Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Šifrovací algoritmus RSA s výplňovou schémou OAEP

Bakalárska práca

2022 **Martin Janitor**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Šifrovací algoritmus RSA s výplňovou schémou OAEP

Bakalárska práca

Študijný program:	Počítačové siete
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra elektroniky a multimediálnych telekomunikácií (KEMT)
Školiteľ:	prof. Ing. Miloš Drutarovský, CSc.
Konzultant:	

Košice 2022 **Martin Janitor**

Abstrakt v SJ

Hlavným cieľom teoretickej časti bakalárskej práce je uvedenie problematiky šifrovacieho algoritmu RSA a jeho uplatnenie v praxi. Následne som opísal výplňovú schému OAEP, ktorá znemožňuje aplikovaniu špecifických typov útokov na RSA algoritmus. Značná časť bakalárskej práce je venovaná aj problematike generovania kryptograficky bezpečných náhodných čísel. V praktickej časti bakalárskej práce som využil dostupné implementácie RSA algoritmu, ktoré som následne analyzoval a podrobil ich testovaciemu procesu. V rámci testovania som implementoval vlastné zdrojové súbory, ktoré realizovali meranie času pri výpočtoch a overenie korektnosti výpočtov pri RSA algoritme a výplňovej schéme OAEP.

Kľúčové slová v SJ

RSA, OAEP, programovací jazyk C, rand, OpenSSL

Abstrakt v AJ

The main goal of the bachelor thesis is to introduce RSA encryption algorithm and its application in practice. Subsequently, I described the OAEP padding scheme, which prevents the application of specific types of attacks to the RSA algorithm. A significant part of the bachelor thesis is dedicated to generating cryptographically secure random numbers. In the practical part of the bachelor thesis, I used accessible implementations of the RSA algorithm, which I subsequently analyzed and subjected to the testing process. As part of the testing, I implemented my own source files, which realize time measurement in calculations and verification of the correctness of calculations in the RSA algorithm and the OAEP padding scheme.

Kľúčové slová v AJ

RSA, OAEP, programming language C, rand, OpenSSL

Bibliografická citácia

JANITOR, Martin. *Šifrovací algoritmus RSA s výplňovou schémou OAEP* . Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 67s. Vedúci práce: prof. Ing. Miloš Drutarovský, CSc.

65316

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra elektroniky a multimediálnych telekomunikácií

ZADANIE BAKALÁRSKEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Počítačové siete**

Názov práce:

Šifrovací algoritmus RSA s výplňovou schémou OAEP

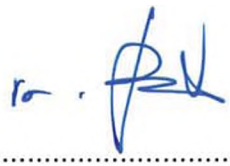
RSA encryption algorithm with OAEP padding

Študent: **Martin Janitor**
Školiteľ: **prof. Ing. Miloš Drutarovský, CSc.**
Školiace pracovisko: **Katedra elektroniky a multimediálnych telekomunikácií**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie bakalárskej práce:

Na základe dostupných informácií naštudujte a opíšte výplňovú schému OAEP (Optimal Asymmetric Encryption Padding) pre šifrovací algoritmus RSA. Vyberte vhodné existujúce implementácie algoritmu RSA v jazyku C, ktoré je možné využiť na vysvetlenie a otestovanie OAEP módu v rámci výučby predmetov zameraných na bezpečnosť počítačových systémov. S využitím jazyka C, dostupných implementácií, kryptografických knižníc a testovacích vektorov pripravte sadu testovacích aplikácií a vhodnú dokumentáciu na otestovanie funkčnosti RSA šifrovania s OAEP módom. Dokumentáciu vytvorte tak, aby bola využiteľná v rámci špecializovaných cvičení. Experimentálne overte základné vlastnosti a spoľahlivosť vybraných knižníc a porovnajte ich vlastnosti. Experimenty realizujte s využitím GCC prekladača na platformách s operačným systémom Windows a Linux.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 27.05.2022
Dátum zadania bakalárskej práce: 29.10.2021



prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 30.5.2022 .....

Vlastnoručný podpis

Poďakovanie

Týmto by som sa chcel poďakovať vedúcemu práce *prof. Ing. Milošovi Drutarovskému, CSc.* za odbornú pomoc pri riešeni práce, cenné informácie a nadobudnutia nových zručností.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Začiatky asymetrickej kryptografie sa píšu v roku 1976 kedy Whitfield Diffie a Martin E. Hellman uverejnili prvý šifrovací algoritmus na báze asymetrického šifrovania. Zaviedli metódu verejného kľúča na dohodu o prenose kľúča, ktorá sa používa dodnes. Základnou myšlienkou kryptografie s verejným kľúčom je verejný kľúč, ktorý je viditeľný pre všetkých účastníkov komunikácie a súkromný, ktorý je viditeľný iba pre účastníka, ktorému patri konkrétny verejný kľuč. Medzi ďalšie typy asymetrických šifrovacích algoritmov patria RSA, ElGamal, Rabin. Využitie asymetrických algoritmov je najmä na prenos zdieľaných kľúčov pre symetrické algoritmy, ktoré disponujú mnohonásobne rýchlejším šifrovaním aj dešifrovaním oproti asymetrickým šifrovacím algoritmom.

V prvej kapitole uvádzam problematiku šifrovacieho algoritmu RSA. Zameriavam sa na koncepčný opis matematického pozdania RSA algoritmu, s ktorým súvisí aj bezpečnosť RSA algoritmu. Opisujem metódy generovania RSA kľúčov, testy na určenie prvočíselnosti, využívané matematické algoritmy, ktoré tvoria základne stavebné bloky šifrovacieho algoritmu RSA. Uvádzam problematiku generovania kryptograficky bezpečných náhodných čísel pre operačné systémy Windows a Linux. Opisujem metódy útokov pre definičný RSA algoritmus, ktoré realizujem na jednoduchých typov príkladoch. Poukazujem na štandard PKCS#1, ktorý stanovuje pravidlá pre využitie šifrovacieho algoritmu RSA v praxi. Poukazujem na možnosti nasadenia implementačných metód RSA algoritmu v praxi.

V druhej kapitole opisujem výplňovú schému OAEP, ktorá slúži na zvýšenie bezpečnosti definičného RSA algoritmu. Šifrovací algoritmus RSA sa využíva v praxi len s dodatočnými kryptografickými metódami, ktoré zabránia špecifickým typom útokov, ktoré sú detailne opísané v prvej kapitole. Jednou z možných metód je výplňová schéma OAEP, ktorá realizuje redundanciu šifrovaného textu s využitím špecifických kryptografických metód. V rámci tejto problematiky uvádzam úvod do problematiky hashovacích funkcií, ktoré sú neoddeliteľnou súčasťou výplňovej schémy OAEP. Názornú implementáciu výplňovej schémy OAEP v spojení so šifrovacím algoritmom RSA uvádzam v obrázkoch uvedených v rámci

kapitoly.

V tretej kapitole sa zaoberám analýzou kryptografických knižníc. Na internete je mnoho dostupných kryptografických knižníc, ktoré majú implementovaný RSA algoritmus. Zvolil som si štyri implementácie kryptografických knižníc, ktoré budem analyzovať. V rámci kapitoly analyzujem knižnice z pohľadu funkčnosti a implementačných nedostatkov. Poukazujem na možnosti výskytu implementačných chýb vo využívaných knižniciach s ktorými sa užívateľ môže stretnúť pri vyhľadávaní dostupných implementácii na internete.

V štvrtej kapitole sa venujem testovaniu kryptografických knižníc. Na testovanie využívam vlastné implementácie testovacích súborov, ktoré som implementoval. Zameriavam sa hlavne na časove nároky výpočtu matematických operácii. Po analýze výsledkov zhodnotím kryptografické knižnice z hľadiska funkčnosti. V praktickej časti poukazujem aj na možnosti využitia nástrojov pre efektívnu kontrolu zdrojových kódov z hľadiska správy pamäte.

1 Asymetrický šifrovací algoritmus RSA

V roku 1978 bol publikovaný šifrovací algoritmus RSA, ktorého autormi sú Ron Rivest, Adi Shamir a Leonard Adleman. Algoritmus je založený na princípe asymetrického šifrovania, v praxi sa môžeme stretnúť aj s názvom kryptografia s verejným kľúčom [1].
Algoritmus využíva pár kľúčov:

- Verejný kľúč $V_k = (e, n)$ – Slúži na šifrovanie správy a je verejne zdieľaný s ostatnými užívateľmi. Verejný kľúč je zložený z verejného exponentu e a modulusu n.
- Súkromný kľúč $S_k = (d, n)$ – Slúži na dešifrovanie správy, ktorá bola zašifrovaná s využitím verejného kľúča. Súkromný kľúč si užívateľ uchováva v tajnosti, nie je verejne zverejnený. Súkromný kľúč je zložený zo súkromného exponentu d a modulusu n.

Okrem šifrovania sa RSA používa aj na vytváranie digitálnych podpisov, pričom vlastník súkromného kľúča je jediný, kto môže podpísať a verejný kľúč umožňuje kedykoľvek overiť platnosť podpisu. Obsahom bakalárskej práce nie je tematika podpisovania správ. Jednou z možností o získaní požadovaných informácii ohľadom podpisovania správ pomocou RSA algoritmu je uvedených napríklad v knihe[2]. Algoritmus RSA uplatňuje svoje využitie hlavne na prenos zdieľaných kľúčov pre rýchlejšie šifrovacie algoritmy, napríklad na prenos zdieľaného kľúča pre symetrický šifrovací algoritmus AES [3].

Princíp šifrovania a dešifrovania v RSA algoritme

Šifrovanie sa realizuje s využitím verejného kľúča rovnicou (1.1).

$$c = m^e \mod n \tag{1.1}$$

Kde c – zašifrovaný text, m – správa, e – verejný exponent, n – modulus. Dešifrovanie sa realizuje s využitím súkromného kľúča rovnicou (1.2). Je možne taktiež zefektívniť rýchlosť dešifrovania s využitím čínskej vety o zvyškoch[2].

$$m = c^d \mod n \tag{1.2}$$

Kde m – správa, c – zašifrovaná správa, d – súkromný exponent, n – modulus.

1.1 Základné metódy využívané v RSA algoritme

Aplikovanie šifrovacieho algoritmu RSA do praxe si vyžaduje dodatočné znalosti z oblasti uloženia veľkých čísel do pamäte počítača, zvolenie vhodných dodatočných kryptografických metód na zvýšenie bezpečnosti RSA algoritmu. V bakalárskej práci sa zameriavam na prácu s programovacím jazykom C, ktorému je prispôsobené vhodné uloženie veľkých čísel BN (ang. Big Number) (ďalej BN) do pamäte počítača z hľadiska využitia dátového typu pole (ang. array). Taktiež v kapitole uvádzam problémy, ktoré sa vyskytujú formou sofistikovaných útokov realizovaných na definičný (ang. textbook) RSA algoritmus, ktorý predstavuje bezpečnostné riziko z hľadiska využitia RSA algoritmu v praxi.

1.1.1 Formát pre uloženie veľkých čísel

Algoritmus RSA pracuje v praxi BN číslami rádovo 1024 bitov a viac. V bakalárskej práci využívam programovací jazyk C, ktorý neobsahuje natívnu premennú pre uloženie čísel väčších ako 1024 bitov. Celé číslo (ang. integer) v jazyku C, je reprezentované 32 bitmi, z toho 1 bit slúži na reprezentáciu znamienka. Maximálna hodnota, ktorú vieme uložiť do premennej typu integer v jazyku C je $2^{31} - 1$. Niektoré GCC prekladače umožňujú využívať aj natívny typ v jazyku C $long\ \ long$, ktorý je reprezentovaný 64 bitmi, čo stále nepostačuje pre implementáciu RSA algoritmu. Riešením je prejsť na inú metódu reprezentácie čísel v pamäti. BN môžeme reprezentovať ako postupnosť vektorov čísel a_i uložených v štruktúre typu pole (ang. array) so základom B (ang. base), $0 \leq a_i < B$, $i = 0 \ldots k - 1$ [4]. BN číslo môžeme reprezentovať so znamienkom aj bez znamienka, v prípade ak je využitá metóda so znamienkom je nutné pridať premennú, ktorá bude uchovávať hodnotu znamienka.

- Bez znamienka – reprezentuje vzorec (1.3).

$$a = \sum_{i=0}^{k-1} a_i B_i = (a_{k-1}...a_1 a_0)_B \tag{1.3}$$

- So znamienkom – reprezentuje vzorec (1.4).

$$a = \pm \sum_{i=0}^{k-1} a_i B_i = \pm (a_{k-1}...a_1 a_0)_B \tag{1.4}$$

Reprezentácia BN v pamäti s využitím dátového typu pole v jazyku C ma výhodu oproti natívnej premennej v tom, že môžeme ukladať ľubovoľne veľké BN číslo pokiaľ to pamäť dovoľuje, pričom s využitím natívnych premenných nás veľkosť premennej obmedzuje, nakoľko pri natívnych premenných ich veľkosť je fixne vopred daná prekladačom.

Príklad:

Uveďme číslo 123 so základom 10, ktoré chceme reprezentovať do dátového typu pole v jazyku C. Majme statické pole A v jazyku C o veľkosti n prvkov. Reprezentáciu čísla môžeme vyjadriť podľa vzťahov (1.5) a (1.6). So vzťahom (1.6) som sa pri riešení praktickej časti bakalárskej práce nestretol, ale je aj takýmto spôsobom možné vyjadriť BN číslo v pamäti. Číslo 123 vyjadríme podľa vzťahu (1.3). Výsledná aplikácia čísla do pamäte je uvedená vo vzťahu (1.5). Jednoduchý algoritmus v jazyku C, ktorý umožňuje konverziu čísla do poľa v jazyku C je uvedený v zdrojovom kóde 1.1.

$$X[0] = 3, \quad X[1] = 2, \quad X[2] = 1, \ \ ...\ 000\ ...\ \ , X[N-1] = 0 \tag{1.5}$$

$$X[0] = 1, \quad X[1] = 2, \quad X[2] = 3, \ \ ...\ 000\ ...\ \ , X[N-1] = 0 \tag{1.6}$$

Zdrojový kód 1.1: Konverzia decimálneho čísla do BN formátu

```
1  /* put the normal int n into the big int A */
2  void make_int (int A[], int n) {
3   int    i;
4
5   /* start indexing at the 0's place */
6   i = 0;
7
8     while (n) {
9            /* put the least significant digit of n into A[i] */
10
11           A[i++] = n % BASE;
12
13           /* get rid of the least significant digit,
14            * i.e., shift right once
15            */
16
17         n /= BASE;
18    }
19
20  /* fill the rest of the array up with zeros */
21   while (i < N) A[i++] = 0;
22 }
```

Pre reprezentáciu BN čísel je možne zvoliť statické pole 1.2 alebo dynamické pole 1.3. Veľkosť statického poľa musí byt vopred fixne daná pri kompilácii, pričom u dynamických sa veľkosť môže meniť za behu programu.

Zdrojový kód 1.2: Uloženie BN čísla do statického poľa

```
1    #define SIZE 45
2    uint32_t BN[SIZE];
```

Zdrojový kód 1.3: Uloženie BN čísla do dynamického poľa

```
1 struct BN{
2     uint8_t sign;
3     uint32_t* BN_vector_array;
4     uint32_t size;
5 };
```

1.1.2 Textbook RSA

Názov textbook RSA je určený pre definičný RSA algoritmus bez dodatočných kryptografických výplňových schém. Operácia šifrovania pre výpočet c podľa vzorca(1.1) je deterministická, pre rovnakú správu m bude vždy rovnaká zašifrovaná správa c. Cieľom textbook RSA je zdôrazniť princíp algoritmu, bez ohľadu na bezpečnosť. V praxi sa textbook RSA nevyužíva, kvôli matematickým útokom, ktoré je možne efektívne aplikovať na zašifrované texty. Útoky, ktoré sú realizované na základe zašifrovaných správ, ktoré sú detailnejšie opísané v podkapitole 1.7 a ich cieľom je využiť matematické metódy na obnovenie pôvodnej správy [5]. Proti zamedzeniu týchto útokov je možne aplikovať dodatočné metódy, ktoré zamedzia realizovať požadované typy útokov. Najpoužívanejšou metódou je optimálna výplň asymetrického šifrovania OAEP (ang. Optimal Asymmetric Encryption Padding)(Ďalej OAEP), ktorá je zadefinovaná v štandarde [6] a stala sa základom pre použitie RSA algoritmu v praxi. Výplňová schéma OAEP realizuje modifikáciu pôvodnej správy pred vstupom do šifrovania, pričom využíva sofistikované metódy, ktoré znemožňujú aplikovanie špecifických typov útokov. Výplňovej schéme OAEP je venovaná v bakalárskej práci osobitná kapitola 2, ktorá podrobne opisuje implementáciu a nasadenie v praxi.

1.2 Proces generovania kľúčov pre RSA algoritmus

V procese generovania RSA kľúčov sa vygeneruje verejný kľúč $V_k = (n, e)$
a k nemu prislúchajúci súkromný kľúč $S_k = (d, n)$ pre jedného užívateľa. Parametrom pre vygenerovanie RSA kľúčov je zadanie vhodnej veľkosti modulusu n, od ktorého závisí bezpečnosť komunikácie. Problematika bezpečnosti RSA algoritmu je uvedená v podkapitole 1.7. Proces vygenerovania RSA kľúčov je uvedený v algoritme (1).

Algorithm 1 Algoritmus generovania RSA kľúčov

VSTUP: L - Dĺžka modulusu v bitoch.

VÝSTUP: (n, e) – Verejný kľúč (d, n) – Súkromný kľúč.

1: Vygeneruj náhodné prvočíslo p dĺžky $2^{(L-1/2)} + 1 \leq p \leq 2^{(L-1)} - 1$ bitov.
2: Vygeneruj náhodné prvočíslo q dĺžky $2^{(L-1/2)} + 1 \leq q \leq 2^{(L-1)} - 1$ bitov.
3: Vypočítaj modulus $n = p * q$.
4: Vypočítaj Eulerovu funkciu $\phi(n) = (p-1) * (q-1)$.
5: Zvoľ verejný exponent e, tak aby platilo $1 < e < \phi(n)$ a zároveň $\gcd(e, \phi(n)) = 1$.
6: Vypočítaj multiplikatívny inverzný prvok $d \equiv e^{-1} \mod \phi(n)$.

