SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-5860

IMPLEMENTÁCIA KRYPTOGRAFICKEJ KNIŽNICE S MCELIECE KRYPTOSYSTÉMOM

DIPLOMOVÁ PRÁCA

2015 **František Uhrecký**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-5860

IMPLEMENTÁCIA KRYPTOGRAFICKEJ KNIŽNICE S MCELIECE KRYPTOSYSTÉMOM

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Pavol Zajac

Bratislava 2015 **František Uhrecký**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2014/2015
Evidenčné číslo: FEI-5384-5860

STU FEI

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. František Uhrecký**
ID študenta:	5860
Študijný program:	Aplikovaná informatika
Študijný odbor:	9.2.9. aplikovaná informatika
Vedúci práce:	doc. Ing. Pavol Zajac, PhD.
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Implementácia kryptografickej knižnice s McEliece kryptosystémom**

Špecifikácia zadania:

Práca nadväzuje na existujúci tímový projekt BitPunch. Cieľom je doplniť a rozšíriť vytvorené riešenie a pripraviť nezávislú kryptografickú knižnicu implementujúcu McEliece kryptosystém.

Úlohy:
1. Analyzujte existujúci stav v oblasti.
2. Navrhnite spôsob nezávislej implementácie McEliece kryptosystému vo forme kryptografickej knižnice.
3. Implementujte riešenie.
4. Otestujte a vyhodnoťte riešenie.

Zoznam odbornej literatúry:

1. McEliece, R J. *The Theory of Information and Coding.* Cambridge: Cambridge University Press, 2004. 397 s. ISBN 0-521-83185-7.
2. Moyle, E. – Kelley, D. *Cryptographic Libraries for Developers.* Hingham: Charles River Media, 2006. 463 s. ISBN 1-58450-409-9.

Riešenie zadania práce od: 22. 09. 2014

Dátum odovzdania práce: 22. 05. 2015

SLOVENSKÁ TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky Ústav informatiky a matematiky 812 19 BRATISLAVA

L. S.

Bc. František Uhrecký
študent

prof. RNDr. Otokar Grošek, PhD.
vedúci pracoviska

prof. RNDr. Otokar Grošek, PhD.
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	František Uhrecký
Diplomová práca:	Implementácia kryptografickej knižnice s McEliece kryptosystémom
Vedúci záverečnej práce:	Pavol Zajac
Miesto a rok predloženia práce:	Bratislava 2015

Práca sa zaoberá implementáciou kryptografickej knižnice BitPunch v jazyku C. BitPunch implementuje McEliece kryptosystém, ktorý patrí medzi postkvantové kryptosystémy. Jedná sa minimalistické open source riešenie. Prvá kapitola popisuje problematiku McEliece kryptostému. Druhá kapitola sa venuje ASN.1 notácii a známym implementáciám McEliece kryptostému a ich porovnaniu. Porovnáva sa funkcionalita, rýchlosť a veľkosť knižníc. Nasledujúce kapitoly sú venované samotnému návrhu knižnice BitPunch. Popisuje sa jej štruktúra a význam použitých prvkov. Ďalej sa popisuje použité testovacie prostredie a vysvetľujú sa jeho princípy. Posledná kapitola prezentuje dosiahnuté výsledky.

Kľúčové slová: McEliece, postkvantová kryptografia, kryptografická knižnica, BitPunch

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	František Uhrecký
Diploma Thesis:	Implementation of cryptographic library with McEliece cryptosytem
Supervisor:	Pavol Zajac
Place and year of submission:	Bratislava 2015

Thesis deals with implementation of cryptographic library BitPunch. BitPunch implements one of the post-quantum cryptosystems called McEliece cryptosystem. BitPunch is minimalistic open source solution. First chapter describes McEliece cryptosystem. Second chapter introduces ASN.1 notation used for key serialization. The third chapter deals with known McEliece implementations and its comparison. Comparison is focused on functionality, speed and size of libraries. The second part of the thesis is devoted to BitPunch library: its implementation, architecture, and testing modules. Final chapter summarizes achieved results.

Keywords: McEliecie, post-quantum cryptography, crypto library, BitPunch

Vyhlásenie autora

Podpísaný František Uhrecký čestne vyhlasujem, že som diplomovú prácu Implementácia kryptografickej knižnice s McEliece kryptosystémom vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Bratislava, dňa 17.5.2015

.....

podpis autora

Poďakovanie

Touto cestou by som chcel poďakovať vedúcemu práce a všetkým čo mi pomáhali.

Obsah

Zoznam obrázkov a tabuliek

Zoznam algoritmov

Zoznam zdrojových kódov

Zoznam skratiek a značiek

ASN.1 - Abstract Syntax Notation number One
CCA2 - Adaptive Chosen-Ciphertext Attack
JVM - Java Virtual Machine
MECS - McEliece Cryptosystem
OID - Object Identifier
PKCS - Public Key Cryptosystem
w/o - without
w/ - with

Úvod

V dnešnej modernej dobe sa kladie veľký dôraz na bezpečnosť komunikácie. Drvivá väčšina komunikácie prebieha elektronicky a pre jej zabezpečenie sa využíva asymetrická kryptografia. Bezpečnosť dnes používaných asymetrických kryptosystémov je založená na matematických problémoch, pre ktoré sú známe algoritmy pre kvantové počítače, ktoré tieto problémy riešia. Síce ešte neexituje kvantový počítač, alebo o tom nevieme, ale bolo by vhodné nájsť alternatívu asymetrického kryptosystému, ktorý patrí do postkvantovej kryptografie.

Vhodným kandidátom na postkvantovú kryptografiu je McEliece kryptosystém [13]. Existuje málo implementácií McEliece kryptosystému, pričom niektoré z nich slúžia len pre testovacie účely. Preto sme sa rozhodli implementovať práve McEliece kryptosystém a vytvoriť tak open source kryptografickú knižnicu BitPunch [8]. Projekt BitPunch začal ako tímový projekt [7]. Výsledkom práce bola funkčná implementácia McEliece kryptosystému v jazyku C. Implementácia poskytovala základný McEliece kryptosystém využívajúci Goppa kód a adaptovanú Pointcheval CCA2 konverziu. Spomínaná CCA2 konverzia vyžaduje hašovaciu funkciu, z čoho vznikla závislosť na OpenSSL crypto knižnici.

Cieľom diplomovej práce bolo pokračovať na projekte BitPunch. Hlavným cieľom bolo prerobiť projekt BitPunch na nezávislú kryptografickú knižnicu. Čo predstavovalo navrhnúť vhodnú architektúru knižnice a odstrániť závislosť na OpenSSL knižnici. Implementovali sme modulárne riešenie, čím dosahujeme prehľadnosť a umožňujeme jednoducho rozširovať knižnicu. Tak isto sme implementovali testovacie prostredie, ktoré zabezpečuje plynulosť vývoja.

Kapitola 1 uvádza dva základné pojmy, Poskvantová kryptografia a McEliece kryptosystém. Popisuje základné princípy McEliece kryptosystému a CCA2 bezpečnosť kryptosystému.

Kapitola 2 predstavuje existujúce riešenia. Dozvieme sa o ASN.1 notácii, ktorá zabezpečuje interoperabilitu. Ďalej sa popisujú existujúce implementácie McEliece kryptosystému a ich porovnanie.

Kapitola 3 sa venuje návrhu implementácie knižnice BitPunch. Popisuje použitú architektúru a princípy. Čitateľ sa dozvie všetko potrebné o knižnici, vrátane použitej konvencie, návrhu modulov, správe kontextov a návrhu ASN.1 notácie pre serializáciu kľúčov.

V Kapitole 4 predstavujeme implementáciu testovacieho prostredia, jeho štruktúru a vyhodnocovaciu logiku.

V záverečnej Kapitole 5 zhodnotíme výsledky nášho úsilia. Nachádza sa tam zhrnutie implementácie a výsledky výkonnostných a pamäťových testov.

Táto práca bola realizovaná ako súčasť projektu "Secure implementation of post-quantum cryptography", NATO Science for Peace and Security Programme Project Number: 984520.

1 Analýza

V kapitole si priblížime dva základné pojmy, Poskvantová kryptografia (časť 1.1) a McEliece kryptosystém (časť 1.2). Predstavíme základný princíp kryptosystému, popíšeme algoritmy šifrovania, dešifrovania, generovania kľúčového páru a CCA2 konverzie systému.

1.1 Postkvantová kryptografia

Pod pojmom postkvantová kryptografia si môžeme predstaviť súhrn kryptosystémov, ktoré sú odolné voči útokom na kvantovom počítači. Dnes medzi bežne používané asymetrické kryptosystémy patria RSA, DSA alebo ECDSA. Ich bezpečnosť je založená na problematike diskrétneho logaritmu, ktoré sú riešiteľné pomocou Shorovho algoritmu [2] na kvantovom počítači.

1.2 McEliece Kryptosystém

Kryptosystém navrhol Robert McEliece v roku 1978 [13]. Je jedným z prvých kryptosystémov, ktoré využívali v procese šifrovania náhodnosť. McEliece kryptosystém je založený na teórii kódovania, pričom pôvodný návrh využíva Goppa kódy. Jedná sa o asymetrický kryptosystém. Bezpečnosť kryptosystému je založená na dekódovacom probléme, ktorý patrí medzi NP-úplné problémy. Zatiaľ nie je známy algoritmus, ktorý by riešil tento problém.

Nasledujúce algoritmy popisujú generovanie kľúčového páru, šifrovanie a dešifrovanie správy.

Generovanie kľúčového páru Alg. 1 popisuje proces generovania kľúčového páru. V rámci generovania kľúčového páru potrebujeme zostrojiť Goppa kód. Goppa kód je definovaný ireducibilným monickým polynómom. Pre nájdenie daného polynómu môžeme použiť algoritmus z [7]. Algoritmus testuje ireducibilnosť náhodne zvoleného polynómu daného stupňa. Pre zostrojenie kontrolnej matice kódu je použitý algoritmus uvádzaný v [16]. Pomocou Gaussovej eliminačnej metódy [7] upravíme kontrolnú maticu na systematický tvar a následne je možné vytvoriť generujúcu maticu. Výstupom algoritmu je dvojica: privátny a verejný kľúč. Privátny kľúč - trojica (S, G, P) (náhodná singulárna matica, generujúca matica kódu, permutačná matica). Verejný kľúč - dvojica $(\widehat{G}, t)$ (zamaskovaná generujúca matica a počet chýb, ktoré vie kód opraviť).

Algoritmus 1 Generovanie kľúčového páru

1. Majme náhodne zvolený goppov ireducibilný polynóm g nad poľom $GF(2^m)$ stupňa t,
2. maticu G s rozmermi $k \times n$, generujúcu Goppa kód $\Gamma = (\alpha_1, ..., \alpha_n, g)$, s dimenziou $k = n - td$,
3. náhodne zvolenú binárnu regulárnu maticu S s rozmermi $k \times k$,
4. náhodne zvolenú binárnu permutačnú maticu P s rozmermi $n \times n$,
5. následne vypočítame maticu $\widehat{G} = SGP$, ktorá bude mať rozmery $k \times n$,
6. kde verejný kľúč je dvojica $(\widehat{G}, t)$, kde t je maximálny počet chýb, ktoré kód opraví, privátny kľúč je trojica (S, G, P).

