SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-98022

IMPLEMENTÁCIA QC-MDPC KRYPTOSYSTÉMU NAD GF(4)

DIPLOMOVÁ PRÁCA

2023 **Bc. Tomáš Vavro**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-98022

IMPLEMENTÁCIA QC-MDPC KRYPTOSYSTÉMU NAD GF(4)

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Mgr. Tomáš Fabšič, PhD.

Bratislava 2023 **Bc. Tomáš Vavro**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023
Evidenčné číslo: FEI-16607-98022

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Tomáš Vavro**
ID študenta:	98022
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Mgr. Tomáš Fabšič, PhD.
Vedúci pracoviska:	doc. Ing. Milan Vojvoda, PhD.
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Implementácia QC-MDPC kryptosystému nad GF(4)**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Pokrok vo vývoji kvantového počítača má vážne dôsledky aj pre kryptografiu. Je známe, že dostatočne výkonné kvantové počítače budú vedieť efektívne riešiť problém faktorizácie čísla na prvočísla a problém diskrétneho logaritmu. Na náročnosti riešenia týchto problémov je založená bezpečnosť v súčasnosti používaných asymetrických kryptosystémov (napríklad RSA). To znamená, že v prípade existencie dostatočne výkonného kvantového počítača by súčasné asymetrické kryptosystémy už neboli bezpečné. Niektoré odhady hovoria, že takto výkonné kvantové počítače by mohli existovať už o 10 rokov. Je preto dôležité, pracovať na vývoji nových asymetrických kryptosystémov, ktoré budú odolné voči útokom kvantového počítača, a ktoré by mohli nahradiť súčasné asymetrické kryptosystémy.

Na dôležitosť tejto témy upozornil aj americký inštitút pre štandardy a technológiu NIST v správe Report on Post-Quantum Cryptography. NIST zároveň vyhlásil súťaž Post-Quantum Cryptography Standardization Process s cieľom navrhnúť nové kryptografické štandardy odolné voči kvantovým počítačom.

Do aktuálne prebiehajúceho štvrtého kola súťaže sa dostal aj McEliece kryptosystém s QC-MDPC kódmi. V tomto kryptosystéme sa výpočty realizujú nad poľom GF(2). V roku 2019 bola publikovaná aj verzia QC-MDPC McEliece kryptosystému nad GF(4). Implementácia tejto verzie ale nebola dosiaľ zverejnená.

Cieľom práce je vytvoriť implementáciu QC-MDPC McEliece kryptosystému nad GF(4).

Úlohy:
1. Oboznámte sa s QC-MDPC McEliece kryptosystémom nad GF(4).
2. Vytvorte implementáciu QC-MDPC McEliece kryptosystému nad GF(4).
3. Otestujte Vašu implementáciu a vyhodnoťte výsledky.

Zoznam odbornej literatúry:

1. M. Baldi, G. Cancellieri, F. Chiaraluce, E. Persichetti and P. Santini, "Using Non-Binary LDPC and MDPC Codes in the McEliece Cryptosystem," 2019 AEIT International Annual Conference (AEIT), Florence, Italy, 2019, pp. 1-6, doi: 10.23919/AEIT.2019.8893339.

Termín odovzdania diplomovej práce:	12. 05. 2023
Dátum schválenia zadania diplomovej práce:	05. 05. 2023
Zadanie diplomovej práce schválil:	prof. Dr. Ing. Miloš Oravec – garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Tomáš Vavro
Diplomová práca:	Implementácia QC-MDPC kryptosystému nad GF(4)
Vedúci záverečnej práce:	Mgr. Tomáš Fabšič, PhD.
Miesto a rok predloženia práce:	Bratislava 2023

V tejto práci sa zaoberáme QC-MDPC kryptosystémom nad $GF(4)$, ktorého použitie bolo navrhnuté autormi Baldi a spol. v roku 2019. Autori zároveň definujú aj základný symbol flipping (SF) dekodér, ktorý v každej iterácii preklápa jeden symbol chybového vektora. Existuje niekoľko dekodérov, ktoré sa dajú použiť na dekódovanie binárnych QC-MDPC kódov pri dešifrovaní v McEliecovom kryptosystéme, a ktoré v každej iterácii preklápajú viacero bitov. Tieto dekodéry sme adaptovali na použitie v QC-MDPC McEliecovom kryptosystéme nad $GF(4)$. Dekodéry, ktoré sme navrhli, a základný SF dekodér sme implementovali v jazyku C a vyhodnotili ich pravdepodobnosť zlyhania dekódovania (DFR) a rýchlosť z hľadiska počtu iterácií. Všetky dekodéry, ktoré sme navrhli, dosiahli zrýchlenie dekódovania oproti základnému SF dekodéru. Navyše, pre vhodné nastavenia parametrov sú DFR niektorých z týchto dekodérov porovnateľné s DFR základného SF dekodéra.

Kľúčové slová: QC-MDPC nad GF(4), symbol flipping, McEliecov kryptosystém

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Bc. Tomáš Vavro
Master's thesis:	Implementation of the QC-MDPC cryptosystem over GF(4)
Supervisor:	Mgr. Tomáš Fabšič, PhD.
Place and year of submission:	Bratislava 2023

In this work, we study the QC-MDPC cryptosystem over $GF(4)$, which was proposed by Baldi et al. in 2019. The authors also define a basic symbol flipping (SF) decoder that flips one symbol of the error vector in each iteration. There are several decoders that can be used to decode binary QC-MDPC codes in the McEliece cryptosystem, which flip multiple bits in each iteration. We adapted these decoders for use in the QC-MDPC McEliece cryptosystem over $GF(4)$. We implemented these decoders and the basic SF decoder in C and evaluated their decoding failure rate (DFR) and speed in terms of the number of iterations. All of the decoders we proposed achieved faster decoding compared to the basic SF decoder. Moreover, for appropriate parameter settings, the DFR of some of these decoders is comparable to the DFR of the basic SF decoder.

Keywords: QC-MDPC over GF(4), symbol flipping, McEliece cryptosystem

Vyhlásenie autora

Podpísaný Tomáš Vavro čestne vyhlasujem, že som diplomovú prácu Implementácia QC-MDPC kryptosystému nad GF(4) vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci. Uvedenú prácu som vypracoval pod vedením Mgr. Tomáša Fabšiča, PhD.

V Žiline dňa 10.05.2023

.....

Podpis autora

Poďakovanie

Ďakujem vedúcemu práce Mgr. Tomášovi Fabšičovi, PhD. za jeho odbornú pomoc, trpezlivosť a ochotu.
Ďakujem stredisku HPC na Slovenskej technickej univerzite v Bratislave, ktoré je súčasťou Slovenskej infraštruktúry pre vysokovýkonné počítanie (projekt SIVVP, ITMS kód 26230120002, financovaný z prostriedkov Európskeho fondu regionálneho rozvoja, ERDF), za výpočtový čas a dostupné zdroje.

Obsah

Zoznam obrázkov a tabuliek

Zoznam algoritmov

Úvod

V posledných rokoch napreduje vývoj v oblasti kvantovej výpočtovej techniky. Hoc kvantové počítače so sebou prinesú rozmach v mnohých vedných disciplínach, predstavujú nevídanú hrozbu pre súčasné kryptografické štandardy. Je známy algoritmus pre kvantové počítače, ktorý dokáže efektívne riešiť problém rozkladu na prvočísla či problém diskrétneho logaritmu. Ide o Shorov algoritmus. [1]

Nakoľko bezpečnosť mnohých asymetrických kryptosystémov, ktoré sú aktuálnymi štandardami, závisí na predpoklade, že uvedené problémy nemožno riešiť efektívne, tieto kryptosystémy sa považujú za zlomené, ak uvažujeme útočníka s dostatočne výkonným kvantovým počítačom. Z toho dôvodu existuje potreba štandardizácie nových kryptosystémov, ktoré budú voči týmto zariadeniam odolné.

Americký inštitút NIST preto v roku 2016 vyhlásil súťaž Post-Quantum Cryptography Standardization [2], ktorá si kladie za cieľ vybrať vhodných kandidátov na štandardizáciu. Do tejto súťaže bolo zapojených mnoho návrhov kryptosystémov, z ktorých niektoré podľahli kryptoanalýze. Tri kryptosystémy, ktoré sú zapojené do súťaže v aktuálnom - štvrtom - kole, sú založené na teórii kódovania.

Pravdepodobne najznámejším z nich je McEliecov kryptosystém, ktorý predstavil v roku 1978 R. McEliece. [3] Tento asymetrický kryptosystém je postavený nad Goppa kódmi. Jeho výhodou je, že obstál v teste časom. Je známy pomerne dlho a napriek tomu odolal kryptoanalýze. Problémom však je, že kľúče v tomto kryptosystéme sú veľmi veľké.

Na riešenie tohto problému boli navrhnuté nové varianty McEliecovho kryptosystému, ktoré sú založené QC-MDPC kódoch. [4] Kandidát s najpriaznivejšou veľkosťou kľúčov v aktuálnom kole súťaže inštitútu NIST je BIKE. [5] Vo svojich skorších verziách bol založený práve na binárnom QC-MDPC McEliecovom kryptosystéme. V aktuálnej verzii používa Niederreiterov kryptosystém nad binárnymi QC-MDPC kódmi, ktorý je mu príbuzný. [6]

Vďaka špecifickej štruktúre QC-MDPC kódov je možné zmenšiť veľkosť používaných kľúčov v QC-MDPC McEliecovom kryptosystéme, no zároveň existuje nenulová pravdepodobnosť, že zlyhá dešifrovanie. To je totiž závislé na použití konkrétneho dekodéra pre binárne QC-MDPC kódy. Bežne používané dekodéry sú založené na bit-flipping (BF) dekodéri, ktorý vo svojej dizertačnej práci navrhol R. Gallager. [7] Sú iteratívne a svoju činnosť vykonávajú buď dovtedy, kým sa im nepodarí úspešne dekódovať vstupnú správu, alebo kým neuplynie maximálny počet iterácií. V takom prípade dekódovanie zlyhalo.

Tento fakt zneužíva GJS útok, ktorý meria pravdepodobnosti zlyhania dekódovania, a na základe týchto údajov je schopný zrekonštruovať použitý súkromný kľúč. [8] Z toho dôvodu

je žiadúce používať také dekodéry, ktoré zlyhávajú s veľmi malou pravdepodobnosťou.
Zároveň však chceme rýchle dekodéry. Existuje viacero variánt dekodérov založených na BF dekodéri pre binárne QC-MDPC. Niektoré v každej iterácii vykonajú len jednu zmenu v hľadanom chybovom vektore, niektoré ich vykonajú viac. V prvej kapitole tejto práce popisujeme okrem nevyhnutných pojmov z teórie kódovania aj niektoré z týchto variantov. Uvádzame aj dekodér pre QC-MDPC kódy nad GF(4), tzv. symbol flipping (SF) dekodér. [9]
SF dekodér je rozšírením BF dekodéra, a teda tiež ide o iteratívny dekodér. V každej iterácií vykoná len jednu zmenu v chybovom vektore. V druhej kapitole predstavujeme naše návrhy, ako rozšíriť SF dekodér o možnosť v každej iterácii vykonať viac takýchto zmien, podobne ako to docieľujú predstavené varianty BF dekodéra pre binárne QC-MDPC kódy. Základný SF dekodér, ako aj jeho varianty, sme implementovali v podobe knižnice v jazyku C. Vo zvyšku druhej kapitoly popisujeme dizajnové princípy a detaily fungovania tejto knižnice.
V poslednej, tretej kapitole, prezentujeme nastavenia vykonaných experimentov a interpretujeme ich výsledky.

1 Teoretické základy

V tejto kapitole predstavíme základné pojmy z teórie kódovania a QC-MDPC McEliecov kryptosystém. Tiež načrtneme, ako je tento kryptosystém formulovaný nad inými konečnými poľami, než GF(2). Na záver predstavíme existujúce dekodéry pre obe varianty.

1.1 Základné pojmy z teórie kódovania

Definícia 1.1.1 [10] ***supp(v)*** *(z angl. support) označuje množinu pozícií nenulových prvkov vo vektore v, t. j. supp*$(v) = \{i : v_i \neq 0\}$.

Definícia 1.1.2 [11] *w(v) označuje* ***Hammingovu váhu*** *vektora v. Tá je definovaná ako počet jeho nenulových prvkov, t. j.* $w(v) = |supp(v)|$. ***Hammingovu vzdialenosť*** *vektorov u a v rovnakej dĺžky n označujeme d(u, v) a je to počet prvkov, v ktorých sa u a v odlišujú, t. j.* $d(u, v) = |\{i : u_i \neq v_i\}|$.

V nasledujúcom texte budeme operovať s pojmami ako kód či kódové slovo. Nech K je nejaké konečné pole a n je kladné celé číslo. Kódom nazývame ľubovoľnú množinu vektorov $C \subseteq K^n$ a kódovými slovami jednotlivé vektory z tejto množiny. Nulové kódové slovo je nulový vektor.

Takáto definícia kódu je však veľmi všeobecná. Zväčša budeme pracovať s konkrétnymi typmi kódov, ktoré majú dodatočné vlastnosti. Jedným takým typom sú lineárne kódy, ktoré definujeme neskôr v tejto kapitole.

Pred tým však ešte uvedieme, že pri kódoch môžeme zisťovať, akú majú minimálnu vzdialenosť. Minimálna vzdialenosť kódu C (značená $d(C)$) je definovaná ako najmenšia Hammingova vzdialenosť spomedzi vzdialeností všetkých párov kódových slov, t. j. $d(C) = min\{d(u, v) : u, v \in C, u \neq v\}$. Minimálna vzdialenosť ovplyvňuje, aké množstvo chýb pri prenose vie daný kód detegovať, resp. opraviť. Kód vie detegovať $s \leq d(C) - 1$ chýb alebo opraviť $t \leq \lfloor \frac{d(C)-1}{2} \rfloor$ chýb [11].

