Vizualizace vybraných moderních šifrovacích algoritmů

Milan Janovič

Bakalářská práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Milan Janovič**
Osobní číslo:	**A18047**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Visualizace vybraných moderních šifrovacích algoritmů**
Téma práce anglicky:	**Visualization of Selected Modern Encryption Algorithms**

Zásady pro vypracování

1. Nastudujte a popište problematiku moderní kryptografie.
2. Vyberte vhodné algoritmy moderní kryptografie a nastudujte je.
3. Zvolte vhodné implementační a visualizační prostředky.
4. Proveďte samotnou implementaci a visualizaci vybraných algoritmů.
5. Vhodně vyhodnoťte a reprezentujte výsledky.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. PIPER, F. C. a Sean MURPHY. Kryptografie. Praha: Dokořán, 2006. Průvodce pro každého. ISBN 80-736-3074-5.
2. SINGH, Simon. Kniha kódů a šifer: Utajování od starého Egypta po kvantovou kryptografii. Praha: Dokořán, 2003. Aliter (Argo: Dokořán): Dokořán). ISBN 80-86569-18-7.
3. OULEHLA, Milan a Roman JAŠEK. Moderní kryptografie. [Praha]: IFP Publishing, 2017, 186 s. ISBN 9788087383674.
4. PAAR, Christof a Jan PELZL. Understanding cryptography: a textbook for students and practitioners. Berlin: Springer, 2010, xviii, 372 s. Dostupné z: doi:9783642041013

Vedoucí bakalářské práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **3. prosince 2021**
Termín odevzdání bakalářské práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne …….

podpis studenta

ABSTRAKT

Táto bakalárska práca sa zaoberá implementáciou webovej stránky pomocou jazyka Python 3.x a framework-u Django. V teoretickej časti práca všeobecne analyzuje a na príkladoch popisuje problematiku modernej kryptografie a vybraných algoritmov modernej kryptografie. V praktickej časti práca na základe získaných poznatkov navrhuje a v neposlednom rade aj realizuje webovú stránku, ktorá implementuje vybrané algoritmy a umožňuje ich vizualizáciu "krok po kroku". Týmto docielime ich rýchlejšie a hlavne jednoduchšie pochopenie. Práca v praktickej časti taktiež popisuje štruktúru samotnej aplikácie, taktiež uvádza niekoľko návrhov na jej zlepšenie a stručne vyhodnocuje možnosti aplikácie práce v praxi.

Klíčová slova: kryptografia, algoritmus, webová stránka, vizualizácia, Python

ABSTRACT

This bachelor thesis deals with implementation of web page using programming language Python version 3.x and Django framework. In the theoretical part the thesis analysis and describes main issues and principles of modern cryptography and chosen algorithms of modern cryptography. In the practical part the thesis, based on the acquired knowledge, designs and creates the web page which implements the chosen algorithms and allows their visualisation in "step-by-step" manner. In the practical part the thesis also describes structure of the web page, gives a couple of suggestions for improving the web page and briefly summarizes options for using the web page in practise.

Keywords: cryptography, algorithm, web page, visualisation, Python

POĎAKOVANIE

Chcel by som sa poďakovať pánovi Ing. Petrovi Žáčkovi, Ph.D. za jeho ochotu, rady, trpezlivosť, pohotovú komunikáciu, dôveru a v neposlednom rade jeho odbornosť v oblasti, ktorú práca rieši, bez ktorej by tvorba tejto práce nebola možná.

PREHLÁSENIE

Prehlasujem, že odovzdaná verzia bakalárskej práce a elektronická verzia nahraná do IS/STAG sú totožné.

OBSAH

ÚVOD

Ľudia už od počiatkov prvých civilizácií pociťovali potrebu utajiť isté, citlivé informácie pred inými, nepovolanými osobami. V počiatkoch sa jednalo len o jednoduché slovné správy, ktoré mohli obsahovať informácie o pohyboch vojsk, mieste stretnutia, prípadne sa mohlo jednať aj o milostné listy. V každom prípade šifrovanie umožňovalo utajiť obsah týchto správ a v ideálnom prípade absolútne znemožniť ich rozlúštenie nepovolanou osobou. Kryptológia teda sprevádza civilizácie po veľmi dlhú dobu.

Avšak tak ako sa vyvíjali a zdokonaľovali civilizácie v priebehu dejín, rovnako sa musela vyvíjať aj kryptológia a jej metódy. Kryptológia preto prešla, od dnes už primitívnych substitučných šifier, až po veľmi sofistikované metódy zabezpečenia informácií aké ponúka napríklad biometria alebo kvantová kryptografia.

Nároky na zabezpečenie neustále rastú a pokrok v oblasti zabezpečenia je teda nielen žiadaný, ale aj nutný. Aj keď si to s najväčšou pravdepodobnosťou väčšina ľudí neuvedomuje, ich každodenný život sprevádza kryptografia a jej metódy zabezpečenia. Príkladom môže byť už len fakt, že takmer každý z nás považuje niečo ako platbu alebo výber kartou, prípade komunikáciu cez internet za samozrejmosť. Avšak obe tieto činnosti ako aj mnohé iné si vyžadujú zabezpečenie a šifrovanie tejto komunikácie.

Táto práca sa preto snaží v prvej kapitole teoretickej časti stručne oboznámiť čitateľa so základnými kryptologickými pojmami. V nasledujúcej kapitole, na príkladoch objasní, vývoj kryptológie a jej metód. V tretej kapitole teoretickej časti potom uvádza čitateľa do problematiky modernej kryptológie a v neposlednom rade taktiež teoreticky objasňuje vybrané algoritmy modernej kryptológie. V praktickej časti sa práca zaoberá implementáciou týchto algoritmov na webovej stránke, ktorá bola vytvorená ako jej súčasť. Teda bližšie opisuje prostriedky použité na vývoj webovej stránky ako aj jej štruktúru.

Práca by teda mala slúžiť na lepšie pochopenie kryptografie a jej metód od jej počiatkov až po v súčasnosti používané metódy zabezpečenia. Taktiež by mala priblížiť čitateľovi princípy fungovania moderných šifrovacích algoritmov na vybraných algoritmoch, ktorých vizualizácia je spracovaná "krok po kroku" a bude teda, ako pevne veríme, pre čitateľa jednoduchšie dané princípy a algoritmy pochopiť. A to najmä vzhľadom k tomu, že nami vybrané algoritmy nie sú, v súčasnosti, tak známe ako bežne používané štandardy a ich názorné, vizuálne spracovanie, o ktoré usilujeme, nie je dostupné. Aj napriek tomu, že sú veľmi moderné a v súčasnosti často používané. Čo sú hlavné dôvody prečo sme sa rozhodli

zvoliť práve tieto algoritmy. Pri štúdiu nami zvolených algoritmov je teda nutné vychádzať takmer výhradne z technickej dokumentácie, ktorá je často pre čitateľa oveľa ťažšie pochopiteľná a v prípade niektorých algoritmov aj ťažko dostupná.

Taktiež je nutné podotknúť, že implementácia všetkých troch algoritmov sa obmedzuje na spracovanie jedného bloku, teda nevyužíva režimy činnosti. Samotná stránka, ktorá algoritmy implementuje je v anglickom jazyku, a to preto, že preklad veľkého množstva pojmov z oblasti IT býva často náročný a nepresný.

I. TEORETICKÁ ČASŤ

1 ZÁKLADNÉ POJMY

Predtým než budeme našu pozornosť venovať princípom klasickej a modernej kryptografie ako aj samotným algoritmom, je nutné objasniť základné kryptografické pojmy a koncepty. Táto kapitola sa preto bude venovať ich čo možno najstručnejšiemu objasneniu, pre potreby nasledujúcich kapitol.