Šifrovanie správy Alg. 2 popisuje proces šifrovania správy. Šifrovanie správy je veľmi rýchly proces. Vstupom algoritmu je správa m (otvorený text) a verejný kľúč $(\widehat{G}, t)$. Šifrovanie je len zakódovanie správy zamaskovaným kódom pomocou generujúcej matice $\widehat{G}$ a následné pridanie t chýb ku kódovému slovu. Výstupom je šifrový text c (kódové slovo s chybou).

Algoritmus 2 Šifrovanie správy

1. Majme správu $\mathbf{m}$ ako binárny k-bitový vektor,
2. vypočítame $\widehat{\mathbf{c}} = \mathbf{m}\widehat{G}$,
3. náhodne zvolíme n-bitový binárny vektor $\mathbf{e}$ s hamingovou váhou t, tzv. chybový vektor,
4. pripočítame chybový vektor ku zakódovanej správe $\mathbf{c} = \widehat{\mathbf{c}} + \mathbf{e}$, čím dostaneme šifrový text $\mathbf{c}$.

Dešifrovanie správy Alg. 3 popisuje proces dešifrovania správy. Vstupom algoritmu je správa c (šifrový text) a privátny kľúč (S, G, P). Nelegitímny príjemca nevie dekódovať správu c na základe znalosti $(\widehat{G}, t)$, ale legitímny príjemca vie. Pomocou inverzných matíc S^{-1} a P^{-1} a dekódovacieho algoritmu Dec transformuje správu a dekóduje ju (teda dešifruje). Výstupom algoritmu je potom správa m (otvorený text). Dekódovanie správy je náročný proces. Existuje viacero efektívnych dekódovacích algoritmov napr. Pattersonov algoritmus [16].

Algoritmus 3 Dešifrovanie správy

1. Majme matice S a P ku ktorým vypočítame inverzné matice S^{-1} a P^{-1},
2. vypočítame $\mathbf{c'} = \mathbf{c}P^{-1}$,
3. pomocou dekódovacieho algoritmu Dec kódu Γ dekódujeme $\mathbf{c'}$ na $\widehat{\mathbf{m}}$,
4. následne vypočítame otvorený text $\mathbf{m} = \widehat{\mathbf{m}}S^{-1}$.

Pattersonov dekódovací algoritmus Alg. 4 popisuje Pattersonov dekódovací algoritmus. Vstupom algoritmu je šifrový text $c' = cP^{-1}$ (aplikácia inverznej permutácie na šifrový text) a kontrolná matica H, ktorej prvky patria do $GF(2^m)$. Výstupom je chybový vektor e', na ktorý keď aplikujeme permutáciu P, dostaneme $e = e'P$. Na výpočet odmocniny polynómu z poľa $GF(2^m)/[x]$ môžeme použiť algoritmus uvádzaný v [7]. Pri počítaní lokátora chyby $\sigma(x)$ si musíme uvedomiť, že už nepočítame modulo $g(x)$.

Algoritmus 4 Pattersonov dekódovací algoritmus

1. Určíme syndróm slova $S_{c'}(x) = c'\mathbf{H}(x^{t-1}, ..., x, 1)$,
2. vypočítame inverziu syndrómu $S_{c'}^{-1}(x)$,
3. nech je $\tau(x) = \sqrt{S_{c'}^{-1}(x) + x}$,
4. nájdeme dva polynómy $a(x)$ a $b(x)$ také, že $b(x)\tau(x) = a(x)\ mod\ (g(x))$, kde $deg(a) \leq \lfloor \frac{t}{2} \rfloor$,
5. určíme lokátor chyby $\sigma(x) = a^2(x) + xb^2(x)$,
6. zrekonštruujeme chybový vektor $e' = (\sigma(\alpha_0), \sigma(\alpha_1), ..., \sigma(\alpha_{n-1})) \oplus (1, ..., 1)$.

1.2.1 CCA2 bezpečnosť

Pôvodný návrh McEliece kryptosystému nie je kryptograficky bezpečný. Existujú viaceré útoky typu Adaptive Chosen-Ciphertext Attack (CCA2) uvádzané v [21, 15, 11]. Cieľom je, aby bol CCA2 bezpečný, preto hovoríme o CCA2 bezpečnej konverzii. V ideálnom prípade CCA2 konverzia transformuje otvorený text na náhodný reťazec, ktorý je potom zašifrovaný klasickým McEliece kryptosystémom. Vhodnou CCA2 konverziou zabezpečíme integritu správy a matica verejného kľúča môže byť uložená v systematickom tvare. Pri použití CCA2 konverzie je bezpečnosť závislá od parametrov systému (n, k, t) ako aj od parametrov konverzie.

Pri niektorých konverziách, ako napr. pri Overbeck alebo adaptovanej Pointcheval

konverzii [16] dochádza k zníženiu zložitosti útoku hrubou silou [21].

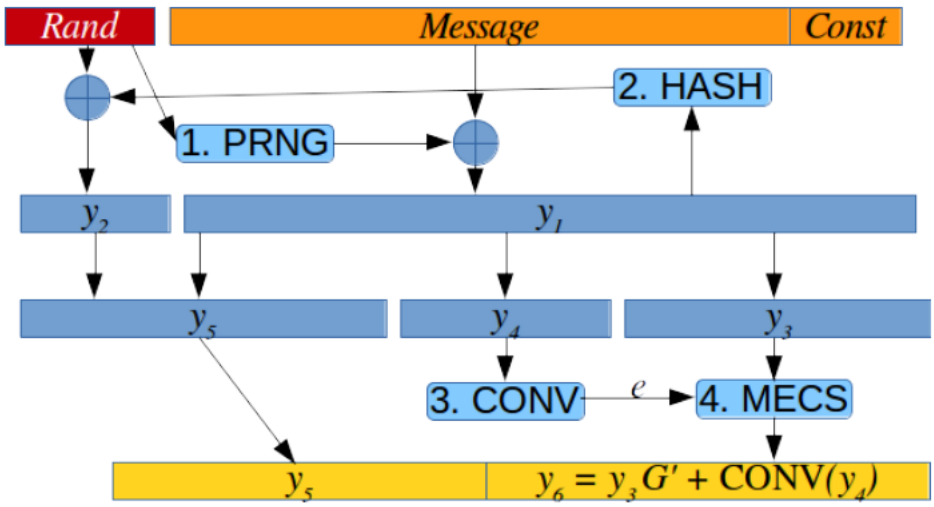
Adaptovaná Pointcheval CCA2 bezpečná konverzia Jedná sa o konverziu veľmi podobnú Pointcheval konverzii a má nasledovnú formu:

(1) Otvorený text vstupujúci do pôvodného MECS je $\widehat{m} = r_1||hash(m||r_2)$, kde r_1 je náhodný bitový vektor dĺžky $k - l$ a výstup hašovacej funkcie je neodlíšiteľný od náhodného l-bitového vektora,

(2) šifrový text je trojica $(c_1, c_2, c_3) = (\widehat{m}\widehat{G} + e, hash(r_1) + m, hash(e) + r_2)$.

Popis útoku Pri zlej voľbe parametrov [21] (vrátane l), dochádza zníženiu zložitosti nasledovného útoku. Ak $\widehat{G}$ je v systematickom tvare a správa m je odlíšiteľná od náhodného reťazca, tak útočník vie extrahovať r_1 časť. Tento bitový vektor je ovplyvnený približne $t' = t(k-l)/n$ chybami z vektora e. Útočník vie, že $r_1' = r_1 + e$. Teraz sa útočník pokúsi určiť t' chýb na pozíciách 0 až $k - l - 1$. Ak je $m' = c_2 + hash(r_1' + e_1)$ pre daný chybový vektor odlíšiteľná od náhodného vektora, tak uspel a dešifroval správu $m = m'$.

Kobara-Imai CCA2 bezpečná gamma konverzia Na obr. 1 je zobrazená efektívna Kobara-Imai CCA2 bezpečná gamma konverzia. Veľkosť správy sa natiahne pridaním *Rand* a *Const* (viď. obr. 1). Následne sa toto natiahnutie zredukuje zakódovaním informácie do chybového vektora e.



Obrázok 1: [15] Kobara-Imai CCA2 bezpečná gamma konverzia MECS

Nasledujúci zoznam vysvetluje pojmy použité v schéme:

- *Rand* - náhodný kľúč relácie (veľkosť je daná bezpečnosťou symetrickej šifry),

- *PRNG* - kryptograficky bezpečný pseudonáhodný generátor (vo všeobecnosti to môže byť symetrická šifra),
- *Const* - verejná konštanta,
- *HASH* - kryptograficky bezpečná hašovacia funkcia,
- *CONV* - konverzná funkcia (invertibilná), ktorá počíta chybový vektor e (veľkosti n a váhou t),
- *MECS* - štandardný McEliecie (chybový vektor e je pridaný argument).

Šifrovanie Alg. 5 popisuje šifrovanie správy pri použití gamma konverzie. Vstupom algoritmu je: verejný kľúč (G', t), správa m a konštanta $Const$, ktorých celková dĺžka (dĺžka $m||Const$) je rovná výstupu $Prng(Rand)$. $Rand$ slúži ako náhodný kľúč relácie, pomocou ktorého zašifrujeme správu m a konštantu $Const$ na y_1. Pomocou hašovacej funkcie $HASH$ utajíme kľúč $Rand$, dostaneme $y_2 = HASH(y_1) + Rand$. Následne rozdelíme y_1 na tri časti y_1', y_3 a y_4. Časť y_4 použijeme na odvodenie chybového vektora s váhou t pomocou $CONV$, časť y_3 ako vstup do McEliece šifrovacieho algoritmu $MECS_{enc}$ a časť y_1' zreťazíme s utajeným kľúčom y_2 tak, že $y_5 = y_2||y_1'$. Výstupom algoritmu je šifrový text $c = y_5||(y_3G' + CONV(y_4))$.

Algoritmus 5 Kobara-Imai CCA2 bezpečná gamma konverzia - algoritmus šifrovania

1. Majme správu m, konštantu $Const$ a zvoľme náhodné r-bitové číslo $Rand$,
2. vypočítajme $y_1 = Prng(Rand) + (m||Const)$,
3. vypočítajme $y_2 = HASH(y_1) + Rand$,
4. nech $y_1 = y_1'||y_4||y_3$,
5. nech $y_5 = y_2||y_1'$,
6. vypočítajme chybový vektor $e = CONV(y_4)$,
7. zašifrujme y_3 pomocou McEliece kryptosystému tak, že $y_6 = MECS_{enc}(e, y_3) = y_3G' + CONV(y_4)$,
8. šifrový text je $c = y_5||y_6$.