Definícia 1.1.3 [10] *Binárnu operáciu* $\star : (K^n, K^n) \mapsto K^n$ *definovanú predpisom* $u \star v = (u_0v_0, u_1v_1, \ldots, u_{n-1}v_{n-1})$ *nazývame* ***Schurov súčin*** *(z angl. Schur product).*

Príklad 1.1.1 *Majme vektory* $u = (0, 1, 1, 0, 0)$ *a* $v = (1, 0, 1, 0, 1)$ *nad GF(2). Potom* $u \star v$ *je nový vektor s nasledujúcimi prvkami:* $u \star v = (0 \cdot 1, 1 \cdot 0, 1 \cdot 1, 0 \cdot 0, 0 \cdot 1) = (0, 0, 1, 0, 0)$. *Inak povedané, Schurov súčin je súčin dvoch vektorov po zložkách.*

Definícia 1.1.4 [11] *Nech K je konečné pole, nech n je kladné celé číslo a nech* $C \subseteq K^n$. *Hovoríme, že C je* ***(n, k)-lineárny kód****, ak je lineárnym podpriestorom lineárneho priestoru* K^n *dimenzie k.*

Číslo n z definície 1.1.4 nazývame dĺžka lineárneho kódu. Ak sú hodnoty n a k zrejmé z kontextu alebo nie sú potrebné pre pochopenie danej problematiky, vynecháme ich zo značenia lineárneho kódu.
Pre lineárny kód C platí, že jeho minimálna vzdialenosť $d(C)$ sa dá spočítať ako najmenšia Hammingova váha spomedzi váh všetkých jeho nenulových kódových slov, t. j. $d(C) = min\{w(v) : v \in C, v \neq \mathbf{0}\}$. [11]

Definícia 1.1.5 [11] *Nech C je lineárny kód dĺžky n nad konečným poľom K. Matica G typu $k \times n$, ktorej riadky tvoria bázu lineárneho podpriestoru C, sa nazýva* ***generujúca matica*** *kódu C. Matica H typu $(n-k) \times n$ s plnou hodnosťou, pre ktorú platí $GH^T = \boldsymbol{0}$, kde $\boldsymbol{0}$ je nulová matica veľkosti $k \times (n-k)$, sa nazýva* ***kontrolná matica****.*

Ekvivalentne môžeme povedať, že pre kontrolnú maticu platí $HG^T = \mathbf{0}$, pričom v tomto prípade je $\mathbf{0}$ nulová matica typu $(n-k) \times k$. Pomocou generujúcej matice vieme kódovať vektory. To znamená, že vektor $v \in K^k$ zobrazíme na kódové slovo $v' \in K^n$. Urobiť to môžeme pomocou jednoduchého násobenia maticou G: $v' = vG$.
Ak je matica G v blokovom tvare, pričom jej prvý blok je jednotková matica, tak hovoríme, že G je v štandardnom tvare. Ak zakódujeme vektor v ako vektor v' pomocou matice G v štandardnom tvare, tak prvých k pozícií vo vektore v' je totožných s vektorom v. V prípade, že pri prenose vektoru v' nedošlo ku chybám, jeho dekódovanie spočíva v prečítaní prvých k pozícií, čo je veľmi jednoduchá operácia. Navyše, ku matici G v štandardnom tvare vieme pomerne jednoducho nájsť kontrolnú maticu H. Ak je $G = (I|P)$, tak $H = (-P^T|I)$. [11]
Matica H je užitočná, okrem iného, na určenie, či je daný vektor kódovým slovom alebo nie.

Definícia 1.1.6 [11] *Nech K je konečné pole, nech $v \in K^n$ a nech H je kontrolná matica kódu C typu $(n-k) \times n$.* ***Syndróm vektora*** *v je vektor $s = Hv^T$.*

Ekvivalentne môžeme syndróm definovať ako vektor $s = vH^T$. V oboch prípadoch platí, že v je kódové slovo vtedy a len vtedy, ak s je nulový vektor.
Je užitočné si uvedomiť, že riadky matice H sú vlastne kontrolné rovnice, do ktorých môžeme dosadiť prvky vektorov, a overiť ich platnosť. Syndróm vektoru je pri tomto pohľade na maticu H úzko spätý s počtom rovníc, ktorým daný vektor nevyhovuje. Počet nesplnených kontrolných rovníc (značený *upc* z angl. unsatisfied parity checks) vzhľadom na vektor v je rovnaký ako Hammingova váha jeho syndrómu, t. j. $upc = w(s) = w(vH^T)$. Situáciu ilustrujeme na príklade 1.1.2.

Príklad 1.1.2 *Uvažujme konečné pole* $K = GF(4) = GF(2)[x]/(x^2 + x + 1)$. *Koreň polynómu* $(x^2 + x + 1)$ *budeme označovať* α. *Potom* $K = GF(4) = \{0, 1, \alpha, \alpha + 1\}$.
Uvažujme maticu $G = \left(\begin{smallmatrix} 1&0&0&2&0&1 \\ 0&1&0&2&1&0 \\ 0&0&1&0&0&1 \end{smallmatrix}\right)$ *a maticu* $H = \left(\begin{smallmatrix} 2&2&0&1&0&0 \\ 0&1&0&0&1&0 \\ 1&0&1&0&0&1 \end{smallmatrix}\right)$ *nad* $GF(3)$. *Overenie rovnosti* $GH^T = \boldsymbol{0}$ *nechávame na čitateľovi. Nech* $m = (1, 0, 1)$. *Zakódujeme vektor* m: $mG = (1, 0, 1, 2, 0, 2) = m'$. m' *je kódové slovo, a preto má nulový syndróm. Skúsme si však predstaviť, že kódové slovo* m' *prenášame zašumeným komunikačným kanálom, ktorý spôsobí, že prijímateľ správy* m' *v skutočnosti zachytí správu* $m'' = (1, 0, 1, 1, 0, 2)$. *Syndróm tejto správy je* $s = m''H^T = (0, 0, 1)$.
Ak o matici H *uvažujeme ako o trojici kontrolných rovníc so šiestimi premennými* $v_1, \ldots, v_6$, *dostávame homogénnu sústavu:*

$$2v_1 + 2v_2 + v_4 = 0 \tag{1}$$

$$v_2 + v_5 = 0 \tag{2}$$

$$v_1 + v_3 + v_6 = 0 \tag{3}$$

Ak do týchto rovníc dosadíme prvky vektora m'', *zistíme, že rovnice* (1) *a* (2) *sú splnené, ale rovnica* (3) *nie.*

Treba zdôrazniť, že v príklade 1.1.2 sme pre ilustráciu významu syndrómu vyšetrovali počet nesplnených rovníc pre celú správu m''. Dekodéry, ktoré si predstavíme v neskorších kapitolách, však určujú počet nesplnených kontrolných rovníc - *upc* (z angl. unsatisfied parity check) - pre jednotlivé pozície vo vektore m''. K tejto problematike sa vrátime v neskorších kapitolách.

1.2 Cyklické matice

Definícia 1.2.1 [9] *Štvorcová matica* M *typu* $k \times k$ *v tvare* $\begin{pmatrix} a_0 & a_1 & \ldots & a_{k-1} \\ a_{k-1} & a_0 & \ldots & a_{k-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \ldots & a_0 \end{pmatrix}$ *sa nazýva* ***cyklická matica***.

V tejto práci budeme vždy predpokladať, že prvky cyklickej matice pochádzajú z poľa K. Množina cyklických matíc typu $k \times k$ s operáciami sčítania a násobenia matíc tvorí okruh, ktorý je izomorfný s okruhom polynómov $K[x]/(x^k - 1)$ s operáciami sčítania a násobenia polynómov. Ak uvažujeme cyklickú maticu M s prvkami pomenovanými rovnako ako v definícii 1.2.1, tak môžeme tento izomorfizmus zapísať predpisom $M \mapsto m(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1}$. [4]
Význam tohto izomorfizmu spočíva v tom, že namiesto používania maticových operácií vieme použiť tie polynomiálne. To má zmysel napríklad pri hľadaní inverzie cyklickej

matice. Tú vieme vypočítať ako inverziu polynómu, ktorý je k nej izomorfný, modulo $x^k - 1$. Tento výpočet je možné zrealizovať napríklad pomocou rozšíreného Euklidovho algoritmu.

1.3 QC, MDPC a QC-MDPC kódy

Definícia 1.3.1 [10] *Hovoríme, že lineárny kód C je **kvázi-cyklický** (QC), ak existuje taká jeho kontrolná matica H, ktorá je zložená z cyklických blokov.*

Vo všeobecnosti môže platiť, že matica H kvázi-cyklického kódu môže mať $R \times N$ cyklických blokov veľkosti $k \times k$, kde $R = (n-k)/k$ a $N = n/k$, avšak všetky kryptosystémy, ktoré popisuje táto práca, definujú maticu H tak, že sa skladá z 1×2 cyklických blokov veľkosti $k \times k$.

Definícia 1.3.2 [4] *Hovoríme, že (n, k, w)-lineárny kód C je **MDPC** (moderate density parity-check), ak existuje taká kontrolná matica kódu C, ktorej riadky majú konštantnú Hammingovu váhu $w = \mathcal{O}(\sqrt{n \log n})$.*

Ku definícii 1.3.2 musíme urobiť dve poznámky. Po prvé, treba si uvedomiť, že k danému kódu C môže existovať viacero rôznych kontrolných matíc H, pričom nie všetky majú vhodné vlastnosti z hľadiska váhy riadkov. Keď však v ďalších kapitolách použijeme pojem "kontrolná matica MDPC kódu", vždy tým myslíme takú maticu H, ktorá tieto vlastnosti má.

Po druhé, váha riadkov kontrolnej matice MDPC kódu je síce nízka, ale nesmie byť príliš nízka. Kódy s ešte nižšou váhou riadku, než je pri MDPC kódoch, sa nazývajú LDPC (z angl. low-density parity-check) a v súčasnosti sa už nepovažujú za vhodné na použitie v kryptosystémoch, ktorými sa zaoberá táto práca. [10]

Definícia 1.3.3 [10] *Hovoríme, že lineárny kód C je **QC-MDPC**, ak je QC a zároveň aj MDPC.*

Nech H je kontrolná matica QC-MDPC kódu zapísaná v zápise po blokoch nasledujúcim spôsobom: $H = (\, H_0 \mid H_1 \mid \cdots \mid H_{N-1} \,)$. Ďalej nech blok H_{N-1} je regulárna matica. Ku takto definovanej matici H vieme spočítať generujúcu maticu G predpisom

$$G = \left(\begin{array}{c|c} \mathcal{I} & \begin{array}{c} -(H_{N-1}^{-1} \cdot H_0)^T \\ -(H_{N-1}^{-1} \cdot H_1)^T \\ \vdots \\ -(H_{N-1}^{-1} \cdot H_{N-2})^T \end{array} \end{array} \right). \tag{4}$$

Tento predpis sme čerpali z [8].

1.4 QC-MDPC McEliecov kryptosystém

QC-MDPC McEliecov kryptosystém je asymetrický kryptosystém založený na teórii kódovania (z angl. code-based). Má viacero variantov používajúcich rôzny typy kódov. V tejto práci budeme pracovať s QC-MDPC kryptosystémom nad konečným poľom K. Tento kryptosystém bol navrhnutý v [9]. Je zovšeobecnením binárneho QC-MDPC McEliecovho kryptosytému [4], na ktorom bol postavený BIKE-1 (verzia 3.2 BIKE [12]). BIKE je jeden z príspevkov do súťaže inštitútu NIST, ktorá si kladie za cieľ štandardizovať kryptosystémy odolné voči kvantovým počítačom. [2]

Teraz tento kryptosytém predstavíme. Nech matica G je generujúca matica QC-MDPC kódu, ktorý s vysokou pravdepodobnosťou opravuje t chýb, a matica H je jeho kontrolná matica. G bude hrať úlohu verejného a H súkromného kľúča. Generovanie týchto matíc sa bude opierať o izomorfizmus, ktorý sme predstavili v kapitole 1.2.

Najprv vygenerujeme rovnomerne náhodne vektory $h_0, h_1 \in K^k$ s Hammingovou váhou w. Tieto vektory budú reprezentovať koeficienty polynómov $h_0(X)$ a $h_1(X)$ z okruhu $K[X]/(X^k - 1)$. Z týchto polynómov skonštruujeme cyklické bloky H_0 a H_1 matice H. Tým je vygenerovaná matica H.

Príklad 1.4.1 *Uvažujme $k = 5$ a $w = 3$. Vygenerujeme vektory $h_0 = (1, \alpha, 0, 0, \alpha + 1)$ a $h_1 = (0, 1, 0, \alpha, \alpha + 1)$. Tieto vektory reprezentujú koeficienty polynómov $h_0(X) = 1 + \alpha X + (\alpha + 1)X^4$ a $h_1(X) = X + \alpha X^3 + (\alpha + 1)X^4$ v okruhu $GF(4)[X]/(X^5 - 1)$. Polynóm $h_0(X)$ zodpovedá cyklickému bloku $H_0 = \begin{pmatrix} 1 & \alpha & 0 & 0 & \alpha{+}1 \\ \alpha{+}1 & 1 & \alpha & 0 & 0 \\ 0 & \alpha{+}1 & 1 & \alpha & 0 \\ 0 & 0 & \alpha{+}1 & 1 & \alpha \\ \alpha & 0 & 0 & \alpha{+}1 & 1 \end{pmatrix}$ a polynóm $h_1(X)$ zasa cyklickému bloku $H_1 = \begin{pmatrix} 0 & 1 & 0 & \alpha & \alpha{+}1 \\ \alpha{+}1 & 0 & 1 & 0 & \alpha \\ \alpha & \alpha{+}1 & 0 & 1 & 0 \\ 0 & \alpha & \alpha{+}1 & 0 & 1 \\ 1 & 0 & \alpha & \alpha{+}1 & 0 \end{pmatrix}$.*

Maticu G, ktorá jej zodpovedá, spočítame použitím rovnosti (4) v kapitole 1.3. V našom prípade je matica H v tvare $H = (\, H_0 \mid H_1 \,)$. Potom matica G bude:

$$G = \left(\mathcal{I} \,\middle|\, -\left(H_1^{-1} H_0\right)^T \right) \tag{5}$$

Maticu H_1^{-1} spočítame ako polynomiálnu inverziu $h_1^{-1}(X) \pmod{X^k - 1}$ napríklad pomocou rozšíreného Euklidovho algoritmu. Ak H_1 nie je regulárna matica, tak sa generovanie opakuje od začiatku. Proces generovania kľúčov je popísaný v algoritme 1.

Šifrovanie je relatívne jednoduchý proces. Otvorenú správu $m \in K^k$ najprv zakódujeme na kódové slovo $m' = mG$ a potom spôsobíme t chýb v m'. Inak povedané, rovnomerne náhodne vygenerujeme chybový vektor $e \in K^n$ s Hammingovou váhou t a pričítame ho po zložkách ku m'. Tak vznikne zašifrovaná správa c. Tento proces je popísaný v algoritme 2.

Algoritmus 1 Generovanie kľúčov

Vstupy: Parametre k a w, konečné pole K

Výstupy: Kľúče (G, H)

1: $h_0 \xleftarrow{\$} K^k,$ kde $w(h_0) = w$
2: $h_1 \xleftarrow{\$} K^k,$ kde $w(h_1) = w$
3: $h_0(X) \xleftarrow{\text{inicializuj podľa}} h_0$
4: $h_1(X) \xleftarrow{\text{inicializuj podľa}} h_1$
5: **if** $h_1^{-1}(X) \pmod{X^k - 1}$ existuje **then**
6: $\quad g(X) \leftarrow h_1^{-1}(X)h_0(X)$
7: $\quad G' \xleftarrow{\text{inicializuj podľa}} g(X)$
8: $\quad G \leftarrow \left(\mathcal{I} \mid -G'^T \right)$
9: $\quad H_0 \xleftarrow{\text{inicializuj podľa}} h_0(X)$
10: $\quad H_1 \xleftarrow{\text{inicializuj podľa}} h_1(X)$
11: $\quad H \leftarrow \left(H_0 \mid H_1 \right)$
12: $\quad$ **return** (G, H)
13: **else**
14: $\quad$ **goto** 1

Algoritmus 2 Šifrovanie

Vstupy: Otvorená správa $m \in K^k$, parameter t, kľúč G, pole K

Výstupy: Zašifrovaná správa c

1: $e \xleftarrow{\$} K^n,$ kde $w(e) = t$
2: $c \leftarrow mG + e$
3: **return** c

Dešifrovanie zašifrovanej správy c (popísané v algoritme 3) spočíva v použití efektívneho dekodéra pre QC-MDPC kódy, ktorý má za úlohu odhaliť použitý chybový vektor e, resp. odstrániť všetkých t chýb, ktoré boli vytvorené vo vektore m'. Keď sa mu to podarí, stačí prečítať prvých k pozícií z vektora m', a tak získať otvorenú správu m. Dekódovanie však môže byť neúspešné, preto dekodér môže vrátiť chybu dekódovania E namiesto chybového vektora e.