1.1 Kryptológia

Kryptológia je vo všeobecnosti označovaná ako veda o utajení správ a ochrane prenosu informácie. V súčasnosti zastrešuje tri pod odbory – kryptografiu, kryptoanalýzu a steganografiu. [1]

Kryptografia je technický obor používajúci matematické metódy za účelom návrhu a tvorby šifrovacích systémov. Pod pojmom šifrovací systém rozumieme systém, ktorý slúži na "premenu" informácie za účelom učiniť ju nezrozumiteľnou pre tretiu stranu. Taktiež sa zaoberá ich následnou inováciou, vývojom a v neposlednom rade aj ich štandardizáciou. Dôležitým faktom je že, kryptografia navrhuje šifrovacie protokoly tak, že informáciu zašifrujeme, ale ďalej ju už neskrývame. [1]

Kryptoanalýza by sa dala označiť za opak, už spomínanej kryptografie, kedže sa snaží nachádzať slabiny, "zadné vrátka" a vo všeobecnosti odhaľovať nedokonalosti šifrovacích algoritmov. Zaoberá sa teda testovaním odolnosti už známych ako aj novo vzniknutých algoritmov a možnosťami útokov na tieto algoritmy za účelom ich prelomenia a získania tajnej informácie. [1]

Mohlo by sa teda zdať nelogické zverejňovať princípy fungovania šifrovacích protokolov (systémov) a algoritmov v nich použitých. Ponúka sa teda otázka, či by sa bezpečnosť týchto algoritmov nezvýšila ich utajením. Avšak opak je pravdou, pretože tajný algoritmus by potom bolo možné testovať len vo veľmi obmedzenej miere. Kryptoanalýza a jej techniky testovania sú preto nevyhnuté pre zabezpečenie čo možno najväčšej miery bezpečnosti a uistenia, že daný algoritmus je odolný voči útokom. [2]

Steganografia je časť kryptológie, ktorá sa na rozdiel od kryptografie nesnaží danú informáciu zašifrovať a teda znemožniť získanie jej obsahu v prípade, že zašifrovanú informáciu získame. Cieľom steganografie je skryť samotnú existenciu danej tajnej informácie (správy). Medzi jej najznámejšie metódy patrí napríklad použitie neviditeľných atramentov, ale aj mnohé iné modernejšie metódy úpravy obrazu, zvuku tak aby zmena

nebola viditeľná alebo počuteľná, ale aby toto médiu obsahovalo tajnú informáciu. V praxi je taktiež častá kombinácia kryptografických a steganografických metód, teda informáciu najskôr zašifrujeme a následne skryjeme aby sme takto docielili ešte väčšiu bezpečnosť. [1]

1.2 Kryptografické termíny

Otvorený text (OT) je pôvodný text správy pred zašifrovaním. [1]

Šifrový text (ŠT) je text správy po zašifrovaní, zvoleným šifrovacím algoritmom (systémom). [1]

Kód je často nesprávne chápaný a zamieňaný za šifru. Šifra správu (informáciu) mení do "nečitateľnej" podoby aby sa zamedzilo tretej strane získať obsah správy, zatiaľ čo kód správu mení za účelom jej prenosu cez komunikačné médium prípadne médiá. [1]

Kľúč je jedným zo vstupných parametrov šifrovacieho ako aj dešifrovacieho procesu, nie je teda možné bez neho správu zašifrovať resp. dešifrovať. [1]

Kľúčový priestor prestavuje množinu všetkých možných kľúčov v danom šifrovacom systéme. [1]

Heslo je podobne ako v prípade kódu často nesprávne interpretované a zamieňané za kľúč. Pričom kľúč je jedným zo vstupov šifrovacieho systému a teda ovplyvňuje výstup z tohto systému. Heslo slúži na autentizáciu na základne iného zdieľaného tajomstva. [1]

Abeceda textu zahŕňa množinu znakov otvoreného a šifrového textu. [1]

Štatistická charakteristika jazyka je vlastnosť jazyka, ktorá charakterizuje frekvenciu výskytu konkrétneho znaku v texte daného jazyka. Túto vlastnosť využíva kryptoanalytická technika nazývaná diferenciálna kryptoanalýza, ktorá sa opiera o fakt, že pri substitúcií (zámene) znaku OT za vždy rovnaký znak ŠT sa prenášajú štatistické vlastnosti abecedy OT do abecedy ŠT. To znamená, že pokiaľ vieme, že OT bol napísaný napríklad v anglickom jazyku možeme si dohľadať tabuľku výskytu jednotlivých znakov v anglickom jazyku a následne vykonať frekvenčnú analýzu ŠT – zistíme frekvenciu výskytu znakov v ŠT. Následne porovnáme frekvenciu výskytu znakov v OT a ŠT. [3]

Pokiaľ sa v OT najčastejšie vyskytuje písmeno "E" a následne písmeno "T" (platí pre anglický jazyk) a v ŠT napríklad písmeno "H" a následne písmeno "Y" možeme predpokladať, že "E" bolo substituované za "H" a "T" za "Y". Týmto spôsobom možeme následne pomocou znalosti jazyka OT a korekcie možných štatistických anomálií pomerne jednoducho zistiť obsah ŠT.

1.3 Šifrovacie systémy

Nasledujúca podkapitola na elementárnej úrovni popisuje typy šifrovacích systémov, princípy ich fungovania, ako aj ich výhody a nevýhody.

1.3.1 Symetrické šifrovacie systémy

Využívajú symetrické šifrovacie algoritmy. Všetky šifrovacie systémy boli až do roku 1976 založené práve na symetrických šifrovacích algoritmoch. Princípom je existencia dvoch "strán", ktoré používajú šifrovaciu a dešifrovaciu metódu (algoritmus) pre ktorú zdieľajú jeden (a ten istý) tajný kľúč. Tento typ šifrovacích systémov a algoritmov je aj napriek ich veku stále veľmi rozšírený a využívaný veľkej miere. [2]

Nevýhodou týchto systémov je nutnosť zdieľať tajný kľúč medzi účastníkmi komunikácie vopred ešte pred zahájením komunikácie samotnej. Ako aj fakt, že s rastúcim počtom účastníkov komunikácie narastá počet potrebných kľúčov. Výhodou symetrických systémov sú ich výrazne nižšie výpočtové nároky pri tvorbe kľúčov ako aj jednoduchosť šifrovacích algoritmov (v porovnaní s asymetrickými systémami).

1.3.2 Asymetrické šifrovacie systémy

V roku 1976 Whitefield Diffie, Martin Hellman a Ralph Merkle prestavili úplne iný typ šifrovacích algoritmov – asymetrické šifrovacie algoritmy alebo taktiež algoritmy s verejným kľúčom (VK). [2]

Jedná sa o algoritmy, ktoré na rozdiel od symetrických šifrovacích algoritmov na zašifrovanie využívajú jeden kľúč – VK, ktorý je voľne dostupný a na dešifrovanie potom kľúč druhý tzv. privátny kľúč (PK), ktorý by mal poznať výhradne jeho vlastník. [1] Je známe, že existuje vzťah medzi VK a PK, ale je založený na princípe tzv. jednocestných funkcií. Za jednocestné funkcie označujeme funkcie o ktorých platí, že je možné ich v rozumnom (polynomiálnom) čase realizovať len v jednom smere. Teda je pomerne jednoduché vypočítať výstup zo vstupu (napríklad za pár minút), ale naopak je veľmi náročné získať v rozumnom čase vstup zo známeho výstupu (napríklad za niekoľko rokov). Exituje veľké množstvo jednocestných funkcií, príkladom môže byť problém faktorizácie čísel na ktorom je založený algoritmus RSA. [2]

Tento typ systémov eliminuje hlavné nevýhody systémov symetrických. Teda nie je potrebné kľúč zdieľať pred začiatkom komunikácie a rovnako nie je pri zvýšenom počte účastníkov komunikácie potrebné generovať väčší počet kľúčov – postačuje jeden kľúčový

pár (VK a PK). Tieto výhody, ale majú svoju cenu. Konkrétne sa jedná o zvýšenú výpočtovú zložitosť algoritmov samotných, ako aj komplikované matematické vzťahy medzi kľúčmi. Týmto logicky stúpa čas potrebný na šifrovanie/dešifrovanie pomocou týchto algoritmov a teda ich rýchlosť je nižšia v porovnaní s algoritmami symetrickými. Rovnako sa limituje objem dát, ktoré sme schopný zašifrovať. Poslednou, avšak o nič menej dôležitou nevýhodou je aj nutnosť dobre zabezpečiť VK a to najmä pred jeho zámenou.

