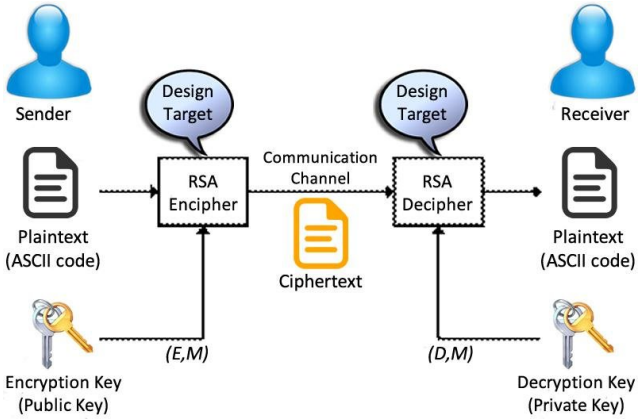
IMPLEMENTÁCIA A PRELOMENIE RSA

Vladimír MARCIN
(xmarci10@stud.fit.vutbr.cz)

2.5.2020

I. ÚVOD

Cieľom projektu bolo sa zoznámiť s asymetrickým šifrovaním, konkrétne s algoritmom RSA. Následne bolo potrebné naimplementovať generovanie parametrov RSA (súkromný a verejný kľúč), šifrovanie, dešifrovanie a prelomenie RSA šifry pomocou faktorizácie slabého verejného modulu.



Obr. 1: Bloková schéma súkromnej komunikácie pomocou RSA.

II. GENEROVANIE PARAMETROV

Generovanie kľúčov je najkomplexnejšia časť RSA šifry. Cieľom je vygenerovať pár (*verejný kľúč*, *privátny kľúč*). Znie to jednoducho?! Nanešťastie, zlé/slabé generovanie robí RSA veľmi náchylným na rôzne spôsoby útokov. Generovanie parametrov tak prebieha nasledovne:

- **Generovanie prvočísiel:** Vstupom je požadovaná veľkosť verejného modulu B v bitoch. Po načítaní sú vygenerované dve prvočísla p a q. Počet bitov jednotlivých čísiel sa určí následovne: $b_p = \lceil B/2 \rceil$ a $b_q = \lfloor B/2 \rfloor$. Aby sme zabezpečili čo najväčšiu náhodnosť a taktiež dostatočnú „vzdialenosť" oboch čísel, je každé z nich generované po bajtoch. To znamená, že pre každé z nich je vytvorené pole o veľkosti $\lceil b/8 \rceil$ a hodnota každého bajtu je generovaná nezávisle. Pred generovaním každého z oboch z čísiel je taktiež generátor pseudonáhodných čísel inicializovaný novou hodnotou zo súboru `/dev/urandom`. Keďže vygenerovaná sekvencia bajtov je náhodná po vygenerovaní musíme zabezpečiť, že p a q neprekračujú požadovaný počet bitov. Za týmto účelom je pre najvyšší bajt oboch čísiel vypočítaná maska, ktorá nastaví požadovaný `MSB` na 1 a prípadné jednotky nad ním vynuluje.
  Týmto postupom sme však získali iba dve náhodné čísla definovanej dĺžky, no požiadavkou je aby p a q boli prvočísla. Za týmto účelom je `LSB` oboch čísiel nastavený na 1 (žiadne párne číslo nemôže byť prvočíslom) a následne je spustený *Rabin-Millerov test prvočíselnosti*[1] (`miller_rabin()`). Ten dokáže rýchlo a s dosť veľkou pravdepodobnosťou určiť, či je vstupné číslo prvočíslom. Pri 64 opakovaniach (nami zvolená hodnota) je pravdepodobnosť, že sa tento test pomýli len $\frac{1}{2^{128}}$[1]. Táto hodnota je tak malá, že situácia kedy by bol výsledok tohto testu nesprávny je takmer nemožná. V prípade, že vráti `false` (hodnota nieje prvočíslo) k vygenerovanému číslu je pripočítaná hodnota 2 (testujeme iba nepárne čísla) a test prebieha znova. To sa bude opakovať tak dlho kým nenájdeme najbližšie prvočíslo.
  Po vygenerovaní oboch čísiel je aj napriek malej pravdepodobnosti vykonaná kontrola na rovnosť oboch čísel. V prípade, že by sa rovnali je číslo q generované nanovo.
- **Verejný modul:** Keď už máme vygenerované dve prvočísla (p a q), môžeme jednoducho určiť hodnotu verejného modulu n ako:

$$n = p \cdot q$$

- **Vyhodnotenie Eulerovej funkcie $\phi(n)$:** Následne je vypočítaná hodnota $\phi(n)$ (angl. Totient), ktorá je použitá pri určovaní verejného a privátneho kľúča. Keďže poznáme prvočíselný rozklad modulu n je výpočet $\phi(n)$ veľmi jednoduchý a to:

$$\phi(n) = (p-1) \cdot (q-1)$$

- **Verejný kľúč:** Verejným kľúčom e je obecne hodnota z intervalu $\langle 3, \phi(n)\rangle$, ktorá spĺňa, že $NSD(e, \phi(n)) = 1$. V našej implementácii je preto ako počiatočná hodnota zvolené číslo 3 a následne sa kontroluje spomínaná podmienka. Za verejný exponent e je potom zvolené najbližšie prvočíslo spĺňajúce túto podmienku. Nájdenie najväčšieho spoločného deliteľa je implementované vo funkcii `gcd()`.