Algoritmus 3 Dešifrovanie

Vstupy: Zašifrovaná správa $c \in K^n$, kľúč H

Výstupy: Otvorená správa m alebo chyba E

1: $m' \leftarrow$ dekódovanie správy c
2: **if** $m' = E$ **then**
3: **return** E
4: **else**
5: $m \xleftarrow{\text{prvých } k \text{ prvkov z}} m'$
6: **return** m

Existujú rôzne dekodéry. Niektoré z nich si popíšeme v nasledujúcich kapitolách. Ešte musíme poznamenať, že doteraz jediný rozdiel medzi binárnym QC-MDPC McEliecovým kryptosystémom a QC-MDPC McEliecovým kryptosystémom nad poľom K bolo práve použité pole. V binárnej verzii bolo $K = GF(2)$. V ďalších kapitolách však už budeme rozlišovať medzi týmito variantami. Binárny variant tohto kryptosystému je viac preskúmaný, a teda preň existuje aj viac dekodérov.

Ešte uvedieme, že pre 80-bitovú bezpečnosť autori [9] odporúčajú 3 možnosti nastavení kryptosystému (viď tabuľka 1). Hodnota $\bar{t}$ v tejto tabuľke označuje minimálnu Hammingovu váhu chybového vektora, pri ktorej je bezpečnosť 80 bitov. Pre všetky uvedené parametre však odporúčajú použiť Hammingovu váhu chybového vektora $t = 84$. Nastavenie pre binárnu variantu kryptosystému autori prevzali z článku [4].

Pole	w	k	$\bar{t}$
$GF(2)$	45	4801	84
$GF(4)$	37	2339	71
$GF(8)$	37	1583	68

Tabuľka 1: Odporúčané nastavenia kryptosystému

1.5 Bit-flipping dekodér

Mnohé z dekodérov, ktoré sa používajú na dekódovanie QC-MDPC kódov sú založené na dizertačnej práci R. G. Gallagera o LDPC kódoch. [7] Bit-flipping (BF) dekodér, ktorý prezentujeme v tejto kapitole, nie je výnimkou. Viacero variánt BF dekodéra na použitie pri dekódovaní QC-MDPC kódov bolo navrhnutých v [4]. V tejto kapitole preberáme popis algoritmu podľa [9].

BF dekodér (algoritmus 4) pre binárne QC-MDPC kódy je iteratívny dekodér. Svoju činnosť začína inicializovaním hľadaného chybového vektora e ako nulový vektor a spočíta syndróm správy c ako $s = cH^T$. Potom v každej iterácii vyhodnotí počet nesplnených kontrolných rovníc, do ktorých prispievajú jednotlivé bity správy. Túto hodnotu budeme pre j-ty bit správy označovať upc_j a môžeme ju jednoducho spočítať ako Hammingovu váhu súčinu syndrómu a j-teho stĺpca matice H po zložkách, t. j. $upc_j = w(s \star H_{*,j})$, kde $H_{*,j}$ označuje j-ty stĺpec matice H[1]. Dekodér nájde taký index j, pre ktorý je spočítaná hodnota najvyššia. Na tejto pozícii preklopí bit chybového vektora a aktualizuje syndróm $s = s + H_{*,j}$. Tu si treba uvedomiť, že výpočet syndrómu v $GF(2)$ je vlastne sčítanie tých stĺpcov matice H, kde má správa c jednotky. Preto na jeho aktualizáciu stačí pričítať stĺpec, ktorý je na tej pozícii, kde bol preklopený bit chybového vektora.

Uvedený algoritmus v každej iterácii preklopí len jeden bit. To znamená, že pri Hammingovej váhe chybového vektora $t = 84$ potrvá úspešné dekódovanie aspoň 84 iterácií. Zároveň treba povedať, že nie je garantované, že algoritmus úspešne dekóduje zašifrovanú správu v priebehu presne 84 iterácií: v niektorých iteráciách mohol byť preklopený bit, ktorý síce dosahoval najvyššiu hodnotu upc_j v tej-ktorej iterácii, no v skutočnosti nešlo o bit, ktorý by bol nastavený v chybovom vektore použitom pri šifrovaní. Nie je dokonca ani garantované, že tento dekodér bude v dekódovaní úspešný. Môže sa stať, že po uplynutí maximálneho počtu iterácií nebude syndróm správy (ktorý sme postupne aktualizovali v priebehu dekódovania) nulový. V takom prípade hovoríme o chybe dekódovania. Toto nie je situácia, ktorá nastáva len pri BF dekodéri. Všetky dekodéry, ktoré v tejto práci uvedieme, môžu pri dekódovaní zlyhať.

Pravdepodobnosť, s akou nastáva chyba dekódovania, budeme značiť skratkou DFR (z angl. decoding failure rate). DFR závisí nielen od použitého dekodéru, ale aj od nastavení

[1] Čitateľ si isto všimne, že sme trochu zvoľnili zápis vektorov. Ak sme počítali syndróm pomocou vzorca $s = Hc^T$, tak ide o stĺpcový vektor dĺžky k. Naopak, ak sme ho počítali ako $s = cH^T$, tak s je riadkový vektor dĺžky k. Striktne vzaté, v druhom prípade nie je Schurov súčin vo výraze $upc_j = w(s \star H_{*,j})$ definovaný. Pre tento prípad implicitne uvažjeme výpočet pomocou vzorca $upc_j = w(s^T \star H_{*,j})$. Podobný predpoklad platí aj pre zvyšok kapitoly.

kryptosystému. Predstavme si situáciu, že pri šifrovaní generujeme chybový vektor tak, že je jeho Hammingova váha vyššia, než je predpokladaný počet chýb, ktoré dokáže opraviť zvolený kód C. V takomto prípade je rozumné predpokladať, že dekódovanie bude zlyhávať častejšie.

Algoritmus 4 Bit-flipping dekodér

Vstupy: Zašifrovaná správa $c \in K^n$, kľúč H, maximálny počet iterácií $ITER$

Výstupy: Chybový vektor e alebo chyba dekódovania E

1: $e \leftarrow \mathbf{0}$
2: $s \leftarrow cH^T$
3: **for** $i \leftarrow 0..ITER - 1$ **do**
4: $\quad upc_{max} \leftarrow -1$
5: $\quad pos \leftarrow -1$
6: $\quad$ **for** $j \leftarrow 0..n - 1$ **do**
7: $\quad\quad upc_j \leftarrow w(s \star H_{*,j})$
8: $\quad\quad$ **if** $upc_j > upc_{max}$ **then**
9: $\quad\quad\quad upc_{max} \leftarrow upc_j$
10: $\quad\quad\quad pos \leftarrow j$
11: $\quad s \leftarrow s + H_{*,pos}$
12: $\quad e_{pos} \leftarrow e_{pos} + 1$
13: **if** $s = \mathbf{0}$ **then**
14: $\quad$ **return** e
15: **else**
16: $\quad$ **return** E

DFR je jedna z metrík, ktoré sa používajú na vyhodnotenie kvality dekodéra. Vo všeobecnosti platí, že dekodér, ktorý zlyháva menej často, je považovaný za lepší, než ten, ktorý zlyháva častejšie. Ak zlyhá dekódovanie, znamená to, že zlyhalo celé dešifrovanie. Hypotetický protokol, ktorý by bol založený na tomto kryptosystéme, by musel na tento fakt reagovať, napr. žiadosťou o opätovné poslanie správy. Potenciálny útočník by mohol úmyselne posielať správy šifrované s použitím vhodne konštruovaných chybových vektorov a zaznamenávať si zlyhania. Zo štatistickej analýzy týchto údajov by potom mohol zistiť niektoré vlastnosti matice H, ba dokonca ju kompletne zrekonštruovať. Na tomto princípe je založený tzv. GJS útok (pomenovaný podľa autorov Guo, Johansson, Stankovski). [8] Na to, aby bol takýto útok úspešný, musí sa útočníkovi podariť zachytiť viacero zlyhaní dekódovania, resp. dešifrovania. Čím nižšia je DFR, tým viac správ musí útočník poslať.

Pre dostatočne nízku hodnotu DFR by bol tento počet tak vysoký, že GJS útok by bol prakticky nerealizovateľný.

1.6 Modifikácie BF dekodéra

Jedným z problémov algoritmu 4, ktorý sme uviedli v kapitole 1.5 je, že preklápa v jednej iterácii len jediný bit. Existuje však viacero modifikácií BF dekodéra, ktoré sa zameriavajú na preklápanie viacerých bitov v každej iterácii. Ako prvý si predstavíme variant, ktorý navrhli autori [4]. Spočíva v tom, že sa v každej iterácii určí maximálna hodnota $upc_{max} = \max\{upc_j : 0 \leq j < n\}$. Potom sa preklopia všetky bity, pre ktoré platí $upc_j \geq upc_{max} - \delta$, kde δ je malé nezáporné celé číslo. Celý postup je uvedený v algoritme 5.

Algoritmus 5 Bit-flipping dekodér s použitím parametra δ

Vstupy: Zašifrovaná správa $c \in K^n$, kľúč H, maximálny počet iterácií $ITER$, parameter δ

Výstupy: Chybový vektor e alebo chyba dekódovania E

1: $e \leftarrow \mathbf{0}$
2: $s \leftarrow cH^T$
3: $upc \leftarrow \mathbf{0}$
4: **for** $i \leftarrow 0..ITER - 1$ **do**
5: $\quad upc_{max} \leftarrow -1$
6: $\quad$ **for** $j \leftarrow 0..n - 1$ **do**
7: $\quad\quad upc_j \leftarrow w(s \star H_{*,j})$
8: $\quad\quad$ **if** $upc_j > upc_{max}$ **then**
9: $\quad\quad\quad upc_{max} \leftarrow upc_j$
10: $\quad$ **for** $j \leftarrow 0..n - 1$ **do**
11: $\quad\quad$ **if** $upc_j >= upc_{max} - \delta$ **then**
12: $\quad\quad\quad e_j \leftarrow e_j + 1$
13: $\quad s \leftarrow (c + e)H^T$
14: **if** $s = \mathbf{0}$ **then**
15: $\quad$ **return** e
16: **else**
17: $\quad$ **return** E

Druhou možnosťou je upraviť algoritmus 4 tak, aby v každej iterácii preklápal všetky bity, ktorých hodnota upc_j presahuje vopred zvolenú prahovú hodnotu T (z angl. threshold).

[7] Ak sa prahová hodnota počíta podľa aktuálnej váhy syndrómu (nie je predpočítaná), nazývame ju adaptívna prahová hodnota (z angl. adaptive threshold). [13] [10] Keďže ide o funkciu syndrómu, označíme ju $T(s)$. Tento dekodér je popísaný v algoritme 6.

Algoritmus 6 Bit-flipping dekodér s prahovou hodnotou

Vstupy: Zašifrovaná správa $c \in K^n$, kľúč H, maximálny počet iterácií $ITER$

Výstupy: Chybový vektor e alebo chyba dekódovania E

1: $e \leftarrow \mathbf{0}$
2: $s \leftarrow cH^T$
3: **for** $i \leftarrow 0..ITER - 1$ **do**
4: **for** $j \leftarrow 0..n - 1$ **do**
5: $upc_j \leftarrow w(s \star H_{*,j})$
6: **if** $upc_j > T(s)$ **then**
7: $e_{pos} \leftarrow e_{pos} + 1$
8: $s \leftarrow (c + e)H^T$
9: **if** $s = \mathbf{0}$ **then**
10: **return** e
11: **else**
12: **return** E

Čitateľ si môže všimnúť, že algoritmy 5 a 6 aktualizujú syndróm až po preklopení všetkých bitov. Syndróm by bolo možné aktualizovať okamžite po preklopení pozície v chybovom vektore. Takéto modifikácie by pravdepodobne mali vplyv na DFR dekodérov. [10]
Nakoľko sa v algoritmoch 5 a 6 preklápa viacej bitov naraz, je možné, že sa vyskytne aj viacej chýb v rámci jednej iterácie. Z toho dôvodu môžu byť DFR týchto dekodérov horšie, než v prípade BF.
Ešte poznamenáme, že správne nastavenie parametra δ, resp. predpis funkcie $T(s)$, je kľúčové. V prípade, že parameter δ je príliš nízky, resp. výstupy funkcie $T(s)$ sú príliš vysoké, dekódovanie potrvá dlho. Príliš vysoké hodnoty δ, resp. nízke výstupy funkcie $T(s)$, by zasa znamenali preklápanie príliš veľkého množstva bitov, čím sa ešte navýši pravdepodobnosť preklopenia nesprávnych bitov a vo výsledku aj neúspechu dekódovania.

1.7 Symbol flipping dekodér

Autori [9] navrhujú relatívne priamočiare rozšírenie algoritmu 4 na dekódovanie QC-MDPC kódov nad poľom K. Jednoduché preklápanie bitov ako pri BF dekodéri už nie je možné, lebo chybový vektor mohol obsahovať aj iné hodnoty než 0 a 1.

Tento dekodér je pomenovaný symbol-flipping (SF). Funguje tak, že v každej iterácii otestuje pre všetky pozície v chybovom vektore všetky nenulové prvky poľa K a vyhodnotí, pre ktoré nastavenie najviac poklesla váha syndrómu, t. j. najviac sa zvýšil počet platných rovníc. Tento proces je popísaný v algoritme 7.

Algoritmus 7 Symbol flipping dekodér

Vstupy: Zašifrovaná správa $c \in K^n$, kľúč H, maximálny počet iterácií $ITER$

Výstupy: Chybový vektor e alebo chyba dekódovania E

1: $e \leftarrow \mathbf{0}$
2: $s \leftarrow cH^T$
3: **for** $i \leftarrow 0..ITER - 1$ **do**
4: **if** $s = \mathbf{0}$ **then**
5: **return** e
6: $\sigma_{max} \leftarrow -1$
7: $a_{max} \leftarrow 0$
8: $pos \leftarrow -1$
9: **for** $j \leftarrow 0..n - 1$ **do**
10: **for** $\forall a \in K$, kde $a \neq 0$ **do**
11: $\sigma_j \leftarrow w(s) - w(s - aH_{*,j})$
12: **if** $\sigma_j > \sigma_{max}$ **then**
13: $\sigma_{max} \leftarrow \sigma_j$
14: $a_{max} \leftarrow a$
15: $pos \leftarrow j$
16: $s \leftarrow s - a_{max}H_{*,pos}$
17: $e_{pos} \leftarrow e_{pos} + a_{max}$
18: **if** $s = \mathbf{0}$ **then**
19: **return** e
20: **else**
21: **return** E

Podobne, ako BF dekodér (algoritmus 4), aj tento dekodér dokáže preklopiť len jeden symbol za iteráciu. Preto aj v tomto prípade platí, že pri Hammingovej váhe chybového vektora $t = 84$ potrvá dekódovanie aspoň 84 iterácií. Hlavným cieľom tejto práce je pokus o rozšírenie tohto dekodéra o preklápanie viacerých symbolov v jednej iterácii.