1.3.3 Hybridné systémy

Hybridné systémy spájajú "to najlepšie z oboch svetov". Asi najpoužívanejším variantom hybridného prístupu je postup pri ktorom na zašifrovanie dát použijeme symetrický kľúč a teda aj symetrické techniky šifrovania (algoritmy) a následne tento kľúč zašifrujeme pomocou VK, teda využijeme asymetrické techniky šifrovania. Tento mechanizmus predstavuje akési "zapuzdrenie" kľúča. Motiváciou pre používanie tohto typu systémov je kombinácia ich silných stránok. V tomto prípade sa snažíme využiť jednoduchosť a rýchlosť symetrických systémov, ako aj to, že umožňujú šifrovať väčšie objemy dát. A súčasne sa snažíme eliminovať potrebu zdieľať symetrický kľúč pred začiatkom komunikácie samotnej, pomocou asymetrického kľúčového páru. [4]

2 KLASICKÁ KRYPTOGRAFIA

Kapitola venujúca sa klasickej kryptografií stručne zhrnie historický vývoj kryptografie od jej počiatkov po dnešnú (modernú) kryptografiu, ktorej bude venovaná samostatná kapitola. Historici sa domnievajú, že kryptografia je stará ako písmo samo. Jej počiatky siahajú až do roku 2000 pred Kristom kedy Egypťania vyzdobovali hrobky panovníkov hieroglyfmi. Bez ohľadu nato, kedy v skutočnosti kryptografia vznikla, stala sa neoddeliteľnou súčasťou komunikácie civilizácií v priebehu dejín. Bola neodmysliteľnou súčasťou napríklad diplomatickej alebo vojenskej komunikácie a dalo by sa povedať, že bola využívaná hlavne za účelom ochrany obsahu citlivých dokumentov pri ich transporte z jedného miesta na druhé. V tomto období bola založená primárne na transformácií znakov OT do na prvý pohľad nezrozumiteľného a bezvýznamového zhluku písmen alebo znakov. Za "náhodným" rozložením sa ale v skutočnosti skrývala istá utajovaná logika, ktorá v tomto období do istej miery plnila funkciu dnešných tajných kľúčov. [5] [6]

2.1 Transpozičné šifry

Ako už napovedá sémantika slova "transpozícia" jedná sa o šifry, ktoré sú založené na princípe zmeny pozície jednotlivých znakov OT, čím vznikne ŠT. Tento typ šifrovania využívala už spartská civilizácia, okolo roku 5 pred Kristom, ktorá používala kryptografické zariadenie nazývané *SCYTALE*. Jednalo sa o valec okolo ktorého sa niekoľkokrát vedľa seba omotal tenký kus pergamenu (alebo kože) na ktorý bola následne napísaná správa (OT). Akonáhle bola táto tenká páska z valca odmotaná prestavovala len dlhú sekvenciu nesúvisiacich znakov až do bodu kedy bola spätne namotaná na valec s presne rovnakým priemerom ako valec pôvodný. V priebehu dejín sa samozrejme vyskytli aj iné typy transpozičných šifier, kedy ŠT vznikal z OT pomocou rozdielneho spôsobu zápisu OT a ŠT. Napríklad možeme ŠT vytvoriť tak, že OT zapíšeme po stĺpcoch namiesto riadkov, prípadne môžeme zameniť poradie riadkov/stĺpcov alebo riadkov a stĺpcov súčasne. Aj napriek tomu, že tento typ spôsob šifrovania dát bol objavený civilizáciami už v staroveku, je dodnes súčasťou mnohých moderných šifrovacích systémov (v kombinácií s inými spôsobmi šifrovania). [7]

2.2 Substitučné šifry

Rovnako ako v prípade transpozičných šifier, substitučný spôsob šifrovania objavili civilizácie už v staroveku, ale aj napriek tomu je súčasťou šifrovacích systémov dodnes.

Princíp fungovania týchto šifier možeme opäť odvodiť od sémantiky slova "substitúcia", čiže nahradenia/zámeny napríklad znaku abecedy OT za znak abecedy ŠT (ktorá samozrejme musí byť nejakým spôsobom rozdielna od abecedy OT). [2] [7]

2.2.1 Monoalfabetické substitučné šifry

Monoalfabetické substitučné šifry sú najjednoduchšie zo skupinu substitučných šifier. Využívajú tú istú abecedu pre OT a ŠT. Ako prvý tento typ šifrovania využil Julius Caesar, ktorý údajne využíval posun šifrovej abecedy o tri znaky "ďalej" v abecede oproti abecede OT, pričom posledné tri znaky sa stali znakmi prvými. [8]

Pokiaľ by sme teda pre OT použili abecedu "ABCDEFGHIJKLMNOPRSTUVXYZ" abeceda ŠT by vyzerala nasledovne "DEFGHIJKLMNOPRSTUVXYZABC". Slovo "Caesar" by sme potom zašifrovali ako "FDHVDU".

Samozrejme posun abecedy môžeme vykonať o ľubovoľný počet znakov, nie len o 3. Tento typ šifier potom označujeme ako ROT X, kde *x* je číslo označujúce počet znakov o ktorý abecedu posunieme. [9] Napr. ROT13 by predstavoval posun abecedy o 13 znakov, čiže akoby zrkadlový obraz abecedy pôvodnej.

Šifry typu ROT teda šifry s jednoduchým lineárnym posunom, ale pomerne jednoducho podliehajú diferenciálnej kryptoanalýze, pričom platí, že s rastúcou dĺžkou ŠT rastie aj pravdepodobnosť úspechu dešifrovania, pretože čim je text dlhší tým výraznejšie sa v ňom prejaví štatistická charakteristika výskytu znakov v danom jazyku. Existuje množstvo spôsobov akými možnosti tieto šifry "posilniť" a zvýšiť ich zložitosť za účelom zlepšenia ich bezpečnosti. Príkladom môže byť využitie bigramového šifrovania (šifrovanie po dvojiciach znakov) v kombinácií s odstránením pôvodných medzier čo výrazne zvyšuje náročnosť dešifrovania. Prípadne možeme využiť tzv. Homofónne šifrovanie, ktoré spočíva prideľuje niektorým znakom abecedy OT viacero znakov ŠT. Teda by sme pre jeden a ten istý znak abecedy OT mohli použiť viacero znakov abecedy ŠT. Čím sa zníži počet výskytov konkrétneho znaku v ŠT, čo logicky spôsobí skreslenie štatistickej charakteristiky jazyka a tým sťaží použitie diferenciálnej kryptoanalýzy. Avšak aj v tomto prípade by si útočník mohol s pomocou znalosti jazyka OT, dôvtipu, korekcie chýb a dedukcie vytvoriť zoznam všetkým párov medzi abecedou OT a ŠT. [3]