VRÁŤ: (n, e) - Verejný kľúč (d, n) - Súkromný kľúč.

1.3 Kryptografický štandard PKCS#1

Kryptografické štandardy verejného kľúča PKCS (ang. Public Key Cryptography Standards) sú špecifické štandardy vytvorené spoločnosťou RSA Laboratories v spolupráci s vývojármi bezpečnostných systémov na celom svete za účelom zavádzania pravidiel pre kryptografiu s verejným kľúčom. Dokumenty PKCS sa stali široko používanými a implementovanými v praxi [6].V rámci publikácie je uvedených 15-násť PKCS štandardov, ďalej sa budem zaoberať štandardom PKCS #1, ktorý je primárne mojím cieľom pre analýzu tejto práce. PKCS #1 je kryptografický štandard pre kryptografiu s verejným kľúčom založenú na RSA algoritme a je definovaný v štandarde [6].

PKCS #1 štandard obsahuje:

1. Kryptografické primitíva:
   - **RSAEP** (ang. RSA Encryption Primitive) – Zašifruje správu pomocou verejného kľúča.
   - **RSADP** (ang. RSA Decryption Primitive) – Dešifruje šifrovaný text pomocou súkromného kľúča.
   - **RSASP1** (ang. RSA Signature Primitive 1) – Vytvorí podpis na základe správy pomocou súkromného kľúča.
   - **RSAVP1** (ang. RSA Verification Primitive 1) – Overí podpísanú správu pomocou verejného kľúča.

2. Šifrovacie schémy:
   - **RSAES-OAEP** (ang. RSA Encryption/decryption Scheme OAEP) – RSA šifrovacia a dešifrovacia schéma s využitím výplňovej schémy OAEP.
   - **RSAES-PKCS1-V1_5** (ang. RSA encryption/decryption scheme in version 1.5) – Schéma šifrovania/dešifrovania ako prvá štandardizovaná vo verzii 1.5 PKCS #1.
3. ASN.1 syntax pre reprezentáciu kľúčov a identifikáciu schém:
   - **Štruktúra verejného kľúča.**
   - **Štruktúra súkromného kľúča.**

Štandard PKCS #1 obsahuje taktiež schémy založené na podpisovaní správ s využitím RSA algoritmu. Tie neboli náplňou tejto bakalárskej práce, preto som ich neuviedol v predošlých bodoch zoznamu. Najaktuálnejšou verziou PKCS#1 štandardu je verzia RFC-8017 2.2 [6], ktorá bola vydaná v roku 2016 spoločnosťou RSA Laboratories [6].

1.4 Matematické pozadie RSA algoritmu

Proces generovania RSA kľúčov si vyžaduje aplikovanie špecifických typov algoritmov, ktoré sú z hľadiska koncepcie algoritmu optimalizované pre prácu s BN číslami. V tejto kapitole opíšem základne typy algoritmov, ktoré sa využívajú pri generovaní RSA kľúčov. V tejto kapitole taktiež opisujem aj konverziu čísel a textových reťazcov.

1.4.1 Eulerova funkcia

Eulerova funkcia označovaná ako $\phi(n)$ udáva počet všetkých prirodzených čísel x, pre ktoré platí podmienka $1 \leq x \leq n$ a zároveň pre čísla x a n platí aby ich najväčším spoločným deliteľom bola pravé jednotka, teda $\gcd(x, n) = 1$. Inak povedané Eulerova funkcia $\phi(n)$ udáva počet všetkých prirodzených čísel nesúdeliteľnych s n [1].
Ak $\phi(p)$, kde p je prvočíslo platí rovnica (1.7), ktorá poukazuje na skutočnosť, že ak p je prvočíslo, potom pre všetky k, $k < p$ plati $\gcd(k, p) = 1$. Vyplýva to z definície prvočísel, pre ktoré platí, že sú deliteľne iba s jednotkou a sebou samým.

$$\phi(p) = p - 1 \tag{1.7}$$

1.4.2 Euklidov algoritmus

Algoritmus RSA vykonáva matematické operácie s BN číslami, ktoré sú opísané v podkapitole (1.1.1) a je veľmi pravdepodobné že nebudeme poznať ich rozklad na prvočísla. Euklidov algoritmus patri medzi najznámejšie algoritmy na vyhľadávanie najväčšieho spoločného deliteľa GCD (ang. Greatest Common Divisor) dvoch prirodzených čísel bez rozkladu na prvočísla. Detailnejší opis Euklidovho algoritmu je možne vyhľadať napríklad v knihe [7].

Algorithm 2 Euklidov algoritmus

Input: a a b, $a \geq b$

Output: "$d = \gcd(a, b)$"

1: **while** $b \neq 0$ **do**
2: $\quad r \leftarrow a \bmod b$
3: $\quad a \leftarrow b$
4: $\quad b \leftarrow r$
5: **end while**
6: **return** "a"

1.4.3 Rozšírený Euklidov algoritmus

Rozšírený Euklidov algoritmus je rozšírením Euklidovho algoritmu na výpočet najväčšieho spoločného deliteľa dvoch čísel a a b, ktorý vypočíta naviac aj hodnoty x a y, pre ktoré platí rovnica (1.8). Slúži na výpočet multiplikatívneho inverzného prvku [8]. V RSA algoritme ho môžeme uplatniť na výpočet súkromného exponentu d, ktorý môže byt realizovaný výpočtom (1.9).

$$a * x + b * y = \gcd(a, b) \tag{1.8}$$

$$\begin{aligned} \phi(n) * x + e * d &= \gcd(e, \phi(n)) \\ e * d &\equiv 1 \mod \phi(n) \\ d &\equiv e^{-1} \mod \phi(n) \end{aligned} \tag{1.9}$$

Algorithm 3 Rozšírený Euklidov algoritmus

Input: a a b, $a \geq b$

Output: "$d = \gcd(a, b)$, x a y, ktoré vyhovujú rovnici (1.8) "

1: **if** $b = 0$ **then**
2: $d = a$, $x = 1$, $y = 0$
3: **return** "(d, x, y)"
4: **end if**
5: $x_2 = 1$, $x_1 = 0$, $y_2 = 0$, $y_1 = 1$
6: **while** $b > 0$ **do**
7: $q = a/b$, $r = a - q * b$, $x = x_2 - q * x_1$, $y = y_2 - q * y_1$
8: $a = b$, $b = r$, $x_2 = x_1$, $x_1 = x$, $y_2 = y_1$, $y_1 = y$
9: **end while**
10: $d = a$, $x = x_2$, $y = y_2$
11: **return** "(d, x, y)"

1.4.4 Algoritmy na konverziu čísel

Kryptografické primitíva I2OSP a OS2IP realizujú konverziu medzi celým kladným číslom a znakovým reťazcom. Tieto kryptografické primitíva sú definované v štandarde [6]. Hlavnou ideou využitia týchto primitív v RSA algoritme je skutočnosť, že RSA algoritmus je založený na matematických operáciách, ktoré dokážu vykonávať matematické operácie len s využitím čísel. Matematické operácie nedokážu priamo vykonávať operácie s textovými reťazcami. Kryptografické primitíva I2OSP a OS2IP realizujú konverziu čísla na textový reťazec a opačne s využitím ASCII tabuľky.

Konverzia z celého čísla na oktetový reťazec – I2OSP

Algoritmus realizuje konverziu kladného celého čísla na oktetový reťazec špecifickej dĺžky. Realizácia konverzie je uvedená v algoritme 4. V RSA algoritme sa využíva pri dešifrovacom procese. Postup implementácie I2OSP do RSA algoritmu:

- Získaj zašifrovanú správu c pomocou RSA algoritmu reprezentovanú ako celé číslo.
- Pomocou funkcie I2OSP vykonaj konverziu z celého čísla na oktetový reťazec podľa algoritmu 4.

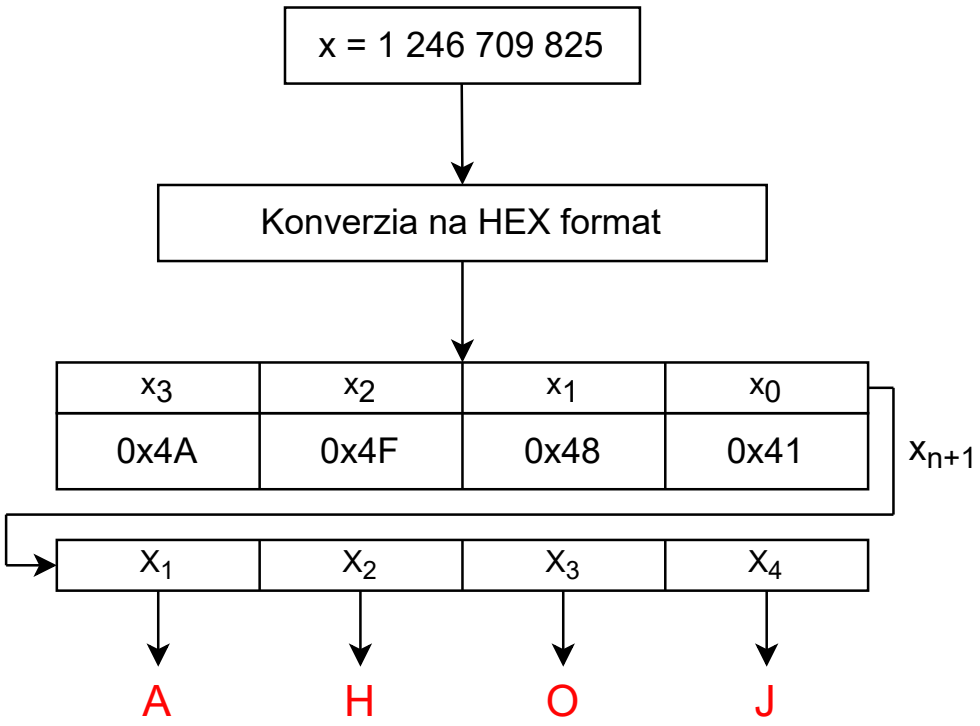
Algorithm 4 Konverzia celého čísla na oktetový reťazec

Input: x – Celé kladné číslo, $xLen$ – Požadovaná dĺžka výstupneho reťazca

Output: "X – Požadovaný reťazec dĺžky $xLen$ "

1: **if** $x \geq 256^{xLen}$ **then**
2: **return** "Príliš veľké číslo"
3: **end if**
4: $x = x_{(xLen-1)}256^{(xLen-1)} + x_{(xLen-2)}256^{(xLen-2)} + ... + x_1256 + x_0$,
kde $0 \leq x_i < 256$ (Ak je jeden alebo viacero oktetov nevyužitých budú nahradené 0, ak $x < 256^{(xLen-1)}$).
5: Nech oktet X_i ma hodnotu čísla $x_{(xLen-i)}$ pre $1 \leq i \leq xLen$.
6: **return** Oktetový reťazec $X = X_1X_2...X_{xLen}$

Príklad konverzie celého čísla na oktetový reťazec:



Obr. 1.1: Konvzia celého čísla na oktetový reťazec

Konverzia z oktetového reťazca na celé číslo – OS2IP

Algoritmus realizuje konverziu oktetového reťazca na celé kladné číslo. Realizácia konverzie je uvedená v algoritme 5. V RSA algoritme sa využíva pri šifrovacom procese.

Postup implementácie I2OSP do RSA algoritmu:

- Zvoľ správu m, reprezentovanú textovým reťazcom.
- Pomocou funkcie OS2IP vykonaj konverziu z textového reťazca na celé číslo podľa algoritmu 5.

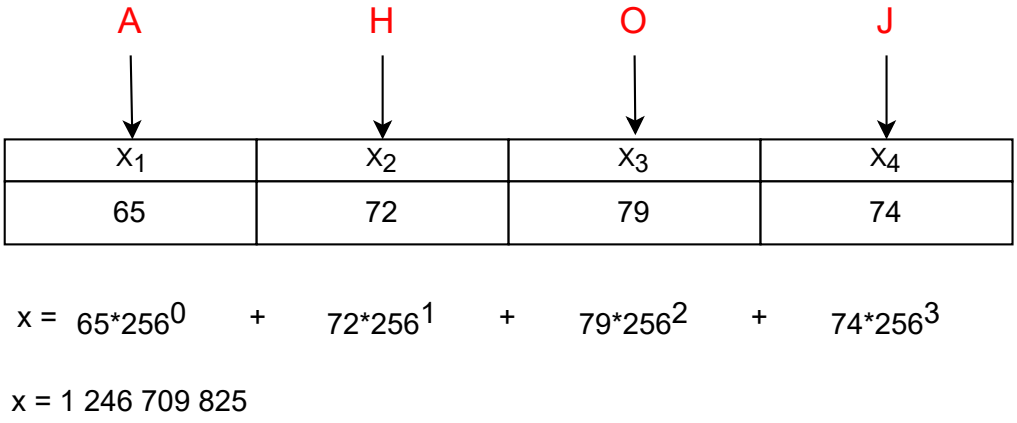
Algorithm 5 Konverzia oktetového reťazca na celé číslo

Input: X – Oktetový reťazec

Output: "x – Celé kladné číslo "

1: Nech $X_1X_2...X_{xLen}$ sú oktetové reťazce z X od prvého až po posledný a nech $x_{(xLen-i)}$ bude hodnota čísla oktetu X_i pre $1 \leq i \leq xLen$.
2: Nech $x = x_{(xLen-1)}256^{(xLen-1)} + x_{(xLen-2)}256^{(xLen-2)} + ... + x_1256 + x_0$.
3: **return** "x"

Príklad konverzie oktetového reťazca na celé čislo:



Obr. 1.2: Konverzia oktetového reťazca na celé číslo

1.5 Testy na určenie prvočíselnosti

Pri hľadaní veľkých prvočísel sa môžu uplatniť rôzne algoritmy na testovanie prvočíselnosti, kedy by skúmaním čísla či je prvočíslo s využitím rozkladu na prvočinitele nebol najvhodnejší spôsob z pohľadu komplexnosti a výpočtovej rýchlosti.

Testy na určenie prvočíselnosti môžeme klasifikovať do dvoch skupín:

1. **Deterministické** – Testy poskytujú jasný výsledok či testované číslo je prvočíslo alebo nie je prvočíslo.

2. **Ne-deterministické** – Testy poskytujú výsledok iba s určitou pravdepodobnosťou. Pre zvýšenie hodnoty pravdepodobnosti sa realizujú opakované testy daného algoritmu. Medzi najznámejšie ne-deterministické testy na prvočíselnosť patria:

   - Millerov–Rabinov test prvočíselnosti (1.5.1).
   - Fermatov test prvočíselnosti (1.5.2).
   - Solovay–Strassenov test prvočíselnosti [9].

1.5.1 Millerov-Rabinov test

Millerov-Rabinov test prvočíselnosti patri medzi najrozšírenejšie testy na prvočíselnosť, svoje uplatnenie nachádza aj v kryptografii. Millerov-Rabinov test je deterministický, výsledok určenia prvočíselnosti nemusí byť pravdivý, platí iba s určitou pravdepodobnosťou. Test využíva princíp Malej Fermatovej (1.5.2) vety (1.12), pričom sa snaží eliminovať problém s Carmichaeloými číslami [10]. Ak testované číslo n je nepárne, platí $n-1$ je párne číslo. Číslo $n-1$ vieme zapísať ako $2^s * r$, kde $s \geq 1$ a r je nepárne číslo. Ak z postupnosti čísel (1.10) platí, $a^r \equiv 1 \mod n$ alebo ak pre niektoré z čísel tejto postupnosti platí $a^{2^{s-1}*r} \equiv (n-1) \mod n$, pre $s \geq 1$, pre náhodné číslo a, $1 < a \leq n-1$, potom číslo n môže byť prvočíslo s určitou pravdepodobnosťou. Ak n je zložené číslo, potom existuje najmenej $3/4$ svedkov (ang. witness) v intervale $1 < a < n$, že n je zložené číslo [11] [12].

$$a^q, a^{2*q}, a^{2^2*q}, ..., a^{2^{k-1}*q}, a^{2^k*q} \tag{1.10}$$

Millerov-Rabinov test overuje prvočíselnosť s pravdepodobnosťou uvedenou vo vzorci (1.11). Parameter t určuje počet iterácii testu. Zvyšovaním hodnoty parametru t je možné zvýšiť pravdepodobnosť prvočíselnosti testovaného čísla n. Implementácia Millerovho-Rabinovho testu prvočíselnosti je uvedená v algoritme 6.

$$1 - \frac{1}{4^t} \tag{1.11}$$

Časová zložitosť Millerovho-Rabinovho testu je $O(t * \log 3n)$[13].