2 Návrh riešenia

V tejto kapitole najprv predstavíme úpravy SF dekodéra, ktoré navrhujeme, a potom vysvetlíme náležité implementačné detaily týkajúce sa našej knižnice.

2.1 Návrh dekodéra SF s použitím parametra δ

Rozšírenie základného SF dekodéra tak, aby preklápal viac symbolov v jednej iterácii sa dá urobiť obdobne, ako je tomu v algoritme 5. Pre každú pozíciu a každý nenulový prvok v poli $GF(4)$ vypočítame σ_j. Do pomocného poľa *sigmas* na pozícii j si uložíme maximálnu hodnotu σ_j, ktorá bola dosiahnutá pre j-tu pozíciu chybového vektora a do pomocného poľa *values* si uložíme prvok, pre ktorý bola dosiahnutá. Súbežne s tým hľadáme aj maximálnu hodnotu σ_j naprieč všetkými pozíciami j, značenú σ_{max}.

Po tom, ako ukončíme iteráciu cez všetky pozície chybového vektora, iterujeme cez ne ešte raz, no tentoraz preklápame tie pozície, pre ktoré $sigmas_j \geq b$, kde b je prahová hodnota, ktorá je v každej iterácii počítaná ako $b = max\{\sigma_{max} - \delta, 0\}$. Tým sa obmedzujeme na preklápanie výhradne tých pozícií, kde sme dosiahli nezáporné hodnoty σ_j. To znamená, že vyžadujeme, aby preklopenie symbolu nezvýšilo počet *upc*. Dekodér je popísaný v algoritme 8.

Ako môžeme vidieť, algoritmus 8 aktualizuje hodnotu syndrómu hneď po preklopení symbolu. V tomto prípade by sa to však nemalo prejaviť na DFR, nakoľko aktualizovaná hodnota sa použije až v ďalšej iterácií pri výpočte nových hodnôt σ_j.

Algoritmus 8 SF dekodér s použitím parametra δ

Vstupy: Zašifrovaná správa $c \in GF(4)^n$, súkromný kľúč H, maximálny počet iterácií $ITER$, parameter δ

Výstupy: Chybový vektor e alebo chyba dekódovania E

$e \leftarrow \mathbf{0}$
$s \leftarrow cH^T$
for $i \leftarrow 0..ITER-1$ **do**
    if $s = \mathbf{0}$ **then**
        return e
    $\sigma_{max} \leftarrow -1$
    $sigmas \leftarrow \mathbf{-1}$
    $values \leftarrow \mathbf{0}$
    for $j \leftarrow 0..n-1$ **do**
        for $\forall a \in GF(4) \setminus \{0\}$ **do**
            $\sigma_j \leftarrow w(s) - w(s - aH_{*,j})$
            if $\sigma_j > \sigma_{max}$ **then**
                $\sigma_{max} \leftarrow \sigma_j$
            if $\sigma_j > sigmas_j$ **then**
                $sigmas_j \leftarrow \sigma_j$
                $values_j \leftarrow a$
    $b \geq max\{\sigma_{max} - \delta, 0\}$
    for $j \leftarrow 0..n-1$ **do**
        if $sigmas_j \geq b$ **then**
            $s \leftarrow s - values_j * H_{*,j}$
            $e \leftarrow e_j + values_j$
if $s = \mathbf{0}$ **then**
    return e
else
    return E

2.2 Návrh dekodéra SF s použitím prahovej hodnoty $T(s)$

Algoritmus 9 SF dekodér s prahovou hodnotou

Vstupy: Zašifrovaná správa $c \in GF(4)^n$, súkromný kľúč H, maximálny počet iterácií $ITER$, parameter δ

Výstupy: Chybový vektor e alebo chyba dekódovania E

1: $e \leftarrow \mathbf{0}$
2: $s \leftarrow cH^T$
3: **for** $i \leftarrow 0..ITER-1$ **do**
4: **if** $s = \mathbf{0}$ **then**
5: **return** e
6: $thr \leftarrow T(s)$
7: **for** $j \leftarrow 0..n-1$ **do**
8: $\sigma_{max} \leftarrow -1$
9: $a_{max} \leftarrow 0$
10: **for** $\forall a \in GF(4) \setminus \{0\}$ **do**
11: $\sigma_j \leftarrow w(s) - w(s - aH_{*,j})$
12: **if** $\sigma_j > \sigma_{max}$ **then**
13: $\sigma_{max} \leftarrow \sigma_j$
14: $a_{max} \leftarrow a$
15: **if** $\sigma_{max} > thr$ **then**
16: $e \leftarrow e + a_{max}$
17: $s \leftarrow (c+e)H^T$
18: **if** $s = \mathbf{0}$ **then**
19: **return** e
20: **else**
21: **return** E

Podobne, ako v prípade SF dekodéra využívajúceho parameter δ, aj koncept adaptívnych prahových hodnôt sa dá jednoducho prispôsobiť na využitie v SF dekodéri. Na začiatku iterácie spočítame prahovú hodnotu $thr = T(s)$ podľa aktuálnej hodnoty syndrómu. Pre každú pozíciu j najprv určíme hodnoty σ_j pre všetky tri nenulové symboly z $GF(4)$ a potom z nich vyberieme najvyššiu.

Ak je nájdená hodnota väčšia, než prahová hodnota thr, preklopíme pozíciu j v chybovom vektore s použitím symbolu, pre ktorý bola dosiahnutá najvyššia σ_j. Narozdiel od prípadu

algoritmu 8, tu nie je možné aktualizovať syndróm hneď po preklopení, pretože na riadku 11 v algoritme 9 sa na ďalšej pozícii j bude opäť počítať σ_j. Ak by došlo k aktualizácii syndrómu, táto hodnota by sa počítala vzhľadom na novú váhu syndrómu, avšak k aktualizácii prahovej hodnoty thr by došlo až na začiatku ďalšej iterácie. Kvôli tomu by porovnanie na riadku 15 pravdepodobne nebolo validné.
Výpočtu prahu $thr = T(s)$ sa budeme detailne venovať v kapitole 3.6. Na tomto mieste sa však patrí uviesť, že v našom návrhu $T(s)$ nenadobúda záporné hodnoty. Vďaka tomu máme garanciu, že preklopenie symbolu nezvýši váhu syndrómu podobne, ako tomu bolo v prípade SF dekodéra s použitím parametra δ (algoritmus 8).

2.3 Implementácia

Implementovali sme knižnicu na prácu s McEliecovým kryptosystémom nad $GF(4)$ v jazyku C v štandarde C11. Táto knižnica nemá žiadne závislosti na knižniciach tretích strán (s výnimkou štandardnej knižnice jazyka C). Používa build systém CMake.
Definovali sme niekoľko možných kompilačných módov:

- `Debug` na kompiláciu s ladiacimi informáciami a aktívnymi kontrolami validity hodnôt argumentov vo volaniach jednotlivých funkcií.
- `Release` na kompiláciu bez ladiacich informácií a dodatočných kontrol. Tento mód sa pri kompilácii použije ako predvolený, ak používateľ neurčí žiadny iný.
- `Testing` na spustenie unit testov, navyše sa aktivujú ladiace informácie a dodatočné kontroly
- Iné módy slúžia na spúšťanie jednotlivých experimentov popísaných v neskorších kapitolách tejto práce. Koncový používateľ knižnice nebude ostatné módy kompilácie používať. V aktuálnej verzii sú v tejto kategórii módy `Weights` a `Iterations`.

Kompilácia samotná sa v OS Linux s príkazovým riadkom Bash dá vykonať nasledujúcou sekvenciou príkazov:

```
mkdir build # vytvorenie priecinku na kompilaciu
cd build    # presun do tohto priecinku
cmake -DCMAKE_BUILD_TYPE=Mode .. # vygenerovanie kompilacnych skriptov
cmake --build . # kompilacia
./mdpc-gf4 # spustenie
```

`Mode` je potrebné nahradiť jedným z uvedených kompilačných módov.

Knižnica je k dispozícii verejne prostredníctvom platformy GitHub a má relatívne prívetivú open-source licenciu (zvolili sme GNU GPL v3), aby bolo možné zdrojový kód auditovať, nájsť prípadné chyby a prispievať vylepšenia. Zároveň sme však docielili, že všetky rozšírenia knižnice musia tiež byť open-source.

Z hľadiska dizajnu bola knižnica navrhnutá tak, aby spĺňala niekoľko zásad. Po prvé, cieľom bolo poskytnúť čo najväčšiu rýchlosť v móde `Release`, preto sú v ňom vypnuté niektoré dodatočné kontroly. Po druhé, väčšina funkcií, ktoré potrebujú alokovať pamäť na svoj výstup, očakáva, že pamäť bude alokovaná vopred v dostatočnej kapacite a že ju dostanú ako jeden zo vstupných argumentov. Funkcie, ktoré toto pravidlo nemôžu dodržať, o tom používateľa informujú v dokumentácii. Toto pravidlo nedodržujú napr. inicializačné funkcie, ktorých primárnou úlohou je práve správna alokácia pamäte pre dané dátové štruktúry. Pre všetky funkcie s výnimkou tých, ktoré majú na starosti inicializáciu, však platí, že ak funkcia alokuje pamäť, sama ju aj uvoľní, keď už ju viac nepotrebuje. Ku každej inicializačnej funkcii existuje korešpondujúca deinicializačná funkcia, aby sme zaistili, že používateľ správne uvoľní používanú pamäť. Deinicializačné funkcie sú vždy referencované v dokumentácii ku inicializačným funkciám.

Hlavným zámerom tohto návrhu bolo obmedzenie množstva alokácií. K tejto téme sa ešte vrátime v tejto kapitole, najprv je však potrebné povedať, aké dátové štruktúry knižnica definuje. Prvky poľa $GF(4)$ sú reprezentované typom `gf4_t`. Prvky $GF(4)$ sú vlastne polynómy s koeficientami z $GF(2)$, takže ich je možné reprezentovať ako postupnosti bitov, kde je prvý bit sprava nastavený, ak je absolútny člen nenulový, a druhý bit sprava je nastavený, ak je koeficient pri α nenulový. Prvok 0 teda reprezentujeme ako 00, prvok 1 ako 01, prvok α ako 10 a prvok $\alpha + 1$ ako 11. Obdobný typ reprezentácie je možné použiť aj na iné polia typu $GF(2^n)$, kde n je kladné celé číslo.

Pri tejto reprezentácii je sčítanie (aj odčítanie) možné vykonať ako exkluzívny logický súčet po bitoch. Operácie súčinu a delenia vykonávame pomocou predvypočítaných Cayleyho tabuliek. Tieto tabuľky sú alokované staticky.

Polynómy nad $GF(4)$ reprezentujeme pomocou štruktúry `gf4_poly_t`. Obsahuje pole koeficientov typu `gf4_t`, kapacitu tohto poľa a stupeň polynómu. Koeficienty sú ukladané zľava doprava, tzn. koeficient pri absolútnom člene je prvou položkou v poli. Dátový typ `gf4_poly_t` aj ako všeobecné vektory nad $GF(4)$. V takom prípade sa zväčša ignoruje nastavenie stupňa polynómu, pretože ide o časovo náročnú operáciu. Tento fakt má dôležité následky. Na takto používaných inštanciách `gf4_poly_t` nie je možné použiť operácie, ktoré očakávajú na svojom vstupe polynómy. Ak si používateľ praje manipulovať s vektormi ako s polynómami, musí použiť funkciu na nastavenie správneho stupňa

(`gf4_poly_adjust_degree`).

Funkcie, ktoré so štruktúrami typu `gf4_poly_t` manipulujú ako s vektormi nad $GF(4)$, to uvádzajú vo svojej dokumentácii. Ak je výstupom nejakej funkcie vektor, no funkcia napriek tomu nastavila jeho stupeň, je to uvedené v jej dokumentácii.

Nad polynómami sme definovali obvyklé operácie ako sčítanie, násobenie, delenie a zvyšok po delení. Tiež sme implementvali inverziu polynómu pri polynomiálnom module (pomocou rozšíreného Euklidovho algoritmu). V móde `Debug` je možné, aby sa pole koeficientov zväčšilo, ak je to potrebné. To je však v rozpore s našimi požiadavkami pre mód `Release`. V tomto móde operácie očakávajú dostatočnú kapacitu alokovanej pamäte. Tento dizajn sa môže zdať obmedzujúci, v praxi tomu však nie je. Našim cieľom nebolo implementovať všeobecnú knižnicu na prácu s polynómami, iba nevyhnutné minimum potrebné na prácu s QC-MDPC McEliecovým kryptosystémom nad $GF(4)$. Pri tomto použití sú veľkosti všetkých polynómov vopred známe a odvíjajú sa od veľkosti cyklických blokov použitých matíc. Preto je možné alokovať ich vopred s dostatočnou kapacitou.

Pri generovaní kľúčov sa musí počítať inverzia polynómu pri danom module. Funkcia, ktorá túto inverziu počíta, implementuje klasický rozšírený Euklidov algoritmus. V prvých verziách knižnice sa pri každej iterácií algoritmu vytvárali (alokovali) nové polynómy na ukladanie medzikrokov výpočtu inverzie. Tento prístup bol veľmi pomalý a spôsoboval, že generovanie kľúčov mohlo trvať rádovo minúty. V aktuálnej implementácii je výpočet inverzie implementovaný tak, aby vykonal konštantný počet alokácií a znovupoužíval pamäť, ktorú v danom momente nepotrebuje na iný účel.

Pri generovaní kľúčov a pri šifrovaní je potrebné generovať náhodné vektory nad $GF(4)$. Preto sme implementovali funkcie na vytvorenie náhodných vektorov nad $GF(4)$ s danou Hammingovou váhou aj bez špecifikovania Hammingovej váhy. V oboch prípadoch sa na generovanie náhodných čísel používa funkcia štandardnej knižnice *rand*. Počiatočná hodnota sa nastavuje funkciou *srand* podľa aktuálneho času. Vygenerované vektory sa ukladajú ako inštancie `gf4_poly_t`. Narozdiel od iných funkcií, ktoré pristupujú ku inštanciám `gf4_poly_t` ako ku vektorom, funkcie generujúce náhodné vektory vždy korektne nastavia stupeň polynómu. Vďaka tomu je možné použiť ich výstupy ako operandy polynomiálnych operácií.