Dešifrovanie Alg. 6 popisuje dešifrovanie správy pri použití gamma konverzie. Vstupom algoritmu je privátny kľúč, správa $c = y_5||y_6$ a konštanta $Const$. Legitímny príjemca zrekonštruuje chybový vektor e a y_3 z y_6 pomocou štandardného McEliece

dešifrovacieho algoritmu $MECS_{dec}$. Následne zrekonštruuje y_4 rovné $CONV^{-1}(e)$. Potom vypočíta kľúč relácie $Rand$ tak, že $Rand = HASH(y_1) + y_2$ a vypočíta správu m a konštantu $Const'$ ako $Prng(Rand)+y_1$. Následne môže overiť integritu správy porovnaním konštanty $Const$ a $Const'$.

Algoritmus 6 Kobara-Imai CCA2 bezpečná gamma konverzia - algoritmus dešifrovania

1. Majme správu $c = y_5||y_6$ a konštantu $Const$,
2. dešifrujme y_6 pomocou McEliece kryptosystému $(e, y_3) = MECS_{dec}(y_6)$, kde e je vypočítaný chybový vektor,
3. vypočítajme $y_4 = CONV^{-1}(e)$,
4. nech $y_5 = y_2||y_1'$, zrekonštruujme y_2 a y_1 tak, že $y_1 = y_1'||y_4||y_3$,
5. vypočítajme $Rand = y_2 + HASH(y_1)$,
6. vypočítajme $(m||Const') = PRNG(Rand) + y_1$,
7. ak $Const'$ je rovná $Const$, tak je zachovaná integrita správy,
8. otvorený text je správa m.

2 Existujúce riešenia

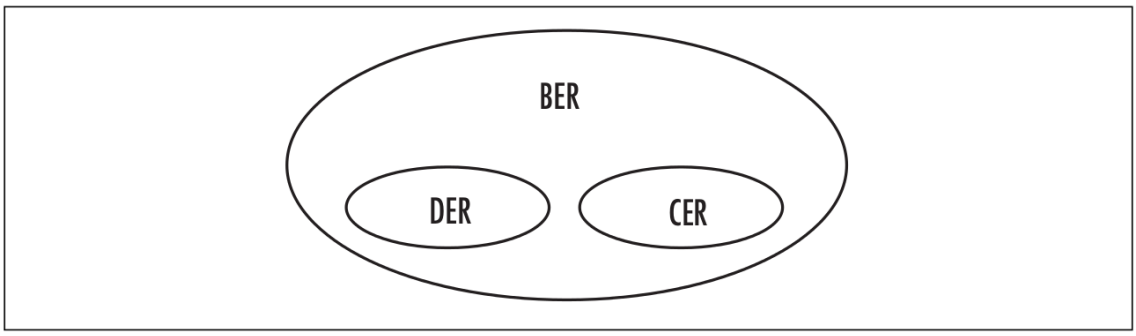
Cieľom práce je implementovať kryptografickú knižnicu implementujúcu McEliece kryptosystém. Keďže už existujú viaceré riešenia, ktoré implementujú McEliece kryptosystém, je dôležité mať o týchto riešeniach prehľad. Následne z nich môžeme vychádzať. Pre zabezpečenie užívateľského/vývojárskeho komfortu je dobré používať otvorené štandardy, ktorými vieme zabezpečiť interoperabiltu medzi viacerými riešeniami. Medzi takéto štandardy patrí napr. telekomunikačný štandard ASN.1 [9].

2.1 ASN.1

Abstract Syntax Notation One (ASN.1) patrí medzi ITU-T[1] štandardy pre kódovanie a reprezentáciu bežných dátových typov, ako reťazce (bitové, znakové), číselné typy, zložené dátové typy a iné. Jednoducho povedané, ASN.1 špecifikuje ako kódovať dáta tak, aby ich vedeli interpretovať aj nástroje tretej strany [17, 9].

Špecifikácia ITU-T X.680 dokumentuje štandard ASN.1. Používa sa v kryptografii, kde formálne špecifikuje kódovanie na bajtovej úrovni rôznych dátových typov, ktoré nie sú priamo prenositelné. Štandard zabezpečí ich deterministickosť.

ASN.1 podporuje štandardné kódovacie pravidlá ako Basic Encoding Rules (BER), Canonical Encoding Rules (CER) a Distinguished Encoding Rules (DER) (obr. 2). Tieto tri modely špecifikujú ako kódovať a dekódovať ASN.1 rovnakého typu.



Obrázok 2: [17] ASN.1 Kódovacie pravidlá

BER pravidlá umožňujú variácie kódovania pre rovnaké dáta. Ľubovolné ASN.1 kódované pomocou CER alebo DER môže byť dekódované pomocou BER, ale nie naopak. Táto vlastnosť vyplýva aj z obr. 2. Ľubovolné dátové typy môžu byť popísané BER a potom sa môžu aplikovať pravidlá CER alebo DER. Kompletná ASN.1 špecifikácia je

[1]International Telecommunication Union - Telecommunication Standardization Sector

komplexnejšia, ale v kryptografii majú najväčší význam DER pravidlá.

ASN.1 bola štandardizovaná v roku 1984 [9] organizáciou CCITT[2]. V roku 1989 CCITT vydala špecifikácie X.208 a X.209 ktoré popisujú ASN.1 a BER pravidlá. Posledné špecifikácie pochádzajú z roku 2008. Štandard ISO 8824 je rozdelený do 4 častí a popisuje samotnú ASN.1 notáciu. Štandard ISO 8825 je rozdelený do 5 častí a popisuje kódovacie pravidlá. Konkrétne časť ISO 8825-1 | ITU-T X.690 špecifikuje BER, CER a DER pravidlá.

2.1.1 ASN.1 OBJECT IDENTIFIER typ

OBJECT IDENTIFIER (OID) je identifikačný mechanizmus vyvinutý organizáciami ITU-T a ISO[3]/IEC[4]. Jedná sa o stromovú štruktúru reprezentovanú číselným označením s bodkami. Začínajúc organizáciou, typom štandardu, podkategóriami a iné. Jedno OID nesmie byť použité na iný účel, než bolo pôvodne určené. Tým zabezpečíme jednoznačnosť.

OID je často používané v asymetrickej kryptografii. Používa sa na špecifikovanie hašovacích algoritmov použitých v certifikátoch, použitých šifrovacích algoritmov, operačných módov šifier a iné.

Existuje OID repozitár, kde je prehľad registrovaných OID (`http://oid-info.com`). Nemusí obsahovať všetky existujúce OID. Dobrovoľníci pridávajú do repozitára nové OID, ktoré následne schváli administrátor repozitára.

Príkladom môže byť algoritmus MD5, ktorého OID je 1.2.840.113549.2.5, čo na prvý pohľad vyzerá komplikovane, ale po zobrazení hierarchie pomocou OID Repository (`http://oid-info.com/get/1.2.840.113549.2.5`) dostaneme: *iso(1) member-body(2) us(840) rsadsi(113549) digestAlgorithm(2) md5(5).* Alebo napríklad OID 1.3.6.1.4.1.8301.3.1.3 *iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) 8301 3 1 3* definuje *Post Quantum Cryptography project.* V prípade, keď aplikácia (nie samotný dekóder) dekóduje OID, tak vie aký algoritmus má použiť.

2.2 Knižnice - súčasný stav

Existuje viacero kryptografických knižníc, niektoré sa zaoberajú postkvantovou kryptografiou. Nižšie popísané knižnice/nástroje implementujú McEliece kryptosystém. Vychádzajú z pôvodného návrhu McEliece, preto používajú Goppa kód. Jedná sa o implementácie v jazyku C#, Java, C++ alebo C. Niektoré sú závislé na externých knižniciach

[2]International Telegraph and Telephone Consultative Committee, dnešné ITU-T
[3]International Organization for Standardization
[4]International Engineering Consortium

ako OpenSSL alebo NTL.

2.2.1 BitPunch

BitPunch vznikol v roku 2013 ako tímový projekt [7]. Autormi sú A. Gulyás, M. Klein, J. Kudláč, F. Machovec a F. Uhrecký. Výsledkom projektu je open source implementácia McEliece kryptosystému v jazyku C. Pôvodná verzia 0.0.1, je závislá iba na hašovacej funkcii z knižnice OpenSSL. Knižnica disponuje nasledovnou funkcionalitou:

- matematické operácie nad poľom GF(2^m):
  - reprezentácia pomocou bitových polí,
  - operácie s maticami, vektormi, polynómami,
  - rozšírený Euklidov algoritmus, odmocnina (polynómy z poľa $GF(2^m)/[x]$),
- generovanie kľúčového páru pre McEliece kryptosystém s parametrami $m = 11$, $t = 50$,
- šifrovanie, dešifrovanie jedného bloku,
- Pointcheval CCA2 konverzia.

Zhrnutie McEliece Knižnica implementuje McEliece kryptosystém pre ktorý poskytuje nasledovnú funkcionalitu:

- Generovanie kľúčového páru,
- šifrovanie, dešifrovanie,
- klasický návrh McEliece PKCS,
- CCA2 bezpečné schémy:
  - Pointcheval CCA2 konverzia,
- jazyk C, závislosť na ext. hašovacej funkcii (z OpenSSL).

2.2.2 BouncyCastle

Kryptografická knižnica, ktorej počiatky siahajú do roku 2000 [12]. Momentálne poskytuje bohaté API implementované v jazyku Java a C#. V jazyku Java poskytuje tzv. provider pre Java Cryptography Extension (JCE), čo je Java API pre kryptografické operácie. Ďalej pre Java Cryptography Architecture (JCA), čo je architektúra pre kryptografické riešenia v Jave [14]. Do JCA patrí aj JCE. Knižnica poskytuje prácu s ASN.1 objektami. Implementuje odľahčené API pre TLS protokol (podľa RFC 2246, RFC 4346) a iné [12].

Zhrnutie McEliece Knižnica implementuje McEliece kryptosystém pre ktorý poskytuje nasledovnú funkcionalitu:

- Generovanie kľúčového páru,
- šifrovanie, dešifrovanie, podpis,
- klasický návrh McEliece PKCS,
- CCA2-bezpečné schémy [5]:
  - Fujisaki-Okamoto konverzia,
  - Kobara-Imai koverzia,
  - Pointcheval konverzia,
- export kľúčov do ASN.1 formátu [20]:
  - OID 1.3.6.1.4.1.8301.3.1.3.4.1 McEliece PKCS
  - OID 1.3.6.1.4.1.8301.3.1.3.4.2.x McEliece CCA2 konverzie
- jazyk JAVA, závislosť od JVM.

Pre všetky spomínané konverzie poskytuje knižnica aj podpisové schémy.

2.2.3 Calculator

Calculator v0.1 je jednoduchá demonštrácia McEliece kryptosystému. Jej autorom je M. Repka, pochádza z roku 2014. Je implementovaná v jazyku C++ a využíva matematické operácie z knižnice NTL. Aplikácia umožňuje generovať kľúčový pár pre rôzne parametre kryptosystému a uložiť ho do súboru, šifrovať a dešifrovať súbor. Ďalej je možné spustiť výkonnostné testy pre dané parametre m a t. V rámci testu sa budú generovať náhodné kľúče. V testoch sa zohľadní chybový vektor s hammingovou váhou s vopred definovaným rozpätím.