[1] Miller–Rabin primality test: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

- **Privátny kľúč:** Poslednou hodnotou, ktorú potrebujeme určiť je privátny kľúč d. Ten sa jednoducho určí ako multiplikatívny doplnok k e vzhľadom na $\phi(n)$. Na to slúži *Rozšírený Euklidov algoritmus*[2] implementovaný vo funkcii `invert()`.

Výstupom prvej fázy nášho riešenia je teda n-tica:

```
$ p q n e d
```

III. Šifrovanie a Dešifrovanie

Ďalšou fázou je šifrovanie a dešifrovanie pomocou vygenerovaných parametrov z prvej fázy. Vstupom pre šifrovanie je hodnota verejného kľúča e, verejného modulu n a nezašifrovanej (otvorenej) správy m. Správu potom šifrujeme pomocou nasledujúceho vzorca:

$$c = m^e \ \% \ n$$

Dešifrovanie prebieha obdobne s tým rozdielom, že namiesto verejného kľúča bude na vstupe privátny d a namiesto otvoreného textu bude text zašifrovaný c. Dešifrujeme pomocou vzorca:

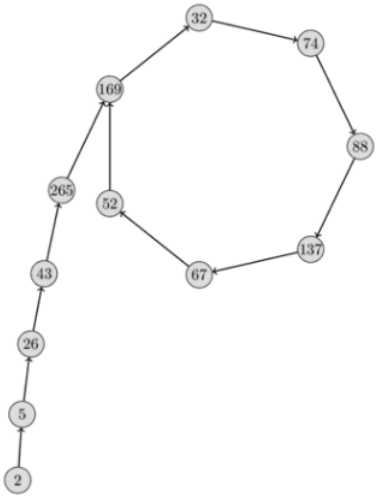
$$m = c^d \ \% \ n$$

Proces šifrovanie resp. dešifrovania je implementovaný vo funkcii `rsa_endecrypt()`. A výstupom je buď `$ c` alebo `$ m` (záleží od zvoleného módu).

IV. Prelomenie RSA

Poslednou fázou bolo prelomenie slabého kľúča RSA pomocou faktorizácie verejného modulu. V teórii čísel je dnes najúčinnejším klasickým algoritmom na faktorizáciu celých čísel väčších ako 10^{100} algoritmus GNFS[3]. Jeho implementácia však nieje triviálna a preto sme hľadali algoritmus, ktorý by bol výrazne rýchlejší ako naivné delenie no rozumne náročný na implementáciu. Podľa [2] presne takéto požiadavky spĺňa algoritmus *Pollard's rho* v kombinácii s postupným delením pre malé prvočísla.

- **Postupné delenie:** Podľa vyššie spomenutého postupu sa implementovaný algoritmus najprv snaží nájsť hľadané faktory pomocou postupného delenia. Keďže vieme, že verejný modul na vstupe je získaný vynásobením dvoch prvočísiel, budeme ho postupne deliť všetkými prvočíslami menšími ako $1\,000\,000$. Keďže test prvočíselnosti každého čísla z intervalu $\langle 2, 1\,000\,000)$ by bol dosť časovo náročný, je za týmto účelom vytvorené statické pole všetkých prvočísiel z uvedeného intervalu (viz. súbor `primes.h`). To zabezpečí získanie vyššieho prvočísla jednoduchou inkrementáciou indexu. Práve týmto postupom sme schopní získať jeden z faktorov v pomerne krátkom čase, v prípade, že je dostatočné malý ($< 1\,000\,000$) takže potenciálne nebezpečný. Táto časť algoritmu je implementovaná vo funkcii `trial_division()`.
- **Pollard's rho:** V prípade, že sú oba faktory verejného modulu $> 1\,000\,000$, predchádzajúci krok algoritmu neuspeje a na rad prichádza faktorizačná metóda Pollard rho. Príjemnými vlastnosťami tejto metódy sú (1) jej malá priestorová zložitosť, (2) rádovo vyššia rýchlosť oproti naivnému deleniu (3) a to, že je založená na jednoduchých myšlienkach, ktoré môžu byť taktiež použité aj v iných oblastiach.
  Nech faktory nášho verejného modulu sú $n = pq$. Algoritmus sa pozerá na pseudonáhodnú sekvenciu $x_i = x_0, f(x_0), f(f(x_0)), \ldots$ kde f je polynomiálna funkcia, zvyčajne sa používa $f(x) = x^2 + c \ mod \ n$, kde $c = 1$. V skutočnosti nás, ale nezaujíma sekvencia x_i. Čo nás zaujíma viac je sekvencia $x_i \ mod \ p$. Keďže f je polynomiálna funkcia a všetky hodnoty sú z rozsahu $\langle 0, p)$ táto sekvencia sa začne skôr, či neskôr cyklicky opakovať. Podľa tzv. *narodeninového paradoxu* sa táto sekvencia začne opakovať po $\mathcal{O}(\sqrt{p})$ prvkoch. Vizualizáciu tejto situácie možno vidieť na obrázku Fig. 2.