Myslíme si, že je podstatné uviesť, ako presne sa náhodné vektory generujú. Generovanie vektora dĺžky s bez požiadavky na jeho konkrétnu Hammingovu váhu je priamočiare a popisuje ho algoritmus 10. Generovanie náhodného vektora dĺžky s s danou Hammingovou váhou b sa vykonáva tak, že jeho prvých b pozícií sa náhodne nastaví na nenulové prvky z $GF(4)$ a zvyšných $s - b$ pozícií sa nastaví na nuly. Potom sa vykoná premiešanie pomocou

Fisher-Yatesovho algoritmu [14] (resp. Knuthovho algoritmu P z [15]). Uvedený proces je zachytený v algoritme 11.

Algoritmus 10 Generovanie náhodného vektora

Vstupy: Alokovaný výstupný vektor v, počet položiek s

1: $deg \leftarrow 0$
2: **for** $i \leftarrow 0..s-1$ **do**
3: $\quad v_i \xleftarrow{\$} GF(4)$
4: $\quad$ **if** $v_i \neq 0$ **then**
5: $\quad\quad deg \leftarrow i$

Algoritmus 11 Generovanie náhodného vektora s danou Hammingovou váhou

Vstupy: Alokovaný výstupný vektor v, počet položiek s, Hammingova váha b

1: **for** $i \leftarrow 0..b-1$ **do**
2: $\quad v_i \xleftarrow{\$} GF(4) \setminus \{0\}$
3: **for** $i \leftarrow b..s-1$ **do**
4: $\quad v_i \leftarrow 0$
5: **for** $i \leftarrow 0..s-2$ **do** ▷ Premiešanie pomocou Fisher-Yatesovho algoritmu.
6: $\quad j \xleftarrow{\$} \{u : i \leq u \leq size-1\}$
7: $\quad v_i, v_j \leftarrow v_j, v_i$
8: $deg \leftarrow 0$
9: **for** $i \leftarrow 0..s-1$ **do**
10: $\quad$ **if** $v_i \neq 0$ **then**
11: $\quad\quad deg \leftarrow i$

Kľúče sa ukladajú v štruktúrach `encoding_context_t` a `decoding_context_t`. Štruktúra `decoding_context_t` obsahuje polynómy typu `gf4_poly_t` h_0 a h_1, ktoré zodpovedajú prvým riadkom cyklických blokov H_0 a H_1. Ďalej obsahuje veľkosť cyklického bloku a jeho Hammingovu váhu. Navyše sú v nej uložené všetky údaje, ktoré sú špecifické pre dekódovanie tým-ktorým dekodérom. Napr. dekodér, ktorý na svoje fungovanie potrebuje parameter δ bude mať tento parameter uložený práve v dekódovacom kontexte. Vďaka tomu môžu mať všetky funkcie, ktoré implementujú jednotlivé dekodéry, rovnaké rozhranie a je principiálne možné prepínať medzi nimi za behu programu pomocou smerníkov na funkcie.

Štruktúra `encoding_context_t` v sebe drží informácie potrebné na kódovanie, t. j. reprezentáciu matice G. Na výpočet matice G sme v predchádzajúcej kapitole uviedli rovnicu

(5). Táto rovnica síce je správna, v našom prípade je však výhodné mierne ju upraviť. Prvý blok matice G je jednotková matica, preto nie je potrebné ukladať ju v pamäti. Nakoľko pracujeme nad poľom $GF(4)$, mínus vo výpočte druhého bloku je irelevantné. Navyše, nie je potrebné počítať transpozíciu, stačí vhodne upraviť algoritmus kódovania. Vo výsledku sa teda v kódovacom kontexte ukladá polynóm $second\_block\_G(X) = h_0(X)h_1^{-1}(X)$ zodpovedajúci prvému riadku druhého bloku matice G s vynechaním transpozície. Ďalej sa v tomto kontexte ukladá veľkosť cyklického bloku.

Výpočet matice G je priamo závislý od vygenerovanej matice H, preto nám prišlo rozumné pre oba kontexty vytvoriť spoločnú inicializačnú funkciu (`contexts_init`). Vytvoria sa v nej obidva kontexty a alokuje sa potrebná pamäť pre polynómy (použitím inicializačnej funkcie pre polynómy). Realizuje sa tu algoritmus 1 s jednou obmenou. Našim cieľom je vygenerovať taký polynóm $h_1(X)$, ktorý je invertibilný mod $X^k - 1$. Za týmto účelom sa opakovane používa rozšírený Euklidov algoritmus, ktorý buď nájde inverziu alebo nám dá informáciu o tom, že $h_1(X)$ nie je pri danom module invertibilný. Výpočet inverzie pomocou rozšíreného Euklidovho algoritmu je však z hľadiska výpočtovej náročnosti veľmi drahá operácia, preto je vhodné vopred vylúčiť polynómy, pri ktorých vieme ľahko zistiť, že invertibilné nie sú.

Nakoľko pracujeme nad poľom $GF(4)$, platí $X^k - 1 = X^k + 1$. Tento polynóm môžeme jednoducho rozložiť:

$$X^k + 1 = (X + 1) \sum_{i=0}^{k-1} X^i$$

Ak má polynóm $h_1(X)$, ktorý sme vygenerovali, vo svojom rozklade polynóm $X + 1$, je súdeliteľný s $X^k + 1$, a teda nebude mať pri tomto module inverziu. Našťastie, vieme jednoducho overiť, či platí $(X + 1) \mid h_1(X)$. Stačí zistiť, či je 1 koreňom $h_1(X)$. To nastáva vtedy, keď $\sum_{i=0}^{k-1} h_{1i} = 0$, pričom h_{1i} je koeficient pri X^i v $h_1(X)$. Po vygenerovaní koeficientov polynómu $h_1(X)$ rovno overíme, či ich súčet nie je nulový. Ak je, polynóm $h_1(X)$ generujeme znova.

Ďalej priblížime, ako fungujú niektoré kľúčové operácie používajúce tieto kontexty. Uviedli sme, že nie je potrebné transponovať druhý blok matice G, ak vhodne upravíme kódovanie. Pri kódovaní správy m najprv skopírujeme celú správu do výstupnej zakódovanej správy m'. To zodpovedá násobeniu správy m jednotkovým blokom. Potom nasleduje násobenie druhým blokom matice G. V prípade, že matica G je reprezentovaná ako v rovnici (5), jej druhý blok je $\left(H_1^{-1}H_0\right)^T$ (vynechali sme mínus). Do správy m' postupne pridávame skalárne súčiny m so stĺpcami druhého bloku. Keď druhý blok nie je transponovaný, počítame namiesto toho skalárne súčiny m s riadkami druhého bloku. Nakoľko v pamäti

ukladáme len koeficienty prvého riadku druhého bloku, je zároveň potrebné počítať cyklické posuny, resp. vhodne indexovať prvý riadok. Kódovanie popisuje algoritmus 12.
Pri výpočte syndrómu s správy c počítame postupne skalárne súčiny c so stĺpcami matice H^T. Ani v tomto prípade však nie je potrebná transpozícia, postačí počítanie skalárnych súčinov c s riadkami H. To vieme opäť docieliť správnou indexáciou. Pri veľkosti cyklického bloku k vieme výpočet syndrómu rozložiť na prvú časť, kde sa počíta násobenie prvej polovice c s riadkami H_0, a druhú časť, kde sa zasa počíta násobenie druhej polovice c s riadkami H_1. Nie je však potrebné implementovať výpočet týchto dvoch častí dvoma cyklami, postačí len jeden od 0 po veľkosť bloku. Výpočet syndrómu je popísaný v algoritme 13.

Algoritmus 12 Kódovanie správy m

Vstupy: Alokovaný výstupný vektor $m' \in GF(4)^{2k}$, vstupný vektor $m \in GF(4)^k$, veľkosť cyklického bloku k, druhý blok matice G *second_block_G*

1: $m'_{0..k-1} \leftarrow m_{0..k-1}$
2: **for** $i \leftarrow k..1$ **do**
3: $\quad tmp \leftarrow 0$
4: $\quad$ **for** $j \leftarrow 0..k-1$ **do**
5: $\quad\quad tmp \leftarrow tmp + m_j * second\_block\_G_{(i+j) \bmod k}$
6: $\quad m'_{2k-i} \leftarrow tmp$

Algoritmus 13 Výpočet syndrómu správy c

Vstupy: Alokovaný výstupný vektor $s \in GF(4)^k$, vstupný vektor $c \in GF(4)^{2k}$, veľkosť cyklického bloku k, polynómy h_0 a h_1

1: **for** $i \leftarrow k..1$ **do**
2: $\quad tmp \leftarrow 0$
3: $\quad$ **for** $j \leftarrow 0..k-1$ **do**
4: $\quad\quad tmp \leftarrow tmp + c_j * h_{0,(i+j) \bmod k}$
5: $\quad\quad tmp \leftarrow tmp + c_{k+j} * h_{1,(i+j) \bmod k}$
6: $\quad s_{k-i} \leftarrow tmp$

Pri implementácii SF dekodéra je potrebné aktualizovať syndróm pripočítaním konkrétneho stĺpca matice H. Presnejšie, po preklopení pozície *pos* v chybovom vektore tak, ako je tomu v algoritme 7 na riadku 18, je potrebné pričítať ku syndrómu a_{max}-násobok stĺpca $H_{*,pos}$. Opäť ide o problém, ktorý je možné riešiť vhodnou indexáciou. Najprv je potrebné určiť,

či *pos* označuje stĺpec, ktorý je v bloku H_0 alebo v bloku H_1. Potom sa iteruje súčasne cez syndróm a cez prvý riadok daného bloku vo vhodnom poradí. Spočíta sa a_{max}-násobok prvku na danej pozícii v riadku a upraví sa prvok na danej pozícii v syndróme. Aktualizáciu syndrómu popisuje algoritmus 14.

Podobnou logikou, ako uvedená aktualizácia syndrómu, sa riadi aj výpočet σ_j. Jediným rozdielom je, že syndróm sa v tomto prípade nemodifikuje, len sa určí jeho nová Hammingova váha.

Algoritmus 14 Aktualizácia syndrómu

Vstupy: Alokovaný výstupný vektor $s \in GF(4)^k$, pozícia *pos*, veľkosť cyklického bloku k, polynómy h_0 a h_1, hodnota $a_{max} \in GF(4)$

1: **if** $pos < k$ **then**
2: $\quad h \leftarrow h_0$
3: **else**
4: $\quad h \leftarrow h_1$
5: $\quad pos \leftarrow pos - k$
6: $x \leftarrow pos$
7: $i \leftarrow 0$
8: **repeat**
9: $\quad s_i \leftarrow s_i + h_x * a_{max}$
10: $\quad x \leftarrow x - 1 \bmod k$
11: $\quad i \leftarrow i + 1$
12: **until** $x \neq pos$

2.4 Použitie knižnice

Považujeme za vhodné uviesť aj schematický príklad použitia našej knižnice. Zatiaľ sme totiž popísali len stavebné bloky našej knižnice. Používateľ však vôbec nemusí interagovať s kódovaním, dekódovaním či generovaním náhodných vektorov. Namiesto toho má k dispozícii funkcie na šifrovanie a dešifrovanie, ktoré vykonajú určitú mieru úkonov namiesto neho.

Predstavme si, že používateľ chce vygenerovať náhodnú správu, zašifrovať ju a v zápätí aj dešifrovať. Môže to docieliť nasledujúcou postupnosťou krokov:

1. inicializácia kontextov s kľúčmi (`contexts_init`)
2. inicializácia vektorov na uloženie otvoreného textu, zašifrovanej správy a dešifrovanej správy (`gf4_poly_init`)

3. vygenerovanie náhodného otvoreného textu (`random_gf4_poly`)
4. zašifrovanie otvoreného textu (`enc_encrypt`)
5. dešifrovanie zašifrovaného textu (`dec_decrypt`)
6. overenie výsledku (pomocou podmienky kontrolujúcej návratovú hodnotu funkcie `dec_decrypt`)
7. uvoľnenie vektorov (`gf4_poly_deinit`)
8. uvoľnenie kontextov (`contexts_deinit`)

Parametre kryptosystému ako veľkosť bloku a váha riadku bloku je potrebné nastaviť pri volaní funkcie `contexts_init` (krok 1), pretože na nich závisí generovanie kľúčov. Počet chýb v chybovom vektore je možné špecifikovať pri šifrovaní.
Používateľ má možnosť vykonať niekoľko nastavení dekódovania, ktoré sa bude vykonávať pri dešifrovaní: pomocou smerníka na funkciu si môže zvoliť, aký dekodér sa použije, prípadne zmeniť niektoré parametre dekodéra (v prípade, že zvolený dekodér má nejaké nastaviteľné parametre). To môže docieliť upravením zodpovedajúcich polí štruktúry `decoding_context_t`.

3 Výsledky experimentov

V tejto kapitole vysvetlíme, aké experimenty sme vykonali, aké boli ich nastavenia a v akom výpočtovom prostredí prebehli. Potom zhodnotíme ich výsledky.

3.1 Hardvérové a softvérové prostredie

Všetky experimenty, ktoré popíšeme v tejto kapitole, prebiehali na zariadení s nasledujúcimi špecifikáciami:

- Hardvér:
  - Intel® Core™i5-10400 @ 2.90GHz × 12 vláken
  - RAM 32GB DDR4-3200 s použitím vhodného XMP profilu
- Softvér:
  - Operačný systém Fedora Linux 37 Workstation
  - Kompilátor gcc verzia 13.0.1
  - Python interpreter verzia 3.11.2, knižnica numpy verzie 1.24.2, matplotlib 3.7.1, seaborn 0.12.2

3.2 Paralelizácia experimentov

Pri popise experimentov vždy uvádzame, aké boli nastavenia kryptosystému a koľko dekódovaní sme vykonali. Aby sme zmenšili pravdepodobnosť, že sa pri niektorom experimente vygenerovali anomálne kľúče, ktoré by ovplyvnili výsledky, v každom experimente sme používali viacero kľúčov. Z toho dôvodu pri popisoch experimentov uvádzame aj počet použitých kľúčov s počet správ, ktoré sme dekódovali daným kľúčom.

Takéto rozdelenie experimentov nám naviac poskytlo možnosť extrémne jednoducho ich paralelizovať. Keď jeden proces vygeneruje kľúče, vykoná s nimi potrebný počet dekódovaní. Medzitým môže iný proces môže vygenerovať ďalšie kľúče a dekódovať s nimi.

Každý proces zapíše svoje výsledky do vlastného súboru. Procesy medzi sebou vôbec nemusia zdieľať pamäť (ani kvôli kľúčom, ani kvôli deskriptorom súborov), a teda nie je potrebné navrhovať ich vzájomné vylučovanie. Keď ukončia svoje vykonávanie, stačí spojiť ich výsledky do jedného výstupu.

Naviac, kreáciu procesov nie je potrebné riešiť na úrovni knižnice. Knižnica obsahuje len implementáciu experimentov. Tie sú parametrizované počtom kľúčov a počtom dekódovaní na každý kľúč. Tieto hodnoty sa načítavajú ako argumenty z príkazového riadku. Pomocou

jednoduchého Bash skriptu potom vieme spustiť potrebný počet procesov, pričom každý má za úlohu spracovať len určitý podiel z celkového počtu kľúčov. Navyše výstupné súbory stačí vytvárať pomocou presmerovaní výstupu.