Algorithm 6 `Millerov-Rabinov test`

Input: n, t

Output: `Prvočíslo s určitou pravdepodobnosťou, Zložené číslo`

1: Nájdi celé čísla r a s aby platilo $(n-1) = 2^s * r$
2: **for** i from 1 to t **do**
3: Vygeneruj náhodné číslo a, $1 < a < n-2$
4: **if** $a^r \equiv 1 \mod n$ **then**
5: **return** "`Prvočíslo s určitou pravdepodobnosťou`"
6: **end if**
7: **for** $j = 1$ to $s-1$ **do**
8: **if** $a^{2^j * r} \equiv (n-1) \mod n$ **then**
9: **return** "`Prvočíslo s určitou pravdepodobnosťou`"
10: **end if**
11: **end for**
12: **end for**
13: **return** "`Zložené číslo`"

1.5.2 Test s využitím Malej Fermatovej vety

Test prvočiselnosti, ktorý je založený na Malej Fermatovej vete (1.12). Test sa zaraďuje medzi ne-deterministické testy prvočiselnosti. Z Malej Fermatovej vety vyplýva ak p je prvočíslo a a je celé číslo, $2 < a < n-2$ a $a \nmid p$ potom platí dôkaz (1.12). Ak by ekvivalencia neplatila, číslo p je zložené číslo a svedkom (ang. witness) je a. Značným problémom v teste je výskyt Carmichaelových čísel, ktoré vyhovujú rovnici (1.12) a pritom testované číslo a nie je prvočíslo. Implementácia s využitím Fermatovho testu je uvedená v algoritme 7. Časová zložitosť Fermatovho testu je $O(t * \log 2n)$ [14].

$$a^{p-1} \equiv 1 \mod p \tag{1.12}$$

Algorithm 7 `Fermatov test prvočíselnosti`

Input: n, t

Output: `Prvočíslo s určitou pravdepodobnosťou, Zložené číslo`

1: **for** i from 1 to t **do**
2: Vygeneruj náhodné číslo a, $2 < a < n - 1$
3: **if** $a^{n-1} \not\equiv 1 \mod n$ **then**
4: **return** "`Zložené číslo`"
5: **end if**
6: **end for**
7: **return** "`Prvočíslo s určitou pravdepodobnosťou`"

1.6 Náhodné čísla v kryptografii

Medzi základné stavebné bloky kryptografických systémov patria aj náhodné čísla, ktoré majú široké spektrum využitia v praxi.

Využitie náhodných číselv kryptografii [15]:

- Generovanie kľúčov pre kryptografické algoritmy.
- Zamedzenie deterministickosti.
- Výplňových schémach.
- Testoch prvočíselnosti.

Základné rozdelenie generátorov náhodných čísel:

- **PRNG** — Sú založene na využívaní algoritmov, ktoré sú periodické. Výstup z takýchto generátorov nie je náhodný, preto sa nazývajú pseudonáhodné generátory náhodných čísel. Medzi najznámejšie pseudonáhodné generátory náhodných čísel patrí lineárny kongruentný generátor.
- **TRNG** — Vyznačujú sa vysokou mierou entropie. Patria do skupiny nedeterministických generátorov náhodných čísel. Sú založené na fyzikálnych javoch, ktorých výstupná hodnota je nepredvídateľná [16].

Cieľom bakalárskej práce nie je detailný opis generátorov náhodných čísel, ale poukázať na možnosti generovania náhodných čísel na operačných systémoch (Ďalej OS) Linux a Windows. Opis a klasifikáciu generátorov náhodných čísel je

možné vyhľadať napríklad v knihe [17].
Na testovanie miery kvality vygenerovaných náhodných čísel sa využívajú štatistické FIPS testy. Problematiku ohľadom testovania náhodných čísel je možné vyhľadať v článku[18].

Problematika funkcie rand() v kryptografii

Medzi najviac využívanú funkciu generovania náhodných čísel v jazyku C patrí funkcia $rand()$, ktorá je zadefinovaná v systémovej knižnici stdlib.h [19] a je nezávislá od OS. Funkcia $rand()$ sa zaraďuje do skupiny PRNG. Funkcia $rand()$ je implementovaná ako lineárny kongruentný generátor. Lineárny kongruentný generátor na základe počiatočnej hodnoty (ang. seed) generuje periodickú postupnosť, ktorá je vopred daná využitým algoritmom. Z pohľadu kryptografie sa to javí ako značný problém. Problematika lineárneho kongruentného generátora v kryptografických systémoch je uvedená v článku [20]. Výhodou funkcie $rand()$ je vysoká rýchlosť generovania náhodných dát.

Generovanie kryptograficky bezpečných náhodných čísel v prostredí OS Windows

Na generovanie kryptograficky bezpečných čísel pre OS Windows môžeme využiť funkciu $rand\_s()$, ktorá je zadefinovaná v systémovej knižnici stdlib.h a je dostupná iba pre OS Windows. Definícia funkcie $rand\_s()$ je zobrazená v zdrojovom kóde 1.4. Vstupným parametrom funkcie $rand\_s()$ je smerník na typ $unsigned\,int$, do ktorého sa uloží vygenerované náhodné číslo. Výstupným parametrom je dátový typ $errno\_t$. Ak vrátená hodnota je 0, generovanie prebehlo úspešne [21]. Názornú ukážku generovania kryptograficky bezpečných náhodných čísel na OS Windows je možné realizovať s využitím zdrojového kódu 1.5.

Zdrojový kód 1.4: Definícia funckie rand()_s

```
1 errno_t rand_s(unsigned int* randomValue);
```

Zdrojový kód 1.5: Generovanie náhodných čísel OS Windows v jazyku C

```
#define _CRT_RAND_S

#include <stdio.h>
#include <stdlib.h>

#define POCET_BAJTOV 20

int main(){

 errno_t err;
 int number;

 for(int i=0;i<POCET_BAJTOV;i++){
    err = rand_s( &number );
        if (err != 0)
        {
            printf_s("The rand_s function failed!\n");
        }
    printf("%02X ", (char)number):
 }
 return 0;
}
```

Generovanie kryptograficky bezpečných náhodných čísel v prostredí OS Linux

Operačný systém Linux na generovanie náhodných čísel využíva metódu načítania náhodných čísel zo súborov. Generátor náhodných čísel zhromažďuje okolité rušenia z ovládačov zariadení a iných zdrojov do zásobníka entropie. Z dát uložených v zásobniku entropie sa vytvárajú náhodné čísla, ktoré sú následne uložene do súborov [22].
Linux má k dispozícii dve kategórie súborov, kde sa ukladajú náhodné čísla:

- **Súbor** $/dev/random$ – je určený na poskytovanie vysokokvalitného, kryptograficky bezpečného náhodného výstupu a vráti iba výstup, pre ktorý je k dispozícii dostatočný náhodný vstup na generovanie výstupu. Ak nie je k dispozícii dostatočný náhodný vstup, čítania zo súboru $/dev/random$ sa zablokujú [23].

- **Súbor** $/dev/urandom$ – poskytuje spoľahlivý zdroj náhodného výstupu, avšak výstup nebude generovaný z rovnakého množstva náhodného vstupu, ak nie je k dispozícii dostatočný vstup. Čítania zo súboru $/dev/urandom$ vždy vrátia požadované množstvo výstupu bez blokovania. Ak nie je k dispozícii dostatočný náhodný vstup, generátor náhodných čísel spracuje alternatívny vstup [23].

Názornú ukážku generovania kryptograficky bezpečných náhodných čísel na OS Linux je možné realizovať s využitím zdrojového kódu 1.6.

Zdrojový kód 1.6: Generovanie náhodných čísel OS Linux v jazyku C

```
1  #include <stdio.h>
2
3  #define POCET_BAJTOV 20
4
5  int main(){
6
7  FILE* fp=fopen("/dev/urandom","r");
8  if (fp==NULL)
9      return 1;
10
11 for(int i=0;i<POCET_BAJTOV;i++){
12  printf("%02X ",fgetc(fp));
13 }
14  return 0;
15 }
```

1.7 Bezpečnosť RSA algoritmu

Najznámejším útokom na RSA algoritmus je faktorizačný útok. Ak existujú efektívne algoritmy pre problém faktorizácie celého čísla, potom možno RSA úplne prelomiť. Takýto efektívny algoritmus zatiaľ neexistuje. Hľadanie takého efektívneho algoritmu je dôležitým nevyriešeným problémom v teórii čísel [24].

1.7.1 Útok s využitím faktorizácie čísel

Faktorizácia sa radí medzi útoky hrubou silou (ang. brute force-attack). Cieľom útoku je rozložiť číslo na súčin dvoch prvočísel. Verejný kľúč RSA algoritmu je zložený z verejného exponentu a modulusu $V_k(e, n)$. Ak sa podarí útočníkovi rozložiť

modulus n na súčin prvočísel p a q, bude schopný vypočítať $\phi(n)$ a tým pádom aj súkromný exponent d [25].

Súčasným najrýchlejším algoritmom na faktorizáciu BN čísel je GNFS (ang. General Number Field Sieve) algoritmus [26].

Kľúčovým elementom pre bezpečnosť RSA algoritmu je zvolenie vhodnej veľkosti modulusu. Veľkosť modulusu závisí od požadovanej bezpečnosti akú si užívateľ zvolí. Všeobecne platí čim väčší modulus, tým väčšia bezpečnosť, ale aj pomalšie vykonávanie operácií v RSA algoritme. Dĺžky modulusu odporúčané štandardom ETSI sú uvedené v tabuľke 1.1 [27]. Dĺžka optimálnej veľkosti modulusu by sa mala voliť po zvážení bezpečnostných potrieb:

- Hodnota chránených údajov.
- Ako dlho je potrebné údaje chrániť.

Parameter	1 - 3 rokov	6 rokov
Dĺžka kľúča [bitov]	$>$ 1900	$>$3000

Tabuľka 1.1: Odporúčaná dĺžka modulusu štandardom ETSI

1.7.2 Útok na rovnakú hodnotu modulusu

Predpokladajme že v komunikácii sa nachádzajú užívatelia A a B. Užívateľ A aj užívateľ B si vygenerujú svoje vlastné RSA kľúče, pričom ich modulus bude rovnaký. Zjavním problémom pri tomto type útoku je, rovnaká správa zašifrovaná dvoma rôznymi exponentmi, pričom užívatelia zdieľajú rovnaký modulus. Útočník dokáže obnoviť pôvodnú správu bez znalosti niektorého z dešifrovacích exponentov [28]. Nech m je správa, ktorú budú užívatelia šifrovať. Užívatelia disponujú rôznymi exponentmi e_A a e_B a zdieľajú rovnaký modulus. Vykonaním šifrovacieho procesu dostavajú zašifrované správy c_A a c_B zobrazené v rovnici (1.13).

$$c_A = m^{e_A} \mod n \ \ a \ \ c_B = m^{e_B} \mod n \tag{1.13}$$

Útočník pozná verejne dostupné n, e_A, e_B , c_A a c_B. Útočník pomocou rozšíreného Euklidovho algoritmu dokáže nájsť hodnoty r a s, pre ktoré platí.

$$re_A + se_B = 1 \tag{1.14}$$

Za predpokladu, že r alebo s je záporné číslo, predpokladajme že v tomto prípade je číslo r záporné, vieme s využitím rozšíreného Euklidovho algoritmu vypočítať c_A^{-1}. Dosadením týchto hodnôt do rovnice (1.15) dostávame pôvodnú správu m.

$$(c_A^{-1})^{-r} * c_B^s = m \mod n \tag{1.15}$$

1.7.3 Útok na nízku hodnotu verejného exponentu

Útok s využitím nízkej hodnoty verejného exponentu e je záložný na rovnici (1.16), kde hodnoty e, n su prvky verejného kľúča $VK = (e, n)$ a m je počiatočná správa a c je zašifrovaná správa [29].

$$c = m^e \mod n \quad m < n^{\frac{1}{e}} \tag{1.16}$$

Útočník dokáže získať pôvodnú správu realizovaním rovnice (1.17).

$$m = c^{\frac{1}{e}} \tag{1.17}$$

V praxi na zvolenie verejného exponentu e sa využívajú vyššie hodnoty. Najčastejšou zvolenou hodnotou je 4-té Fermatovo číslo [1].

1.7.4 Útoky realizované na zašifrované texty

Tieto typy útokov sa nazývajú aj CCA) (ang. Chosen-Ciphertext útoky, ktoré využívajú zašifrovaný text a implementačné nedostatky RSA algoritmu na realizáciu útokov [30].

ÚTOK č.1

Predpokladajme že máme k dispozícii užívateľov A, B. Užívateľ B zašifruje správu m pre užívateľa A s využitím verejného kľúča užívateľa A (1.18).

$$c = m^{e_a} \mod n_a \tag{1.18}$$

Útočník na obnovenie správy zvolí náhodné čísla r, $r < n_a$ a s využitím verejného kľúča užívateľa A vypočíta hodnoty(1.19).

$$\begin{aligned} x &= r^{e_a} \mod n_a \\ y &= xc \mod n_a \\ t &= r^{-1} \mod n_a \end{aligned} \tag{1.19}$$

[1] 4-té Fermatovo číslo = 65 537

Následne útočník zašle užívateľovi A žiadosť o podpis zašifrovanej správy y. Užívateľ A zašle podpísanú správu útočníkovi v tvare (1.20). Metódy podpisovania správ nie sú náplňou tejto bakalárskej práce, bližšie informácie k podpisovaniu správ v RSA algoritme sú dostupné napríklad v knihe [2].

$$u = y^{d_a} \mod n_a \tag{1.20}$$

Útočník realizuje výpočet za získanie pôvodnej správy, ktorá bola zašifrovaná užívateľom B (1.21).

$$tu \mod n_a = r^{-1}y^{d_a} \mod n_a = r^{-1}x^{d_a}c^{d_a} \mod n_a = m \tag{1.21}$$

ÚTOK č.2

Predpokladajme že máme k dispozícii účastníka A a účastníka B. Účastník B zašifruje správu pre účastníka A v tvare (1.22).

$$c = m^{e_a} \mod n_a \tag{1.22}$$

Útočník môže realizovať MITM útok a riadiť komunikáciu medzi účastníkmi A a B. Útočník odchytí zašifrovanú správu c a upraví ju na tvar (1.23) a zašle ju účastníkovi A.

$$c' = (c * 2^{e_a}) \mod n_a \tag{1.23}$$

Účastník A dešifruje správu v tvare (1.24). Účastník A tak získal podvrhnutú správu, ktorá je odlišná od pôvodnej správy, ktorú zaslal účastník B.

$$c^{d_a} \mod n_a = (c*2^{e_a})^{d_a} \mod n_a = c^{d_a} * 2^{ed_a} \mod n_a = m*2 \mod n_a \tag{1.24}$$

2 Optimálna výplň asymetrického šifrovania – OAEP

Výplňová schéma OAEP bola vytvorená v roku 1994 autormi Mihir Bellare a Phillip Rogaway [31]. V predošlej kapitole (1.7) som sa zaoberal špecifickými útokmi na textbook RSA, ktoré sa zameriavali na matematické nedostatky RSA algoritmu. Výplňová schéma OAEP slúži na čiastočne zamedzenie týmto typom útokov. Výplňová schéma OAEP je definovaná v štandarde [6].

Predpokladajme, že šifrovaný text odoslaný príjemcovi spozoroval útočník, ktorý potom postupuje nasledovne:

1. Útočník zvolí otvorený text, ktorý by sa po zašifrovaní mohol podobať originálnemu zašifrovanému textu odosalnému príjemcovi.
2. Útočník zašifruje otvorený text pomocou verejného kľúča známeho príjemcu. Ak sa výsledok zhoduje s pozorovaným šifrovaným textom, potom bol odhad správny ak nie, útočník sa pokúsi o ďalší odhad [5].

2.1 Stavebné bloky výplňovej schémy OAEP

Aby výplňová schéma OAEP bola odolná voči CCA útokom, musí obsahovať kryptografické metódy, ktoré budu zaručovať bezpečnosť.

Základne kryptografické metódy využívane v OAEP:

- **Náhodné čísla** – Slúžia na zamedzenie deterministickosti šifrovanej správy.
- **Hashovacie funkcie** – Slúžia na overenie integrity šifrovanej správy.
- **Separátor 01** – Slúži na kontrolu dešifrovanej správy po dešifrovaní.

2.1.1 Hashovacie funkcie

Hashovacie funkcie patria do kategórie jednocestnych funkcii (ang. one way function) tzn. pre danú správu M je jednoduché vypočítať $H(M)$ podľa (2.1), ale inverznú operáciu z $H(M)$ vypočítať M je veľmi obtiažné. Cieľom tejto bakalárskej práce nie je detailný opis hashovacích funkcii, ale principiálne využitie pre výplňovú schému OAEP. Detailnejší opis ohľadom hashovacích funkcií je uvedený v knihe [32].

$$M \mapsto H(M) \tag{2.1}$$

Výplňová schéma OAEP využíva hashovacie funkcie, ktoré slúžia na overenie integrity pôvodnej správy. V OAEP je doporučené využívať hashovacie funckcie z rodiny SHA (ang. Secure Hash Algorithm), ktoré sú uvedené v tabuľke 2.1 a sú doporučené štandardom [6].

	SHA-1	SHA-256	SHA-384	SHA-512
Výstup	160	256	384	512
Dĺžka správy	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Veľkosť slova	32	32	64	64
*EKB	80	128	192	256

Uvedené hodnoty v tabuľke sú v bitoch

*EKB= Ekvivalentná kryptografická bezpečnosť

Tabuľka 2.1: Porovnanie hashovacích funkcií

2.1.2 Funkcia generovania masky MGF–1

Funkcia generovania masky (ang. mask generation function) je kryptografická funkcia podobná hashovacej funkcii. Hashovacia funkcia ma fixnú veľkosť dĺžky výstupu, pričom MGF ma variabilnú dĺžku výstupu. Základným stavebným blokom MGF sú hashovacie funkcie, ktoré sa vykonávajú v rundách, taktiež hashovacie funkcie zabezpečujú kryptografickú bezpečnosť funkciám generovania masky. Vstupom MGF je oktetový reťazec variabilnej dĺžky a výstupnom je oktetový reťazec požadovanej dĺžky. MGF sú deterministické, výstupný oktetový reťazec závisí na vstupnom oktetovom reťazci. Vo výplňovej schéme OAEP je MGF využívaná dvakrát vo forme Feistelovej štruktúry [6] [33]. Algoritmus MGF je uvedený v zdrojovom kóde 8.