Obr. 2: Vizualizácia sekvencie $x_i \ mod \ p$, kde $n = 2206637, p = 317, x_0 = 2$ a $f(x) = x^2 + 1$ [2].

Stále však ostáva jedna veľká otázka. Ako môžme uvažovať sekvenciu $x_i \ mod \ p$, keď nepoznáme p?
Odpoveď nieje až taká náročná. V sekvencii sa nachádza cyklus $\{x_i \ mod \ p\}_{i \leq j}$ vtedy a len vtedy ak sa tam nachádzajú dva indexy $s, t \leq j$ také, že $x_s \equiv x_t \ mod \ p$. To možno prepísať na tvar $x_s - x_t \equiv 0 \ mod \ p$, čo je ekvivalentné s $p \mid gcd(x_s - x_t, n)$. Takže ak v našej sekvencii nájdeme dva indexy s a t také, že $g = gcd(x_s - x_t, n) > 1$, našli sme hľadaný cyklus a teda aj faktor g nášho verejného modulu n. Pozorný čitateľ si môže všimnúť, že je možná situácia kedy $g = n$. V takom prípade je algoritmus spustený znova s novou hodnotou x_0.
Na detekciu cyklu v našej implementácii používame

[2]Extended Euclidean algorithm: https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

[3]General number field sieve: https://en.wikipedia.org/wiki/General_number_field_sieve

Brentovu modifikáciu pôvodného Pollard rho algoritmu. V nej existuje iba jedna kópia x_i, ktorá sa vyhodnocuje v každom cykle a porovnáva sa s uloženou hodnotou y. Hodnota y sa mení iba v prípade, že i je mocnina základu b. Brent, naviac zistil, že ako najoptimálnejšie je za b zvoliť číslo 2 [3]. Základnú kostru implementovaného algoritmu možno vidieť v Algoritme 1. Pre viac detailov viz funkcia `pollard_rho()`.

Algoritmus 1 Základná schéma algoritmu Pollards's Rho Brent. Funkcia $f(x) = x^2 + 1 \ mod \ n$.

$x_0 \leftarrow random(\langle 2, n))$
$y \leftarrow x_0$
for $i = 0; ...; i++$ **do**
  $x_i \leftarrow f(x_i)$.
  $g \leftarrow GCD(|x_i - y|, n)$
  if $g > 1$ **then**
    return "Found factor g"
  end if
  if i is power of 2 **then**
    $y \leftarrow x_i$
  end if
end for
return "Failed :("

V. Experimenty

Experimenty boli vykonané tak, že pomocou módu generovania výslednej aplikácie (`-g`) bol vygenerovaný verejný modul žiadanej dĺžky. Následne sme pomocou neho a príslušného verejného kľúča zašifrovali náhodnú správu (`-e`) a pustili sme aplikáciu v móde lámania šifry (`-b`). Pre každú dĺžku verejného modulu sme spustili test 10x a výsledky môžno vidieť v tabuľke nižšie. Variabilita časov je spôsobená vždy inou novo vygenerovanou počiatočnou hodnotou x_0.

Experimenty bežali na stroji s procesorom CORE i7-7700HQ (2.80 GHz) a systémom Ubuntu 18.04 (64-bit).

Tabuľka I: Experimentálne vyhodnotenie lámania RSA pomocou faktorizácie. Prvé dva riadky tabuľky (oddelené čiarou) predstavujú lámanie pomocou čiastočného delenia, nakoľko boli faktory verejného modulu $< 1\,000\,000$. Na lámanie dlhších modulov už bol použitý algoritmus Pollard rho popísaný vyššie.

počet bitov n	min	max	avg
20	0.002 s	0.007 s	0.006 s
40	0.006 s	0.019 s	0.012 s
60	0.016 s	0.044 s	0.035 s
80	0.164 s	0.648 s	0.429 s
100	1.889 s	29.459 s	17.877 s
120	20 m 43.989 s	26 m 45.590 s	23 m 33.510 s

VI. Záver

V rámci tohoto projektu bol naštudovaný asymetrický šifrovací algoritmus RSA a bola naimplementovaná aplikácia (v jazyku `C` s použitím knižnice `gmp.h`) schopná generovať parametre RSA, šifrovať, dešifrovať RSA správy a taktiež prelomiť RSA šifru v prípade, že bola použitá nedostatočná dĺžka verejného modulu.

Literatúra

[1] "*How RSA Works With Examples*." http://doctrina.org/How-RSA-Works-With-Examples.html, máj 2012.

[2] J. Kogler, "*Integer factorization*." https://cp-algorithms.com/algebra/factorization.html#toc-tgt-10, marec 2019.

[3] "*Brent's Factorization Method*." https://mathworld.wolfram.com/BrentsFactorizationMethod.html, 2015.

VII. Acknowledgements

Táto práca bola financovaná z fondu Marcin Family 😄.