Predstavme si, napríklad, že potrebujeme vykonať 10 000 dekódovaní. Jeden spôsob, ako to docieliť, je vygenerovať 100 kľúčov a s každým dekódovať 100 správ. Môžeme vytvoriť 10 procesov, z ktorých každý bude generovať 10 kľúčov a vykoná s každým potrebných 100 dekódovaní.

Nakoľko procesor, na ktorom prebiehali výpočty, má 12 vláken a experimenty vždy vyťažili jeho jadrá na 100%, nikdy sme nevytvorili viac ako 12 procesov naraz. Vytvoreným procesom sme navyše zvýšili prioritu príkazom `nice`. Uvedené nemalo vplyv na DFR ani na rýchlosť dekódovania z hľadiska počtu iterácií, iba na rýchlosť dekódovania z hľadiska času. Tú však vo výsledkoch neuvádzame.

3.3 DFR dekódera SF

Autori [9] testovali DFR implementácie SF dekodéra (algoritmus 7) pre QC-MDPC kódy nad $GF(2)$, $GF(4)$ a $GF(8)$. V chybovom vektore nastavili vyšší počet chýb, než je odporúčaná hodnota $w(e) = 84$. Ide o korektný spôsob, ako umelo navýšiť DFR. Tá by totiž pri odporúčaných nastaveniach bola veľmi nízka a jej meranie by bolo veľmi nepraktické (bolo by potrebné vykonať veľmi vysoký počet dekódovaní, aby sme zaznamenali aspoň niekoľko zlyhaní dekódovania).

Zároveň však zmenšili aj veľkosť bloku: namiesto odporúčaného $k = 2339$ sa v grafe 2 v [9] uvádzalo $k = 2293$. Aj zmenšenie veľkosti bloku je možné použiť na navýšenie DFR, boli sme však prekvapení, že autori zároveň navýšili počet chýb aj menili veľkosť bloku.

Neboli sme si istí, či v prípade nastavenia $k = 2293$ nejde o preklep, pretože nastavenia veľkosti blokov pre kryptosystémy nad inými poľami ($GF(2)$ a $GF(8)$) zodpovedali uvádzaným odporúčaniam.

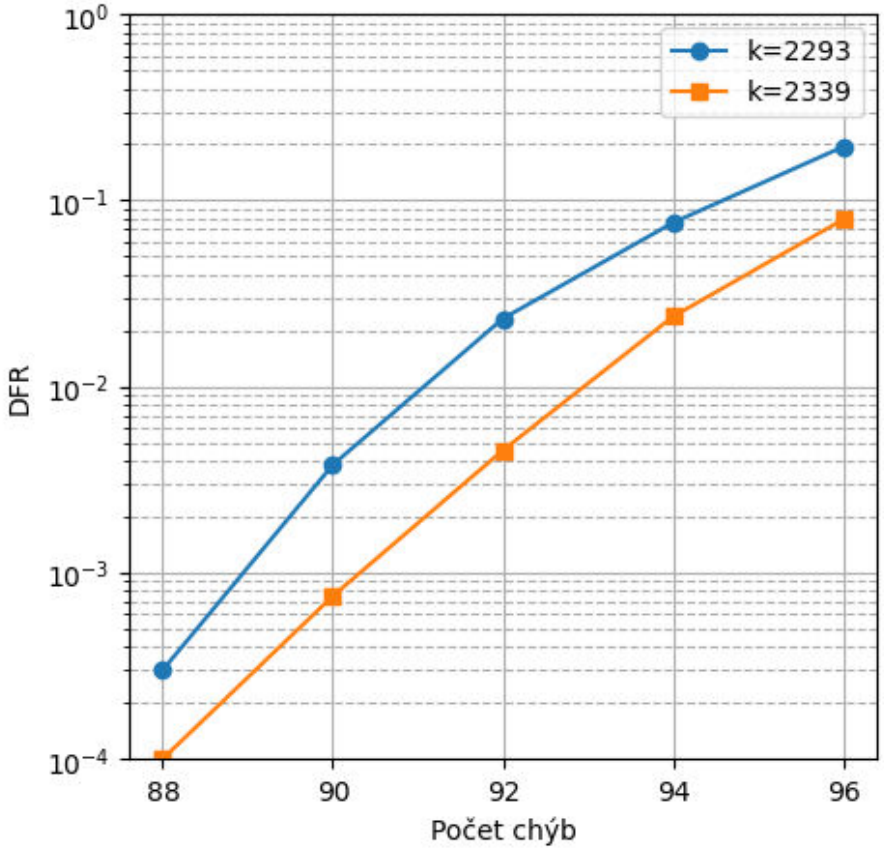
Preto sme sa rozhodli otestovať obe veľkosti bloku so základným SF dekodérom. Všetky ostatné nastavenia boli identické pre oba testy. V oboch prípadoch sme testovali všetky počty chýb $w(e) \in \{88, 90, 92, 94, 96\}$. Autori [9] navyše testovali pre všetky dekodéry aj $w(e) = 98$ a $w(e) = 100$. Vyššie nastavenia $w(e)$ sme v žiadnom experimente nepoužili z dôvodu časovej náročnosti. Nižšie nastavenia zasa nie sú relevantné pre porovnanie našich výsledkov s tými, ktoré prezentujú autori, pretože najnižšia hodnota $w(e)$, s ktorou testovali SF nad $GF(4)$ bola práve 88.

Váha riadku cyklického bloku matice H bola $w = 37$. Pri veľkosti bloku $k = 2293$ sme pre každý počet chýb sme vygenerovali 100 kľúčov a s každým kľúčom sme vykonali 100

dekódovaní. Pri veľkosti bloku $k = 2339$ sme použili 200 kľúčov a 100 dekódovaní na kľúč. V tomto prípade sme navýšili počet kľúčov, pretože dekódovanie zlyhávalo menej často a chceli sme dosiahnuť presnejšie hodnoty pre nízke nastavenia $w(e)$. Maximálny počet iterácií bol v oboch prípadoch $ITER = 200$.

Počty zlyhaní, ktoré sme namerali, sú uvedené v tabuľke 2. DFR pre daný počet chýb a veľkosť bloku $k = 2293$ je počet zlyhaní vydelený 10 000 a pre veľkosť bloku $k = 2339$ vydelený 20 000.

Nakoľko autori svoje výsledky uviedli iba vo forme grafu, aj my sme sa rozhodli zakresliť naše výsledky do grafu, aby bolo možné ich jednoducho porovnať s článkom [9]. Tento graf je na obrázku 1.



Obr. 1: Graf vývoja DFR pre rôzne veľkosti bloku pri SF dekodéri.

Vidíme, že krivka DFR pre nastavenie $k = 2293$ je viac podobná výsledkom autorov podľa obrázku 2 z článku [9]. Pre každé nastavenie $w(e)$ je DFR pri veľkosti bloku $k = 2339$ výrazne nižšia, než pre identické nastavenie $w(e)$ pri $k = 2293$. To isté platí aj pri jej porovnaní s krivkou, o ktorej autori [9] tvrdia, že reprezentuje DFR pre nastavenie veľkosti bloku $k = 2293$.

$w(e)$	88	90	92	94	96
$k = 2293$	3	38	232	760	1943
$k = 2339$	2	15	91	476	1582

Tabuľka 2: Namerané počty zlyhaní na 10 000 dekódovaní pre $k = 2293$, resp. na 20 000 dekódovaní pre $k = 2339$ pri SF dekodéri.

Tieto výsledky naznačujú, že naša prvotná podozrievavosť voči uvedenej veľkosti bloku $k = 2293$ nebola opodstatnená a zdá sa, že autori skutočne vyhodnocovali DFR pri takom nastavení k. Z toho dôvodu sme pri testoch DFR ďalších dekodérov používali nastavenie $k = 2293$.

3.4 Zber dát na neskoršiu analýzu

Základný SF dekodér, ktorý sme implementovali, dosahoval v testovaní DFR výsledky porovnateľné s tými, ktoré uviedli autori [9], preto sme pristúpili ku zberu dát, ktoré vypovedajú o správaní tohto dekodéra v priebehu jeho činnosti. Považovali sme za užitočné pochopiť, ako sa tento dekodér správa pri použití odporúčaných nastavení kryptosystému, t. j. $k = 2339$, $w = 37$, $w(e) = 84$. Maximálny počet iterácií bol $ITER = 200$.

Pre tieto nastavenia sme vygenerovali 10 kľúčov a vykonali sme 10 šifrovaní nasledovaných dešifrovaniami na každý kľúč. Pri každom šifrovaní sme zaznamenali použitý chybový vektor. Pri každom dešifrovaní sme zaznamenali informácie o priebehu dekódovania: v každej iterácii sme uložili aktuálnu Hammingovu váhu syndrómu a hodnoty σ_j pre všetky pozície j a všetky symboly. Všetky dekódovania boli úspešné.

Zaznamenané dáta používame v neskorších experimentoch, kde ich analýza ovplyvňuje nastavenia modelov, ktoré sme testovali. Mohla by sa však naskytnúť otázka, ako môžu dáta, ktoré sme namerali pri dekódovaní s odporúčanými nastaveniami kryptosystému, byť užitočné pri experimentoch, v ktorých sú iné nastavenia veľkosti bloku a počtu chýb v chybovom vektore. Netreba zabúdať, že zmenšenie veľkosti bloku a navýšenie počtu chýb slúži len na to, aby sa umelo navýšila DFR na merateľné hodnoty, nie na to, aby sme ich použili pri reálnom použití kryptosystému.

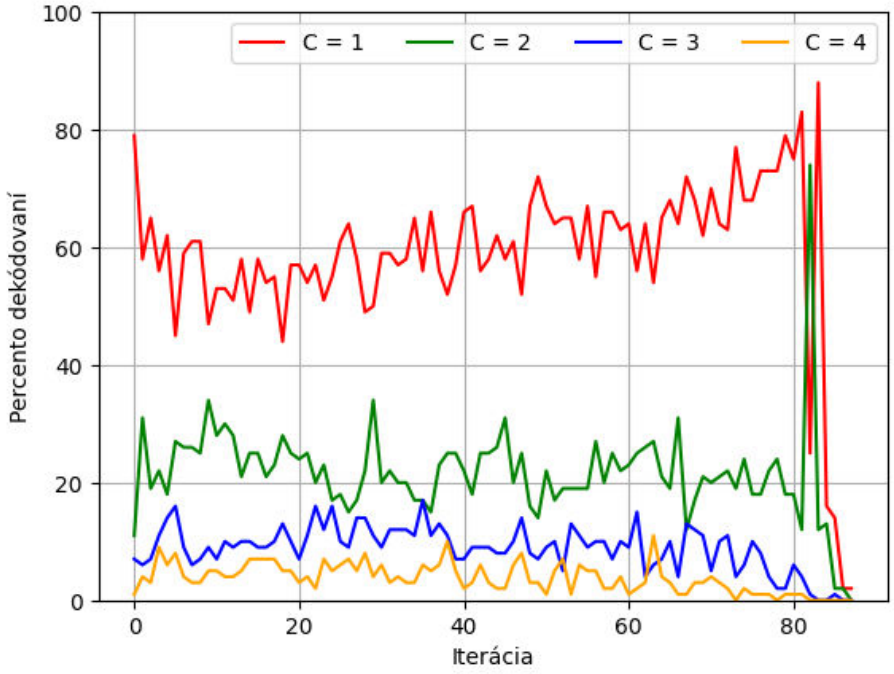
V tejto práci sme navrhli niekoľko dekodérov, ktoré potrebujú na svoje fungovanie dodatočné parametre. Našim cieľom je nájsť také hodnoty týchto parametrov, ktoré sú použiteľné práve s odporúčanými nastaveniami kryptosystému, preto sme na zber dát popísaný v tejto kapitole použili práve tieto nastavenia. Až pri vyhodnotení DFR našich návrhov zmenšíme veľkosť bloku a navýšime počet chýb.

Ešte uvedieme, že všetky zozbierané dáta sú k dispozícii online prostredníctvom platformy

GitHub spolu so skriptami v jazyku Python, ktoré sme napísali na ich spracovanie. Vďaka tomu je možné vykonať kontrolu niektorých tvrdení, ktoré v práci uvádzame.

3.5 Vyhodnotenie dekodéra SF s využitím parametra δ

V prípade SF dekodéra s využitím parametra δ (algoritmus 8) je potrebné otestovať rôzne nastavenia tohto parametra. Malo by ísť o malú, nezápornú, celočíselnú hodnotu. Otázkou je, či má zmysel testovať nastavenie $\delta = 0$. Také nastavenie by znamenalo, že sa preklopia symboly na tých pozíciách j, pre ktoré platí $\sigma_j = \sigma_{max}$.
Zmysel to bude mať vtedy, keď sa v jednotlivých iteráciách objaví viacero pozícií, ktoré dosahujú σ_{max}. Aby sme zistili, či to tak je, analyzovali sme dáta zozbierané podľa popisu z kapitoly 3.4. Pre každé zo 100 vykonaných dekódovaní sme v každej iterácii určili $C = |\{j : \sigma_j = \sigma_{max}\}|$. Získané hodnoty sme zoskupili podľa čísla iterácie naprieč dekódovaniami. Spočítali sme, v koľkých prípadoch zo 100 dekódovaní v jednotlivých iteráciách nastáva situácia, že $C = 1$, potom $C = 2$, $C = 3$ a $C = 4$ ($C = 5$ a vyššie boli zväčša outliery). Výsledky sme naniesli na graf 2.



Obr. 2: Percento z celkového počtu dekódovaní, v ktorých sa dosahujú hodnoty $C = 1$, $C = 2$, $C = 3$ a $C = 4$, v jednotlivých iteráciách.

Je zjavné, že vo väčšine dekódovaní bola v jednotlivých iteráciách len jedna pozícia, na ktorej sa dosiahla σ_{max}. Napriek tomu však pre každú iteráciu platí, že relatívne veľké

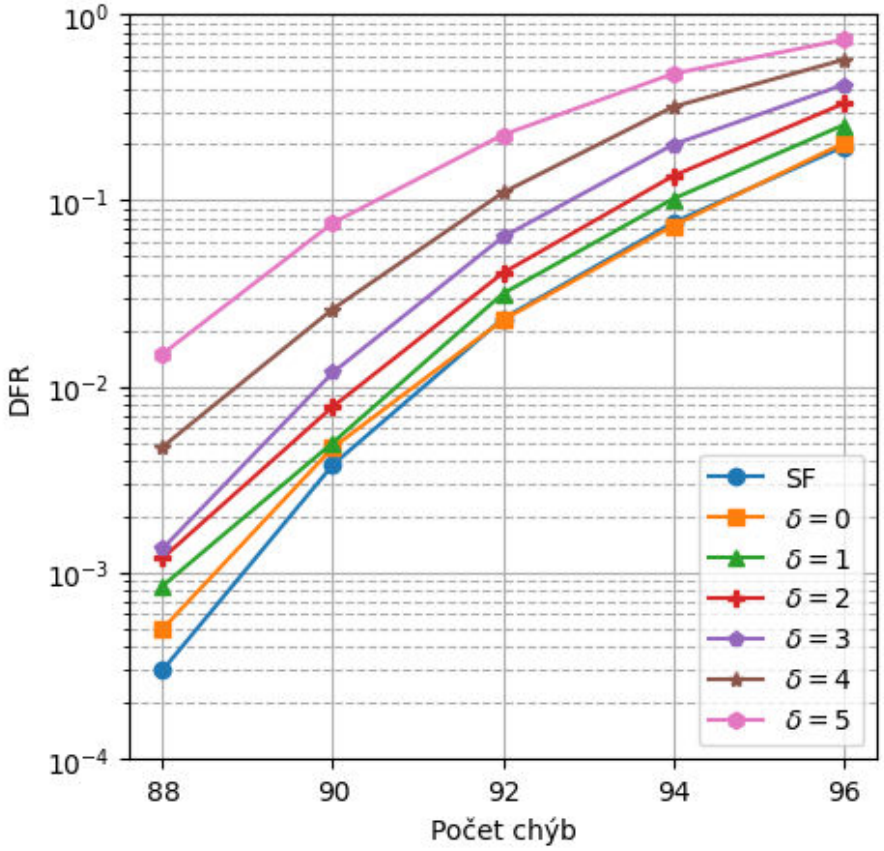
percento dekódovaní malo v danej iterácii viacero takých pozícií. Z toho usudzujeme, že má zmysel testovať nastavenie $\delta = 0$.