Zhrnutie McEliece Nástroj implementuje McEliece kryptosystém pre ktorý poskytuje nasledovnú funkcionalitu:

- Generovanie kľúčového páru,
- šifrovanie, dešifrovanie,
- klasický návrh McEliece PKCS,

- export/import kľúčov do/z binárneho formátu,
- jazyk C++, závislosť na NTL knižnici.

2.2.4 Flea

Flea je open source (LGPL, BSD licencia) implementácia McEliece kryptosystému v jazyku C, prvá verzia z roku 2013. Autorom knižnice je Falko Strenzke [4, 18, 19]. Strenzke vo svojej práci použil už existujúcu open source implementáciu HyMES (The Hybrid McEliece Encryption scheme) [3], ktorej autormi sú Bhaskar Biswas a Nicolas Sendrier. Vylepšil výkon knižnice, zameral sa na postranné kanály a na útoky vnášaním chýb.

HyMES implementácia sa pôvodne odlišuje od klasického návrhu McEliece kryptosystému. Prenášaná informácia sa kóduje aj do chybového vektora. Pre hľadanie koreňov polynómu pre nájdenie chybového vektora používa Berlekampov algoritmus (Berlekamp Trace Algorithm, BTA) [1]. Algoritmus je ale efektívny len pre Goppov polynóm malého stupňa. Ďalej využíva rôzne time-memory tade-offs, čiže je použiteľná hlavne pre výkonnejšie platformy ako PC. Cieľom Strenzkeho bolo vytvoriť implementáciu, ktorá by fungovala aj na obmedzených zariadeniach.

Strenzke implementoval nasludujúce zmeny:

- Odstránil informáciu kódovanú v chybovom vektore,
- implementoval ochranu proti postranným kanálom a útokom vnášaním chýb,
- dešifrovanie bez kontrolnej matice, čo šetrí pamäť použitú pre privátny kľúč,
- kódovacie a dekódovacie funkcie,
- implementácia Overbeck CCA2 [5],
- upravil spracovanie chýb (návratových hodnôt) v kóde.

Zmenu parametrov systému n (dĺžka kódu) a t (stupeň Goppovho polynómu, počet chýb, ktoré vie kód opraviť) je potrebné zmeniť už predkompiláciou zdrojových súborov. Nastavenia parametrov sa nachádzajú v súbore `include/api/flea/code_based.h`.

Zhrnutie McEliece Knižnica implementuje McEliece kryptosystém pre ktorý poskytuje nasledovnú funkcionalitu:

- Generovanie kľúčového páru,
- šifrovanie, dešifrovanie,

- klasický návrh McEliece PKCS,
- CCA2 bezpečné schémy [5]:
  - Overbeck konverzia,
- jazyk C, nezávislosť na ext. knižniciach,
- potreba prekompilovať pri zmene parametrov systému,
- vhodné na obmedzené platformy.

2.3 Porovnanie implementácií

Testované boli nasledujúce implementácie McEliece kryptosystému:

1. MECS Calculator (M. Repka),
2. BitPunch 0.0.1 MECS (F. Uhrecký a kolektív),
   `https://github.com/FrUh/BitPunch`
3. Flea 0.1.1, HyMES (F. Strenzke),
   `http://www.cryptosource.de/`
4. Java BouncyCastle McEliecePKCS trieda,
   `https://www.bouncycastle.org/`

Testovacia platforma mala nasledovnú konfiguráciu:

- **CPU** Intel Core i5-2430M CPU @ 2.40GHz × 4
- **RAM** 2 × 4GB DDR3 1333MHz
- **OS** Ubuntu 14.04.1 LTS, Linux 3.13.0-34-generic
- **GCC** 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

Tabuľka 1: Porovnanie SW implementácií MECS

	KeyGen [ms]	Enc [μs]	Dec [ms]	Veľkosť knižnice Shared [KiB]	Static [KiB]
1. Calculator	1395	124	36.6	92 (+2MiB)	116 (+4MiB)
2. BitPunch	866	62	3.9	64	96
3. a) Flea w/o H	46	34	0.6	160	232
3. b) Flea w/ H	44	34	0.2	160	232
4. BouncyCastle	1096	201	8.6	583 (celý BC 3MiB)	

Najrýchlejšou implementáciou z testu je knižnica Flea. Knižnica poskytuje dešifrovanie s alebo bez predpočítanej kontrolnej matice H. Ako vidieť z tabuľky s použitím matice prebehlo dešifrovanie rýchlejšie. BouncyCastle a Calculator sú pomalšie. Hlavným dôvodom pre BouncyCastle je, že knižnica funguje na Java Virtual Machine (JVM). Calculator využíva knižnicu NTL, ktorá nie je optimalizovaná na rýchlosť. BitPunch dosahuje pomerne dobré výsledky, s ohľadom na to, že sa jedná o začínajúci projekt.

V rámci veľkostí knižníc dosahuje najlepšie výsledky BitPunch a Flea. Calculator musí obsahovať knižnicu NTL, ktorá zaberá 4 MiB. BouncyCastle vyžaduje pre svoju funkčnosť celú knižnicu BC, ktorá obsahuje potrebné API.

3 Knižnica BitPunch

V kapitole sa zameriame na knižnicu BitPunch. Popíšeme návrh knižnice, organizáciu modulov a konvenciu. Vysvetlíme správu interných štruktúr a používanie kontextov. Keď pochopíme základné princípy organizácie knižnice, tak ju dokážeme rozširovať o nové moduly.

3.1 Základné pojmy

Na začiatok si definujme základné pojmy, ktoré popisujú knižnicu, alebo sú v nej použité. Predpokladáme, že čitateľ má programátorské znalosti, tak objasníme len tie pojmy, ktoré sú potrebné pre pochopenie návrhu knižnice. Medzi tieto pojmy patria:

Modul Modul predstavuje skupinu súborov, ktorá poskytuje určitú funkcionalitu. Napríklad matematický modul, ktorý implementuje matematické operácie. Moduly sa nachádzajú v samostatných zložkách, pričom môžu byť závislé od ostatných modulov. Moduly sprehľadňujú organizáciu knižnice a umožňujú jednoduchšiu správu. Ďalej môžeme skompilovať jednoúčelovú knižnicu, v ktorej vypneme nepotrebné moduly.

Kontext Jazyk C nie je objektovo orientovaný, tak túto vlastnosť implementujeme pomocou kontextov. Kontext predstavuje vnútorný stav modulu. Kontext si môžeme predstaviť ako špecifickú štruktúru, ktorá si uchováva atribúty potrebné pre jej beh (analogicky atribúty/metódy tried). Uľahčuje používanie (užívateľ sa nezaujíma o internú organizáciu) a implementáciu (API sa nemení, ale vnútorná štruktúra sa môže). Ďalej môže urýchliť procesy, pri ktorých je možné použiť techniku time memory trade-off.

3.2 Konvencia

Navrhnutá konvencia knižnice nám pomáha jednoduchšie sa orientovať v kóde. Každé pravidlo má svoj význam a svojím spôsobom urýchľuje používanie. Postupne si prejdeme všetky pravidlá, ktoré by sa mali dodržať pri úpravách zdrojových kódov knižnice.

3.2.1 Odsadzovanie

Nasledujúca časť kódu znázorňuje použitý štýl odsadzovania, používanie zátvoriek a medzier. Znak použitý pre odsadenie je tabulátor.

Zdrojový kód 1: Príklad odsadenia

```
int i;
if (p1->deg != p2->deg) {
    return -1;
```

```
    }
    for (i = 0; i <= p1->deg; i++) {
        if (p1->coef[i] != p2->coef[i]) {
            return i + 1;
        }
    }
    return 0;
```

3.2.2 Názvoslovie

Všetky časti knižnice sú jednoznačne identifikovateľné, že patria do knižnice. Slúžia k tomu viaceré prefixy, ktoré upresňujú význam jednotlivých častí.

Prefixy Pre názvoslovie funkcií, štruktúr alebo dátových typov sú definované viaceré prefixy. Tieto prefixy uľahčujú vyhľadávanie dostupnej funkcionality knižnice. Prefixy sú definované na základe toho do akého modulu patria, s akým dátovým typom pracuje funkcia alebo či sa jedná o definíciu dátového typu. Pre prefixy platia nasledujúce pravidlá:

BPU_* Predstavuje základný prefix pre všetky funkcie, makrá a dátové typy knižnice. Následne pokračuje tým, aké dátové typy spracováva, napr. `BPU_gf2`, `BPU_gf2Vec`, `BPU_gf2Mat`, `BPU_gf2x`, `BPU_gf2xVec`, `BPU_gf2xMat`, `BPU_gf2xPoly`, `BPU_perm` alebo podľa modulu `BPU_mecs`, `BPU_code`, `BPU_math`.

BPU_T_* Prefix, ktorý sa používa pre definíciu dátových typov. Ďalej sa rozlišuje o aký dátový typ sa jedná, napr. `BPU_T_GF2_`.

BPU_T_EN_* Prefix, ktorý sa používa pre definíciu enumeračných dátových typov. Následne sa rozširuje o modul, do ktorého patrí, napr. `BPU_T_EN_Mecs_` alebo `BPU_T_EN_Code_`.

BPU_EN_* Prefix, ktorý sa používa pre definíciu hodnôt enumeračných dátových typov. Následne sa rozširuje o modul, do ktorého patrí, napr. `BPU_EN_MECS_` alebo `BPU_EN_CODE_`.

BPU_EC_* Prefix, ktorý sa používa pre definíciu návratových hodnôt. Následne sa rozširuje o modul, do ktorého patrí, napr. `BPU_EC_MECS_` alebo `BPU_EC_CODE_`. V prípade všeobecnej návratovej hodnoty, ktorá nie je závislá len na konkrétnom module, za prefixom nasleduje hneď názov chyby.

BPU_CONF_* Prefix, ktorý slúži pre konfiguráciu knižnice. Pri kompilácii je možné zvoliť viaceré konfigurácie. Tie sú zabezpečené pomocou makier, ktoré začínajú

týmto prefixom.

Konvencia funkcií/makier Konvencia funkcií/makier používa nasledovné pravidlá:

1. Všetky funkcie/makrá začínajú prefixom na základe spracovávaných typov argumentov alebo modulu,
2. najskôr sa definujú výstupné argumenty funkcie/makra, následne vstupné argumenty s modifikátorom `const`,
3. každé slovko názvu funkcie/makra začína veľkým písmenom (okrem počiatočného prefixu).

Príklad definície funkcie:

```
int BPU_permGetInv(BPU_T_Perm_Vector *out,
    const BPU_T_Perm_Vector *in);
```

3.3 Moduly knižnice

Knižnica je organizovaná do viacerých modulov. Jednotlivé moduly môžu byť od seba závislé, pričom si poskytujú potrebnú funkcionalitu (obr. 3). Medzi základné moduly knižnice patria:

- `asn1` - poskytuje import/export kľúčov pomocou libtasn1 knižnice (len Goppa kód),
- `code` - poskytuje implementáciu kódov (nGoppa, QC-MDPC [6]),
- `crypto` - poskytuje implementáciu McEliece kryptosystému, CCA2 konverzií, hašovacích funkcií a funkcií súvisiacich s kryptosystémom,
- `math` - poskytuje implementáciu matematických operácií nad poľom $GF(2^m)$, $GF(2^m)/[x]]$ a iné,
- `prng` - poskytuje API pre pseudonáhodný generátor.

