodov, pretože z týchto dvoch prvočísel sa ich vynásobením získa verejný modul $n$, ktorý je súčasťou verejného kľúča a pri potenciálnej snahe o prelomenie by útočník uhádol prvočísla $p$ a $q$ jednoduchým odmocnením $p = q = \sqrt{n}$. Testovanie prvočísel je implementované pomocou metódy Miller-Rabin opísanej na konci sekcie.

Po náleze prvočísel sa vypočíta verejný modul $n$ a $\phi(n) = (p-1) \cdot (q-1)$ čím sa označuje Eulerova funkcia (*angl. Euler's totient function*). Hodnota $\phi(n)$ je použitá pri hľadaní verejného exponenta, značíme $e$, ktorý je náhodne vybraný z intervalu $(1, \phi(n))$ a musí preňho platiť, že $gcd(e, \phi(n)) = 1$, kde funkcia $gcd(x, y)$ vracia najväčší spoločný deliteľ čísel $x$ a $y$. Tú zastupuje Euklidov algoritmus implementovaný vo funkcii `gcd_custom`. Pre náhodný výber čísel sa pri prvočíslach $p, q$ a exponente $e$ využívajú funkcie `mpz_urandomb` resp. `mpz_urandomm`.

Posledným krokom je výpočet súkromného exponenta $d$ tak, aby $e \cdot d = 1 \bmod \phi(n)$, čo môžeme upraviť do tvaru $d = e^{-1} \bmod \phi(n)$. Keďže platí $gcd(e, \phi(n)) = 1$, potrebnú multiplikatívnu inverziu verejného exponenta efektívne získame rozšíreným Euklidovským algoritmom implementovaným vo funkcii `extended_euclidean_algorithm`. Hlavne pri malej šírke verejného modulu, teda nízkej hodnote $B$, môžu nastať situácie, kedy $e = d$ a v takomto prípade končí program chybovou hláškou. Minimálnou hodnotou parametra $B$ je číslo 5, pretože v prípade menšej hodnoty nie je možné nájsť dve rôzne prvočísla $p$ a $q$, až na jediný prípad, kedy pre $B = 4$ síce existuje dvojica rôznych prvočísel $p = 3$, $q = 2$ ale neexistuje také $e \in \mathbb{Z}$, pre ktoré platí $1 < e < \phi(n)$, v dôsledku toho, že $\phi(n) = 2$.

**Miller-Rabin test** Miller-Rabin pravdepodobnostný test prvočíselnosti je algoritmus, ktorý určuje či je dané číslo prvočíslom pomocou modulárneho umocňovania, malej Fermatovej vety a faktu, že umocnenie $\pm 1$ na druhú modulo $p$ je 1, kde $p$ je prvočíslo väčšie než 2. Test je implementovaný vo funkcii `miller_rabin_test` a riadi sa dvomi parametrami: testované číslo a počet iterácií. Výstupom je rozhodnutie, či je dané číslo *pravdepodobne prvočíslo* alebo je číslo *zložené*. Miera chybovosti v rozhodnutí sa s každou iteráciu znižuje štvornásobne. V praxi sa často používa počet iterácii 64, kedy platí, že pravdepodobnosť nesprávneho určenia prvočíselnosti je $4^{-64} = 2^{-128}$. Túto pravdepodobnosť považujeme za veľmi nízku a preto počet operácii pre Miller-Rabin test je pre tento projekt nastavený takisto na číslo 64.

[1] https://gmplib.org/

## Šifrovanie, dešifrovanie

Úloha zahŕňa aj použitie parametrov RSA na účely šifrovania a dešifrovania správy. Šifrovaním dostaneme z otvorenej správy $message$ zašifrovanú správu $cipher = message^e \bmod n$. Reverzným postupom je dešifrovanie, kedy sa správa v otvorenom tvare získa nasledovne: $message = cipher^d \bmod n$. Prípad kedy šifrujeme za pomoci verejného kľúča a dešifrujeme privátnym kľúčom sa používa pre účely utajenia informácie. Opačné použite, teda šifrovanie súkromným a dešifrovanie verejným kľúčom sa používa taktiež, napríklad pri elektronickom podpise. Z princípu algoritmu RSA je zrejmé, že RSA sa používa na šifrovanie takých správ, ktoré sú kratšie ako verejný modul. Dokonca, podľa štandardu by správa nemala mať viac ako $k - 11$ bytov, keďže dĺžka verejného modulu $k$ má byť aspoň 12 bytov. Tento projekt však povoľuje aj modul kratší ako 12 bytov a nezahŕňa implementáciu tzv. *paddingu*, preto je pre zašifrovanie správy nutné splniť iba podmienku, že správa je kratšia ako verejný modul. Ak táto podmienka splnená nie je, program končí chybovou hláškou.

## Prelomenie RSA

Poslednou časťou projektu bolo prelomenie RSA pomocou faktorizácie slabého verejného modulu. Najjednoduchšou metódou faktorizácie čísel je metóda pokusného delenia. Jej výhodou je, že určite vráti prvočíselného deliteľa, nie je tak potrebné ďalšie testovanie. Avšak táto metóda má význam iba pri faktorizácii malých čísel, prípadne na odstránenie malých faktorov pri faktorizácii veľkých čísel. Nehodí sa ale na faktorizáciu veľkých čísel, kvôli jej časovej zložitosti. Pre riešenie tejto úlohy bol preto implementovaný faktorizačný algoritmus Johna Pollarda, ktorého čas behu je úmerný druhej odmocnine veľkosti najmenšieho primárneho faktora čísla . Pred samotným volaním faktorizačného algoritmu sa testuje, či modul, ktorý chceme faktorizovať nie je mocnina čísla alebo prvočíslo. V prípade mocniny sa faktory získajú jednoduchým odmocnením, avšak ak je modul prvočíslo, program končí chybovou hláškou. Po úspešnej faktorizácii je už postup k rozlúsknutiu správy priamočiary a skladá sa z výpočtu druhého faktoru, Eulerovou funkciou dostaneme $\phi(n)$, a rozšíreným Euklidovským algoritmom aj privátny exponent $d$. Nakoniec, správu v otvorenom tvare získame dešifrovaním: $message = cipher^d \bmod n$.