Testovali sme DFR pre všetky nastavenia $\delta \in \{0, 1, 2, 3, 4, 5\}$. Použili sme veľkosť bloku $k = 2293$, váhu $w = 37$ a všetky $w(e)$ z množiny $\{88, 90, 92, 94, 96\}$. Pre $w(e) = 88$ sme vykonali 20 000 dekódovaní (200 kľúčov, 100 dekódovaní na kľúč). Pre ostatné hodnoty $w(e)$ sme vykonali 10 000 dekódovaní (100 kľúčov, 100 dekódovaní na kľúč). Maximálny počet iterácií bol $ITER = 200$. Namerané počty zlyhaní sme uviedli v tabuľke 3.

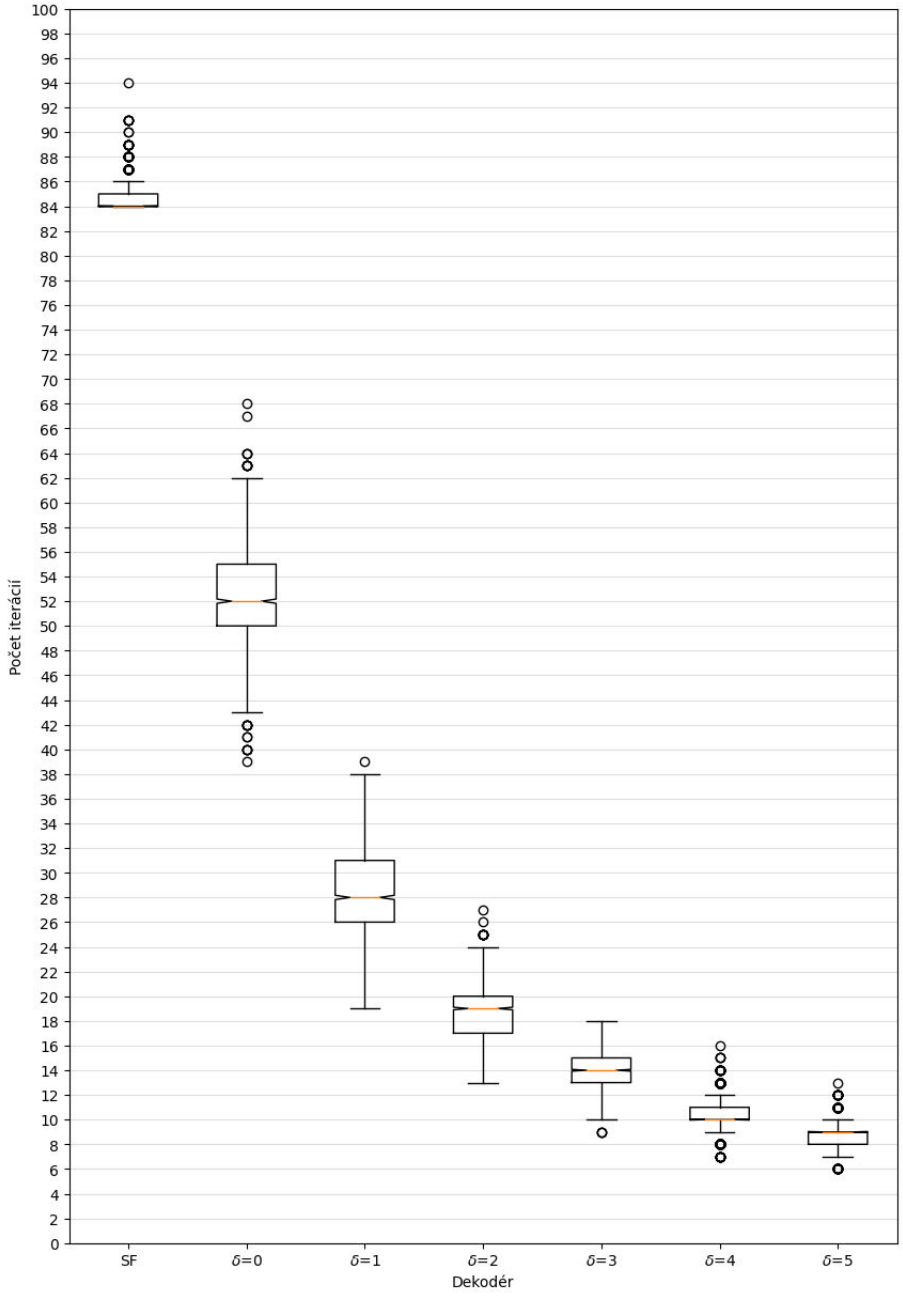
Taktiež sme spočítali vývoj DFR a naniesli ho na graf 3. Zároveň sme na tento graf vykreslili aj vývoj DFR pre základný SF dekodér s veľkosťou bloku $k = 2293$, aby bolo možné jednoducho ho porovnať so zvyšnými výsledkami.

w(e)	88	90	92	94	96
$\delta = 0$	10	47	228	732	2028
$\delta = 1$	17	50	316	1020	2538
$\delta = 2$	24	78	408	1354	3308
$\delta = 3$	27	119	637	1995	4175
$\delta = 4$	95	261	1102	3183	5693
$\delta = 5$	300	757	2254	4793	7293

Tabuľka 3: Namerané počty zlyhaní na 20 000 dekódovaní pre $w(e) = 88$, resp. na 10 000 dekódovaní pre ostatné $w(e)$ vzhľadom na nastavenie parametra δ.



Obr. 3: Graf vývoja DFR pre rôzne nastavenia parametra δ pri SF dekodéri.



Obr. 4: Krabicové diagramy reprezentujúce trvania dekódovaní z hľadiska počtu iterácií pre jednotlivé nastavenia δ.

Vidíme, že nastavenie parametra $\delta = 0$ má podobný vývoj DFR ako základný SF dekodér pri nastavení $w(e) > 90$. Pre menšie váhy chybového vektora je však krivka DFR lepšia v prípade SF dekodéra. Pri zvyšujúcej sa hodnote δ môžeme pozorovať zhoršujúcu sa DFR. Našou primárnou motiváciou na zavedenie SF dekodéra, ktorý využíva parameter δ, bolo urýchlenie dekódovania. Aby sme zistili, do akej miery jednotlivé nastavenia δ urýchľujú dekódovanie, vykonali sme 2000 dekódovaní (20 kľúčov, 100 dekódovaní na kľúč) pre

odporúčané nastavenia kryptosystému (podľa tabuľky 1). Maximálny počet iterácií bol $ITER = 200$. Zaznamenávali sme, koľko iterácií trvalo dekódovanie. Ten istý experiment sme vykonali aj so základným SF dekodérom. Údaje sme vizualizovali pomocou krabicových diagramov (graf 4).

Základný SF dekodér podľa očakávaní potrebuje na dekódovanie aspoň 84 iterácií. Prekvapivo, už nastavenie $\delta = 0$ výrazne urýchľuje dekódovanie. Aj najpomalšie dekódovania pri $\delta = 0$ sú rýchlejšie, než dekódovania s použitím SF dekodéra. Najčastejšie dekódovanie skončilo po 52 iteráciách. Nastavenie $\delta = 1$ je viditeľne rýchlejšie než $\delta = 0$ s mediánom na hodnote 28. Pri $\delta = 2$ opäť vidíme zrýchlenie, už však nie je také výrazné. Pri $\delta = 3$ je medián na úrovni 14 iterácií. Pri $\delta = 5$ je medián aj tretí kvartil na hodnote 9 a iba outliery dosahujú hodnotu vyššiu ako 10.

Autori [4] pri BF dekodéri pre binárne QC-MDPC kódy s použitím parametra δ odporúčajú na $\delta \approx 5$, aby sa dosiahlo dekódovanie do 10 iterácií. Ide o podobnú hodnotu, ako sme namerali pre $\delta = 5$ aj my.

3.6 Vyhodnotenie dekodéra SF s využitím prahovej hodnoty

Na to, aby SF dekodér s využitím adaptívnej prahovej hodnoty $T(s)$ fungoval korektne, musí byť funkcia $T(s)$ vhodne zvolená. Hodnota funkcie $T(s)$ pre konkrétny syndróm s by mala predstavovať hranicu medzi hodnotami σ_j na pozíciách, kde $j \in supp(e)$, a σ_j na pozíciách, kde $j \notin supp(e)$. Z uvedeného popisu sa črtá možná stratégia na aproximáciu $T(s)$.

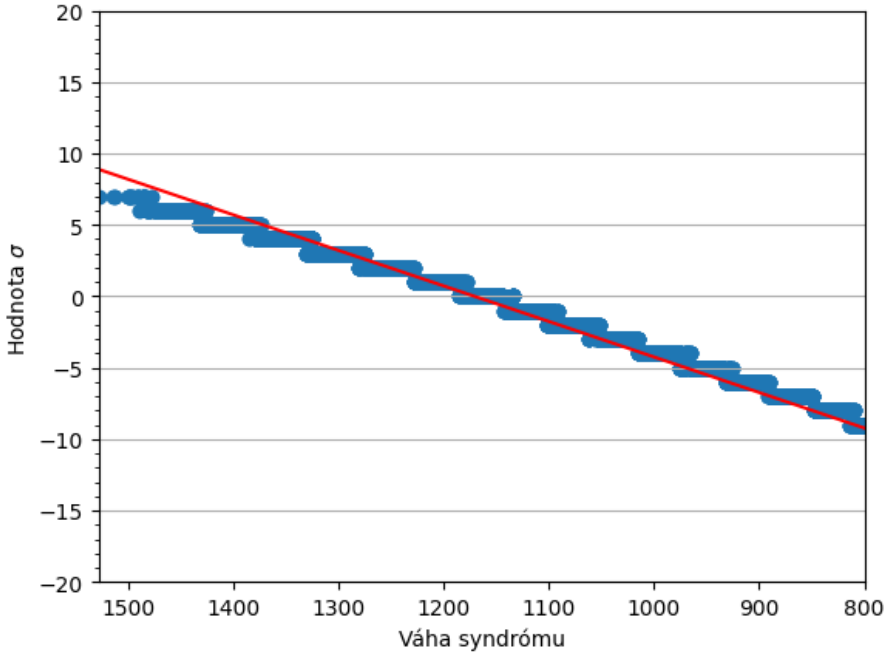
Opäť sme analyzovali dáta získané pri zbere popísanom v kapitole 3.4. Vytvorili sme zoznam všetkých váh, ktoré nadobúdal syndróm naprieč všetkými dekódovaniami. Pre každú váhu sme našli iterácie, v ktorých ju syndróm dosiahol, a priradili sme jej zoznam všetkých hodnôt σ_j, ktoré boli v týchto iteráciách vypočítané. Nakoľko sme pri zbere dát ukladali aj použité chybové vektory, vedeli sme tieto zoznamy ďalej rozdeliť na "správne" hodnoty σ_j (také σ_j, kde $j \in supp(e)$ a "nesprávne" hodnoty σ_j (také σ_j, kde $j \notin supp(e)$) hodnoty σ_j.

Vo výsledku sme teda ku každej nameranej váhe syndrómu priradili dva zoznamy: jeden zoznam takých σ_j, ktoré značia, že pozíciu j je žiadúce preklopiť, a jeden zoznam takých σ_j, ktoré značia, že pozíciu j nie je žiadúce preklopiť.

Pre každú váhu syndrómu sme potom určili takú hodnotu σ (ktorá sa nemusí nevyhnutne nachádzať medzi spočítanými σ_j), pre ktorú platí, že väčšina "nesprávnych" σ_j je menšia než σ a väčšina "správnych" σ_j je väčšia ako σ. Tak sme získali množinu dvojíc váh

syndrómov a zodpovedajúcich σ.

Na tieto dvojice sme aplikovali lineárnu regresiu a získali sme jednu možnú aproximáciu funkcie $T(s)$. Nájdená lineárna funkcia bola $f_0(s) = 0.0248577875w(s) - 29.1143817$. Graf tejto funkcie spolu s nájdenými σ pre jednotlivé váhy syndrómu je na obrázku 5.



Obr. 5: Nájdené hodnoty σ vzhľadom na váhu syndrómu a k nim aproximovaná priamka.

Aproximovaná funkcia však relatívne rýchlo klesá do záporných hodnôt. Navyše, možné hodnoty σ_j sú celé čísla, preto aj výstupy $T(s)$ by mali byť celočíselné. Aby sme zohľadnili oba tieto fakty, definovali sme prvú možnú funkciu na výpočet prahových hodnôt nasledujúcim spôsobom:

$$T_0(s) = max\{\lfloor f_0(s) \rfloor, 0\}$$

Treba tiež dodať, že táto priamka je pravdepodobne priveľmi optimistický odhad prahovej hodnoty, pretože σ je nastavená vždy tak, aby bolo možné preklopiť čo najviac pozícií j, kde $j \in supp(e)$. Z pozorovania dát však vieme, že zoznamy "správnych" a "nesprávnych" σ_j sa často výrazne prelínajú. Pri príliš nízkej prahovej hodnote teda bude dochádzať aj ku preklápaniu "nesprávnych" pozícií j.

Preto sme sa definovali viacero ďalších funkcií, ktoré sme založili na paralelných priamkach vzhľadom na nájdenú lineárnu aproximáciu:

$$T_i(s) = max\{\lfloor f_0(s) + i \rfloor, 0\},$$

kde $i \in \{1, 2, 3, 4, 5\}$. Potom sme testovali DFR pre všetky definované funkcie.