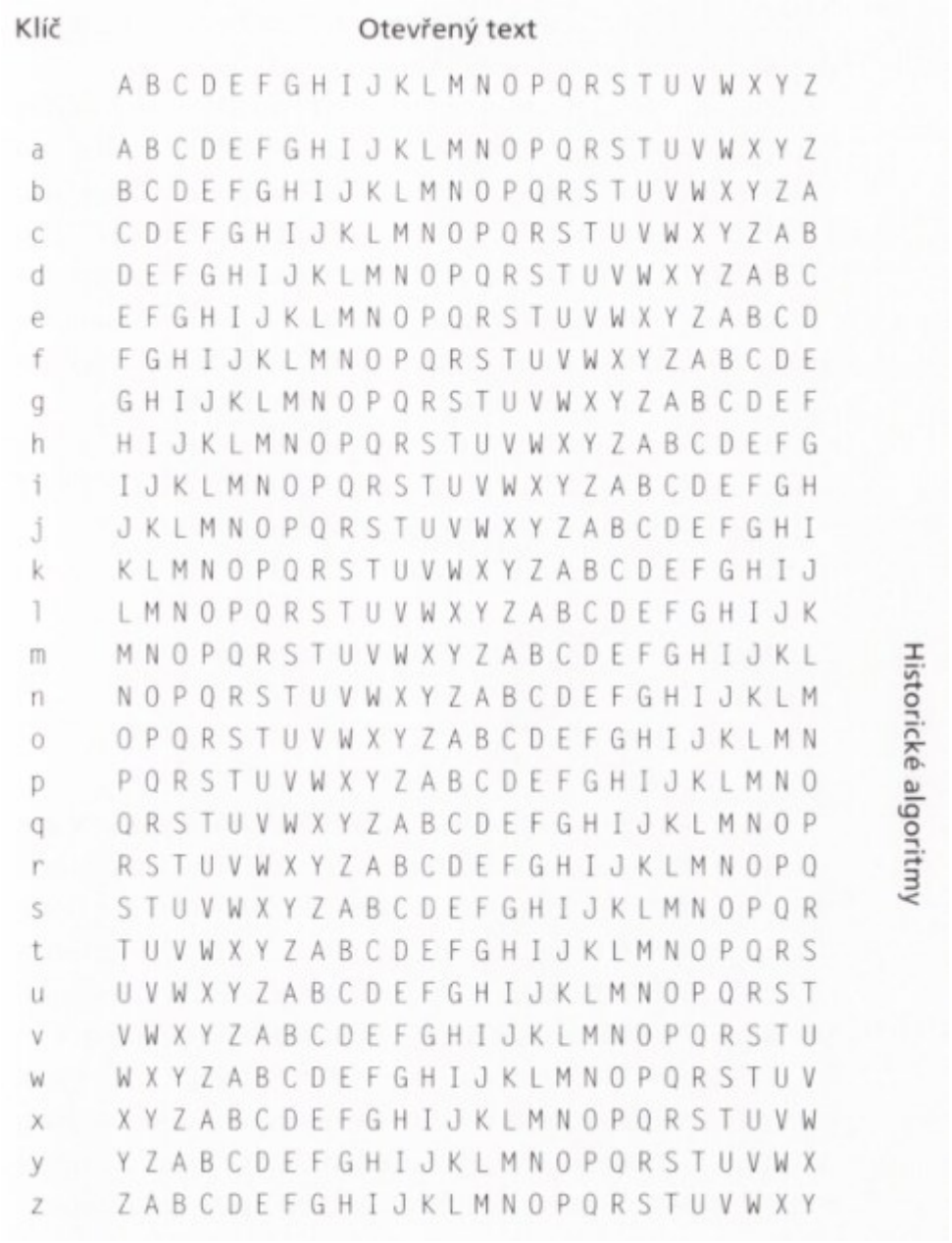
2.2.2 Polyalfabetické substitučné šifry

V predchádzajúcej kapitole sme uviedli niekoľko spôsobov akými možno "rozhodiť" štatistickú charakteristiku jazyka a tým sťažiť použitie diferenciálnej kryptoanalýzy pri dešifrovaní. Ďalšou možnosťou je použitie šifier ktoré pre abecedu OT využívajú hneď niekoľko abecied ŠT – polyalfabetické substitučné šifry. V prípade týchto šifier platí, že ten istý znak OT môže zastupovať hneď niekoľko znakov ŠT (a naopak), v závislosti na rôznych faktoroch. [3]

Vigenerova šifra

Aj napriek tomu, že koncept polyalfabetických šifier popísal talian Leon Batista Alberti už v 60 rokov 15. storočia. Asi najznámejším príkladom manuálnej polyalfabetickej šifry je Vigenerova šifra pomenovaná po francúzskom kryptografovi Blaise de Vigenerovi. Alberti síce vykonal asi najvýznamnejší posun v kryptografií vtedajšieho tisícročia, ale nezrealizoval svoj koncept. Ten zrealizovali až jeho pokračovatelia, medzi najznámejších patrí už spomínaný Vigener. Vigener koncept svojej šifry publikoval v už v roku 1586, ale širšieho rozšírenia sa dočkala až o 200 rokov neskôr. Šifra v priebehu jej používania dostala dokonca prezývku "Le Chiffre Indechiffrable" alebo teda "Neprelomiteľná šifra", kedže prelomiť ju sa podarilo až Babbagovi a Kasiskému v polovici 19 storočia. [3] [10]

Šifra pozostáva z 26 abecied zapísaných pod seba do tzv. Vigenerovho štvorca. Jedná sa vlastne o 26 pod seba zapísaných abecied s lineárnym posunom, počínajúc posunom o 1. Nad a vedľa tohoto štvorca je potom zapísaná abeceda OT a kľúča. Pričom abeceda zapísaná nad štvorcom pri šifrovaní predstavuje znak OT a abeceda zapísaná do stĺpca vedľa štvorca potom znak kľúča. Za znak ŠT potom volíme znak v tabuľke v rovnakom stĺpci ako znak OT a súčasne v rovnakom riadku ako znak použitého kľúča, ktorý je v prípade potreby opakovaný do potrebnej dĺžky. [3] [10]



Obrázok 1– Vigenerov štvorec [3]

Pokiaľ by sme teda podľa tabuľky na **Obrázok 1** zašifrovali napríklad slovo “PARALYZA”, pomocou kľúčového slova “KLUC” šifrovanie by vyzeralo nasledovne:

Otvorený text	P	A	R	A	L	Y	Z	A
Kľúč	K	L	U	C	K	L	U	C
Šifrový text	Z	L	L	C	V	J	T	C

Ako môžeme vidieť v slove “PARALYZA” sa písmeno “A” vyskytuje až trikrát pričom dvakrát sme ho zašifrovali ako “C“ a raz ako “L“, rovnako si možeme všimnúť, že písmeno “L“ sme v šifrovom texte použili pre dve rozdielne písmená “A“ a “R“. Čo ako už bolo

spomínané výrazne sťažuje aplikáciu diferenciálnej kryptoanalýzy, ako aj prelomenie šifry vo všeobecnosti.

2.2.3 Nomenklátory a klamače

Nomenklátory a klamače sú ďalším prostriedkom používaným na sťaženie prelomenia substitučných šifier. Pod pojmom nomenklátor rozumieme špeciálny symbol (kódové slovo), ktoré sa priraďovalo často sa vyskytujúcim slovám v OT. Mohlo sa jednať napríklad o mená osobností a pod. Klamače sú podobne ako nomenklátory špeciálne symboly, prípadne znaky, avšak nemajú žiadnu informačnú hodnotu (význam) a do textu sú pridané len z dôvodu zmätenia prípadného nepriateľa. [11]

2.2.4 Polygrafické substitučné šifry

V doteraz rozoberaných šifrách prebiehala substitúcia jedného znaku OT za jeden znak ŠT. Síce sme spomenuli, že v prípade homofónnych substitúcií a polygrafických substitúcií sa jeden znak OT mohol premietnuť do viacerých, rôznych znakov v ŠT, ale šifrovanie vždy prebiehalo znak po znaku.