Algorithm 8 MGF-1

Input: $mgfSeed$ seed, z ktorého sa generuje maska, oktetový reťazec
$maskLen$ dĺžka masky v oktetoch, najviac 2^{32} hLen
$Hash$ Typ hashovacej funckie, s dĺžkou hLen oktetov

Output: $mask$ maska, oktetový reťazec dĺžky maskLen

1: **if** $maskLen > 2^{32} hLen$ **then**
2: **return** "`Maska príliš dlhá`"
3: **end if**
4: Nech T je oktetový reťazec.
5: **for** i from 0 to $\frac{maskLen}{hLen} - 1$ **do**
6: C = I2OSP($counter$,4)
7: $T = T$ || Hash($mgfSeed$ || C)
8: **end for**
9: **return** "`Vráť` T `o dĺžke` $maskLen$ `oktetov`"

2.1.3 Náhodný oktetový reťazec SEED

V OAEP je SEED náhodne vygenerovaný oktetový reťazec pomocou generátorov náhodných čísel. Odporúča sa, aby boli pseudonáhodné oktety v OAEP generované nezávisle pre každý proces šifrovania, najmä ak rovnaké údaje sú vstupom do viacerých šifrovacích procesov [6]. Cieľom pridania SEED do výplňovej schémy OAEP je zamedzenie deterministickosti zašifrovanej správy.

2.2 Realizácia šifrovania a dešifrovania v OAEP

Pre realizáciu šifrovania a dešifrovania s využitím výplňovej schémy OAEP je nutné vopred zvoliť vhodnú hashovaciu funkciu pre realizáciu šifrovania aj dešifrovania. Odporúčané hashovacie funkcie pre vyplňovú schému OAEP sú uvedené v tabuľke 2.1. Následne je nutnosťou prispôsobiť dĺžku správy, ktorú budeme šifrovať. Dĺžka správy sa odvíja od zvolenej hashovacej funkcie, aby platila rovnosť (2.2).

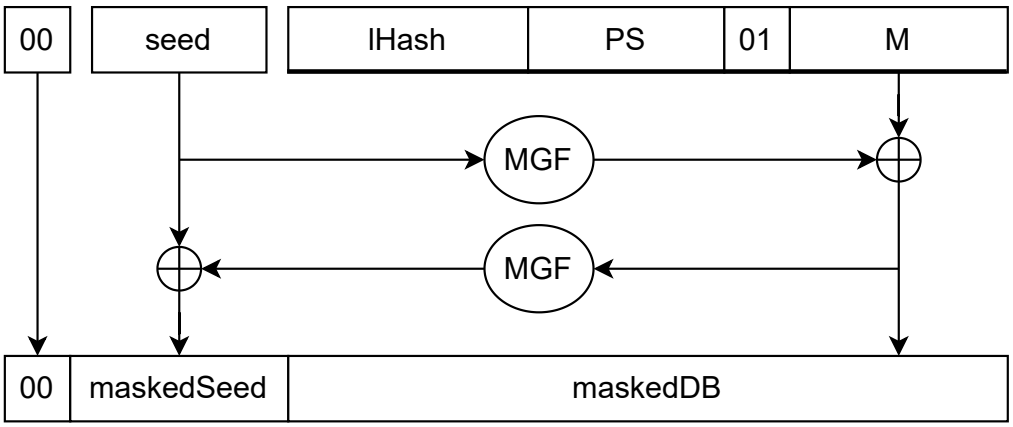
$$mLen \leq k - 2 * hLen - 2 \tag{2.2}$$

Kde $mLen$ — Dĺžka správy, k — Dĺžka modulusu, $hLen$ — Dĺžka výstupu zvolenej hashovacej funkcie.

Šifrovanie sa realizuje pomocou algoritmu (9). Na obrázku 2.1 je znázornená bloková schéma šifrovacieho procesu.

Algorithm 9 EME-OAEP-Encode(M, L, emLen)

1: $hash$ – Hashovacia funkcia, (n, e) – Verejný kľúč, M – správa, $emLen$ - Dĺžka správy M, L – Label
2: Ak $L >$ dĺžka maximálnej správy pre hashovaciu funkciu - Vráť chybu, label príliš dlhý
3: Ak $mLen > emLen - 2hLen - 1$, Vráť príliš dlhá správa.
4: Vygeneruj oktetový reťazec PS o veľkosti $emLen - mLen - 2hLen - 1$, ktorý bude obsahovať iba hodnotu 0.
5: Nech $lHash = Hash(L)$, o velkosti $hLen$.
6: Zreťaz $pHash, PS, sprvu\ M$ a výplňový charakter $01\ DB = lHash \parallel PS \parallel 01 \parallel M$.
7: Vygeneruj náhodný oktetový reťazec dĺžky $hLen$.
8: Nech $dbMask = MGF(seed, emLen - hLen)$.
9: Nech $maskedDB = DB \oplus dbMask$.
10: Nech $seedMask = MGF(maskedDB, hLen)$.
11: Nech $maskedSeed = seed \oplus seedMask$.
12: Nech $EM = maskedSeed \parallel maskedDB$.
13: Výstup EM.



Obr. 2.1: Šifrovanie správy s využitím OAEP

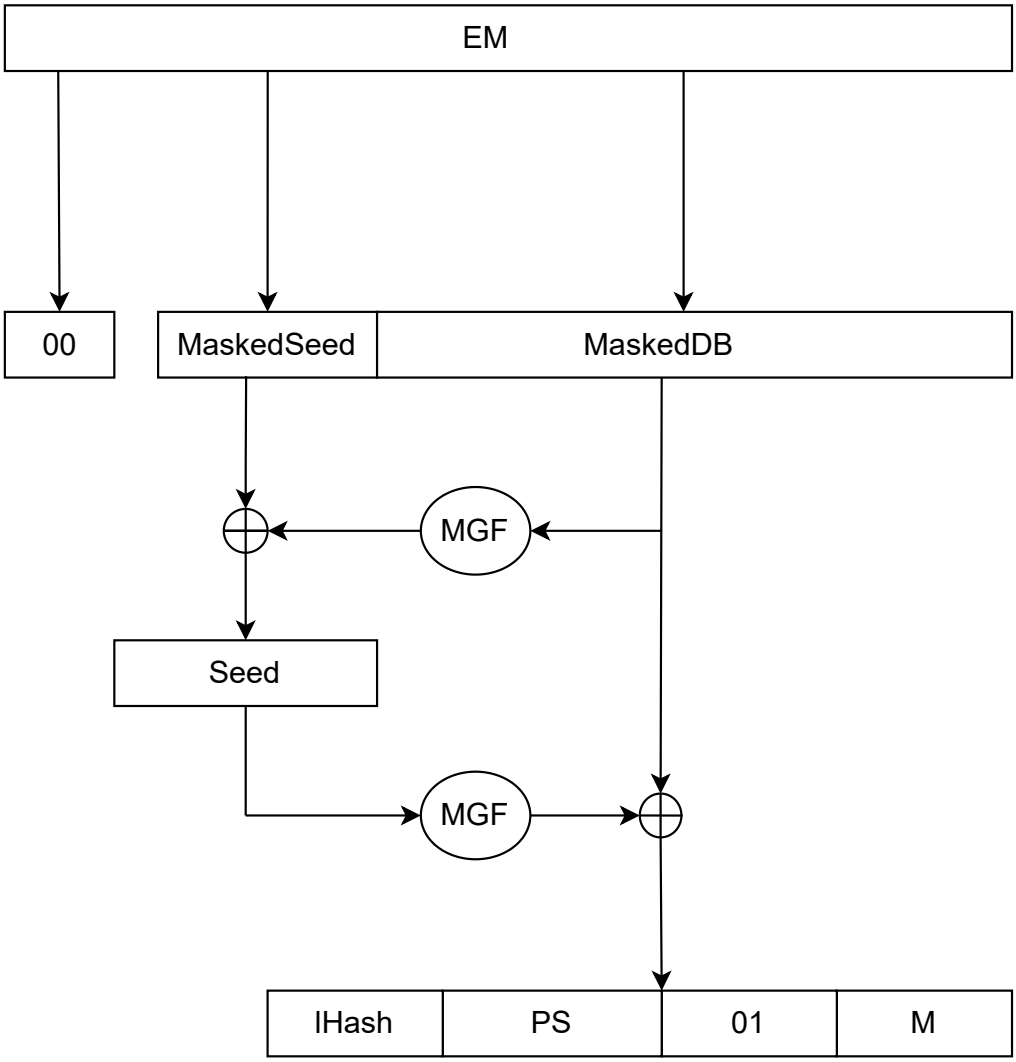
Dešifrovanie sa realizuje pomocou algoritmu (10) Na obrázku 2.2 je znázornená bloková schéma dešifrovacieho procesu.

Algorithm 10 EME-OAEP-Decode(EM, L)

1: $Hash$ – hashovacia funckia, k – DĺžkA modulusu, L - Label
2: Ak $L >$ dĺžka maximálnej správy pre hashovaciu funkciu - Vráť chybu, label príliš dlhý.
3: Ak $emLen < 2hLen + 1$, Dešifrovacia chyba.
4: Nech $maskedSeed$ je prvých $hLenoktetov\ EM$ a nech $maskedDB$ sú zvyšné oktety dlzky $emLen - hLen$.
5: Nech $seedMask = MGF(maskedDB, hLen)$.
6: Nech $seed = maskedSeed \oplus seedMask$.
7: Nech $dbMask = MGF(seed, emLen - hLen)$.
8: Nech $DB = maskedDB \oplus dbMask$.
9: Nech $lHash = Hash(L)$, dlzky $hLen$ oktetov.
10: Oddel DB na oktetový reťazec $pHAsh$‘ pozostávajúci z prvých hLen oktetov z DB, PS pozostávajúci zo sekvencie 0, 01 oktetu a správy M

$$DB = lHash' \parallel PS \parallel 01 \parallel M$$

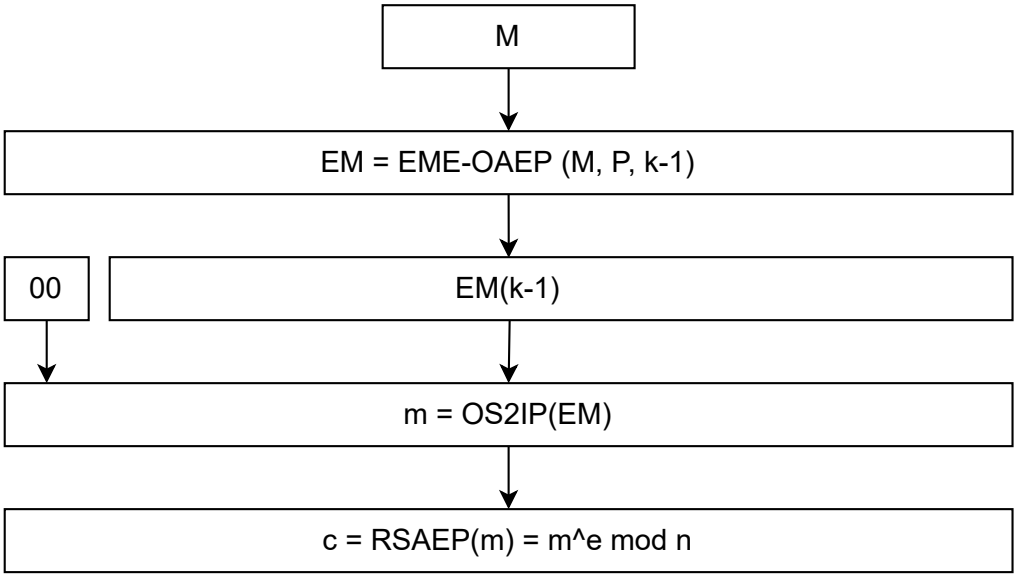
Ak sa v DB za sekvenciou 0 nenachádza charakter 01 - Dešifrovacia chyba.
11: Ak $lHash$‘ $\neq lHAsh$ - Dešifrovacia chyba.
12: Vráť správu M.



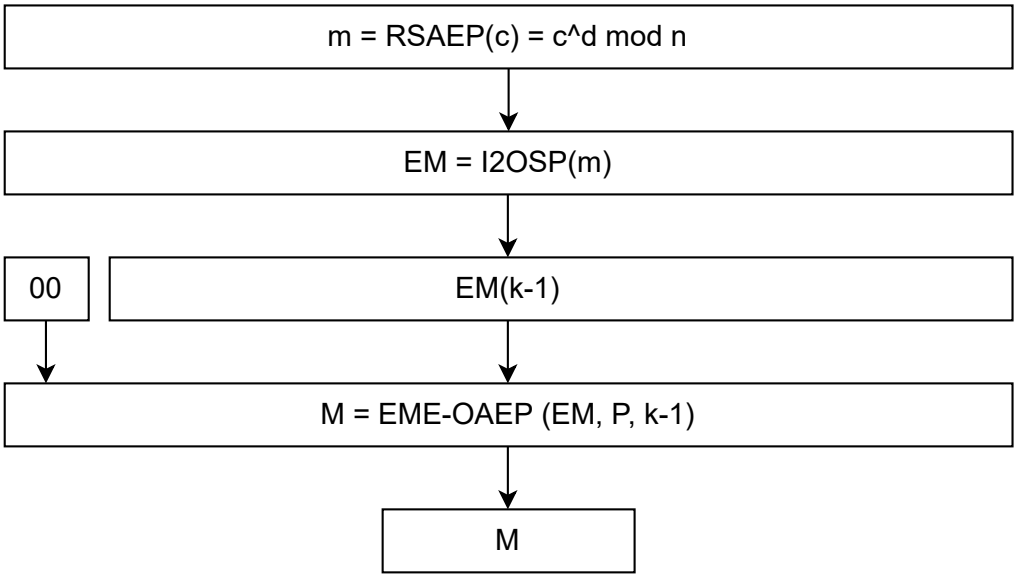
Obr. 2.2: Dešifrovanie správy s využitím OAEP

2.3 Výplňová schéma OAEP v spojení s RSA

Najčastejšiu možnosťou využitia výplňovej schémy OAEP v praxi je v spojení s RSA algoritmom. Proces šifrovania v spojení s RSA algoritmom je uvedený na obrázku 2.3. Proces dešifrovania v spojení s RSA algoritmom je uvedený na obrázku 2.4.



Obr. 2.3: Šifrovanie OAEP v spojení s RSA



Obr. 2.4: Dešifrovanie OAEP v spojení s RSA

3 Kryptografické knižnice

Na internete je možné vyhľadať množstvo dostupných implementácii kryptografických knižníc v jazyku C od amatérskych implementácii až po vysoko sofistikované implementácie, ktoré sú využívané v praxi. V tejto bakalárskej práci som sa zameral hlavne na vyhľadávanie knižníc, ktoré obsahujú a dokážu vykonávať operácie s využitím BN čísel.

Pri vyhľadávaní kryptografických knižníc na internete ma zaujali tieto kryptografické knižnice:

1. Kryptografická knižnica MCRYPTO.
2. Kryptografická knižnica KOKKE - komplexná implementácia.
3. Kryptografická knižnica STUDENT - amatérska implementácia.
4. Kryptografická knižnica OPENSSL - sofistikovaná implementácia.

Z hľadiska uplatnenia kryptografických knižníc v praxi môžeme kryptografické knižnice rozdeliť do dvoch skupín, amatérske implementácie a sofistikované implementácie:

Charakteristika amatérskych implementácii:

- Pracujú s malými číslami, ktoré sú reprezentované natívnymi premennými v jazyku C, ktorých rozsah je v intervale $[0, 2^{64} - 1]$.
- Neobsahuju algoritmy na generovanie kľúčov pre RSA algoritmus, spoliehajú sa na zadávanie pevných hodnôt, ktoré si užívateľ svojvoľne zvolí.
- Implementácie algoritmov nie sú odolné voci špecifickým typom útokov.
- Amatérske implementácie zvyknú obsahovať závažné nedostatky v zdrojových kódoch.

Charakteristika sofistikovaných implementácii:

- Pre reprezentovanie BN čísla využívajú statické alebo dynamické polia, v ktorých sú indexované vektory čísel so základnom (ang. base).
- Obsahujú algoritmy na generovanie RSA kľúčov.
- Sú odolné voči špecifickým typom útokov.
- Využívajú optimalizované algoritmy pre čo najviac efektívnu prevádzku.
- Nie sú závisle na platforme [Windows, Linux].
- Na generovanie náhodných čísel využívajú TRNG generátory náhodných čísel.
- Zápis a načítavanie RSA kľúčov zo súboru vykonávajú vo špecifických typoch formátov.

3.1 Dodatočné rozšírenia pre analyzované kryptografické knižnice

V rámci praktickej časti bakalárskej práce som realizoval dodatočne projekty, ktoré obsahujú implementácie hashovacích funkcií a výplňovej schémy OAEP v jazyku C. Realizáciu projektov s hashovacími funkciami a OAEP výplňovou schému bolo nutné realizovať, kvôli tomu, že nie každá implementácia to obsahovala a v ďalších kapitolách sú využívané metódy na porovnanie kryptografických knižníc. Na internete som vyhľadal implementácie hashovacich funkcií v jazyku C, ktoré som následne využil v praktickej časti tejto bakalárskej práce.

Vyhľadané hashovacie implementácie:

1. Hashovacia funckia SHA-1 [1].
2. Hashovacia funckia SHA-256 [2].
3. Hashovacia funckia SHA-512 [3].

[1] Dostupná na: `https://github.com/B-Con/crypto-algorithms`
[2] Dostupná na: `https://github.com/B-Con/crypto-algorithms`
[3] Dostupná na: `https://github.com/orlp/ed25519`

3.1.1 Implementácia hashovacích funkcií

V rámci praktickej časti bakalárskej práce som analyzoval implementácie hashovacich funkcií, aby som ich vedel následne využiť ako rozšírenia pre kryptografické knižnice, ktoré neobsahovali implementácie hashovacích funkcií. Hlavným cieľom vyhľadania implementácii hashovacích funkcii bolo zameranie sa na štandard, ktorý odporúča hashovacie funkcie pre výplňovú schému OAEP [6].Zdrojové súbory k hashovacím funkciám sa nachádzajú v prílohe G.

Problémy s kompatibilitou

V rámci analýzy hashovacích funkcii som narazil na problémy s kompatibilitou, konkrétne u hashovacích funkciách SHA-1 a SHA-256 dostupných v prílohe G v adresároch SHA_1 a SHA_256. Vytvorením vlastného zdrojového súboru pre kontrolu hashovacích funkcii podľa testovacích vektorov na stránke [4] som narazil na problém pri kompilácii projektu na OS Windows. S použitým GCC prekladačom sa pri kompilácii vyskytla chybová hláška duplicity názvu typu premennej $WORD$, ktorá bola použitá v zdrojových kódoch sha1.h a sha256.h. Riešením bolo zmeniť názov typu premennej $WORD$ na ONE_WORD.