V rámci modulov sa ešte môže nachádzať hlbšie členenie ak sa to vyžaduje. Vhodným príkladom je modul `crypto`. Obsahuje nasledovné časti:

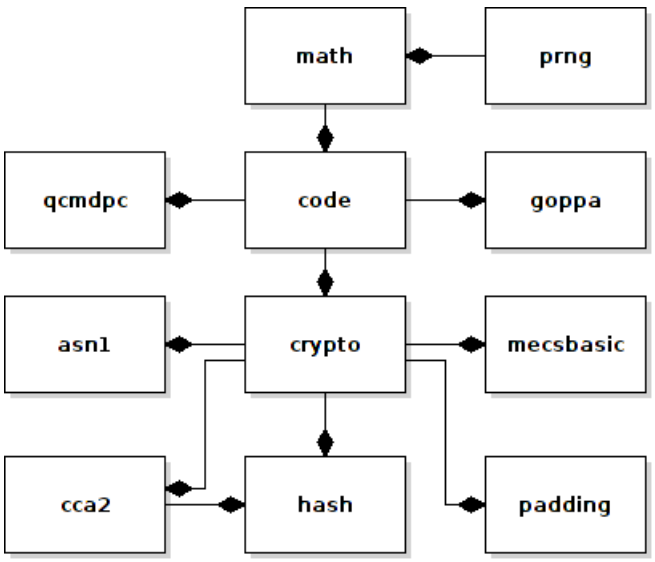
- `cca2` - implementácia CCA2 konverzií,
- `hash` - implementácia hašovacej funkcie z knižnice PolarSSL (`https://polarssl.org/`),

- `mecsbasic` - implementácia základného algoritmu McEliecie kryptosystému,
- `padding` - implementácia paddingu bloku správy.

Fyzická organizácia modulov je riešená podpriečinkami, kde sa nachádzajú zdrojové súbory modulov. Štruktúra je nasledovná:

```
lib/src/bitpunch
├── asn1
├── code
│   └── goppa
├── crypto
│   ├── cca2
│   ├── hash
│   │   └── polarssl
│   │       └── polarssl
│   ├── mecsbasic
│   └── padding
├── math
└── prng
```

Nasledujúci diagram (obr. 3) popisuje vzťahy medzi jednotlivými modulmi. Napríklad modul `crypto` nemôže existovať bez modulu `code`. Ale modul `code` môže existovať bez modulu `crypto`.



Obrázok 3: Vzťahy modulov

3.4 Kontexty

Moduly používajú na interpretáciu vnútorného stavu kontexty. V knižnici rozlišujeme momentálne tri druhy kontextov a to:

1. `Math` kontext - pre matematický modul,

2. `Code` kontext - pre modul pre kódy,

3. `Mecs` kontext - pre kryptografický modul.

Definície kontextov sa nachádzajú vo svojich moduloch. Názvy súborov v ktorých sú definované majú nasledovnú formu:

- `nazovmoduluctx.h`,
- `nazovmoduluctx.c`.

3.4.1 Inicializácia parametrov kontextu

Kontexty potrebujú pre inicializáciu špecifické parametre, od ktorých sú závislé. V našom prípade inicializujeme parametre pre `Mecs` a `Code` kontext. Zabezpečíme to volaním funkcií:

- `BPU_mecsInitParams*` - inicializácia parametrov pre kryptosystém,
- `BPU_codeInitParams*` - inicializácia parametrov kódu.

Týmito funkciami vieme nastaviť interné štruktúry, ktoré reprezentujú parametre systému/kódu. Znak `*` znamená, že môžu existovať rôzne parametre pre rôzne typy systémov/kódov napr. `BPU_mecsInitParamsGoppa`. Všeobecne sa nastavujú parametre reprezentované dátovým typom `union`:

Zdrojový kód 2: Parametre systému/kódu

```
typedef union _BPU_T_UN_Code_Params{
    BPU_T_Goppa_Params *goppa;
    BPU_T_Qcmdpc_Params *qcmdpc;
    // HERE you add your code params structure
}BPU_T_UN_Code_Params;

typedef union _BPU_T_UN_Code_Params BPU_T_UN_Mecs_Params;
```

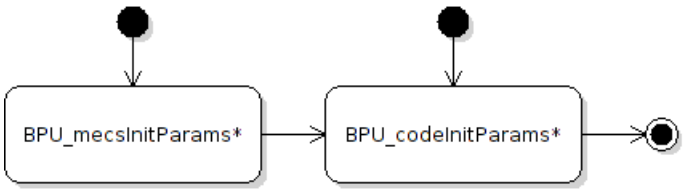
Príklad parametrov pre Goppa kód zobrazuje zdr. kód 3.

Zdrojový kód 3: Parametre Goppa kódu

```
typedef struct _BPU_T_Goppa_Params {
    uint16_t m;
```

```
    uint16_t t;
    BPU_T_GF2_16x mod;
}BPU_T_Goppa_Params;
```

Na diagrame (obr. 4) vidíme priebeh inicializácie parametrov. Užívateľ môže pracovať buď s `Mecs` alebo `Code` parametrami. Vždy potrebuje zavolať len jedno volanie, v horizontálnom smere sa volajú funkcie automaticky.



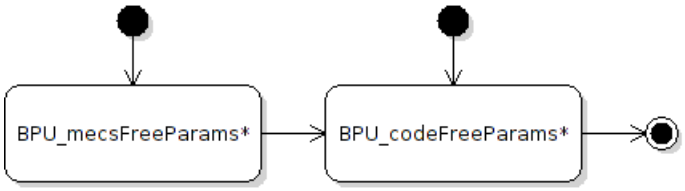
Obrázok 4: Priebeh inicializácie parametrov

Po inicializácii (skončení práce s parametrami) potrebujeme uvoľniť pamäť, čo zabezpečujú funkcie:

- `BPU_mecsFreeParams*` - uvoľnenie parametrov pre kryptosystém,
- `BPU_codeFreeParams*` - uvoľnenie parametrov kódu.

Takýto spôsob inicializácie/uvoľnenia parametrov zabezpečuje to, že užívateľ nemusí definovať štruktúry, ktoré reprezentujú parametre, ale iba zavolá potrebnú inicializačnú funkciu, ktorá má v predpise definované parametre. Týmto spôsobom umožnujeme jednotné volanie pre inicializáciu kontextu.

Tak isto ako pri inicializácii aj pri uvoľnení parametrov platia pravidlá. Na diagrame (obr. 5) vidíme postupnosť volaní funkcií.



Obrázok 5: Priebeh uvoľnenia parametrov

3.4.2 Inicializácia kontextu

Nato aby sme vedeli pracovať s konkrétnym modulom musíme inicializovať jeho kontext. Inicializácia sa realizuje volaním knižničnej funkcie s argumentami. Keďže rozlišujeme tri druhy kontextov, tak k nim prislúchajú aj tri rôzne inicializačné funkcie a to:

- `BPU_mathInitCtx` - inicializácia kontextu pre matematický modul,
- `BPU_codeInitCtx` - inicializácia kontextu pre kódy,
- `BPU_mecsInitCtx` - inicializácia kontextu pre McEliece kryptosystém.

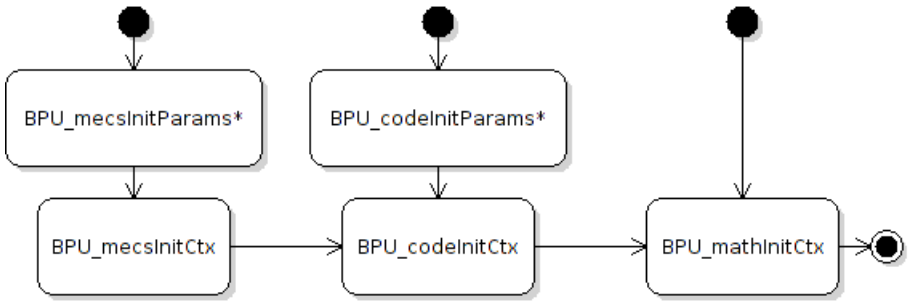
Po skončení používania kontextu, je potrebné uvoľniť pamäť. Na to slúžia funkcie:

- `BPU_mathFreeCtx` - uvoľnenie kontextu pre matematický modul,
- `BPU_codeFreeCtx` - uvoľnenie kontextu pre kódy,
- `BPU_mecsFreeCtx` - uvoľnenie kontextu pre McEliece kryptosystém.

Kontexty majú medzi sebou závislosti. Ako vidno na diagrame (obr. 6), napríklad `Mecs` kontext zavisí na `Code` kontexte a následne `Math` kontexte.

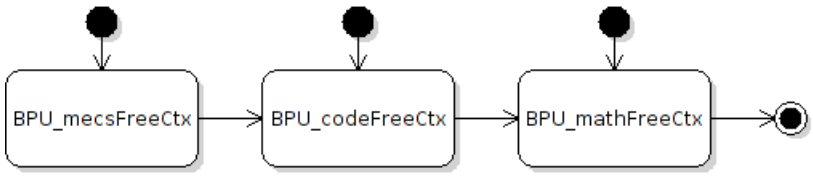
Obrázok 6: Vzťahy kontextov

Tak isto aj inicializácia kontextov má svoje pravidlá. Diagram (obr. 7) zobrazuje, aké sú volania pri inicializácii kontextov (závislosť inicializácií). V prípade, že sa inicializuje `Mecs` kontext, zavolá sa automatický inicializácia `Code` kontextu a `Math` kontextu (`Code` kontext vyvolá túto inicializáciu). Keď chce používateľ pracovať s kontextami, tak reálne volá len vertikálne volania zobrazené v diagrame (preto je tam znázornených viac počiatočných stavov). Horizontálne volania sa volajú automaticky.



Obrázok 7: Priebeh inicializácie kontextov

Uvoľnenie kontextu má tiež svoje pravidlá. Diagram (obr. 8) zobrazuje priebeh volaní funkcií. Užívateľ môže pracovať s ľubovolným kontextom, preto je na diagrame znázornených viacero počiatočných stavov. Horizontálne sa volajú automaticky.