V prípade polygrafických substitúcií hovoríme o tzv. n-grafických substitúciách, kde $n \geq 2$. Čiže znaky nešifrujeme jednotlivo, ale minimálne po dvojiciach, vtedy hovoríme o tzv. bigramových šifrách. [3] [9]

Príkladom bigramovej šifry je napríklad šifra “Playfair”. V jej prípade sa využíva mriežka 5x5 do ktorej vpisujeme na začiatok kľúčové slovo a následne ostatné znaky, ktoré sa v kľúčovom slove nenachádzali. Rovnako si možeme všimnúť, že je nutné jeden znak vynechať, kedže mriežku 5x5 môže obsadiť maximálne 25 znakmi, pričom v abecede ich máme 26. Pre Český ako aj Slovenský jazyk sa tento problém rieši vynechaním písmena “J”. Písmeno “J” a “I” potom zastupuje písmeno “I” v závislosti na kontexte správy. Šifrovanie následne závisí od oboch znakov, ktoré vstupujú do šifrovacieho procesu. Pokiaľ ležia v rovnakom riadku tabuľky nahradíme oba znaky znakmi o jedno miesto vpravo v tabuľke, pokiaľ ležia v rovnakom stĺpci nahradíme ich znakmi o jedno miesto nižšie v tabuľke, pokiaľ neplatí ani jedna z prechádzajúcich podmienok prvé písmeno je nahradené znakom v rovnakom riadku a súčasne v stĺpci druhého písmena. Podobne druhé písmeno je nahradene písmenom v rovnakom riadku a súčasne v stĺpci prvého písmena. Pokiaľ pri prvých dvoch podmienkach nastane situácia, že posun o znak vpravo, prípadne nižšie nie je

možný z dôvodu že sa nachádzame na konci riadku alebo stĺpca prechádzame akoby v kruhu opätovne na začiatok riadku alebo stĺpca. [3]

2.3 Hybridné šifry

Podobne ako v prípade hybridných šifrovacích systémov kedy sme sa pomocou kombinácie princípov symetrickej a asymetrickej kryptografie snažili docieliť vytvorenie čo najlepšieho systému. V prípade hybridných šifier sa kombináciou substitúcie a následnej transpozície (alebo opačného postupu) snažíme docieliť vytvorenie odolnejšej šifry.

Hovoríme teda o metóde akéhosi "skladania" šifier, kedy kombináciou samostatne pomerne slabých šifrovacích algoritmov vzniká algoritmus výrazne silnejší. [3]

3 MODERNÁ KRYPTOGRAFIA

Existuje množstvo míľnikov o ktorých by sme mohli povedať, že vytyčovali akýsi prechodový bod medzi klasickou a modrenou kryptografiou. Jedným z týchto momentov môže byť objavenie Vernamovej šifry v roku 1917, ktorá bola neskôr upravená do podoby ktorú poznáme ako "One-time Pad", ktorá bola označená za jedinú šifru poskytujúcu tzv. bezpodmienečnú bezpečnosť. Túto šifru aj to, čo to bezpodmienečná bezpečnosť vlastne je, rozoberieme ďalej v tejto práci. Ďalším a možno aj výstižnejším míľnikom je rok 1949 kedy Claude Elwood Shannon definoval, že sila algoritmu by mala spočívať v jeho matematickej zložitostí, nie na tajnostiach okolo neho. [12]

Doteraz rozoberané príklady šifrovacích systémov boli často založené na fakte, že prípadný útočník nepoznal princíp akým boli dáta zašifrované (princíp fungovania teda akoby plnil funkciu tajného kľúča) a po odhalení tohto princípu nebol šifrovací systém bezpečný a bol teda nepoužiteľný. Dnešné šifrovacie systémy a algoritmy v nich obsiahnuté nie sú tajné a ich špecifikácie ako aj princípy fungovania sú voľne dostupné. Bezpečnosť týchto algoritmov je potom postavená na predpoklade, že neexistuje počítač alebo počítačový systém ktorý by dokázal tento systém prelomiť v polynomiálnom čase. Hovoríme o tzv. "vypočtovej bezpečnosti" (z angl. "computational security"). Kryptologický systém je potom "výpočtovo bezpečný" pokiaľ najlepší známy algoritmus na jeho prelomenie vyžaduje aspoň t operácií. Táto definícia so sebou ale prináša hneď niekoľko otázok, najpodstatnejšou je otázka, ktorý algoritmus je vlastne najlepší algoritmus na prelomenie daného systému. A taktiež či neexistuje algoritmus o ktorom doposiaľ nevieme, ktorý je lepší, než najlepší známy algoritmus. Môžeme taktiež hovoriť o už spomínanej bezpodmienečnej bezpečnosti (z angl. unconditional security), kedy predpokladáme, že aj keby mal útočník neobmedzený a teda nekonečný výpočtový výkon, nedokázal by (bez znalosti kľúča) dešifrovať ŠT a teda prelomiť daný šifrovací systém. Bezpodmienečná bezpečnosť je však docieliteľná len v prípade Vernamovej šifry (v podobne One-time Pad), o ktorej si ďalej v tejto práci povieme že, má konkrétne nevýhody, kvôli ktorým nie je pre každodennú aplikáciu veľmi praktická. Preto sa v praxi snažíme o už spomínanú výpočtovú bezpečnosť a v princípe sa snažíme nájsť čo najlepší pomer medzi bezpečnosťou daného šifrovacieho systému a investovaným úsilím (čas, výpočtová sila a iné). [2] [12]

Veľmi výraznou črtou moderných šifrovacích systémov je to, že na rozdiel od klasických šifrovacích systémov nepracujú s jednotlivými znakmi OT, ale jednotlivými bitmi. Teda OT je pred zašifrovaním prevedený pomocou kódových tabuliek do sekvencie bitov a až potom

je ďalej spracovaný (zašifrovaný) šifrovacím systémom. Podľa toho akým spôsobom daný šifrovací systém spracováva vstupné dáta (OT) systémy delíme na blokové a prúdové. [3]

Blokové ako aj prúdové šifry (šifrovacie systémy) našli svoje špecifické uplatnenie. Prúdové šifry boli vo všeobecnosti považované za "malé" a rýchle, preto boli používané najmä v zariadeniach s obmedzenými výpočtovými zdrojmi (napríklad v mobilných telefónoch). Blokové šifry boli naopak využívané vo veľkej miere na šifrovanie na internete, kde predstavovali väčšinu šifrovacích procesov. Každopádne, ale existujú aj prúdové šifry používané na šifrovanie v rámci internetu napríklad RC4. [2]

3.1 Symetrické blokové šifrovacie systémy

Blokové šifrovacie systémy šifrujú celé bloky bitov vstupných dát naraz pomocou rovnakého kľúča. Výsledok šifrovania bitu v rámci ktoréhokoľvek bloku závisí od všetkých ostatných bitov v rámci daného bloku. Veľkosť týchto blokov je vopred definovaná pre danú šifru, ktorú chceme použiť. Väčšina šifier v praxi využíva bloky o veľkosti 128 bitov (16 bajtov) napríklad dobre známa šifra AES alebo bloky o veľkosti 64 bitov (8 bajtov) napríklad šifra DES. Pokiaľ by sme teda znaky OT kódovali do binárnej sústavy pomocou kódovej tabuľky ASCII, v ktorej každý znak predstavuje 8 bitovú (jedno bajtovú) sekvenciu, môžeme povedať, že v prípade šifry AES šifrujeme 16 znakov súčasne v rámci jedného bloku a v prípade šifry DES potom 8 znakov. [2] [3]

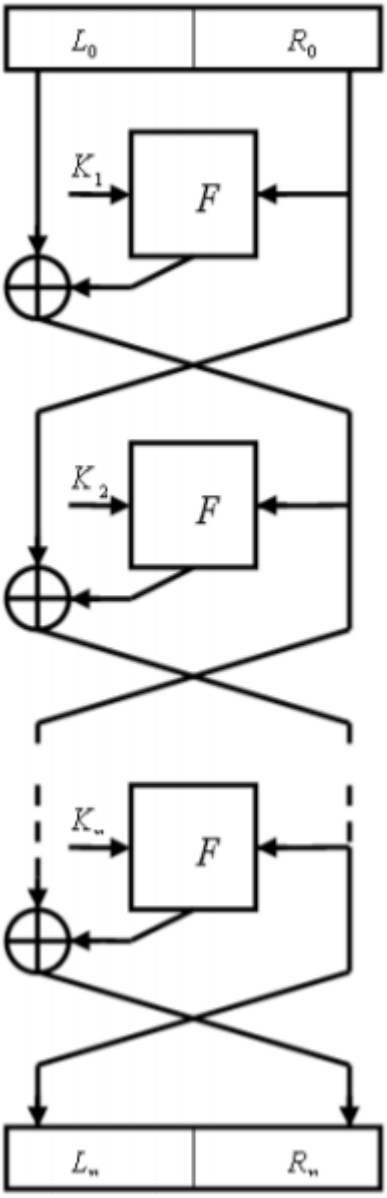
3.1.1 Feistelova štruktúra

Veľké množstvo blokových šifier využíva iteratívny prístup. Teda dáta OT rozdelené na bloky šifrujeme pomocou opakovanej aplikácie šifrovacej funkcie. Vstupom do konkrétnej iterácie je zväčša kľúč a výstup (časť ŠT) z predchádzajúcej iterácie. [13]