Rozšírenie projektu s hashovacími funkciami

Ako som v predošlej podkapitole avizoval, cieľom vyhľadania implementácii hashovacích funkcií bolo rozšírenie výberu hashovacích funkcii pre výplňovú schému OAEP. Implementoval som enumeračný typ $HASH\_FUNCTION$, ktorý dokáže uchovať typ využitej hashovacej funkcie. Tento typ je ďalej využívaný v praktickej realizácii tejto bakalárskej práce v OAEP implantácii, ktorá je detailne opísaná v ďalšej podkapitole. Implementácia enumeračného typu je uvedená v zdrojovom kóde 3.1.

Zdrojový kód 3.1: Enumeračný typ pre uloženie typu hashovacej funkcie

```
1 typedef enum
2 {
3       SHA1_FUNC, /* SHA-1 */
4       SHA256_FUNC, /* SHA-256 */
5       SHA512_FUNC /* SHA-512 */
6 } HASH_FUNCTION;
```

[4] Dostupné na: `https://www.di-mgt.com.au/sha_testvectors.html`

3.1.2 Implementácia výplňovej schémy OAEP

V rámci praktickej realizácie som sa zameral aj na vyhľadanie výplňovej schémy OAEP v jazyku C. Hľadaním na internete som narazil na zaujímavú implementáciu, dostupnú na stránke [5] od autora Rupan. Zdrojové súbory k výplňovej schéme OAEP sa nachádzajú v prílohe F.

Implementácia ma zaujala z nasledovných dôvodov:

- Jednoduchá čitateľnosť zdrojových kódov.
- Obsahovala kryptograficky bezpečné generovanie náhodných čísel pre OS Linux.

Avšak táto implementácia obsahovala aj svoje nedostatky, ktoré som musel modifikovať aby implementácia výplňovej schémy OAEP bola plnohodnotne použiteľná pre kryptografické knižnice využívané v praktickej časti tejto bakalárskej práce. Opis a modifikácia značných nedostatkov tejto implementácie sú uvedené v nasledujúcej podkapitole.

Opis a modifikácia nedostatkov implementácie

Analýzou implementácie som zistil, že táto implementácia je primárne určená pre OS Linux keďže využívala:

- Generovanie náhodných dát zo súboru $/dev/urandom$, ktorý som bližšie popísal v kapitole 1.6.
- Systémovú knižnicu arpa/inet.h, ktorá je dostupná iba pre OS Linux.
- Funkcia $oaep\_decode()$ neumožňovala zápis výsledku po procese dešifrovania do premennej.

Implementácia funkcie $oaep\_decode()$ neumožňovala uloženie výsledku po procese dešifrovania OAEP do premennej. Vo funkcii $oaep\_decode()$ som modifikoval argumenty funkcie pridaním premennej $uint8\_t * decoded$, ktorá slúži na uloženie výsledku. V zdrojovom kóde, dostupnom v prílohe F súbor oaep.c vo funkcii $oaep\_decode()$ som pridal blok kódu 3.2, ktorý umožňuje zápis do premennej.

[5] Dostupná na: `https://github.com/Rupan/rsa`

Zdrojový kód 3.2: Zápis výsledku do premennej decoded

```
1  for(int x=0;x<pass;x++){
2   decoded[x] = EM[k-pass+x];
3  }
```

Za účelom využitia tejto implementácie na operačných systémoch Linux a Windows bolo nutné vykonať:

- Pridanie generovania náhodných čísel pre OS Windows.
- Nahradenie funkcie $htonl()$, ktorá bola využívaná vo funkcii $MGF1()$.

Pridanie generovania náhodných čísel pre OS Windows

V rámci pridania generovania náhodných čísel pre platformu Windows som realizoval modifikáciu funkcie $fill\_random()$ pridaním dodatočného zdrojového kódu, ktorý realizuje generovanie náhodných čísel na OS Windows a je opísaný v podkapitole 1.6. Následne som využil direktívy preprocesora, ktoré umožnili vybrať vhodný generátor náhodných čísel pre konkrétny OS. Blok kódu, ktorý realizuje generovanie kryptograficky bezpečných náhodných čísel na závislosti od OS Linux a Windows je uvedený v zdrojovom kóde 3.3.

Zdrojový kód 3.3: Generovanie kryptograficky bezpečných náhodných čísel na OS Linux a Windows

```
1  #ifdef __linux__
2    int32_t fd, bytes;
3    fd = open("/dev/urandom", O_RDONLY);
4    if(fd == -1) return -1;
5    bytes = read(fd, dest, len);
6    close(fd);
7    if(bytes != (ssize_t)len) return -1;
8
9  #elif defined _WIN32
10   unsigned int num;
11   for (int i = 0; i < len; i++) {
12       errno_t err = rand_s(&num);
13       if(err != 0) return -1;
14       dest[i] = (uint8_t)num;
15   }
16 #endif
```

Modifikácia funkcie MGF-1

Modifikáciu funkcie MGF1 som realizoval kvôli nasledovným dôvodom:

- Pridanie možnosti výberu hashovacej funckie.
- Vytvorenie kódu, ktorý nahradí funkciu $htonl()$.

Pridaním dodatočného typu premennej $HASH\_FUNCTION$ do funkcii $oaep\_encode()$ a $oaep\_decode()$ som vytvoril možnosť pre výber vhodnej hashovacej funkcie. Štruktúra premennej typu $HASH\_FUNCTION$ je uvedená v zdrojovom kóde 3.1.

Funckia $htnol()$ je implementovaná v systémovej knižnici arpa/inte.h, ktorá je dostupná iba na OS Linux. Na internete som vyhľadal dokumentáciu ohľadom funkcie $htnol()$, aby som získal prehlaď, čo daná funkcia realizuje. Dokumentácia funkcie $htonl()$ je dostupná na stránke[6]. Čítaním dokumentácie som získal prehlaď o funkcii a jej využitia. Funkcia slúži na prevod z low-endian na big-endian v sieťovej komunikácii. Rozhodol som sa túto funkciu aj knižnicu arpa/inet.h úplne odstrániť zo zdrojového kódu a nahradiť ju vlastnou implementáciou, ktorá realizuje prevod z low-endian na big-endian. Funkcia realizujúca prevod z low-endian na big-endian je uvedená v zdrojovom kóde 3.4.

Zdrojový kód 3.4: Náhrada funkcie htonl()

```
1 uint32_t x;
2 uint8_t* s = (uint8_t*)&x;
3 uint32_t res=(uint32_t)(s[0]<< 24 | s[1]<< 16 | s[2]<< 8 | s[3]);
```

3.2 Kryptografická knižnica MCRYPTO

Kryptografická knižnica MCRYPTO od autora Dang Nguyen Duc [7]. Autor uvádza, že táto knižnica je určená pre študentov, ktorí sa zaujímajú o kryptografiu s verejným kľúčom a hľadajú komplexnú kryptografickú knižnicu [34]. Autor knižnice poukazuje, že cieľom pre návrh knižnice bolo:

- Komplexnosť.
- Jednoduchá čitateľnosť zdrojových kódov.

[6]Dostupná na:`https://linux.die.net/man/3/htonl`
[7]Dostupná na: `https://code.google.com/archive/p/libmcrypto/`

- Jednoducho použiteľná a modifikovateľná.

Kryptografická knižnica MCRYPTO ma po analýze zdrojových kódov, oslovila bodmi, ktoré už samotný autor opísal v predošlom zozname. Taktiež ma zaujala funkcionalitami, ktoré ponúkala.

Zoznam funkcionalít:

1. Prehľadnosť zdrojových kódov.
2. Názvoslovie premenných, ktoré sú zhodné so štandardom PKCS#1.
3. Implementáciou výplňovej schémy OAEP.
4. Množstvo predprogramovaných funkcii na:
   - Prevod BN čísel medzi dvojkovou, decimálnou, hexadecimálnou a osmičkovou sústavou.
   - Konverziu reťazcov do RADIX-64 formátu (ang. base64 format).
5. Množstvo matematických operácií s BN číslami.
6. Možnosti výberu náhodných čísel podľa parametra (Kryptograficky bezpečné náhodné čísla, lineárny konguentný generátor).
7. Sadou predprogramovaných hashovacích funkcií.
8. Implementáciou na podpisovanie správ pomocou RSA algoritmu.
9. Možnosť modifikovania funkcií na dynamickú správu pamäte.
10. Implementáciou kryptografického algoritmu eliptických kriviek ECC (ang. Elliptic Curve Cryptography).

Následne po zvážení týchto bodov sa táto knižnica stala potenciálnym kandidátom pre následne využite a otestovanie v tejto bakalárskej práci.

3.2.1 Opis kryptografickej knižnice MCRYPTO

Knižnica je napísaná v jazyku C a mala by byť v súlade so všetkými operačnými systémami založenými na Unixe. Knižnica je primárne určená pre OS Linux, pričom realizáciou dodatočných úprav, ktoré sú uvedené v sekcii 3.2.4 môže byť použiteľná aj pre OS Windows.

Knižnica MCRYPTO pozostáva z nasledujúcich časti:

- **Aritmetickej knižnice BIGDIGITS** - Knižnica, ktorá obsahuje implementáciu základných matematických operácii pre BN čísla. Bližšiemu opisu tejto knižnice je venovaná osobitná sekcia 3.2.2.
- **Eliptické krivky** - Knižnica obsahuje implementáciu ECC s využitím GF.
- **RSA** - Knižnica obsahuje implementáciu RSA algoritmu a výplňovej schémy OAEP.

3.2.2 Charakteristika knižnice BIGDIGITS

Autorom knižnice BIGDIGITS je David Ireland [8]. Knižnica BigDigits je voľne dostupná (ang. open source) kryptografická knižnica. Matematické algoritmy, ktoré sú využívané v knižnici sú uvedené v knihe [35]. Knižnica BigDigits je navrhnutá tak, aby pracovala s kladnými číslami [36].

Knižnica ma dve rozhrania, knižnicu „bd“ a základnú knižnicu „mp“:

1. **bd** - Rozhranie využíva dynamickú alokáciu pamäte.
2. **mp** - Rozhranie využíva statické polia s fixnou dĺžkou. Rozhranie mp je v porovní s rozhraním bd rýchlejšie.

3.2.3 Využité generátory náhodných čísel

Knižnica BIGDIGITS využíva dva typy generátorov náhodných čísel:

- **Generátor pseudonáhodných čísel** - Generuje kryptograficky nie bezpečné náhodné čísla. Implementácia algoritmu je realizovaná ako lineárny kongruentný generátor, v tomto prípade to je funkcia $rand()$, ktorá je zadefinovaná v systémovej knižnici stdlib.h. Počiatočná hodnota (ang. seed) je zvolená s využitím funkcie $srand((unsigned)time(NULL))$. Problém lineárneho kongruentného generátora v kryptografii je uvedený v sekcii 1.6. Funkcia $DIGIT\_T\ spPseudoRand(DIGIT\_T\ lower, DIGIT\_T\ upper)$ realizuje generovanie pseudonáhodných čísel v rozsahu [lower, upper]. Zdrojový kód funkcie $spPseudoRand()$ je uvedený z zdrojovom kóde 3.5.

[8] `https://www.di-mgt.com.au/bigdigits.html`

- **Generátor kryptograficky bezpečných čísel** - Na generovanie kryptograficky bezpečných náhodných čísel je využitá funkcia $int\ prng(BYTE*\ buf,\ int\ buf\_size)$. Funkcia $prng()$ generuje náhodné čísla pre OS Linux s využitím súboru $/dev/urandom$, ktorý bol opísaný v sekcii 1.6. Táto funkcia obsahovala iba generovanie náhodných čísel pre OS Linux, úpravou tejto funkcie som realizoval aj generovanie pre OS Windows. Problematike úprav funkcie $prng()$ pre nezávislosť na OS som sa venoval v sekcii 3.2.4 tejto bakalárskej práce.

Zdrojový kód 3.5: Generátor pseudonáhodných čísel basicstyle

```
1  DIGIT_T spPseudoRand(DIGIT_T lower, DIGIT_T upper)
2  {
3          /*      Returns a pseudo-random digit.
4                  Handles own seeding using time
5                  NOT for cryptographically-secure random numbers.
6          */
7          static unsigned seeded = 0;
8          UINT i;
9          double f;
10
11         if (!seeded)
12         {
13                 /* seed with system time */
14                 srand((unsigned)time(NULL));
15         /* Throw away a random few to avoid similar starts */
16                 i = rand() & 0xFF;
17                 while (i--)
18                         rand();
19                 seeded = 1;
20         }
21         f = (double)rand() / RAND_MAX * upper;
22         return (lower + (DIGIT_T)f);
23 }
```

Knižnica obsahovala aj možnosť výberu generátora náhodných čísel, ktorý je zohľadnený v zdrojovom kóde 3.6.

Knižnica ponúkala dve možnosti:

1. **Kryptograficky bezpečné RNG**.

2. **Lineárny kongruentný generátor**.

Zdrojový kód 3.6: Zvolenie RNG v knižnici mcrypto.h

```
1 #define STRONG_RANDOM   1 // Cryptographically secure
2 #define STRONG_RANDOM   0 //Cryptographically unsecure
```

3.2.4 Implementačné nedostatky knižnice MCRYPTO

Knižnica MCRYPTO ako som spomínal v úvode tejto kapitoly je optimalizovaná pre OS Linux. Knižnica bola koncepčne navrhnutá pre 32-bitové operačné systémy.
Realizácia úprav v MCRYPTO knižnici:

- **Funkcia** $prng()$ - Funkcia slúži na generovanie náhodných čísel pre OS Linux. Pre použiteľnosť aj pre OS Windows bolo nutné upraviť zdrojový kód funkcie $prng()$ na tvar 3.7.
- **Architektúra PC systému** - Knižnica MCRYPTO je optimalizovaná pre 32-bitové systémy, pričom využíva natívnu premennú $unsigned\ long$ v jazyku C. Natívna premenná $unsigned\ long$ ma rozličnú veľkosť pre 32-bitové a 64-bitové počítačové systémy, veľkosti sú uvedené v tabuľke 3.1. Riešením bolo nahradiť všetky použité premenné typu $unsigned\ long$ na typ $uint32\_t$, ktorý je definovaný v systémovej knižnici stdint.h. Veľkosť typu $uint32\_t$ je fixne daná pre 32-bitové aj 64-bitové systémy a to 4 bajty.

Nastavenie generátora pre špecifickú platformu:

Zdrojový kód 3.7: Nastavenie Linux generátora náhodných čísel

```
1 #if LINUX_URANDOM
2     /* use linux's /dev/urandom */
3 #else
4     \* use Windows generator */
5 #endif
```

TYP	32-bitový systém (bajty)	64-bitový systém (bajty)
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8

Tabuľka 3.1: Porovnanie veľkosti natívnych premenných

Kritická chyba z pohľadu implementácie OAEP v RSA algoritme

Analýzou testovania knižnice s využitím šifrovacieho algoritmu RSA v spojení s výplňovou schémou OAEP sa v priebehu testovania objavila kritická chyba, ktorá mala za následok v niektorých prípadoch neplatnú dešifrovanú správu po procese dešifrovania správy na strane príjemcu.

Analýza kritickej chyby prebiehala v nasledovných krokoch:

1. Vytvorenie testovacieho súboru, ktorý vygeneruje RSA kľúče a následne zašifruje a dešifruje správu s využitím RSA algoritmu a výplňovej schémy OAEP a overí pôvodnú správu s dešifrovanou správou.

2. Po vyskytnutí chyby som začal analyzovať funkciu, ktorá vykonáva modulárne umocnenie. V tejto implementácii je uvedená ako $mpModExp()$, ktorá je aj základom RSA algoritmu pri šifrovaní a dešifrovaní.
   Vygeneroval som:

   (a) Náhodnú správu m.

   (b) Náhodný exponent e.

   (c) Náhodný modulus n.

   Vypočítal som hodnotu c podľa vzorca (1.1) a výsledok som overoval pomocou nástroja MAGMA [9]. Po vykonaní viacerých testov sa implementácia funkcie $mpModExp()$ nejavila ako chybná.

3. Začal som v cykle spúšťať test, ktorý realizoval šifrovanie a dešifrovanie RSA v spojení s OAEP podľa obrázka 3.2, a nechával som si vypisovať:

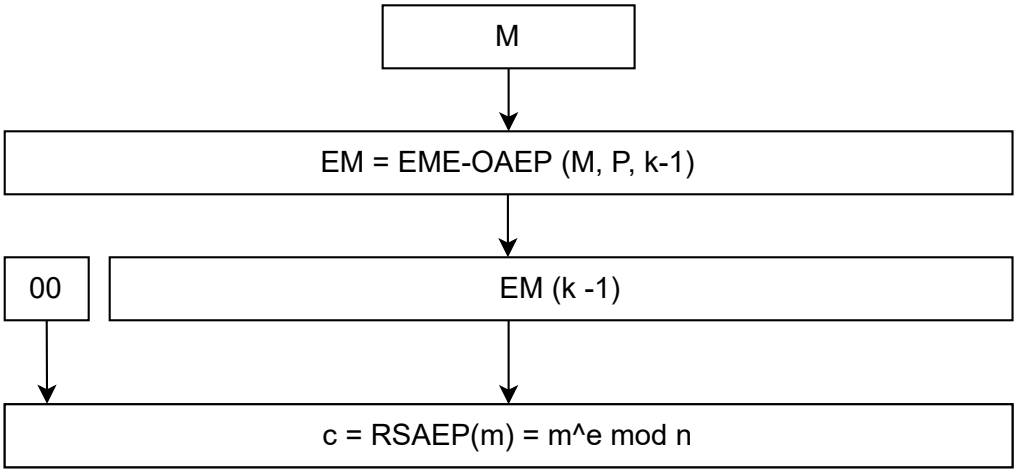
   (a) Zašifrovanú správu po realizácii OAEP.