Najprv sme pre každú funkciu vygenerovali 10 kľúčov a vykonali 100 dekódovaní na kľúč pri nastavení $k = 2339$, $w = 37$ a $w(e) = 84$, t. j. pri odporúčaných parametroch. Maximálny počet iterácií bol $ITER = 20$. Pri dobre nastavenom výpočte prahových hodnôt by sme mali namerať na takto malom počte dekódovaní 0 zlyhaní. Pri $T_0(s)$ sme však namerali 12 zlyhaní. To naznačuje, že výpočet prahových hodnôt je v tomto prípade skutočne príliš optimistický. Pri $T_5(s)$ sme zaznamenali 2 zlyhania. To naznačuje, že výpočet prahových hodnôt pomocou $T_5(s)$ je zasa až príliš pesimistický. Z toho vyplýva, že vhodná funkcia bude ohraničená funkciami $T_0(s)$ a $T_5(s)$.

Pre $T_1(s)$-$T_4(s)$ sme pokračovali s testami DFR pri veľkosti bloku $k = 2293$ a váhach chybového vektora $w(e) \in \{88, 90, 92, 94, 96\}$. Pre $w(e)$ sme vygenerovali 200 kľúčov a vykonali 100 dekódovaní na kľúč. Pre zvyšné $w(e)$ sme vygenerovali 100 kľúčov a vykonali 100 dekódovaní na kľúč. Maximálny počet iterácií bol $ITER = 20$. Namerané počty zlyhaní uvádzame v tabuľke 4 a zodpovedajúci graf vývoja DFR je na obrázku 6. Do tohto grafu sme opäť pridali aj výsledky pre základný SF dekodér s $k = 2293$ a $ITER = 200$, aby ich bolo možné ľahko porovnať s novými dátami.

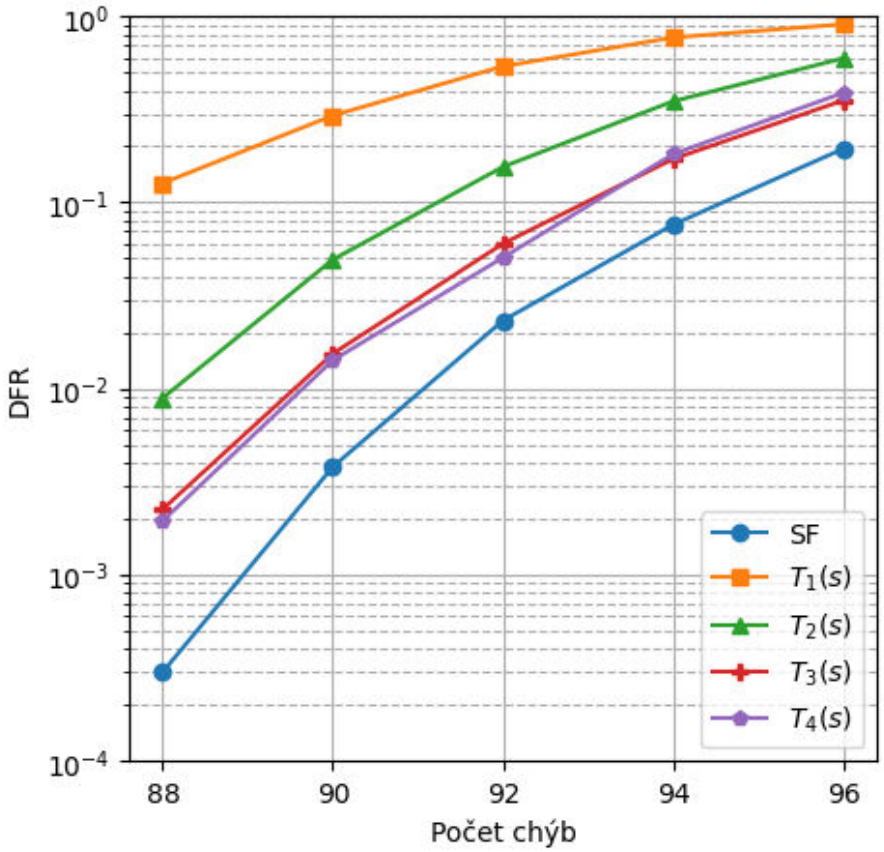
w(e)	88	90	92	94	96
$T_1(s)$	2538	2930	5385	7693	9068
$T_2(s)$	178	493	1554	3496	5947
$T_3(s)$	45	154	601	1712	3533
$T_4(s)$	39	142	506	1819	3881

Tabuľka 4: Namerané počty zlyhaní na 20 000 dekódovaní pre $w(e) = 88$, resp. na 10 000 dekódovaní pre ostatné $w(e)$ vzhľadom na použitú funkciu $T_i(s)$.

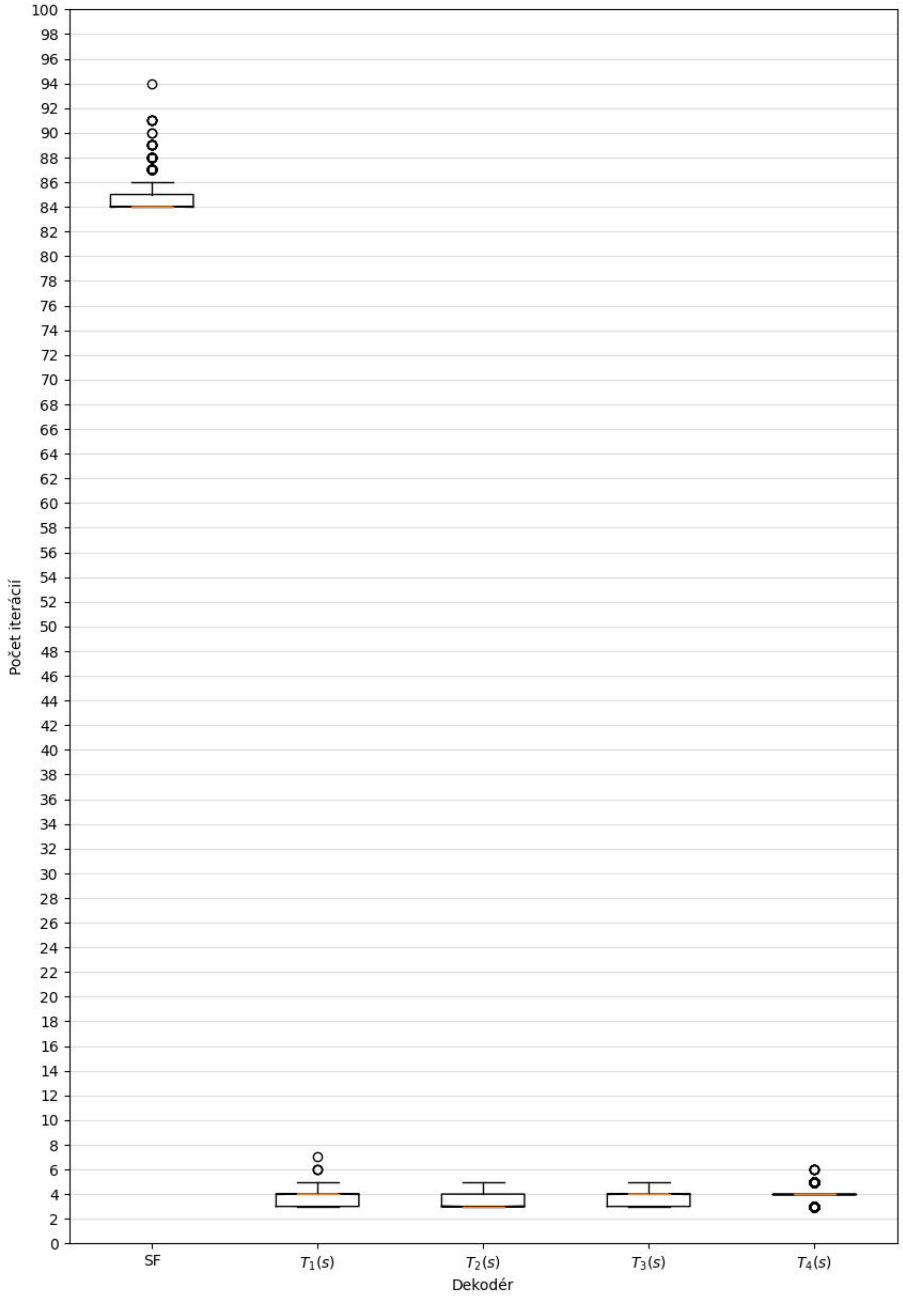
Ako vidíme, SF dekodér má suverénne najlepšiu DFR. Pri nastavení výpočtu prahových hodnôt pomocou $T_3(s)$ a $T_4(s)$ sú si DFR navzájom veľmi podobné. Opäť platí, že primárnou motiváciou na zavedenie dekodéra, ktorý používa výpočet prahových hodnôt, bolo zníženie počtu iterácií dekódovania. Aby sme ich v tomto prípade vyhodnotili, opäť sme použili odporúčané nastavenia krytosystému podľa 1 a $ITER = 20$. Vygenerovali sme 20 kľúčov a s každým sme vykonali 100 dekódovaní. Počty iterácií sme naniesli na krabicové grafy, ktoré vidíme na obrázku 7. Pre jednoduchosť porovnania sme opäť vykreslili aj údaje pre SF dekodér.

Vidíme, že všetky testované dekodéry, ktoré používajú výpočet prahovej hodnoty, úspešne dekódujú správy do 8 iterácií. Najlepšie dopadol dekodér pracujúci s $T_2(s)$, ktorý dosiahol medián počtu iterácií na hodnote 3. Ostatné testované dekodéry s výpočtom prahových

hodnôt mali mediány na hodnote 4. Pri použití funkcií $T_1(s)$ a $T_4(s)$ sa navyše objavovali outliery. Zaujímavé však je, že pri $T_4(s)$ sú medzi outliermi hodnoty 3 a 5. To značí, že väčšina dekódovaní trvala práve 4 iterácie.



Obr. 6: Graf vývoja DFR pre SF dekodéry s použitím rôznych funkcií $T_i(s)$ na výpočet prahových hodnôt.



Obr. 7: Krabicové diagramy reprezentujúce trvania dekódovaní z hľadiska počtu iterácií pre jednotlivé nastavenia $T_i(s)$.

Záver

V tejto práci sme nadviazali na článok [9] a navrhli sme tri dekodéry pre QC-MDPC McEliecov kryptosystém nad $GF(4)$, ktoré si kladú za cieľ urýchliť dekódovanie.

Prvým z nich je SF dekodér s parametrom δ (algoritmus 8). Podarilo sa nám ukázať, že pri vhodnom nastavení parametra δ tento dekodér dokáže svoju činnosť ukončiť do 10 iterácií.

Ďalej sme aproximovali funkciu na výpočet adaptívnych prahových hodnôt a použili vo variante SF dekodéra. Pre niektoré nastavenia tohto výpočtu dokázal dekodér úspešne dekódovať do 5 iterácií.

Pre uvedené dekodéry sme okrem počtu iterácií dekódovania otestovali aj trendy vývoja DFR pre menšie veľkosti blokov s vyššími počtami chýb. Podobnú metodológiu na vyhodnotenie DFR použili aj autori [9] v prípade SF dekodéra.

Nadväzujúce práce sa môžu zamerať na nájdenie lepšieho spôsobu aproximácie výpočtu adaptívnych prahových hodnôt, adaptáciou black-gray (BG) dekodéra pre binárne QC-MDPC kódy a jeho variantov na použitie s QC-MDPC kódmi nad GF(4) či na skúmanie odolnosti týchto dekodérov voči útokom, ako je napr. GJS útok.

Zoznam použitej literatúry

1. SHOR, P. W. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. USA: IEEE Computer Society, 1994, s. 124–134. SFCS '94. ISBN 0818665807. Dostupné z DOI: 10.1109/SFCS.1994.365700.
2. NIST (ed.). *Post-Quantum Cryptography Standardization*. Dostupné tiež z: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.
3. MCELIECE, R. J. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*. 1978, roč. 44, s. 114–116.
4. MISOCZKI, Rafael, TILLICH, Jean-Pierre, SENDRIER, Nicolas a BARRETO, Paulo S. L. M. MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes. In: *2013 IEEE International Symposium on Information Theory*. 2013, s. 2069–2073. Dostupné z DOI: 10.1109/ISIT.2013.6620590.
5. ARAGON, N. et al. (ed.). *Bike: Bit Flipping Key Encapsulation*. 2022. Dostupné tiež z: https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.
6. NIEDERREITER, H. Knapsack-type cryptosystems and algebraic coding theory, Probl. *Control and Inform. Theory*. 1986, roč. 15.
7. GALLAGER, R. G. *Low-density parity-check codes*. 1963. Diz. pr. M.I.T.
8. GUO, Qian, JOHANSSON, Thomas a STANKOVSKI, Paul. A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors. In: CHEON, Jung Hee a TAKAGI, Tsuyoshi (ed.). *Advances in Cryptology – ASIACRYPT 2016*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, s. 789–815. ISBN 978-3-662-53887-6.
9. BALDI, Marco, CANCELLIERI, Giovanni, CHIARALUCE, Franco, PERSICHETTI, Edoardo a SANTINI, Paolo. Using Non-Binary LDPC and MDPC Codes in the McEliece Cryptosystem. In: *2019 AEIT International Annual Conference (AEIT)*. 2019, s. 1–6. Dostupné z DOI: 10.23919/AEIT.2019.8893339.
10. VASSEUR, Valentin. *Post-quantum cryptography: a study of the decoding of QC-MDPC codes*. 2021. Dostupné tiež z: https://theses.hal.science/tel-03254461. Theses. Université de Paris.
11. HILL, R. *A First Course in Coding Theory*. Clarendon Press, 1986. Oxford Applied Linguistics. ISBN 9780198538035.

12. ARAGON, N. et al. (ed.). *Bike: Bit Flipping Key Encapsulation*. 2020. Dostupné tiež z: https://bikesuite.org/files/round2/spec/BIKE-Spec-2020.02.07.1.pdf.

13. CHAULET, Julia. *Etude de cryptosystèmes à clé publique basés sur les codes MDPC quasi-cycliques*. 2017. Dostupné tiež z: https://theses.hal.science/tel-01599347. Theses. Université Pierre et Marie Curie - Paris VI.

14. FISHER, Ronald Aylmer a YATES, Frank. *Statistical tables for biological, agricultural and medical research*. 3rd ed., rev. and enl. London: Oliver a Boyd, 1948. ISBN 0-02-844720-4.

15. KNUTH, Donald E. *The art of computer programming*. 3rd ed. Reading, Mass.: Addison-Wesley, 1997. ISBN 0-201-89684-2.

Prílohy

A Štruktúra elektronického nosiča

/mdpc-gf4

· Priečinok obsahujúci implementáciu a jej dokumentáciu.

/process-data

· Priečinok so skriptom na spracovanie dát a so všetkými dátami zozbieranými podľa popisu uvedeného v kapitole 3.4.

/main.py

· Skript použitý na spracovanie dát a na vykreslenie grafov. Všetky grafy v tejto práci boli vytvorené s použitím tohto skriptu.

dp_vavro.pdf

· Dokument obsahujúci diplomovú prácu.

Implementácia je dostupná aj na `https://github.com/tj314/mdpc-gf4-pure-c`. Skript na spracovanie dát, ako aj všetky dáta, ktoré spracováva sú dostupné na `https://github.com/tj314/mdpc-gf4-data-analysis`.