Obrázok 8: Priebeh uvoľňovania kontextov

3.4.3 Math kontext

Matematický modul (`math`) používa vlastný kontext. Zdrojový súbor kontextu sa nachádza v `bitpunch/math/mathctx.h`. Definícia kontextu je nasledovná:

Zdrojový kód 4: Math kontext

```
typedef struct _BPU_T_Math_Ctx {
    BPU_T_GF2_16x *exp_table;
    BPU_T_GF2_16x *log_table;
    BPU_T_GF2_16x mod;
    uint8_t mod_deg;
    int ord;
}BPU_T_Math_Ctx;
```

mod, mod_deg Reprezentuje polynóm nad poľom GF2 a stupeň polynómu. Polynóm ďalej definuje konečné pole $GF(2^m)$.

exp_table Reprezentuje exponenciálnu tabuľku pre konečné pole $GF(2^m)$. Keď g je generátorom poľa, tak index v tabuľke i nám povie, koľko je $e = g^i \ mod \ (mod)$, kde $e = exp\_table[i]$. Momentálne sa v knižnicu používa generátor rovný 2.

log_table Reprezentuje logaritmickú tabuľku pre konečné pole $GF(2^m)$. Indexom tabuľky je prvok e z nášho poľa. Hodnota na tomto indexe nám povie, na koľkú musíme umocniť generátor g aby sme dostali daný prvok e. Inak zapísané $e = g^i \ mod \ (mod)$, kde $i = log\_table[e]$.

Exponenciálna a logaritmická tabuľka implementuje vhodný time memory trade-off. Výrazne to urýchľuje umocňovanie v konečnom poli (často používaná operácia), keďže sa jedná len o pozretie do tabuľky. Jedná sa o pomerne malé konečné pole, ktoré obsahuje 2^{11} prvkov (pre $m = 11$). Takže tento trade-off nie je problém používať ani implementovať.

Inicializácia kontextu Predpis funkcie pre inicializáciu kontextu je nasledovný:

Zdrojový kód 5: Predpis funkcie pre inicializáciu Math kontextu

```
int BPU_mathInitCtx(
    BPU_T_Math_Ctx **ctx,
    const BPU_T_GF2_16x g,
    const BPU_T_GF2_16x mod);
```

ctx Funkcia vyžaduje smerník na kontext, ktorý inicializuje.

g Je generátor konečného poľa $GF(2^m)$, ktoré ktoré budeme generovať.

mod Viď 3.4.3.

3.4.4 Code kontext

Modul pre kódy používa komplexnejší kontext. Niektoré parametre sú všeobecné pre viaceré kódy, preto sa nachádzajú priamo v kontexte. Pre špecifické parametre kódov sa používa dátový typ `union _BPU_T_UN_Code_Spec` viď zdr. kód 8, ktorý definuje tieto špecifické vlastnosti (napr. štruktúra pre Goppa kódy viď zdr. kód 9). Zdrojový súbor kontextu sa nachádza v `bitpunch/code/codectx.h`. Všeobecná definícia kontextu je nasledovná:

Zdrojový kód 6: Code kontext

```
typedef struct _BPU_T_Code_Ctx {
    BPU_T_EN_Code_Types type;
    int (* _encode)(
        BPU_T_GF2_Vector *out,
        const BPU_T_GF2_Vector *in,
        const struct _BPU_T_Code_Ctx *ctx);
    int (* _decode)(
        BPU_T_GF2_Vector *out,
        const BPU_T_GF2_Vector *in,
        const struct _BPU_T_Code_Ctx *ctx);
    BPU_T_Math_Ctx *math_ctx;
    BPU_T_GF2_Vector *e;
    BPU_T_UN_Code_Spec *code_spec;

    uint16_t code_len;
    uint16_t msg_len;
    uint8_t t;
```

```
}BPU_T_Code_Ctx;
```

type Definuje typ použitého kódu. Knižnica implementuje Goppa a QC-MDPC[6] kód, ale týmto pádom je ju možné rozšíriť aj o iné typy kódov.

Zdrojový kód 7: Typ kódu

```
typedef enum _BPU_T_EN_Code_Types {
    BPU_EN_CODE_GOPPA,
    BPU_EN_CODE_QCMDPC
    // HERE you can add your code type
}BPU_T_EN_Code_Types;
```

code_len, msg_len, t Parametre vychádzajú z parametrov samotného kryptosystému (m, t). Jedná sa o všeobecné parametre, ktoré definujú dĺžku kódového slova, dĺžku vstupného slova a počet chýb, ktoré vie kód opraviť (nemusí to byť horná hranica).

code_spec Štruktúra, ktorá spadá do typu `union`. V nej sa definujú špecifické vlastnosti kódov, závislé od typu kódu. Pre Goppa kód sa používa `BPU_T_Goppa_Spec`. Takto je možné rozšíriť knižnicu o ďalšie typy kódov, kde programátor si sám zadefinuje, čo si potrebuje uchovávať.

Zdrojový kód 8: Špecifické štruktúry pre kódy

```
typedef union _BPU_T_UN_Code_Spec{
    BPU_T_Goppa_Spec *goppa;
    BPU_T_Qcmdpc_Spec *qcmdpc;
    // HERE you add your code spec structure
}BPU_T_UN_Code_Spec;
```

Nasledujúci kód zobrazuje príklad definície štruktúry, ktorá sa používa pre Goppa kód.

Zdrojový kód 9: Štruktúra pre Goppa kód

```
typedef struct _BPU_T_Goppa_Spec {
    BPU_T_GF2_Matrix *g_mat;
    BPU_T_GF2_16x_Matrix *h_mat;
    BPU_T_GF2_16x_Poly *g;
    BPU_T_Perm_Vector *permutation;
    uint16_t support_len;
}BPU_T_Goppa_Spec;
```

g_mat, h_mat Generujúca a kontrolná matica kódu.

g Generujúci goppov polynóm.

permutation Predstavuje permutáciu, ktorou je maskovaný verejný kľúč, generujúca matica kódu. Implementácia je taká, že táto permutácia je v skutočnosti zložená z dvoch permutácií. Z náhodnej permutácie p_1 a permutácie p_2, ktorá "zamieša" maticu použitú pri generovaní kódu. Permutácia p_2 je zväčša identická, len v prípade, že nie je možné zostrojiť systematický tvar matice, je zložená z $n-1$ náhodných permutácií, kde n je počet pokusov o vytvorenie systematického tvaru.

support_len Počet prvkov poľa.

math_ctx Smerník odkazujúci sa na aktuálny Math kontext (časť 3.4.3).

e Vektor reprezentujúci chybový vektor. Používa sa pri šifrovaní/dešifrovaní správy. V kontexte sa nachádza z toho dôvodu, aby sa uľahčila implementácia CCA2 konverzie.

_encode, _decode Smerník na funkciu kódovania/dekódovania. Zároveň definuje predpis funkcie kódovania(zdr. kód 10)/dekódovania(zdr. kód 11). Funkcie sa nastavia pri inicializácii kontextu podľa zvoleného typu kódu. Týmto spôsobom zabezpečujeme zmenu tela funkcie bez rekompilácie knižnice a rýchly spôsob volania funkcie (bez zbytočných if-ov, lebo meníme referenciu na funkciu).

Zdrojový kód 10: Predpis funkcie encode

```
int (* _encode)(
    BPU_T_GF2_Vector *out,
    const BPU_T_GF2_Vector *in,
    const struct _BPU_T_Code_Ctx *ctx);
```

Zdrojový kód 11: Predpis funkcie decode

```
int (* _decode)(
    BPU_T_GF2_Vector *out,
    const BPU_T_GF2_Vector *in,
    const struct _BPU_T_Code_Ctx *ctx);
```

Inicializácia kontextu Predpis funkcie pre inicializáciu kontextu je nasledovný:

Zdrojový kód 12: Predpis funkcie pre inicializáciu Code kontextu

```
int BPU_codeInitCtx(
    BPU_T_Code_Ctx **ctx,
    const BPU_T_UN_Code_Params *params,
    const BPU_T_EN_Code_Types type);
```

ctx Funkcia vyžaduje smerník na kontext, ktorý inicializuje.

params Parametre kódu. Viď časť 3.4.1.

type Viď 3.4.4.

3.4.5 Mecs kontext

Kontext, ktorý reprezentuje stav McEliece modulu (krytosystému). Zdrojový súbor kontextu sa nachádza v `bitpunch/crypto/mecsctx.h`. Definícia kontextu je nasledovná:

Zdrojový kód 13: Mecs kontext

```
typedef struct _BPU_T_Mecs_Ctx {
    BPU_T_EN_Mecs_Types type;
    int (* _encrypt)(
        BPU_T_GF2_Vector *out,
        const BPU_T_GF2_Vector *in,
        const struct _BPU_T_Mecs_Ctx *ctx);
    int (* _decrypt)(
        BPU_T_GF2_Vector *out,
        const BPU_T_GF2_Vector *in,
        const struct _BPU_T_Mecs_Ctx *ctx);
    int (* _genKeyPair)(struct _BPU_T_Code_Ctx *ctx);

    BPU_T_Code_Ctx *code_ctx;
    uint16_t pt_len;
    uint16_t ct_len;
}BPU_T_Mecs_Ctx;
```

type Definuje aký typ McEliece kryptosystému sa použije, s akým typom kódov alebo s akou CCA2 konverziou.

Zdrojový kód 14: Typ kryptosystému

```
typedef enum _BPU_T_EN_Mecs_Types {
```

```
    BPU_EN_MECS_BASIC_GOPPA = 1,
#ifdef BPU_CONF_MECS_CCA2_POINTCHEVAL_GOPPA
    BPU_EN_MECS_CCA2_POINTCHEVAL_GOPPA,
#endif
    BPU_EN_MECS_BASIC_QCMDPC,
#ifdef BPU_CONF_MECS_CCA2_POINTCHEVAL_QCMDPC
        BPU_EN_MECS_CCA2_POINTCHEVAL_QCMDPC,
#endif
}BPU_T_EN_Mecs_Types;
```

code_ctx Smerník odkazujúci sa na aktuálny kontext kódu (časť 3.4.4).

pt_len, ct_len Definujú dĺžku otvoreného/šifrového textu v bitoch. Inicializujú sa na začiatku podľa parametrov kryptosystému a použitého typu (kódu, konverzie).

_encrypt, _decrypt Smerník na funkciu šifrovania/dešifrovania. Zároveň definuje predpis funkcie šifrovania(zdr. kód 15)/dešifrovania(zdr. kód 16). Funkcie sa nastavia pri inicializácii kontextu podľa zvoleného typu kryptosystému. Týmto spôsobom zabezpečujeme zmenu tela funkcie bez rekompilácie knižnice a rýchly spôsob volania funkcie (bez zbytočných if-ov, meníme referenciu).

Zdrojový kód 15: Predpis funkcie encrypt

```
int (* _encrypt)(
    BPU_T_GF2_Vector *out,
    const BPU_T_GF2_Vector *in,
    const struct _BPU_T_Mecs_Ctx *ctx);
```

Zdrojový kód 16: Predpis funkcie decrypt

```
int (* _decrypt)(
    BPU_T_GF2_Vector *out,
    const BPU_T_GF2_Vector *in,
    const struct _BPU_T_Mecs_Ctx *ctx);
```

Inicializácia kontextu Predpis funkcie pre inicializáciu kontextu je nasledovný:

Zdrojový kód 17: Predpis funkcie pre inicializáciu Mecs kontextu

```
int BPU_mecsInitCtx(
    BPU_T_Mecs_Ctx **ctx,
    const BPU_T_UN_Mecs_Params *params,
    const BPU_T_EN_Mecs_Types type);
```

ctx Funkcia vyžaduje smerník na kontext, ktorý inicializuje.

params Parametre kryptosystému. Viď časť 3.4.1.

type Viď 3.4.5.