[9] Online kalulačka pre BN čísla, dostupná na: `http://magma.maths.usyd.edu.au/calc/`

(b) Modulus n.

(c) Dešifrovanú správu po realizácii RSA.

Porovnával som zašifrovanú správu po realizácii OAEP s dešifrovanou správou po realizácii RSA. Ak sa správy nezhodovali všimol som si že hodnota modulusu na MSB (ang. Most Significant Bit) pozícii je menšia ako hodnota EM na MSB pozícii. Problematika uloženia BN čísel do pamäte je uvedená v podkapitole 1.1.1. Opis chyby je uvedený na obrázku 3.3.

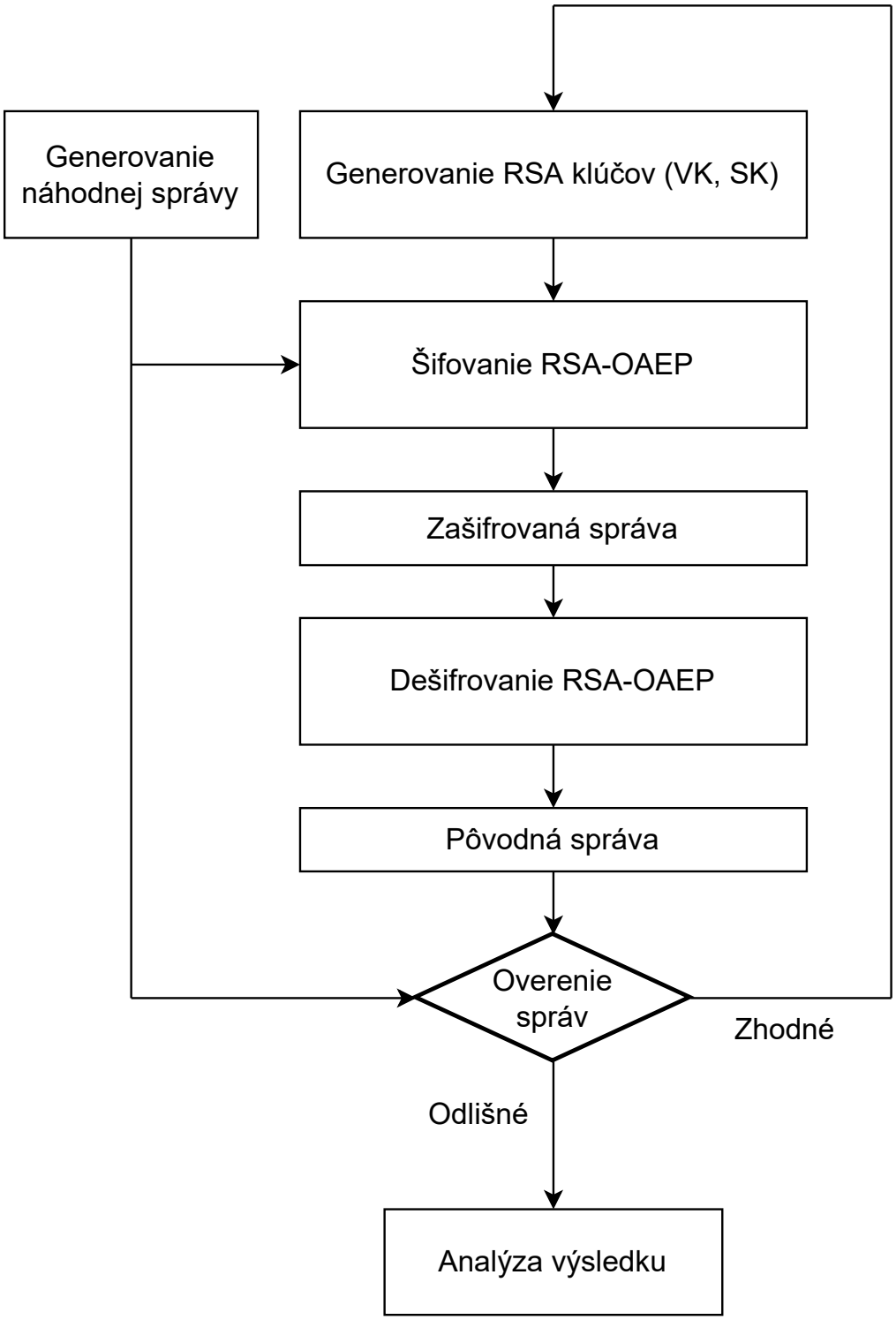


Obr. 3.1: Šifrovanie RSA-OAEP bez funkcie OS2IP

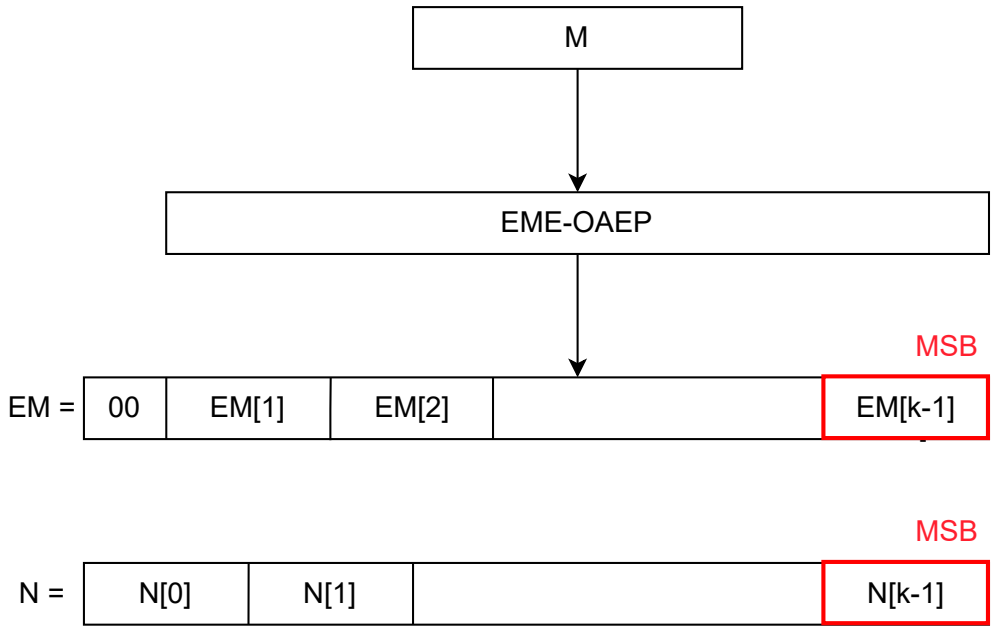
Autor realizoval šifrovanie RSA s využitím OAEP podľa obrázku 3.1, kde je vynechaná funkcia OS2IP, ktorá realizuje konverziu správy na celé číslo. Treba poznamenať že zašifrovaná správa em nie je vo formáte príslušného BN čísla pre danú knižnicu. V praxi môže nastať situácia $em[k-1] > n[k-1]$, kde k je počet bajtov modulusu n čo vedie k nesprávnemu dešifrovanému textu. Riešením bolo pridať blok kódu 3.8, ktorý realizuje konverziu do BN formátu.

Zdrojový kód 3.8: Konverzia textového reťazca do BN formátu

```
1 for(int i=0; i<len; i++){
2     reversed[i] = em[len -1 -i];
3 }
```



Obr. 3.2: Realizácia testovania RSA-OAEP v knižnici MCRYPTO



Obr. 3.3: Kritická chyba v knižnici MCRYPTO

3.3 Kryptografická knižnica STUDENT

Kryptografická knižnica od autora Junwei-Wang[10]. Autor knižnice uvádza že knižnica bola vytvorená za účelom zadania na univerzite.

Knižnicu som si zvolil pre praktickú realizáciu kvôli nasledovným bodom:

- Algoritmy sú prispôsobené práci s BN číslami.
- Disponuje možnosťou meniť veľkosť základu čísla (ang. base) pomocou direktív preprocesora.
- Obsahuje algoritmy na šifrovanie a dešifrovanie RSA algoritmu.
- Obsahuje implementáciu generovania kľúčov pre RSA algoritmus.

3.3.1 Opis kryptografickej knižnice STUDENT

Knižnica je napísaná v programovacom jazyku C a je nezávislá od OS Windows a Linux.

[10]Dostupná na:`https://github.com/junwei-wang/biginteger`

Zloženie kryptografickej knižnice STUDENT:

- Zdrojového súboru $bignum.c$ - V tomto zdrojovom súbore sú implementované prevažne matematické algoritmy pre prácu s BN číslami. Obsahuje aj implementácie funkcií generovania náhodných čísel, Millerov-Rabinov test prvočíselnosti.
- Zdrojového súboru $rsa.c$ - V tomto zdrojovom súbore sú implementované funkcie pre prácu s RSA algoritmom, generovanie kľúčov pre RSA algoritmus, šifrovanie a dešifrovanie s využitím RSA algoritmu.

3.3.2 Opis chýb v kryptografickej knižnici STUDENT

Generovanie náhodných čísel s využitím funkcie rand()

Knižnica na generovanie náhodných čísel využíva funkciu $genrandom()$, ktorá generuje náhodné čísla s využitím funkcie $rand()$, ktorá nie je pre bezpečné generovanie náhodných čísel vhodná. Problematike funkcie $rand()$, je opísaná v sekcii 1.6.

Problémy s pridelenou pamäťou pomocou funkcie malloc()

Knižnica využíva prácu s BN číslami formou dynamického alokovania pamäte pomocou funkcie $malloc()$, ktorá je zadefinovaná v systémovej knižnici stdlib.h. Autor nerealizuje vo funkciách návratovú hodnotu ak sa vyskytne chyba napríklad s alokáciou pamäte. Autor využíva návratový typ funkcie bignum uvedený v zdrojovom kóde 3.11, predpokladá, že prideľovanie pamäte bude vždy korektné.

Autor vo funkcii $int\ millerrabin(bignum\ n,\ int\ t)$ v niektorých prípadoch neuvoľňuje pridelenú pamäť, čo ma za následok zahltenie pamäte počítača, prípadne neočakávane skončenie programu. Autor nerealizuje kontrolu či funkcia $malloc()$ pridelila požadovanú pamäť, čo môže mať za následok neočakávané správanie priebehu programu. Chyby odhalené s neuvoľňovaním pamäte boli odhalené pomocou nástroja Valgrind, ktorý je opísaný v sekcii 4.1. Po analýze chýb som realizoval úpravu, ktorá uvoľnovala nepotrebnú pamäť.

Metóda rekurzívneho výpočtu GCD

Autor realizoval funkciu $bignum\ gcd(bignum\ a,\ bignum\ b)$ formou rekurzívnej funkcie, pričom pri volaniach alokoval veľké množstvo pamäte, čo v niektorých

prípadoch malo za následok neočakávané správanie programu. Chybu som odhalil po kompilácii testovacieho súboru test.c, ktorý bol vopred pripravený autorom knižnice. Kompiláciu som realizoval na OS Windows. Po kompilácii programu a následnom spustení programu zvykol program nekontrolovane spadnúť. Spustením kontroly pamäte pomocou nástroja Valgrind odhalilo chybu. Chyba bola v rekurzívnom volaní funkcie $gcd()$, kedy autor nerealizoval uvoľnenie pamäte. Využite rekurzívneho algoritmu na výpočet GCD nie je v kryptografických systémoch vhodnou voľbou. Funkciu $gcd()$ som prepísal na nerekurzívnu s využitím Euklidovho algoritmu (2).

3.3.3 Rozšírená implementácia knižnice STUDENT

V rámci praktickej časti som realizoval dodatočné implementácie pre rozšírenie kryptografickej knižnice STUDENT.

Cieľom rozšírenia knižnice bolo:

1. Zvýšiť bezpečnosť generovania náhodných dát.
2. Implementovať prácu so súbormi.
3. Rozšíriť knižnicu o možnosť využitia výplňovej schémy OAEP.
4. Úprava implementácie, aby realizovala výpočty so základom 8 alebo 32 bitov.

Pridanie načítavania a zapisovania dát formou využitia súborov

Cieľom bolo vytvoriť špecificky spôsob typ formátu ukladania údajov do súboru. Vytvoril som si vlastný formát pre ukladanie kľúčov do súboru.

Štandard pre ukladanie kľúčov do súboru:

1. Hlavička.
   —–BEGIN PUBLIC KEY—–
   alebo
   —–BEGIN PRIVATE KEY—–
   v závislosti od použitého typu kľúča
2. Exponent v závislosti od použitého typu kľúča
   $[e]$ alebo $[d]$.

3. Reprezentácia BN čísla exponentu(verejný alebo súkromný) A v tvare: A[0] ..... A[x-1], kde X je dĺžka exponentu.

4. Modulus [n].

5. Reprezentácia BN čísla modulusu A v tvare: A[0] ..... A[x-1], kde X je dĺžka modulusu.

6. Päta.
   —-END PUBLIC KEY—-"
   alebo
   —-END PRIVATE KEY—-"

Optimalizovanie knižnice pre možnosť výberu 8 alebo 32 bitového základu čísla

Knižnicu STUDENT som rozšíril aj o možnosť výberu základu (ang. base) čísla. Implementované boli dva základy:

- 8 - bitový základ.
- 32 - bitový základ.

Z hľadiska vykonávania matematických operácii sa knižnica s využitím 32- bitového základu čísla javí ako rýchlejšia. 8-bitový základ slúži iba na porovnanie rýchlosti výpočtov.Názorná ukážka implementácie 8-bitového a 32-bitového základu čísla s využitím direktív preprocesora v jazyku C je znázornená v zdrojovom kóde 3.9.

Zdrojový kód 3.9: Voľba základu čísla s využitím direktív preprocesora

```
1  #ifdef BIT32
2  // -------- 32 bit number system -----------
3  typedef __int128_t block;
4  /* Real value stored in bignumber format */
5  typedef unsigned int var;
6
7  #define B 4294967296
8  #define E 32          // B = 2^E, E = 1, 2, 3, ..., 32
9  #define MASK 0xffffffff
10
11 #elif defined BIT8
12 // -------- 8 bit number system ---------------
13 typedef int64_t block;
14 typedef unsigned char var;
15
16 #define B 256
17 #define E 8
18 #define MASK 0xff
19
20 #endif
```

3.4 KOKKE - kompaktná kryptografická knižnica

Knižnica od autora Kokke [11]. Amatérska kryptografická knižnica, ktorá sa vyznačuje malou veľkosťou. Autor knižnice poukazuje na ciele pri tvorbe tejto knižnice sa zameral hlavne na veľkosť výstupného súboru, aby bola malá. taktiež sa zameral na priehľadnosť zdrojového kódu [37].

Charakteristika knižnice KOKKE:

- Knižnica je platformovo nezávislá.
- Vyznačuje sa malou veľkosťou - Malý kód a binárna veľkosť približne 500 SLOC (ang. Source Lines Of Code).
- Statické prideľovanie pamäte.

[11] Dostupná na: `https://github.com/kokke/tiny-bignum-c`

- Knižnica pracuje s BN číslami, ktoré sú reprezentované ako štruktúra, ktorá obsahuje statické pole 3.12.
- Žiadne použitie dynamickej alokácie pamäte.
- S BN číslami dokáže vykonávať základnmé matematické operácie: sčítanie, odčítanie, násobenie, delenie, modulárne umocnenie, modulo.
- Neobsahuje generovanie RSA klúčov.

Knižnicu KOKKE som si zvolil kvôli jej kompaktnosti a malej veľkosti výstupného súboru. Cieľom zahrnutia knižnice KOKKE do praktického overenia bolo možnosť poukázať na veľkosť výstupného súboru pri zohľadnení časových nárokov. Výhodou knižnice KOKKE je síce jej veľkosť ale je to na úkor vykonávania matematických operácii.

3.4.1 Rozšírenia implementácie knižnice KOKKE

Ako bolo avizované v predošlej sekcii, táto knižnica nedokáže generovať vlastné RSA kľúče. Cieľom rozšírenia pre KOKKE knižnicu bolo vytvoriť funkcie, ktoré budú vedieť načítavať RSA kľúče zo súborov. Implementoval som funkcie, ktoré dokážu načítavať RSA kľúče vo formáte, ktorý som uviedol v podkapitole venovanej kryptografickej knižnici STUDENT 3.3. Knižnica KOKKE teda využíva na šifrovanie a dešifrovanie RSA kľúče, ktoré sú vygenerované knižnicou STUDENT.

3.5 Kryptografická knižnica OpenSSL

Sofistikovaná kryptografická knižnica skladajúca sa z dvoch časti libcrypto a libssl. Šifrovacia knižnica OpenSSL libcrypto implementuje širokú škálu kryptografických algoritmov používaných v rôznych internetových štandardoch. Funkcionalita zahŕňa symetrické šifrovanie, kryptografiu s verejným kľúčom, dohodu o kľúči, manipuláciu s certifikátom, kryptografické hashovacie funkcie, generátory kryptografických pseudonáhodných čísel, autentifikačné kódy správ , funkcie odvodenia kľúčov a rôzne pomocné programy. Šifrovacia knižnica OpenSSL libssl implementuje podporu pre sieťovú komunikáciu [38].

3.6 Porovnanie knižníc z hľadiska BN formátu

Porovnanie kryptografických knižníc z hľadiska:

- Štruktúry BN čísla.
- Reprezentácie čísel formou statického alebo dynamického prideľovania pamäte.

3.6.1 Formát pre uloženie BN čísel

V závislosti od vhodne zvoleného typu formátu pre uloženie BN čísla do pamäte závisia aj výpočtové nároky na vykonávanie matematických operácii. Implementácie zväčša využívajú na uloženie BN čísel postupnosť vektorov číslic, ktoré boli opísane podrobnejšie v podkapitole 1.1.1. V tabuľke 3.2 je možné vidieť porovnanie kryptografických knižníc z hľadiska využitia prideľovanej pamäte.

