Dokumentace k 2. projektu do předmětu KRY – Implementace a prolomení RSA

Vojtěch Havlena, xhavle03

30. dubna 2016

1 Úvod

Tato dokumentace se věnuje popisu zvolených metod a algoritmů pro projekt do předmětu Kryptografie – implementace a prolomení RSA. Cílem práce je implementovat program, který umí generovat parametry RSA, šifrovat, dešifrovat a faktorizovat zadaný veřejný modulus. Pro implementaci byla zvolena knihovna GMP pro manipulaci s velkými čísly a program je napsán v jazyce C++ (je rovněž využíváno GMP rozhraní pro C++).

2 Generování parametrů RSA

První implementovanou částí je generování parametrů RSA. Tato část je implementována ve funkci `GenerateKeys`. Vstupem je velikost veřejného modulu v bitech. Výstupem jsou potom hodnoty (e, n) – soukromý klíč a (d, n) – veřejný klíč. Generování klíčů probíhá podle následujícího schématu.

Algoritmus 1: Schéma generování parametrů RSA

Output: Soukromý klíč (d, n), veřejný klíč (e, n)

1: Generuj dvě velká prvočísla p a q.
2: $n \leftarrow pq$
3: $\phi(n) \leftarrow (p-1)(q-1)$
4: Zvol náhodně e mezi 1 a $\phi(n)$ tak, že $\gcd(e, \phi(n)) = 1$
5: $d \leftarrow inv(e, \phi(n))$
6: **return** (e, n), (d, n)

Nyní se blíže podíváme na jednotlivé části a způsob jejich implementace.

2.1 Generování prvočísel

V první části jsou generovány náhodná prvočísla p, q velká tak, aby $n = pq$ bylo na zadaných B bitů (funkce `GenerateRandomPrime`). Tedy je hledáno prvočíslo p

s $\lfloor B/2 \rfloor$ bity a prvočíslo q s $\lceil B/2 \rceil$ bity. Náhodná prvočísla jsou generovány následovně: Nejprve je vygenerováno náhodné číslo r o daném počtu bitů. Následně jsou nejnižší a dva nejvyšší bity nastaveny na 1 (nejnižším bitem zajistíme, že se jedná o liché číslo). V případě, že bychom 2. nejvyšší bit nenastavili na 1, není zaručeno, že vynásobením dvou takových čísel dostaneme číslo o zadané velikosti. Následně je pomocí randomizovaného algoritmu Solovay–Strassen testováno, zda r je prvočíslo [5]. Pokud ne, je číslo zvětšeno o 2 a opět testováno dokud nezískáme prvočíslo (přesněji není označeno alg. Solovay–Strassen za prvočíslo).

V případě, že již máme vygenerovaná prvočísla p a q, je nutné provést kontrolu, zda tato prvočísla nejsou stejná. Pokud ano, jsou vygenerována nová prvočísla. Nakonec pokud máme prvočísla p, q, $p \neq q$, tak ještě zkontrolujeme, zda $n = pq$ je na požadovaný počet bitů (postupným přičítáním 2 při generování prvočísla jsme mohli dostat příliš velké číslo).

Testování prvočíselnosti je provedeno pomocí již zmíněného algoritmu Solovay–Strassen. Algoritmus zapsaný v pseudokódu vypadá následovně.

Algoritmus 2: Solovay–Strassen

Input: Testované číslo n, počet opakování k
Output: *Comp* pokud je n složené číslo, jinak *probably_prime*

1: **for** *0* **to** k **do**
2: Vyber náhodné celé číslo číslo a z intervalu $[2, n-1]$
3: $x \leftarrow \left(\frac{a}{n}\right)$
4: **if** $x = 0$ **or** $a^{(n-1)/2} \neq x \pmod{n}$ **then**
5: **return** *Comp*
6: **end**
7: **end**
8: **return** *probably_prime*

V programu je pro modulární umocňování velkých čísel typu `mpz_class` použita vestavěná funkce `mpz_powm`. Třída `mpz_class` pro reprezentaci velkých čísel umožňuje používat přetížené operátory pro aritmetické operace (+,*, ...). Není tedy nutné volat speciální funkce a zdrojový kód je podstatně přehlednější. V algoritmu je pomocí $(\frac{a}{n})$ označen Jacobiho symbol, který je možný spočítat následovně [1]:

Algoritmus 3: Výpočet Jacobiho symbolu

Input: Čísla a, n jejichž Jacobiho symbol se má spočítat
Output: Jacobiho symbol $\left(\frac{a}{n}\right)$

```
1:  r ← 1
2:  while a ≠ 0 do
3:      while a mod 2 = 0 do
4:          a ← a/2
5:          if n mod 8 = 3 or n mod 8 = 5 then
6:              r ← −r
7:      end
8:      Prohoď a, n
9:      if a mod 4 = 3 and n mod 4 = 3 then
10:         r ← −r
11:     a ← a mod n
12: end
13: if n = 1 then
14:     return r
15: return 0
```

Jedním z parametrů algoritmu Solovay–Strassen je také počet opakování. V programu jsem použil 100 opakování (tato hodnota byla uvedena v [5]).

2.2 Generování klíčů

V případě, že již máme vygenerovaná různá prvočísla p, q o zadané velikosti, spočítáme $n = pq$ a $\phi(n) = (p-1)(q-1)$. Výpočet veřejného exponentu e potom probíhá následovně. Postupně se generují náhodná celá čísla v intervalu $[2, \phi(n) - 1]$, dokud některé z nich nesplňuje podmínku $\gcd(e, \phi(n)) = 1$. GCD je implementováno pomocí Euklidova algoritmu (funkce `EuclideanAlgorithm`).

Posledním krokem je generování soukromého exponentu d, který je spočítán jako $d = inv(e, \phi(n))$ a inv je operace nalezení inverzního prvku (v programu funkce `Inverse`). Operace inv je implemenvoána pomocí rozšířeného Euklidova algoritmu, který hledá Bézoutovy koeficienty (inverzní prvek vzhledem k modulu $\phi(n)$ je totiž jeden z Bézoutových koeficientů). Veřejný klíč je potom dvojice (e, n), soukromý klíč je dvojice (d, n).

3 Šifrování a Dešifrování

Šifrování zadané zprávy (čísla) podle zadaného klíče je implementováno ve funkci `Cipher`, dešifrování ve funkci `Decipher`. Šifrování je prováděno podle vztahu [5]

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

a dešifrování podle vztahu [5]

$$m = c^d \pmod{n},$$

kde c je zašifrovaná zpráva (číslo) a m je otevřená zpráva (číslo). Při implementaci byla opět využita vestavěná funkce `mpz_powm` pro modulární umocňování.

4 Faktorizace

V této části se budeme zabývat zvolenými algoritmy a implementací faktorizace veřejného modulu. Při faktorizaci se nejprve provede triviální (zkusmé) dělení prvních 1000000 čísel. V implementaci je využito toho, že není třeba zkoušet složená čísla, stačí vyzkoušet všechna prvočísla menší než 1000000 (funkce `TrivialDivision`). Pro nalezení všech prvočísel menší než daná mez je využit Eratosthenovo síto. Použitím tohoto algoritmu jsem se vyhnul zkoušení čísel, které jistě být děliteli nemohou. Navíc dělení veřejného modulu je prováděno uvnitř cyklu Eratosthenova síta (hned jak se zjistí, že dané číslo je prvočíslo). Není tedy nutné nejprve nalezení a testování dělitelnosti provádět odděleně ve dvou cyklech.

V případě, že není nalezen dělitel věřejného modulu pomocí zkusmého dělení, je využita sofistikovanější metoda. V programu je použit algoritmus Pollard ρ s Brentovou detekcí cyklu (Brentova metoda) [2]. Tento algoritmus je implementován ve funkci `BrentMethod`. Jak klasický algoritmus Pollard ρ (s Floydovu detekcí cyklu), tak Brentova metoda využívají narozeninového paradoxu. Algoritmus využívá polynom f pro generování pseudonáhodné posloupnosti $x_{n+1} = f(x_n)$. Polynom je ve tvaru $f(x) = x^2 + c \bmod n$ (hodnota c je v programu zvolena náhodně).