3.5 ASN.1

Knižnica umožňuje serializáciu kľúčov kryptosystému (pre Goppa kód). Je to zabezpečené pomocou knižnice libtasn1. Táto knižnica podporuje prácu s ASN.1 notáciou a umožňuje kódovať výstupné dáta v DER formáte. Ako je popísané v časť 2.1.1, pomocou OID vieme identifikovať algoritmus, ku ktorému patrí použitá notácia. My sme prevzali existujúce OID z projektu BouncyCastle, a to presne:

- 1.3.6.1.4.1.8301.3.1.3.4.1 - McEliece PKCS,
- 1.3.6.1.4.1.8301.3.1.3.4.2.2 - McEliece Pointcheval PKCS.

Použité OID môžeme vyhľadať na stránke Technickej Univerzity Darmstadt (`http://www.hrz.tu-darmstadt.de/itsicherheit/object_identifier/oids_der_informatik__cdc/`).

Definovali sme vlastný predpis notácie pre privátny (zdr. kód 18) a verejný kľúč (zdr. kód 19). Nachádzajú sa v súboroch `asn1/MecsPriKey.asn1` a `asn1/MecsPubKey.asn1`. Notácie obsahujú len nevyhnutné dáta pre obnovenie kľúča.

Zdrojový kód 18: Privátny kľúč ASN.1

```
MecsPriKey ::= SEQUENCE {
    oid OBJECT IDENTIFIER, -- oid
    m INTEGER, -- degree of field polynomial
    t INTEGER, -- error capability of code
    mod INTEGER, -- field polynomial GF(2^m)
    g OCTET STRING, -- goppa polynomial
    p OCTET STRING, -- permutation
    h_mat OCTET STRING -- control matrix H over GF2[x]
}
```

Zdrojový kód 19: Verejný kľúč ASN.1

```
MecsPubKey ::= SEQUENCE {
    oid OBJECT IDENTIFIER, -- OID
    m INTEGER, -- degree of field polynomial
    t INTEGER, -- error capability of code
    g_mat OCTET STRING -- generator matrix GF2
}
```

Implementujeme API pre import/export kľúča z/do súboru. Funkcie sa nachádzajú v module `asn1` v súbore `bitpunch/asn/asn.h`. Ich predpis je nasledovný:

Zdrojový kód 20: Predpis funkcie pre import kľúča

```
int BPU_asn1LoadKeyPair(
    BPU_T_Mecs_Ctx **ctx,
    const char *pri_key_file,
    const char *pub_key_file);

int BPU_asn1LoadPriKey(
    BPU_T_Mecs_Ctx **ctx,
    const char *pri_key_file);

int BPU_asn1LoadPubKey(
    BPU_T_Mecs_Ctx **ctx,
    const char *pub_key_file);
```

Zdrojový kód 21: Predpis funkcie pre export kľúča

```
int BPU_asn1SaveKeyPair(
    BPU_T_Mecs_Ctx *ctx,
    const char *pri_key_file,
    const char *pub_key_file);

int BPU_asn1SavePriKey(
    BPU_T_Mecs_Ctx *ctx,
    const char *pri_key_file);

int BPU_asn1SavePubKey(
```

```
    BPU_T_Mecs_Ctx *ctx,
    const char *pub_key_file);
```

Funkcie pracujú s aktuálnym kontextom kryptosystému a potrebujú cestu k vstupnému/výstupnému súboru.

4 Testovacie prostredie

V rámci projektu bolo potrebné vyvinúť testovacie prostredie. Aby sme vedeli zabezpečiť, že neporušíme existujúcu funkcionalitu zmenou alebo novou funkcionalitou (tzv. regresné testovanie). Takýmto spôsobom uľahčujeme a zlepšujeme vývoj. Preto sme sa rozhodli implementovať jednoduché testovacie prostredie. Naše testovanie prostredie `runTest` je implementované v jazyku Python. Prostredie poskytuje:

- výpis zoznamu existujúcich testov,
- spúšťanie jednotlivých alebo množiny testov,
- základné informácie o priebehu testu,
- detekcia chyby v logu,
- kontrola návratovej hodnoty testu (ak je 0 tak `PASSED` inak `FAILED`),
- umožňuje nastaviť log level na `DEBUG`, `INFO` alebo `ERROR`,
- konečný sumár spustených testov.

4.1 Štruktúra prostredia

Prostredie je organizované v jednoduchej súborovej štruktúre. Koreňovým adresárom je `runtest`. V ňom sa nachádza obslužný skript `runTest`, ktorý zabezpečuje spúšťanie a vyhodnocovanie testov. Po spustení testov sa vytvorí priečinok `results`, v ktorom sa vytvárajú podpriečinky s názvom aktuálneho času spustenia testov. Obsahujú aktuálnu kópiu zdrojových súborov knižnice, nad ktorými sa spustia testy. Správa sa to ako dočasný workspace pre testovacie prostredie. Druhý priečinok je `tests`. Tu sa nachádzajú všetky dostupné testy. Testy sú zaradené do jednotlivých `suite`, skupina testov s podobnými vlastnosťami. Jednote suite-y sú reprezentované priečinkami. Tu môžeme vidieť príklad reálneho prostredia:

```
./runtest
├── results
├── tests
│   ├── builds
│   │   └── testBuild.sh
│   ├── memory
│   │   ├── testLibSize.sh
│   │   ├── testMemLeaks.sh
│   │   ├── testMemRun.c
│   │   └── testMemRun.sh
│   └── register.py
└── runTest
```

4.2 Registrácia testu

Každý test je potrebné zaregistrovať, aby ho bolo možné spustiť. Registrácia testu je jednoduchá. Je reprezentovaný záznamom v zdrojovom súbore pythonu, v štruktúre typu `dictionary`. Súbor sa nachádza v `runtest/tests/register.py`.

Zdrojový kód 22: Príklad registrácie testu

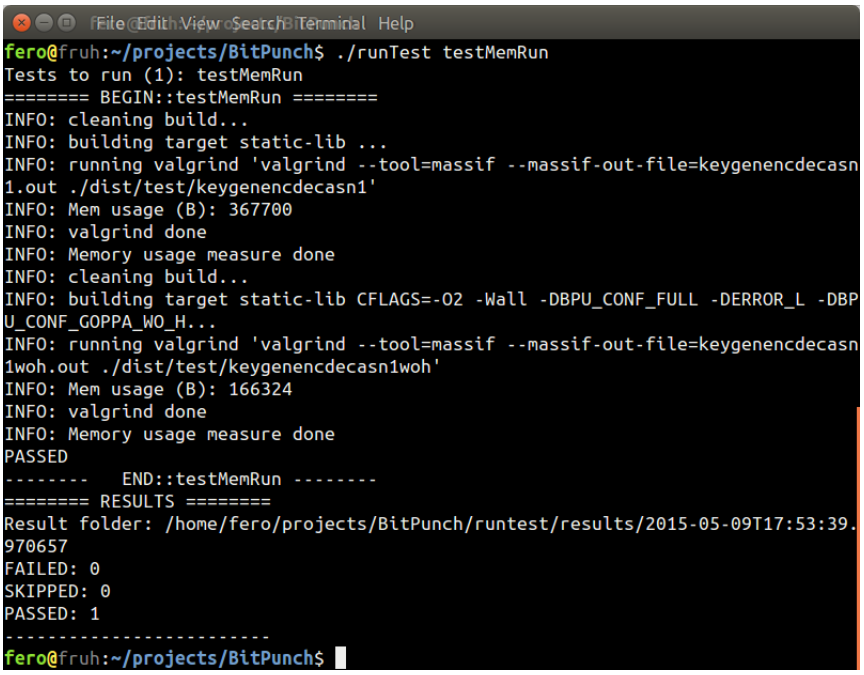
```
TESTS = {
    "testMemLeakWithH" : {
        "suite" : "memory",
        "file" : "testMemLeaks.sh",
        "args" : "test-debug",
        "runLevel" : "regular",
    }
}
```

Význam jednotlivých kľúčov v slovníku je nasledovný:

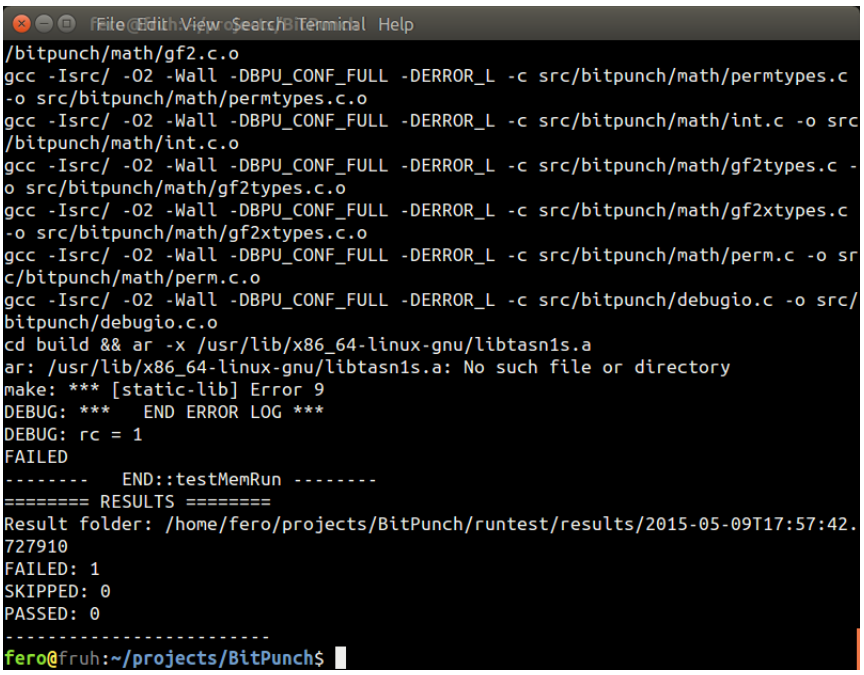
- hlavným kľúčom je názov testu napr. `testMemLeakWithH`,
- `suite` - predstavuje skupinu (priečinok) v akej sa test nachádza,
- `file` - názov spustiteľného súboru testu,
- `args` - argumenty s akými sa spúšťa test,
- `runLevel` - existujú 3 rôzne runLevel hodnoty, podľa náročnosti testu a to `express`, `regular` a `extreme`.

4.3 Vlastnosti testu a vyhodnotenie

Logika testovacieho prostredia je priamočiara. Každý test je reprezentovaný spustiteľným súborom, ktorý v prípade úspechu vracia 0 inak nenulové číslo. Čiže nezáleží na tom v akom programovacom jazyku je test napísaný. Každý test môže vypisovať priebeh na obrazovku. Defaultne sa zobrazujú len správy, ktoré začínajú reťazcom rovným alebo menším log levelu napr. `DEBUG: msg`, `INFO: msg` alebo `ERROR: msg`. V prípade že nastane chyba, návratový kód je rôzny od 0, alebo sa objavil v logu reťazec `ERROR`, vypíše sa celý priebeh testu a vyhlási sa za `FAILED`. Pre ilustráciu priebehu testov viď obr. 9 a obr. 10.