Knižnica	Statické polia	Dynamické polia
MCRYPTO	X	
STUDENT		X
KOKKE	X	
OpenSSL		X

Tabuľka 3.2: Porovnanie knižnic z hľadiska prideľovania pamäte

Formát uloženia BN čísel knižnica MCRYPTO

Zdrojový kód 3.10: Reprezentácia BN čísla BIGDIGITS implementácia

```
1 #deine MAX_DIG_LEN 64 /* 2048 bits*/
2
3 typedef uint32_t DIGIT_T;
4 DIGIT_T BN_NUM[MAX_DIG_LEN];
```

Formát uloženia BN čísel knižnica STUDENT

Zdrojový kód 3.11: Reprezentácia BN čísla STUDENT implementácia

```
1 typedef struct {
2         int sign;
3         int size;
4         block *tab;
5 } bignum;
```

Formát uloženia BN čísel knižnica KOKKE

Zdrojový kód 3.12: Reprezentácia BN čísla KOKKE implementácia

```
1 struct bn
2 {
3   DTYPE array[BN_ARRAY_SIZE];
4 };
```

Formát uloženia BN čísel knižnica OpenSSL

Zdrojový kód 3.13: Reprezentácia BN čísla BIGDIGITS implementácia

```
1  struct bignum_st{
2     BN_ULONG *d;
3     int top;    /* Index of last used d +1. */
4     /* The next are internal book keeping for bn_expand. */
5     int dmax;   /* Size of the d array. */
6     int neg;    /* one if the number is negative */
7     int flags;
8 };
```

4 Experimentálne testovanie kryptografických knižníc

Cieľom experimentálnej časti bolo poukázať na funkčnosť vyhľadaných kryptografických knižníc. V rámci experimentálne časti som realizoval testy na overenie funkčnosti kryptografických knižníc. Opis realizovaných testov a ich výsledky sú uvedené v podkapitole 4.2. Taktiež som sa zameral na analýzu zdrojových kódov z pohľadu správy pamäte s vyžitím nástroja Valgrind.

4.1 Využité nástoje v rámci praktickej časti

- **GCC** - slúži na preklad zdrojových súborov v jazyku C.
- **OpenSSL** - Kryptografická knižnica, ktorá bola popísaná v sekcii 3.5. Knižnicu OpenSSL je možné využiť aj ako aplikáciu bez nutnosti písania zdrojových kódov.
- **Valgrind** - Nástroj slúži na vyhľadávanie únikov z pamäte, detekciu chýb, nesprávneho zápisu do pamäte, neuvoľnenie alokovanej pamäte. Nástroj je dostupný iba pre OS Linux. Detailnejší opis Valgrind nástroja je uvedený na stránke [39].

	GCC	OpenSSL	Valgrind
Windows	Winlibs MinGW-W64 11.2.0	OpenSSL 3.0.3	-
Linux	GCC Debian 11.3.0	OpenSSL 3.0.3	valgrind-3.18.1

Tabuľka 4.1: Verzie využitých nástrojov

4.2 Realizované testy s využitím kryptografických knižníc

V rámci praktickej časti bakalárskej práce som sa zameral na otestovanie kryptografických knižníc, ktoré boli opísane v predošlej kapitole.

Vytvoril som tri hlavné testy a jeden s využitím testovacích vektorov:

- **TEST č.1** – Overenie správnosti výpočtu modulárneho umocnenia.
- **TEST č.2** – Test rýchlosti šifrovania a dešifrovania správ pomocou RSA algoritmu s využitím výplňovej schémy OAEP.
- **TEST č.3** – Test rýchlosti generovania RSA kľúčov.
- **TEST s využitím testovacích vektorov** – Test realizuje overenie výpočtov s využitím testovacích vektorov.

Testy som vykonal 10-krát na OS Linux a Windows, následne som si namerané hodnoty zaznamenal. Do tabuliek som zapísal maximálny čas, minimálny čas a priemerný čas výpočtu.

4.2.1 Zaznamenávanie dĺžky výpočtov

Pri porovnávaní kryptografických knižníc bolo nutné zaviesť metódu, ktorou budeme zaznamenávať namerané hodnoty výpočtov. Využil som metódu, ktorá je uvedená v zdrojovom kóde 4.1.

Meranie času je implementované funkciami:

- $clock\_gettime(CLOCK\_MONOTONIC, start)$ - Spúšťa meranie času.
- $clock\_gettime(CLOCK\_MONOTONIC, end)$ - Ukončuje meranie času.

Následne je výsledok vypísaný v sekundách.

Zdrojový kód 4.1: Zdrojový kód pre realizáciu merania časových údajov

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define BILLION 1000000000L

int main(void){
    double diff;
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);
    // Merany algoritmus
    clock_gettime(CLOCK_MONOTONIC, &end);

    diff = (end.tv_sec - start.tv_sec) +
    (double) (end.tv_nsec - start.tv_nsec) / (double)BILLION;

    printf("elapsed time = %lf seconds\n",diff);
    return 0;
}
```

4.2.2 Realizácia testu č.1

Test realizoval overenie správnosti výpočtu modulárneho umocnenia. Na overenie správnosti výpočtov som implementoval vlastný algoritmus 11. Správnosť výsledkov som overil porovnaním výstupov z použitých knižníc. Časove výsledky realizácie testu sú uvedene v tabuľke 4.2.

Využité zdrojové súbory:

- **KOKKE** – Dostupný v prílohe D v adresári EXTENSIONS_KOKKE\TESTS \test01.c.
- **MCRYPTO** – Dostupný v prílohe C v adresári EXTENSIONS_MCRYPTO \TESTS\test01.c.
- **STUDENT** – Dostupný v prílohe B v adresári EXTENSIONS_STUDENT \TESTS\test01.c.

- **OpenSSL** – Dostupný v prílohe E, súbor test01.c.

Algorithm 11 `Algoritmus na testovanie modulárneho umocnenia`

Input: $TEST\_COUNT$ – Počet iterácii algoritmu

Output: m

1: SEED(4)
2: m[0] = rand() % 254 + 1;
3: e[0] = rand() % 254 + 1;
4: **for** i from 0 to $MODULUS\_LEN$ **do**
5: $n[i] = MAX\_VAL$ // Maximálna hodnota závisí od využitia základu BN čísla.
6: **end for**
7: **for** i from 0 to $TEST\_COUNT$ **do**
8: $c = m^e \mod n$
9: $e = m^c \mod n$
10: $m = c^n \mod e$
11: $n = c^e \mod m$
12: **end for**
13: **return** "m"

Výsledok testu pre nastavenú hodnotu TEST_COUNT =30 v hexadecimálnom tvare:

```
4C7EA7BB 595970F2 BD3FF64C D20D277F C58ECE03 9494769F 1FC4FF3A
9F04C642 4781359A 1B1239FE 1A929E57 8D4DEABB 66A1E540 DE82DF6A
5C48C497 6C6EE3DC 16EC6775 0C012B26 B625DC25 325E4F29 1BF66EE9
8C82F457 1C2CFDB9 C6E750CF 19523C50 14ADD600 00000000 00000000
```

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,172	0,2	0,182	0,112	0,14	0,119
MCRYPTO	2,307	2,956	2,498	2,583	3,097	2,894
STUDENT-32	5,041	5,314	5,149	5,197	6,377	5,584
STUDENT-8	35,105	36,744	35,559	36,635	41,146	38,461
KOKKE	90,551	94,174	92,8	90,717	95,185	92,747

*Výsledky su vyjadrené v sekundách

Tabuľka 4.2: Výsledky testu č.1 pre parameter TEST_COUNT = 30

4.2.3 Realizácia testu č.2

Test realizoval overenie časových nárokov na výpočet šifrovacieho procesu výplňovej schémy OAEP so šifrovacím algoritmom RSA.

Test bol realizovaný s využitím modulusu o dĺžke:

1. 1024 bitov šifrovanie - zaznamenané v tabuľke 4.3.
2. 1024 bitov dešifrovanie - zaznamenané v tabuľke 4.6.
3. 2048 bitov šifrovanie - zaznamenané v tabuľke 4.4.
4. 2048 bitov dešifrovanie - zaznamenané v tabuľke 4.7.
5. 4096 bitov šifrovanie - zaznamenané v tabuľke 4.5.
6. 4096 bitov dešifrovanie - zaznamenané v tabuľke 4.8.

Proces vykonávania testu:

1. Vygeneruj RSA kľúče VK_e, n a SK_d, n špecifickej dĺžky podľa aktuálneho testu. Knižnica KOKKE využíva RSA kľúče vygenerované knížnicou STUDENT.
2. Spusť meranie času.
3. Zašifruj správu pomocou RSA-OAEP.
4. Zastav meranie času a vypíš výsledok.
5. Spusť meranie času.
6. Dešifruj správu pomocou RSA-OAEP.
7. Zastav meranie času a vypíš výsledok.

Knižnice KOKKE a MCRYPTO pracujú s BN číslami, ktoré využívajú statické pole, preto bolo nutné prispôsobiť zadávanie veľkosti polí s využitím makier v jazyku C. V adresároch, kde sa nachádzajú jednotlivé testy príslušných knižníc sa nachádza aj súbor Makefile, v ktorom sa definuje makro MOD_LEN pre vhodnú veľkosť statického poľa.

Využité zdrojové súbory:

- **KOKKE** – Dostupný v prílohe D v adresári EXTENSIONS_KOKKE\TESTS.
  1. 1024-bitový modulus – test02.c.
  2. 2048-bitový modulus – test03, založený na test02.c, s rozdielom nastavenia makra MOD_LEN na hodnotu -DMOD_LEN=2048.
  3. 4096-bitový modulus – test04, založený na test02.c, s rozdielom nastavenia makra MOD_LEN na hodnotu -DMOD_LEN=4096.
- **MCRYPTO** – Dostupný v prílohe C v adresári EXTENSIONS_MCRYPTO \TESTS.
  1. 1024-bitový modulus – test02.c.
  2. 2048-bitový modulus – test03, založený na test02.c, s rozdielom nastavenia makra MOD_LEN na hodnotu -DMOD_LEN=2048.
  3. 4096-bitový modulus – test04, založený na test02.c, s rozdielom nastavenia makra MOD_LEN na hodnotu -DMOD_LEN=4096.
- **STUDENT** – Dostupný v prílohe B v adresári EXTENSIONS_STUDENT \TESTS\test01.c.
- **OpenSSL** – Dostupný v prílohe E, súbor test02.c.

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,00003	0,00004	0,00004	0,00004	0,00007	0,00005
MCRYPTO	0,023	0,029	0,027	0,023	0,025	0,024
STUDENT-32	0,0005	0,029	0,027	0,0005	0,0008	0,001
STUDENT-8	0,003	0,011	0,009	0,004	0,013	0,01
KOKKE	0,0115	0,0133	0,0124	0,0119	0,0124	0,0121

*Výsledky su vyjadrené v sekundách

Tabuľka 4.3: Namerané hodnoty test č.2 1024 bitov – proces šifrovania

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,00007	0,00009	0,00008	0,00005	0,00007	0,00005
MCRYPTO	0,175	0,189	0,186	0,179	0,181	0,18
STUDENT-32	0,003	0,004	0,003	0,00006	0,03	0,0008
STUDENT-8	0,047	0,071	0,052	0,051	0,68	0,061
KOKKE	0,078	0,011	0,092	0,06	0,109	0,072

*Výsledky su vyjadrené v sekundách

Tabuľka 4.4: Namerané hodnoty test č.2 2048 bitov – proces šifrovania

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,0002	0,0003	0,0002	0,0001	0,0003	0,0001
MCRYPTO	1,512	1,733	1,7	1,449	1,492	1,46
STUDENT-32	0,008	0,012	0,01	0,0001	0,019	0,009
STUDENT-8	0,136	0,265	0,155	0,067	0,248	0,178
KOKKE	0,386	0,501	0,397	0,379	0,47	0,421

*Výsledky su vyjadrené v sekundách

Tabuľka 4.5: Namerané hodnoty test č.2 4096 bitov – proces šifrovania

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,0005	0,0006	0,0005	0,0003	0,0008	0,0004
MCRYPTO	0,0005	0,0007	0,0006	0,0003	0,0005	0,0003
SRUDENT-32	0,054	0,065	0,057	0,044	0,06	0,052
STUDENT-8	0,755	1.242	0.096	0,751	1,266	0,84
KOKKE	0,794	1,01	0,917	0,833	0,933	0,875

*Výsledky su vyjadrené v sekundách

Tabuľka 4.6: Namerané hodnoty test č.2 1024 bitov – proces dešifrovania

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,002	0,003	0,002	0,002	0,003	0,002
MCRYPTO	0,177	0,196	0,193	0,171	0,188	0,179
STUDENT-32	0,493	0,5	0,497	0,798	0,855	0,821
STUDENT-8	9,523	10,329	10,015	9,876	10,938	9,996
KOKKE	10,942	14,217	11,081	11,026	15,073	11,922

Tabuľka 4.7: Namerané hodnoty test č.2 2048 bitov – proces dešifrovania

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,007	0,01	0,008	0,007	0,008	0,008
MCRYPTO	1,431	1,7	1,677	1,43	1,491	1,479
STUDENT-8	3,201	3,44	3,41	3,27	5,086	4,168
STUDENT-8	33,405	74,094	61,255	21,678	80,318	42,248
KOKKE	149,62	151,294	151,02	152,151	156,68	153,751

*Výsledky su vyjadrené v sekundách

Tabuľka 4.8: Namerané hodnoty test č.2 4096 bitov – proces dešifrovania

4.2.4 Realizácia testu č.3

Test realizoval overenie časových nárokov na generovanie RSA kľúčov. V tomto teste nie je obsiahnutá knižnica KOKKE, keďže neobsahuje algoritmus na generovanie RSA kľúčov.

Generovali sa kľúče s dĺžkou modulusu:

- 1024 bitov - zaznamenané v tabuľke 4.9.
- 2048 bitov - zaznamenané v tabuľke 4.10.
- 4096 bitov - zaznamenané v tabuľke 4.11.

Využité zdrojové súbory:

- **MCRYPTO** – Dostupný v prílohe C v adresári EXTENSIONS_MCRYPTO \TESTS\test05.c.

- **STUDENT** – Dostupný v prílohe B v adresári EXTENSIONS_STUDENT \TESTS\test05.c.
- **OpenSSL** – Dostupný v prílohe E, súbor test05.c.

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,025	0,027	0,0267	0,016	0,032	0,027
MCRYPTO	1,195	2,007	1,491	1,07	1.82	1,62
STUDENT-32	3,031	12,432	9,104	2,302	10,092	8,74
STUDENT-8	18,191	27,469	19,469	15,016	24,41	20,301

*Výsledky su vyjadrené v sekundách

Tabuľka 4.9: Namerané hodnoty test č.3 1024 bitov

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	0,085	0,375	0,179	0,028	0,116	0,074
MCRYPTO	7,742	9,24	8,709	7,831	11,204	9,024
STUDENT-32	157,432	201,453	463,453	48,79	321,63	224,741
STUDENT-8	119,012	680,882	284,054	140,74	582,74	322,643

*Výsledky su vyjadrené v sekundách

Tabuľka 4.10: Namerané hodnoty test č.3 2048 bitov

	Windows			Linux		
	Min	Max	Priemer	Min	Max	Priemer
OPENSSL	1,362	1,684	1,421	0,967	1,421	1,525
MCRYPTO	61,496	80,377	72,361	82,488	89,655	84,21
STUDENT-32	241,647	823,621	624,61	200,079	289,452	514,431
STUDENT-8	947,207	4087,23	2624,207	842,17	3772,147	1984, 278

*Výsledky su vyjadrené v sekundách

Tabuľka 4.11: Namerané hodnoty test č.3 4096 bitov

4.2.5 Test s využitím testovacích vektorov

Test realizuje overenie správnosti RSA algoritmu, hashovacich funkcií a výplňovej schémy OAEP. Testovací vektor je dostupný na stránke [1]. V tomto teste nie sú obsiahnuté knižnice KOKKE a OpenSSL. Knižnica KOKKE nemá implementovaný algoritmus na generovanie RSA kľúčov. OpenSSL knižnica neumožňuje priamo zadávanie hodnoty SEED v zdrojových súboroch knižnice OpenSSL.

Využité zdrojové súbory:

- **MCRYPTO** – Dostupný v prílohe C v adresári EXTENSIONS_MCRYPTO \TESTS\test_vect.c.
- **STUDENT** – Dostupný v prílohe B v adresári EXTENSIONS_STUDENT \TESTS\test_vect.c.

Inicializačné hodnoty, uvedené v hexadecimálnom tvare:

P:

```
ee cf ae 81 b1 b9 b3 c9 08 81 0b 10 a1 b5 60 01 99 eb 9f 44 ae
f4 fd a4 93 b8 1a 9e 3d 84 f6 32 12 4e f0 23 6e 5d 1e 3b 7e 28
fa e7 aa 04 0a 2d 5b 25 21 76 45 9d 1f 39 75 41 ba 2a 58 fb 65
99
```

Q:

```
c9 7f b1 f0 27 f4 53 f6 34 12 33 ea aa d1 d9 35 3f 6c 42 d0 88
66 b1 d0 5a 0f 20 35 02 8b 9d 86 98 40 b4 16 66 b4 2e 92 ea 0d
a3 b4 32 04 b5 cf ce 33 52 52 4d 04 16 a5 a4 41 e7 00 af 46 15
03
```

e:

```
11
```

M:

```
d4 36 e9 95 69 fd 32 a7 c8 a0 5b bc 90 d3 2c 49
```

SEED:

```
aa fd 12 f6 59 ca e6 34 89 b4 79 e5 07 6d de c2 f0 6c b5 8f
```

Realizácia testu:

[1] Dostupná na: https://www.inf.pucrs.br/~calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf

1. Modifikácia zdrojových súborov, ktoré obsahujú generovanie kľúčov pre RSA algoritmus.
2. Vykonanie testu s dodatočným výpisom.
3. Porovnanie výsledkov s testovacími vektormi.

4.3 Vyhodnotenie dosiahnutých výsledkov

Implementáciou uvedených testov som porovnal rôzne implementácie kryptografických knižníc. V teste č.1 som testoval korektnosť matematickej operácie modulárneho umocnenia. Najlepšie výsledky dosahovala v tomto teste knižnica OpenSSl. Nároky na výpočet modulárneho umocnenia v knižnici KOKKE sú vysoké, pretože analýzou zdrojového kódu a spúšťaním rôznych typov testov som odhalil nie veľmi vhodnú metódu implementácie algoritmu na delenie, ktorú daná knižnica využívala. V teste č.2 som realizoval testovanie rýchlosti šifrovacieho a dešifrovanieho procesu RSA algoritmu s využitím výplňovej schémy OAEP, použil som rôzne veľkosti modulusov. V tomto teste obstala najlepšie knižnica OpenSSL. V rámci testu č.3 som otestoval rýchlosti generovania kľúčov pre RSA algoritmus, pričom som využil rôzne dĺžky modulusov. Využité boli iba knižnice STUDENT, OpenSSL a MCRYPTO. Knižnica KOKKE v tomto teste nebola obsiahnutá, keďže neobsahuje implementáciu na generovanie RSA kľúčov. V tomto teste opäť najlepšie výsledky dosahovala knižnica OpenSSL. V rámci otestovania algoritmov som využil test s testovacími vektormi, v ktorom som overil funkčnosť RSA algoritmu v spojení s výplňovou schémou OAEP. Z hľadiska dosiahnutých výsledkov v predchádzajúcich testov najlepšie výsledky dosahovala kryptografická knižnica OpenSSL.

5 Záver

V teoretickej časti bakalárskej práce som opísal koncepčne RSA algoritmus. Opísal som základné princípy RSA algoritmu pričom som za zameral na využitie základných matematických algoritmov, testovanie prvočíselnosti, generovanie kľúčov pre RSA algoritmus, bezpečnosti RSA algoritmu. V rámci RSA algoritmu som stručne opísal generovanie náhodných čísel, pričom som za zameral na generovanie kryptograficky bezpečných náhodných čísel na operačných systémoch Windows a Linux. V krátkosti som uviedol aj problematiku lineárneho kongruentného generátora. Následne som sa venoval typom útokov na definičný RSA algoritmus, ktoré som detailnejšie opísal v prvej kapitole. Po koncepčnom opise RSA algoritmu a jeho bezpečnostných problémov som uvidel riešenie v podobe využitia výplňovej schémy OAEP. V rámci problematiky výplňovej schémy OAEP som uviedol problematiku hashovacích funkcii, ktoré sú základným stavebným blokom výplňovej schémy OAEP. Opísal som možnosti využitia výplňovej schémy OAEP so šifrovacím algoritmom RSA a ich vzájomného uplatnenia v praxi. Pre rozšírenie problematiky šifrovacieho algoritmu RSA v spojení s výplňovou schémou OAEP som opísal štandard PKCS#1, ktorý sa zaoberá problematikou implementovnia RSA algoritmu do praxe.

V praktickej časti bakalárskej práce som vyhľadal implementácie šifrovacieho algoritmu RSA algoritmu na internete, ktoré som detailnejšie opísal v tretej kapitole. Následne som tieto implementácie podrobil analýze z hľadiska využívaných funkcií, spôsobu prideľovania pamäte, spôsobu generovania náhodných čísel, využitie výplňovej schémy OAEP. Počas analýzy zdrojových kódov sa ukázali implementačné nedostatky kryptografických knižníc. Kryptografické knižnice, ktoré sa javili navonok ako použiteľné pre praktickú realizáciu tejto bakalárskej práce, v skutočnosti obsahovali implementačné nedostatky, ktoré som v tretej kapitole analyzoval a následne koncepčne opísal a navrhol vhodné riešenie na opravu daného problému. Na analýzu zdrojových kódov z hľadiska kontroly prideľovania pamäte som využil nástroj Valgrind, ktorým som analyzoval zdrojové súbory a hľadal implementačné nedostatky súvisiace so správou pamäte.

V praktickej časti som sa zaoberal aj testovaním kryptografických knižníc. Pre realizáciu testovania som si vytvoril vlastné testovacie súbory, ktorými som overoval funkčnosť RSA algoritmu a výplňovej schémy OAEP. Zameral som sa hlavne na meranie času generovania kľúčov rôznych dĺžok pre RSA algoritmus, šifrovania a dešifrovania RSA algoritmu v spojení s výplňovou schémou OAEP. Overoval som aj výpočet RSA algoritmu v spojení s výplňovou schémou OAEP testovacími vektormi, ktoré som vyhľadal na internete. Porovnaním knižníc s vyžitím testovacieho súboru, ktorý obsahoval implementáciu s testovacími vektormi som overil ich funkčnosť výpočtov.

Literatúra

1. MOLLIN, Richard A. *An Introduction to Cryptography*. Chapman Hall/CRC, 2006. ISBN 978-1-58488-618-1.
2. LEVICKY, Dusan. *Aplikovaná kryptografia, Od utajenia správ ku kybernetickej bezpečnosti*. Elfa, 2018. ISBN 9788080862657.
3. KNEBL, Hans Delfs Helmut. *Introduction to Cryptography*. Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-49243-6. Principles and Applications.
4. SHOUP, Victor. *A Computational Introduction to Number Theory and Algebra*. 2008. ISBN 978-0521516440.
5. MARTIN, Keith M. *Everyday Cryptography*. Oxford University Press, 2017. ISBN 978–0–19–109206–0. Fundamental Principles and Applications.
6. K. MORIARTY B. Kaliski, J. Jonsson; RUSCH, A. RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2. *Internet Engineering Task Force* (*IETF*). 2016. Dostupné tiež z: `https://datatracker.ietf.org/doc/html/rfc8017#section-7.1`. The Public-Key Cryptography Standards are specifications produced by RSA Laboratories in cooperation with secure systems developers worldwide for the purpose of accelerating the deployment of public- key cryptography.
7. KOBLITZ, Neal. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1994. ISBN 978-1-4419-8592-7.
8. ALFRED J. MENEZES Paul C. van Oorschot, Scott A. Vanstone. *HANDBOOK of APPLIED CRYPTOGRAPHY*. CRC Press, Inc, 1996. ISBN 0-8493-8523-7.
9. BENJAMIN FINE, Gerhard Rosenberger. *Number Theory, An Introduction via the Density of Primes*. Birkhäuserk, 2016. ISBN 978-3-319-43875-7.
10. MARIA WELLEDA BALDONI, Ciro Ciliberto; CATTANEO, Giulia Maria Piacentini. *Elementary Number Theory Cryptography and Codes*. Springer-Verlag Berlin Heidelberg, 2009. ISBN 978-3-540-69199-0.

11. MA, Dan. The Miller-Rabin primality test. 2014. https://mathcrypto.wordpress.com/tag/r rabin-primality-test/.

12. KLEINBERG, Bobby. *The Miller-Rabin Randomized Primality Test*. 2010-05. Tech. spr. Cornell University. Dostupné tiež z: `https://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf`.

13. MACH, Vladimír. *Rabin-Millerův test prvočíselnosti* [How it was published]. 2010. Dostupné tiež z: `http://mff.wladik.net/09zs/ADS_Rabin-MillerTest.pdf`.

14. TOKUÇ, A. Aylin. *Fermat Primality Test* [How it was published]. 2021. Dostupné tiež z: `https://www.baeldung.com/cs/fermat-primality-test`.

15. BENJAMIN JUN, Paul Kocher. The INTEL Random Number Generator. *Cryptography Research, Inc.* 1999. Dostupné tiež z: `https://www.rambus.com/wp-content/uploads/2015/08/IntelRNG.pdf`.

16. KYUNGROUL LEE, Manhee Lee. True Random Number Generator (TRNG) Utilizing FM Radio Signals for Mobile and Embedded Devices in Multi-Access Edge Computing. 2019. Dostupné tiež z: `https://mdpi-res.com/d_attachment/sensors/sensors-19-04130/article_deploy/sensors-19-04130.pdf?version=1569318578`. doi:10.3390/s19194130.

17. SCHNEIER, Bruce; FERGUSON, Niels. *Practical Cryptography*. John Wiley Sons, 2003. ISBN 0471223573.

18. A Statistical Test Suite forRandom and Pseudorandom Number Generators for Cryptographic Applications. *National Institute of Standards and Technology (NIST)*. 2010. Dostupné tiež z: `https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf`. Authors: Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks,Alan Heckert, James Dray, San Vo.

19. KER, Michael. *rand(3) — Linux manual page*. 2021. Dostupné tiež z: `https://man7.org/linux/man-pages/man3/rand.3.html`. This page is part of release 5.13 of the Linux man-pages project.

20. An Investigation of Sources of Randomness Within Discrete Gaussian Sampling. 2017. Dostupné tiež z: `https://eprint.iacr.org/2017/298`. Seamus Brannigan, Neil Smyth, Tobias Oder, Felipe Valencia, Elizabeth O'Sullivan, Tim Guneysu4 and Francesco Regazzoni.

21. rand$_s$. 2022. Dostupné tiež z: `https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/rand-s?view=msvc-170`. Docuemtation rand$_s function$.

22. KER, Michael. *urandom(4) - Linux man page*. 2021. Dostupné tiež z: `https://man7.org/linux/man-pages/man4/random.4.html`. This page is part of release 5.13 of the Linux man-pages project.

23. *random and urandom Devices*. 2022. Dostupné tiež z: `https://www.ibm.com/docs/en/aix/7.2?topic=files-random-urandom-devices`. článok vytvorený spoločnosťou IBM.

24. YAN, Song Y. *Computational Number Theory And Modern Cryptography*. Higher Education Press, 2013. ISBN 9781118188583.

25. JUST, Jiri; COFFEY, John. *An Assessment of Attacks Strategies on the RSA Public-Key Cryptosystem*. West Florida Pensacola, FL 32514, 2009. Tech. spr. Department of Computer Science The University of West Florida Pensacola, FL 32514. Dostupné tiež z: `https://www.iiis.org/cds2009/cd2009sci/RMCI2009/PapersPdf/R873IE.pdf`. An optional note.

26. KLÍMA, Vlastimil. In: *Dve cisla za 200 000 dolaru*. 2001, kap. 2, s. 182–188. Dostupné tiež z: `http://crypto-world.info/klima/2001/chip-2001-10-182-188.pdf`.

27. ETSI TS 119 312 V1.4.2 (2022-02). 2022. Dostupné tiež z: `https://www.etsi.org/deliver/etsi_ts/119300_119399/119312/01.04.02_60/ts_119312v010402p.pdf`.

28. SCHNEIER, Bruce. *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C*. John Wiley a Sons, Inc, 1996. ISBN 0471128457.

29. LINDELL, Jonathan Katz Yehuda. *Introduction to Modern Cryptography*. CRC Press, 2014. ISBN 978-1-4665-7027-6. Dostupne na.

30. BLEICHENBACHER, Daniel. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS 1. 2006. Dostupné tiež z: `http://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf`.

31. NAIREN CAO, Adam O'Neill; ZAHERI, Mohammad. Toward RSA-OAEP without Random Oracles. 2020. Dostupné tiež z: `https://eprint.iacr.org/2018/1170.pdf`.

32. AUMASSON, Jean-Philippe. *Serious Cryptography: A Practical Introduction to Modern Encryption*. William Pollock, 2018. ISBN 1-59327-826-8.

33. KLÍMA, Vlastimil. In: *RSA v novém svûtle*. 2001, kap. 1, s. 172–175. Dostupné tiež z: `http://crypto-world.info/klima/2001/chip-2001-11-172-175.pdf`.

34. DUC, Dang Nguyen. *libmcrypto*. 2009. Dostupné tiež z: `https://code.google.com/archive/p/libmcrypto/`. This mcrypto library is aimed at students who are studying public-key cryptography.

35. KNUTH, Donald E. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1998. ISBN 978-0201896848.

36. IRELAND, David. *BigDigits multiple-precision arithmetic source code*. [B.r.]. Dostupné tiež z: `https://www.di-mgt.com.au/bigdigits.html#knuth`. BigDigits is a opensource library.

37. *A small multiple-precision integer implementation in C*. [B.r.]. Dostupné tiež z: `https://github.com/kokke/tiny-bignum-c`. Author name on github : Kokke.

38. *OpenSSL Cryptography and SSL/TLS Toolkit*. [B.r.]. Dostupné tiež z: `https://www.openssl.org/`. The OpenSSL Project develops and maintains the OpenSSL software - a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication.

39. *Valgrind*. [B.r.]. Dostupné tiež z: `https://valgrind.org/`. Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.

Zoznam skratiek

AES Advanced Encryption Standard.

ASCII American Standard Code for Information Interchange.

ASN.1 Abstract Syntax Notation One.

BN Big Number.

CCA Chosen-Ciphertext Attack.

ECC Elliptic Curve Cryptography.

ETSI European Telecommunications Standards Institute.

FIPS Federal Information Processing Standards.

GCC GNU Compiler Collection.

GCD Greatest Common Divisor.

GF Galois Field.

GNFS General Number Field Sieve.

I2OSP Integer-to-Octet-String primitive.

MGF Mask generation function.

MITM Man In The Middle.

MSB Most Significant Bit.

OAEP Optimal asymmetric encryption padding.

OS Operating System.

OS2IP Octet-String-to-Integer primitive.

PKCS Public Key Cryptography Standards.

PRNG Pseudo Random Number Generator.

RNG Random Number Generator.

RSADP RSA Decryption Primitive.

RSAEP RSA Encryption Primitive.

RSAES-OAEP RSA Encryption/decryption Scheme OAEP.

RSAES-PKCS1-V1_5 RSA encryption/decryption scheme in version 1.5.

RSASP1 RSA Signature Primitive 1.

RSAVP1 RSA Verification Primitive 1.

SHA Secure Hash Algorithm.

SLOC Source Lines Of Code.

TRNG True Random Number Generator.

Zoznam príloh

Príloha A CD médium – záverečná práca v elektronickej podobe,

Príloha B Kryptografická knižnica STUDENT,

Príloha C Kryptografická knižnica MCRYPTO,

Príloha D Kryptografická knižnica KOKKE,

Príloha E Kryptografická knižnica OpenSSL,

Príloha F Výpľňová schéma OAEP,

Príloha G Hashovacie funkcie,

Príloha H Dokumentácia.

A Obsah CD Media

Príloha A CD médium – záverečná práca v elektronickej podobe,

Príloha B Kryptografická knižnica STUDENT

```
STUDENT/ .2 src/
├── changes.txt
└── source.txt
EXTENSIONS_STUDENT/ ..... Rozšírenia pre knižnicu STUDENT
├── include/
│   ├── ext_bignum.h
│   ├── ext_file.h
│   └── ext_rsa.h
├── src/
│   ├── ext_bignum.c
│   ├── ext_file.c
│   └── ext_rsa.c
└── TESTS/ ..... Príklady názorných testov
    ├── Makefile
    ├── test01.c
    ├── test02.c
    ├── test05.c
    └── test_vect.c
```

Príloha C Kryptografická knižnica MCRYPTO

```
MCRYPTO/
├── include/
│   ├── bigdigits.h
│   ├── hash.h
│   ├── mcrypto.h
│   ├── md5.h
│   ├── pkcs1-rsa.h
│   ├── sha1.h
│   └── sha2.h
├── src/ ..... Zoznam zdrojových kódov
├── changes.txt
└── source.txt
EXTENSIONS_MCRYPTO/ ..... Rozšírenia pre knižnicu MCRYPTO
└── TESTS/ ..... Príklady názorných testov
    └── Makefile
```

```
    │
    │   ├── test01.c
    │   ├── test02.c
    │   ├── test05.c
    │   ├── test06.c
    │   └── test_vect.c
    ├── extensions_mcrypto.h
    └── extensions_mcrypto.c
```

Príloha D Kryptografická knižnica KOKKE

```
        KOKKE/
    ├── bn.c
    ├── bn.h
    ├── changes.txt
    └── source.txt
└── EXTENSIONS_KOKKE/ ..... Rozšírenia pre knižnicu KOKKE
    ├── include/
    │   ├── ext_file.h
    │   └── ext_rsa.h
    ├── src/
    │   ├── ext_file.c
    │   └── ext_rsa.c
    └── TESTS/ ..... Príklady názorných testov
        ├── Makefile
        ├── rsa_private_1024
        ├── rsa_private_2048
        ├── rsa_private_4096
        ├── rsa_public_1024
        ├── rsa_public_2048
        ├── rsa_public_4096
        ├── test_priv
        ├── test_pub
        ├── test_vect.c
        ├── test01.c
        └── test02.c
```

Príloha E Kryptografická knižnica OpenSSL

```
        OPENSSL/
    ├── Makefile
    ├── test01.c
    ├── test02.c
    ├── test05.c
    └── test06.c
```

Príloha F Výpľňová schéma OAEP

```
        OAEP/
    ├── Makefile
    ├── oaep.c
    ├── oaep.h
    └── changes.txt
```

```
├── source.txt
└── test01.c
```

Príloha G Hashovacie funkcie

```
HASH/
├── EXTENSIONS/ ..... Rozšírenia pre hashovacie funkcie
│   ├── ext_sha.c
│   └── ext_sha.h
├── SHA_1/ ..... SHA-1
│   ├── sha1.c
│   ├── sha1.h
│   ├── changes.txt
│   └── source.txt
├── SHA_256/ ..... SHA-256
│   ├── sha256.c
│   ├── sha256.h
│   ├── changes.txt
│   └── source.txt
└── SHA_512/ ..... SHA-512
    ├── sha256.c
    ├── sha256.h
    ├── fixedint.h
    ├── source.txt .2 TESTS/ ..... Príklady názorných testov
    ├── Makefile
    └── test01.c
```

Príloha H Dokumentácia

```
DOCUMENTATION/
├── HASH.txt
├── KOKKE.txt
├── MCRYPTO.txt
├── OAEP.txt
├── OPENSSL.txt
└── STUDENT.txt
```