Klasická verze Pollard ρ, která využívá Floydovu detekci cyklu postupně generuje hodnoty $x = f(x)$ a $y = f(f(y))$ dokud $x \neq y$. V případě, že $x = y$ byl nalezen cyklus a výpočet končí. V každé iteraci se rovněž spočítá $p = \gcd(|x - y|, n)$. Pokud $p > 1$, byl nalezen dělitel veřejného modulu. Nicméně může se stát, že algoritmus nenajde dělitele i když n je složené číslo. V tom případě je nutné změnit polynom f a provést algoritmus znovu [3].

Brentova metoda vychází z klasického Pollard ρ algoritmu, jen používá jinou detekci cyklu (Brentova detekce cyklu). Brentova metoda mj. využívá faktu, že pokud $\gcd(x, n) > 1$ potom i $\gcd(ax, n) > 1$, pro kladné celé číslo a. Místo počítání $\gcd(|x - y|, n)$ v každé iteraci, je tedy možné spočítat součin několika po sobě jdoucích $|x - y|$ a teprve potom hledat pro tento součin a n největší společný dělitel, čímž dochází ke zrychlení celého algoritmu. Zápis algoritmu v pseudokódu je potom následující [2, 4].

Algoritmus 4: Brentova metoda

Input: Číslo n, které se má faktorizovat
Output: Dělitel n

1: Zvol náhodně hodnoty y, c , m z intervalu $[1, n-1]$, $f(x) = x^2 + c \mod n$.
2: $r \leftarrow 1$, $q \leftarrow 1$, $G \leftarrow 1$
3: **do**
4: $x \leftarrow y$
5: **for** $i \leftarrow 1$ **to** r **do**
6: $y \leftarrow f(y)$
7: **end**
8: $k \leftarrow 0$
9: **while** $k < r$ **and** $G = 1$ **do**
10: $ys \leftarrow y$
11: **for** $i \leftarrow 1$ *to* $\min(m, r-k)$ **do**
12: $y \leftarrow f(y)$
13: $q \leftarrow q \cdot |x - y| \mod n$
14: **end**
15: $G \leftarrow \gcd(q, n)$
16: $k \leftarrow k + m$
17: **end**
18: $r \leftarrow 2r$
19: **while** $G = 1$;
20: **if** $G = n$ **then**
21: **do**
22: $ys \leftarrow f(ys)$
23: $G \leftarrow \gcd(|x - ys|, n)$
24: **while** $G = 1$;
25: **return** G

Pro výpočet gcd se využívá již zmíněná funkce `EuclideanAlgorithm`. Brentova metoda byla zvolena, protože klasická metoda Pollard ρ faktorizovala některá čísla do délky 96 bitů mnohem déle než požadovaných 60 s.

5 Závěr

V rámci projektu se podařilo vytvořit program, který umožňuje generovat parametry RSA, přičemž pro testování prvočíselnosti byla zvolena metoda Solovay–Strassen. Dále program umožňuje šifrování a dešifrování zpráv (čísel) a také faktorizaci veřejného modulu. Faktorizace je implementována nejprve pomocí zkusmého dělení s využitím Eratosthenova síta. V případě neúspěchu je dále použita sofistikovanější Brentova metoda, která umožňuje faktorizovat v relativně krátkém čase veřejný modulus do velikosti 96 bitů.

Reference

[1] *Jacobi Symbol Algorithm* [http://2000clicks.com/mathhelp/NumberTh27JacobiSymbolAlgorithm.aspx]. Poslední přístup: 13. 4. 2016.

[2] *Pollard Rho Brent Integer Factorization* [https://comeoncodeon.wordpress.com/2010/09/18/pollard-rho-brent-integer-factorization/]. Poslední přístup: 15. 4. 2016.

[3] *Pollard's rho algorithm* [https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm]. Poslední přístup: 13. 4. 2016.

[4] Brent, R. P. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*. Roč. 20, č. 2. S. 176–184. ISSN 1572-9125.

[5] Nechvatal, J. *Public Key Cryptography*. [b.m.]: National Institute of Standards and Technology, Gaithersburg, MD 20899, 1991. NIST Special Publication 800-2.