Veľké množstvo blokových šifier aplikuje tento iteratívny prístup pomocou Feistelovej štruktúry. Je nutné podotknúť, že sa jedná len o určitý spôsob spracovania dát, nie o konkrétny algoritmus. Princípom je rozdelenie OT na ľavú a pravú časť. Následne v závislosti na šifre, ktorá implementuje Feistelovu štruktúru je na pravú časť aplikovaná konkrétna funkcia danej šifry, s použitím kľúča špecifického pre danú iteráciu (tzv. podkľúča). Na takto upravenú pravú časť a ľavú (zatiaľ nepozmenenú) časť OT aplikujeme logickú funkciu XOR. Následne pravú a ľavú časť medzi sebou vymeníme a postup opakujeme. Teda z pôvodnej pravej časti sa v ďalšej iterácií stáva časť ľavá a z výstupu operácie XOR medzi pôvodnou ľavou časťou a upravenou pravou častou sa stáva časť pravá.

Rozdelenie OT na polovice (ľavú a pravú časť) môžeme nahradiť rozdelením na štyri, osem alebo aj viac častí. Takýto typ šifier označujeme ako zovšeobecnené Feistelove šifry. Počet iterácií je rovnako ako šifrovacia funkcia závislí po konkrétnej použitej šifre. [13] [14]

Schému Feistelovej štruktúry zobrazuje **Obrázok 2**, kde R označuje pravú časť, L ľavú časť, K podkľúč danej iterácie, F šifrovaciu funkciu a $\oplus$ logickú operáciu XOR.

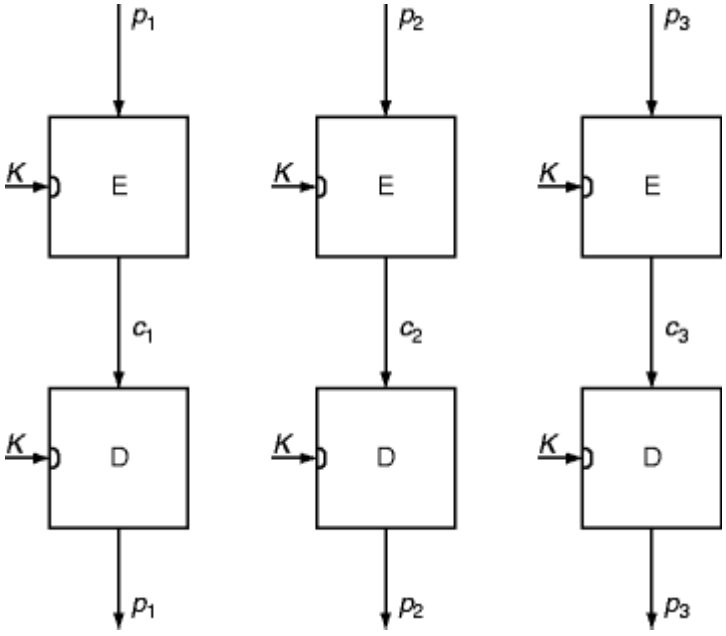


Obrázok 2 – Feistelova štruktúra [14]

3.1.2 Režimy činnosti blokových šifier

V prípade blokových šifier bolo nutné vyriešiť otázku ako zašifrovať bloky dát pomocou *n*-bitovej blokovej šifry, kedy bitová dĺžka OT nebola celočíselným násobkom *n*. Jedným z prístupov bolo doplnenie chýbajúcich bitov pomocou "0" bitov, tak aby výsledná bitová dĺžka bola násobkom *n* (tzv. padding). Alternatívnym prístupom bola štandardizácia režimov činnosti blokových šifier, ktorá umožnila šifrovanie aj blokov ktorých bitová dĺžka bola menšia ako *n*, respektíve teda OT, ktorých bitová dĺžka bola väčšia ako bitová dĺžka jedného bloku zvolenej blokovej šifry a zároveň nebola celočíselným násobkom *n*. [15]

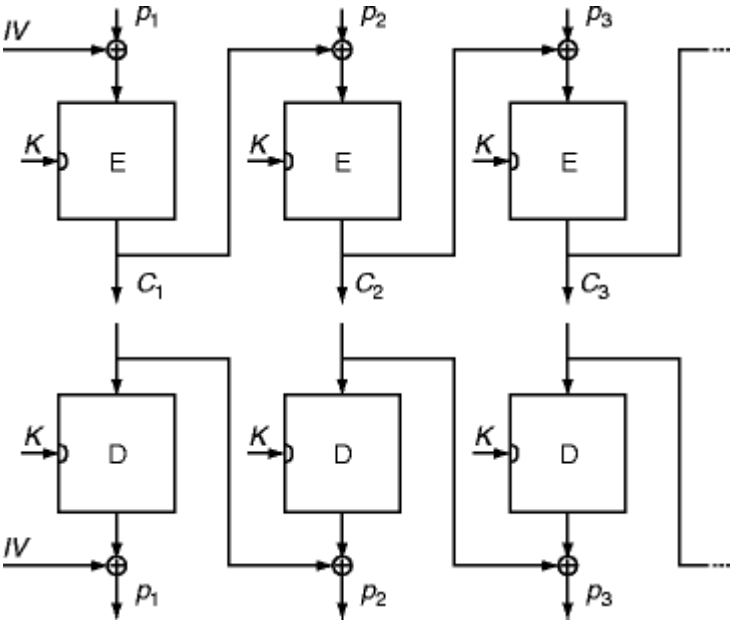
ECB (Electronic Code Book) režim je najjednoduchším režimom činnosti blokových šifier. V tomto režime je OT rozdelený do *x n*-bitových blokov a následne je každý z týchto blokov zašifrovaný bok po bloku v jednotlivých iteráciách samostatne pomocou rovnakého kľúča. Dešifrovanie je potom rovnako aplikované na každý blok zvlášť. Aj napriek tomu, že šifrovanie každého bloku úplne nezávisle na ostatných blokoch garantuje, že prípadné chyby vzniknuté pri šifrovaní v jednom bloku sa neodrazia na ŠT vygenerovanom v ostatných blokoch, nie je používanie tohto režimu činnosti odporúčané. Najmä kvôli jednoduchosti tohto systému a nulovému manažmentu kľúčov. Režim ECB využíva padding ako metódu na zaistenie toho, že bitová dĺžka OT bude celočíselným násobkom *n*. [15]



Obrázok 3 – ECB režim [15]

Schému ECB režimu zobrazuje **Obrázok 3**, kde p_x označuje blok OT, K šifrovací kľúč (rovnaký pre všetky iterácie), E šifrovaciu funkciu, c_x blok ŠT a D dešifrovaciu funkciu.