### Pollard's Rho faktorizačný algoritmus

Myšlienka tohto algoritmu je nasledovná: Predpokladajme, že chceme faktorizovať číslo $n = p \cdot q$, kde $p$ je netriviálny faktor. Funkcia $g(x) = (x^2 + 1) \bmod n$ sa použije na vygenerovanie pseudonáhodnej postupnosti. Nech prvá hodnota $x_0 = 2$ a postupnosť ďalej pokračuje nasledovne: $x_1 = g(2)$, $x_2 = g(g(2))$, atď. Táto postupnosť súvisí s inou postupnosťou $\{x_k \bmod p\}$. Hodnotu $p$ ale dopredu nepoznáme, a teda ani túto postupnosť a v tom je spočíva hlavná myšlienka algoritmu.

Keďže počet možných hodnôt pre tieto postupnosti je konečný, obidve postupnosti sa budú eventuálne opakovať. Vďaka narodeninovému paradoxu vieme, že očakávaný počet $x_k$ predtým ako dôjde k opakovaniu je $\mathcal{O}(\sqrt{N})$, kde $N$ je počet možných hodnôt. Postupnosť $\{x_k \bmod p\}$ sa tak pravdepodobne bude opakovať oveľa skôr ako postupnosť $\{x_k\}$. Akonáhle postupnosť obsahuje opakovanú hodnotu, postupnosť bude ďalej cykliť, pretože každá hodnota závisí iba na predchádzajúcej hodnote. K menu tohto algoritmu preto vedie jeho štruktúra prenesená do grafu podobajúca sa na grécke písmeno $\rho$.

Cyklus sa detekuje pomocou Floydovho algoritmu detekcie cyklu tak, že ak pre dvojicu $x_i$, $x_j$ z postupnosti platí $gcd(|x_i - x_j|, n) \neq 1$ algoritmus hlási cyklus. Hľadáme tak najväčší spoločný deliteľ rôzny od 1, čo ale môže platiť aj pre samotné $n$ a v takomto prípade algoritmus končí chybou a je opakovaný s inými parametrami[2].

[2] https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

Pseudokód implementácie Pollard's Rho algoritmu v tomto projekte:

```
function pollard_rho(n):
    x <- random(2,n-1)
    y <- x
    d <- 1
    while d = 1:
        x <- g(x)
        y <- g(g(y))
        d <- gcd(|x - y|, n)
    if d = n:
        return FAIL
    else:
        return d
```

V samotnej implementácii sa na začiatku ako parameter generuje náhodné číslo z intervalu $\langle 2, n-1 \rangle$, čím sa docieli zmena parametrov algoritmu medzi volaniami. Pre výpočet $gcd$ už nie je použitá vlastná funkcia `gcd_custom` ale GMP funkcia `mpz_gcd` aby bol výpočet čo najefektívnejší. Funkcia `pollard_rho` sa volá dovtedy, kým nevráti výsledok iný ako `FAIL`.

# Evaluácia

Pre účely otestovania bol vytvorený skript `test.py`, ktorý postupne spúšťa implementovaný algoritmus pre generovanie kľúčov nad rôznymi dĺžkami verejného modulu, zašifruje vygenerovanú správu, a výsledky kontroluje dešifrovaním a prelomením RSA. Tento skript je možné spustiť príkazom `make test` z hlavného adresára projektu. Navyše pri lámaní RSA sa utilitou `time` zisťuje čas behu programu a pre každú dĺžku modulu, označujeme $|n|$ sa program otestuje 10-krát a vypočíta najmenší, najväčší a priemerný čas behu. Výsledky evaluácie (Tab. 1) sú extrahované práve z výstupu tohoto testovacieho skriptu.

| $\mid n \mid$ | $min[s]$ | $max[s]$ | $avg[s]$ |
|---|---|---|---|
| 60 | 0.01 | 0.01 | 0.010 |
| 70 | 0.05 | 0.05 | 0.050 |
| 80 | 0.36 | 0.88 | 0.603 |
| 81 | 0.12 | 0.98 | 0.292 |
| 82 | 0.20 | 0.31 | 0.239 |
| 83 | 0.34 | 0.91 | 0.551 |
| 84 | 0.23 | 0.30 | 0.258 |
| 85 | 0.61 | 1.88 | 1.178 |
| 86 | 0.12 | 2.00 | 1.048 |
| 87 | 0.43 | 2.22 | 1.279 |
| 88 | 0.74 | 3.59 | 1.800 |
| 89 | 0.12 | 3.40 | 1.776 |
| 90 | 0.86 | 7.93 | 2.356 |
| 91 | 0.41 | 3.56 | 1.991 |
| 92 | 0.81 | 7.16 | 3.509 |
| 93 | 0.54 | 8.81 | 4.107 |
| 94 | 0.38 | 10.01 | 3.455 |
| 95 | 1.00 | 9.71 | 5.096 |
| 96 | 0.29 | 11.05 | 4.492 |
| 97 | 2.78 | 12.07 | 7.122 |
| 98 | 2.40 | 17.14 | 9.304 |
| 99 | 2.74 | 38.75 | 12.496 |
| 100 | 1.96 | 29.08 | 13.898 |

Tabuľka 1: Výsledky experimentálnej evaluácie prelomenia RSA.