Obrázok 9: Priebeh testu - PASSED



Obrázok 10: Priebeh testu - FAILED

5 Výsledky riešenia

V kapitole si zhrnieme dosiahnuté výsledky nášho riešenia. Ukážeme si čo sme implementovali/zmenili oproti pôvodnej verzii knižnice BitPunch. Ďalej porovnáme výkon a pamäťové nároky aktuálneho riešenia.

5.1 Zhrnutie zmien v knižnici

Knižnica prešla významnými zmenami a kompletnou refaktorizáciou kódu. Zmeny v knižnici implementované v rámci diplomovej práce sa dajú zosumarizovať v nasledovných bodoch (viac detailov je uvedených v Kapitole 3):

- Refaktorizácia kódu a zjednotenie konvencie,
- zavedenie modulárnej architektúry knižnice (základné moduly: `asn1`, `code`, `crypto`, `math`, `prng`),
  - zabezpečuje pridanie nových modulov do knižnice bez majoritných zmien v kóde (napr. nový typ kódu, konverzie),
- nový manažment kontextov modulov (`Mecs`, `Code` a `Math` kontext),
- čiastočná optimalizácia rýchlosti,
- redukcia maximálneho využitia pamäti,
  - optimalizácia maticových operácií,
  - možnosť dešifrovania s/bez predpočítanej kontrolnej matice H,
- import/export kľúčov vo formáte ASN.1 v DER kovaní pomocou knižnice libtasn1 (pre Goppa kód),
- reimplementácia hašovacej funkcie SHA-512 z knižnice PollarSSL,
- pridanie prvku 0 (nula) do konečného poľa,
- pridanie QC-MDPC kódov (spolupráca s A. Gulyás - Implementácia QC-MDPC McEliecovho kryptosystému [6]),
- pripravené prostredie pre implementáciu opatrení pre postranné kanály (M. Klein - Postranné kanály v SW implementácii McElieceovho kryptosystému [10])
- implementácia jednoduchého testovacieho prostredia.

5.2 Výkonostné a pamäťové testy

Podarilo sa nám rozšíriť funkcionalitu knižnice ale aj znížiť jej nároky na výkon a pamäť. Najskôr si predstavíme konfiguráciu testovacieho prostredia:

- **CPU** Intel Core i5-2430M CPU @ 2.40GHz × 4,
- **RAM** 2 × 4GB DDR3 1333MHz,
- **OS** Ubuntu 14.04.1 LTS, Linux 3.13.0-34-generic,
- **GCC** 4.8.2 (Ubuntu 4.8.2-19ubuntu1),
- parametre kryptosystému: m = 11 a t = 50, Goppa kód.

Výsledky rýchlostných testov vidíme v tab. 2:

Tabuľka 2: Porovnanie rýchlosti BitPunch

BitPunch	KeyGen [ms]	Enc [μs]	Dec [ms]	Shared [KiB]	Static [KiB]
1. v0.0.1	866	62	3.9	64	96
2. v0.0.3 wo. H	768	48	52	92	172
3. v0.0.3 w. H	723	48	3.6	92	172

Z tabuľky vidíme, že dosahujeme lepšie výsledky pri šifrovaní, dešifrovaní a generovaní kľúčového páru. Jedine bez predpočítanej kontrolnej matice H priebieha dešifrovanie pomalšie, ale to sme predpokladali, pretože sa táto matica v podstate dopočítava za behu.

Výsledky veľkosti knižnice zobrazuje tab. 3:

Tabuľka 3: Porovnanie veľkosti knižnice

BitPunch	Shared [KiB]	Static [KiB]
1. v0.0.1 with CCA2	64**	96**
2. v0.0.3 FULL	92*	272
3. v0.0.3 FULL w/o print	72*	224
4. v0.0.3 Basic MECS w/o print	56*	208
5. v0.0.3 Basic + CCA2 w/o print	60*	212

- * + 80 pre knižnicu libtasn1,
- ** + veľkosť knižnice OpenSSL lcrypto (SHA-512),
- FULL - enc, dec, keygen, CCA2, ASN.1, Goppa kód.

Veľkosť knižnice narástla, pretože sme pridali novú funkcionalitu. Hlavný nárast pociťujeme pre statickú knižnicu, pretože už implementuje hašovaciu funkciu SHA-512 a tak isto aj ASN.1 z knižnice libtasn1. Ale ako vidíme, pre dynamickú knižnicu vieme dosiahnuť porovnateľné, alebo aj lepšie výsledky (v prípade že nepotrebujeme funkcie typu `print`).

Spotrebu pamäte sme merali pomocou programu `valgrind` a jeho nástroja `massif`. Meranie zaznamenalo maximálnu spotrebu pamäte počas generovania kľúčového páru, šifrovania, dešifrovania jedného bloku a importu/exportu kľúča pomocou ASN.1. Šifrovanie a dešifrovanie prebiehalo s/bez adaptovanej Pointcheval CCA2 konverzie. V tab. 4 vidíme výsledky testov:

Tabuľka 4: Spotreba pamäte

BitPunch	Memory usage [KiB]
1. v0.0.1	8 818
2. v0.0.3 wo. H	162
3. v0.0.3 w. H	362

Pamäťové nároky sme znížili rádovo z MiB na niekoľko KiB. Veľkú úsporu dosahujeme, keď sa nepredpočíta kontrolná matica, čo je ale to na úkor rýchlosti. Toto riešenie je vhodné len na obmedzené zariadenia, kde nie je dostatok pamäte pre beh programu.

Záver

V práci sme sa zaoberali implementáciou kryptografickej knižnice s McEliece kryptosystémom. Cieľom bolo implementovať nezávislú kryptografickú knižnicu s McEliece kryptosystémom. Rozhodli sme sa pokračovať v tímovom projekte BitPunch a pripravili sme open source riešenie, ktoré používa verejný repozitár GitHub (`https://github.com/FrUh/BitPunch`).

Podarilo sa nám prerobiť existujúce riešenie na modulárnu štruktúru s jednotnou správou kontextov. Náš návrh priniesol projektu flexibilitu a umožňuje jednoduchú správu a rozširovanie knižnice. Refaktorizovali sme existujúci kód a zjednotili konvenciu. Zrýchlili sme kritické operácie a výrazne optimalizovali spotrebu pamäte (z MiB na KiB). Kompletné výsledky nášho úsilia sumarizuje Kapitola 5.

Modulárnosť knižnice sme preverili doplnením externej implementácie QC-MDPC kódov (A. Gulyás) bez väčších zásahov do kódu. Pripravili sme knižnicu tak, aby sa dalo na nej v budúcnosti pracovať (prehľadnosť kódu, dostupná online dokumentácia `http://fruh.github.io/BitPunch`). V spolupráci s M. Kleinom sa pracuje na odstránení postranných kanálov.

Veríme, že projekt má perspektívu a bude sa na ňom pokračovať, aby upovedomil o svojej existencii open source komunitu a dostal sa do reálneho používania.

Zoznam použitej literatúry

[1] Berlekamp, E. R. Factoring polynomials over large finite fields. *Mathematics of Computation 24*, 111 (1970), 713–735.

[2] Bernstein, Daniel J and Buchmann, Johannes and Dahmen, Erik. *Post-quantum cryptography*. Springer.

[3] Bhaskar Biswas, Nicolas Sendrier. *Hybrid McEliece, HyMES*. `https://www.rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`, (5.10.2014).

[4] cryptosource GmbH. Research, 2014. `http://www.cryptosource.de` (5.10.2014).

[5] D. Engelbert, R. O., and Schmidt, A. A Summary of McEliece-Type Cryptosystems and their Security. Cryptology ePrint Archive, Report 2006/162, 2006. `https://eprint.iacr.org/2006/162/20060510:130300`, (15.11.2014).

[6] Gulyás, A. Implementácia QC-MDPC McEliecovho kryptosystému, 2015. Diplomová práca.

[7] Gulyás, A., Klein, M., Kudláč, J., Machovec F., Uhrecký, F. Reálna implementácia Code-based cryptography, 2014. Tímový projekt.

[8] Gulyás, A., Klein, M., Kudláč, J., Machovec F., Uhrecký, F. BitPunch - McEliece Cryptographic Library, 2015. `https://github.com/FrUh/BitPunch` (9.5.2015).

[9] ITU. Introduction to ASN.1, 2014. `http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx`, (2.2.2015).

[10] Klein, M. Postranné kanály v SW implementácii McElieceovho kryptosystému, 2015. Diplomová práca.

[11] Kobara, K., and Imai, H. Semantically secure McEliece public-key cryptosystems-conversions for McEliece PKC. In *Public Key Cryptography* (2001), Springer, pp. 19–35.

[12] Legion of the Bouncy Castle Inc. About the legion of the bouncy castle, 2013.
https://www.bouncycastle.org/, (19.11.2014).

[13] McEliece, R. J. A public-key cryptosystem based on algebraic coding theory. *DSN progress report 42*, 44 (1978), 114–116.

[14] Oracle and/or its affiliates. Java Cryptography Architecture (JCA) Reference Guide, 2014.
https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html, (2.12.2014).

[15] Repka, Marek, and Pavol Zajac. Overview of the Mceliece Cryptosystem and its Security. *Tatra Mountains Mathematical Publications 60.1* (2014), 57–83.

[16] Shoufan, A., Wink, T., Molter, G., Huss, S., and Strentzke, F. A novel processor architecture for McEliece cryptosystem and FPGA platforms. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on* (2009), IEEE, pp. 98–105.

[17] St Denis, Tom. *Cryptography for developers*. Syngress (2006).

[18] Strenzke, F. A side-channel secure and flexible platform-independent implementation of the mceliece pkc –flea version 0.1.1–.

[19] Strenzke, F. Efficiency and Implementation Security of Code-based Cryptosystems. Fachbereich Informatik der Technischen Universität Darmstadt.

[20] Technische Universität Darmstatd. OIDs des Fachgebietes CDC der Informatik, 2015.
http://www.hrz.tu-darmstadt.de/itsicherheit/object_identifier/oids_der_informatik__cdc/oids_des_fachgebietes_cdc_der_informatik.de.jsp, (28.01.2015).

[21] Zajac, P. A note on CCA2-protected McEliece Cryptosystem with a systematic public key.

6 Elektronická príloha

Obsahom elektronickej prílohy je:

- diplomová práca vo formáte pdf,
- LaTeX zdrojové súbory diplomovej práce,
- finálna verzia knižnice BitPunch v0.0.4,
- dokumentácia ku knižnici vo formáte html,
  - programátorská dokumentácia,
  - dokumentácia k testovaciemu prostrediu.

Súborová štruktúra priloženého nosiča:

```
/
├── BitPunch
│   ├── BitPunch-src.zip
│   └── BitPunch-doc.zip
├── dp.pdf
├── document-src.zip
└── README.txt
```