CBC (Cipher Block Chaining) režim je najpopulárnejším režimom činnosti blokových šifier. Rovnako ako v prípade ECB režimu je OT rozdelený do x n-bitových blokov a zašifrovaný bok po bloku v jednotlivých iteráciách pomocou rovnakého kľúča. Rozdielom je pridanie operácie XOR medzi blokom dát, ktoré majú byť práve zašifrované a blokom práve zašifrovaných dát (výstupom z prechádzajúcej iterácie blokovej šifry). Výstup z daného bloku teda nezávisí len na použitom šifrovacom algoritme a bloku vstupných dát, ale aj na predchádzajúcich, už zašifrovaných blokoch dát. Pre prvý blok, nemôžeme použiť predchádzajúci blok zašifrovaných dát kedže taký blok logicky neexistuje, preto je pre prvý blok použitý tzv. inicializačný vektor, ktorý by mal predstavovať blok náhodne zvolených dát. Čo znamená, že pre rovnaký kľúč a rovnaký šifrovací algoritmus môžeme zmenou inicializačného vektoru docieliť pre rovnaký OT rozdielne výstupy (ŠT). V režime CBC nie je možné pre väčšinu aplikácií využiť padding na zaistenie n-násobnej bitovej dĺžky OT. Pre CBC režim preto existujú dve metódy riešiace tento problém, prvou je zašifrovanie posledného (neúplného) bloku OT v režime OFB. Druhou metódou je tzv. ciphertext stealing, kedy posledný blok OT doplníme potrebným počtom bitov z predchádzajúceho bloku zašifrovaných dát z pravej strany a tieto dáta potom v predposlednom bloku vynechávame. Pri dešifrovaní je potom potrebné posledný blok dešifrovať skôr ako blok predposledný. Prípadná chyba v i-tom bloku ŠT sa pri dešifrovaní prejaví ako chyba i-teho bloku OT a rovnako sa odrazí v i+1 bloku. [15]



Obrázok 4 – CBC režim [15]

Schému CBC režimu zobrazuje **Obrázok 4**, kde p_x označuje blok OT, *IV* inicializačný vektor *K* šifrovací kľúč (rovnaký pre všetky iterácie), *E* šifrovaciu funkciu, C_x blok ŠT, D dešifrovaciu funkciu a $\oplus$ logickú operáciu XOR

OFB (Output FeedBack) režim je od predchádzajúcich dvoch systémov veľmi odlišný. Zatiaľ čo v režime ECB a CBC bloková šifra slúžila na generovanie ŠT. V tomto režime slúži len ako generátor pseudonáhodnej sekvencie kľúčov pre jednotlivé iterácie. ŠT vzniká ako výsledok operácie XOR medzi OT a kľúčom pre danú iteráciu, ktorý vznikol ako výstup blokovej šifry, ktorej vstupom bol kľúč z predchádzajúcej iterácie. V princípe sa v tomto režime jedná o akúsi simuláciu prúdovej šifry. Pre prvú iteráciu je pre operáciu XOR ako aj pre vstup do blokovej šifry použitý inicializačný vektor. V prípade neúplného posledného bloku dát (a teda na zachovanie *n*-násobnej bitovej dĺžky OT) je postup v OFB veľmi jednoduchý. Jednoducho zašifrujeme posledný neúplný *m*-bitový ($m < n$) blok dát pomocou *j*-bitov ($j = m$) kľúča pre danú iteráciu (z ľavej strany). V režime OFB sa prípadná chyba v *i*-tom bite ŠT pri dešifrovaní prejaví len v *i*-tom bite OT. [15]

CFB (Ciphertext FeedBack) režim je veľmi podobný režimu OFB, kedže ŠT rovnako vzniká ako výsledok operácie XOR medzi OT a kľúčom špecifickým pre danú iteráciu, ktorý je výstupom z blokovej šifry, slúžiacej len ako generátor pseudonáhodnej postupnosti. Rozdielom je to, že zatiaľ čo v režime OFB bol vstupom do blokovej šifry kľúč z predchádzajúcej iterácie v režime CFB je vstupom spätná väzba, teda blok ŠT

z predchádzajúcej iterácie. Pre prvú iteráciu je pre operáciu XOR ako aj pre vstup do blokovej šifry použitý inicializačný vektor rovnako ako v režime OFB. V prípade neúplného bloku je postup rovnaký ako v režime CFB. Prípadná chyba v *i*-tom bloku ŠT sa pri dešifrovaní prejaví v *i*-tom bloku OT a približne *n* nasledujúcich bitoch OT, následne sa dešifrovanie zotaví. [15]

3.1.3 Camellia

Camellia je bloková symetrická šifra, vyvinutá v rámci spolupráce spoločností Mitsubishi Electric Corporation a NTT (Nippon Telegraph and Telephone Corporation) v roku 2000. Odvtedy bola štandardizovaná rôznymi štandardizačnými organizáciami medzi ktoré patrí napríklad ISO, NESSIE, IETF a iné. Camellia využíva bloky o veľkosti 128 bitov a dĺžku kľúča 128, 192 a 256 bitov. [16]

V nasledujúcich podkapitolách si všeobecne vysvetlíme šifrovací algoritmus, dešifrovací algoritmus ako aj algoritmus tvorby podkľúčov a následne bližšie rozoberieme funkcie použité v rámci šifrovacieho procesu a tvorby podkľúčov. V popise budú použité názvy funkcii z originálnej dokumentácie šifry Camellia.

3.1.3.1 Šifrovanie

Obrázok 5 zobrazuje šifrovací proces pre 128 – bitový kľúč. Ako môžeme vidieť šifrovací proces pozostáva z 18 iterácií, rozdelených do troch blokov po šesť iterácií s využitím Feistelovej štruktúry, ktorú sme rozobrali v kapitole **3.1.1**. V každom bloku je potom šesťkrát po sebe volaná funkcia *F* s využitím šiestich 64-bitových podkľúčov, každým pre jednu z iterácií. Ďalej platí, že po 6. a 12. iterácií, čiže po prvom a druhom bloku po šesť iterácií, je volaná funkcia *FL* a FL^{-1}, tieto "medzi-iterácie" budeme označovať decimálnym číslom zodpovedajúcim podielu súčtu iterácií medzi ktorými sa nachádzajú napr. 6,5. Okrem toho je pred prvou iteráciou a po poslednej iterácií použitá logická funkcia XOR. [16]

Teraz prejdeme celý šifrovací ako aj dešifrovací proces ešte raz v sekvenčnom poradí. Pre jednoduchší zápis budeme používať niekoľko notácií, ktoré teraz definujeme. Dolné indexy typu $_{(n)}$ označujú, že daný prvok je *n*-bitov dlhý (napr. $X_{(n)}$). Dolný index typu $_L$ označuje, že sa jedná o ľavú časť daného prvku resp. bitovej sekvencie (napr. X_L) a dolný index typu $_R$ potom označuje, že sa jedná o pravú časť (napr. X_R). Ďalej budeme používať niekoľko symbolov a premenných:

$\oplus$ – označuje bitovú XOR operáciu

$\|$ – označuje zreťazenie dvoch operandov

$\lll_n$ – označuje kruhovú rotáciu operandu o *n*-bitov doľava

$\cap$ – označuje bitovú AND operáciu

$\cup$ – označuje bitovú OR operáciu

$\bar{x}$ – označuje bitový doplnok k *x*

V – označuje blok vstupných dát (blok OT)

S – označuje blok výstupných dát (blok ŠT)

Pred začiatkom samotného šifrovacieho procesu sú z kľúča *K* vygenerované podkľúče $kw_{t(64)}$ (t = 1, 2, 3, 4), $k_{u(64)}$ (u = 1, 2, 3, ..., 18) a $kl_{v(64)}$ (v = 1, 2, 3, 4) – viac o tom akým spôsobom sú kľúče generované v kapitole **3.1.3.3**. Pred prvou iteráciou je na prvý blok vstupných dát a dva zreťazené podkľúče $kw_{1(64)}$ a $kw_{2(64)}$ použitá logická funkcia XOR a výsledok je rozdelený na pravú a ľavú časť, ktoré následne vstupujú do prvej iterácie šifry, teda platí [16] :

$$M_{(128)} \oplus \left(kw_{1(64)} \parallel kw_{2(64)}\right) = L_{0(64)} \parallel R_{0(64)} \qquad (1)$$

Následne pre každú z osemnástich iterácií vykonáme :

$$L_r = R_{r-1} \oplus F(L_{r-1}, k_r), \qquad (2)$$

$$R_r = L_{r-1}. \qquad (3)$$

Čo znamená, že do každej ďalšej z iterácií vstupuje ako pravá strana nezmenená ľavá strana z predchádzajúcej iterácie (L_{r-1}) a ako ľavá strana výsledok operácie XOR medzi pravou stranou z predchádzajúcej iterácie (R_{r-1}) a výstupom funkcie *F*, ktorej vstupnými parametrami bola ľavá strana z prechádzajúcej iterácie (L_{r-1}) a podkľúč pre danú iteráciu (k_r).

Po 6. a 12. iterácií vykonáme [16]:

$$L'_r = R_{r-1} \oplus F(L_{r-1}, k_r) \qquad (4)$$

$$R'_r = L_{r-1}\,, \qquad (5)$$

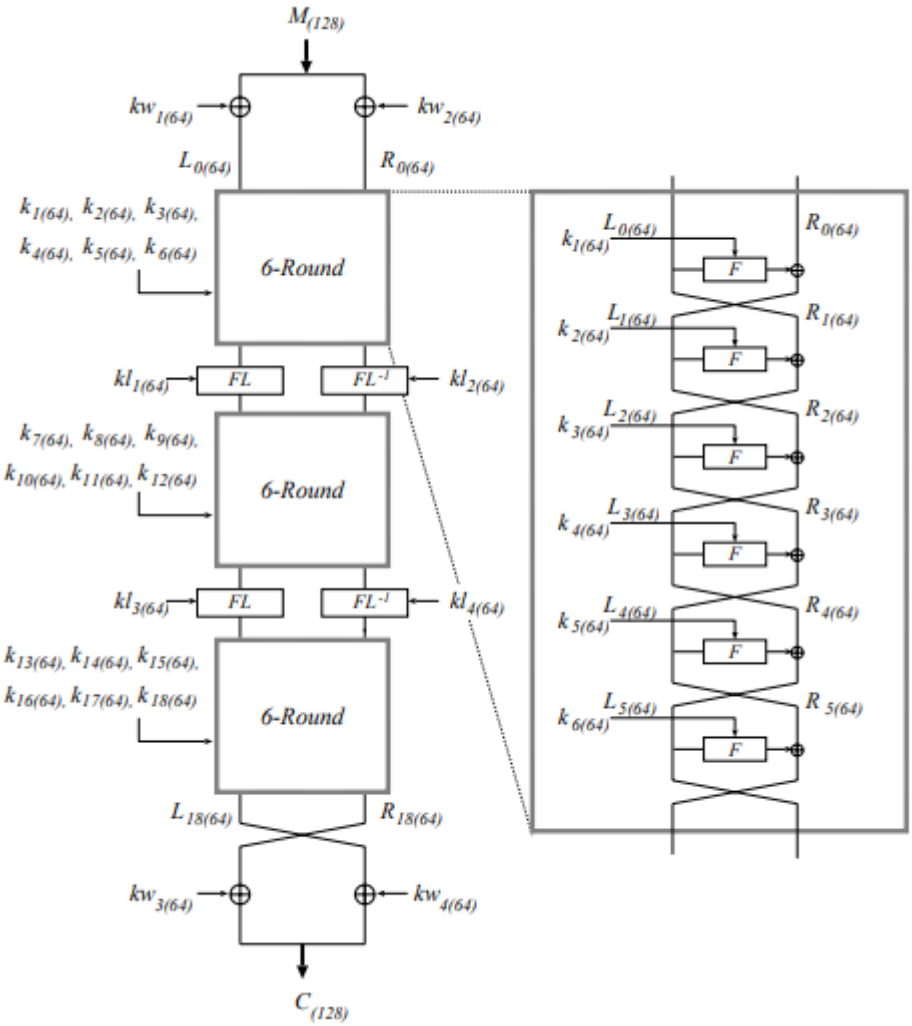
$$L_r = FL\left(L'_r, kl_{\frac{2r}{6}-1}\right), \qquad (6)$$

$$R_r = FL^{-1}\left(R'_r, kl_{\frac{2r}{6}}\right). \quad (7)$$

Čo znamená, že do 6,5. a 12,5. iterácie vstupuje ako ľavá strana, ľavá strana z poslednej iterácie predchádzajúceho bloku (L'_r), ktorá vznikla ako produkt operácie XOR medzi pravou stranou predchádzajúcej iterácie v danom bloku (R_{r-1}) a funkcie *F*, ktorej vstupnými parametrami bola ľavá strana predchádzajúcej iterácie v danom bloku (L_{r-1}) a podkľúč pre danú iteráciu (k_r). Výstupom 6,5. a 12,5. iterácie pre ľavú stranu (L_r) je potom produkt funkcie *FL*, ktorej vstupmi je pôvodná ľavá strana (L'_r) a podkľúč *kl* pre danú iteráciu ($kl_{\frac{2r}{6}-1}$). Pre pravú stranu vstupuje do 6,5. a 12,5. iterácie pravá strana z poslednej iterácie predchádzajúceho bloku (R'_r), ktorú predstavuje ľavá strana predchádzajúcej iterácie v danom bloku (L_{r-1}). Výstupom 6,5. a 12,5. iterácie pre pravú stranu (R_r) je potom produkt funkcie FL^{-1}, ktorej vstupmi je pôvodná pravá strana (R'_r) a podkľúč *kl* pre danú iteráciu ($kl_{\frac{2r}{6}}$).

ŠT je nakoniec výsledok logickej funkcie XOR medzi zreťazenou ľavou a pravou stranou z 18. iterácie a zreťazenými kľúčmi $kw_{3(64)}$ a $kw_{4(64)}$ teda platí [16]:

$$S_{128} = (R_{18(64)} \parallel L_{18(64)}) \oplus (kw_{3(64)} \parallel kw_{4(64)}) \quad (8)$$



Obrázok 5 – Šifrovací proces Camellia pre 128 – bitový kľúč [16]

Obrázok 6 zobrazuje šifrovací proces pre 192 a 256 – bitový kľúč. Môžeme vidieť, že pre tieto bitové dĺžky kľúča šifrovací proces pozostáva z 24 iterácií rozdelených do štyroch blokov po šesť iterácií s využitím Feistelovej štruktúry. V každom bloku je potom šesťkrát po sebe volaná funkcia *F* s využitím šiestich 64-bitových podkľúčov, každým pre jednu z iterácií a po 6., 12. a 18. iterácií, čiže po prvom, druhom a treťom bloku po šesť iterácií, je volaná funkcia *FL* a FL^{-1} rovnako ako v prípade 128 – bitového kľúča. Taktiež rovnako ako v prípade 128 – bitového kľúča je pred prvou a po poslednej iterácií volaná logická funkcia XOR. [16]

Šifrovací proces pre 192 a 256 – bitový kľúč prebieha veľmi podobne ako v prípade 128 – bitového kľúča. Teda pred začiatkom samotného šifrovacieho procesu sú z kľúča *K* vygenerované podkľúče $kw_{t(64)}$ (t = 1, 2, 3, 4), $k_{u(64)}$ (u = 1, 2, 3, ..., 24) a $kl_{v(64)}$ (v = 1, 2, 3, 4, 5, 6) – viac o tom akým spôsobom sú kľúče generované v kapitole **3.1.3.3**. Pred prvou

iteráciou je na prvý blok vstupných dát a dva zreťazené podkľúče $kw_{1(64)}$ a $kw_{2(64)}$ použitá funkcia XOR a výsledok je rozdelený na pravú a ľavú časť, ktoré následne vstupujú do prvej iterácie šifry, teda platí [16]:

$$V_{(128)} \oplus \left(kw_{1(64)} \parallel kw_{2(64)}\right) = L_{0(64)} \parallel R_{0(64)} \tag{9}$$

Vzťahy platiace pre jednotlivé iterácie šifrovacieho procesu ako aj dešifrovacích procesov, ktoré budeme rozoberať v kapitole **3.1.3.2,** sú rovnaké ako v prípade 128 – bitového kľúča a preto ich nebude opätovne slovne interpretovať. Pre prípadnú slovnú interpretáciu týchto vzťahov vyhľadajte časť objasňujúcu šifrovací proces pre 128 – bitový kľúč na začiatku kapitoly **3.1.3.1**.

Následne pre každú z dvadsiatich štyroch iterácií vykonáme [16]:

$$L_r = R_{r-1} \oplus F(L_{r-1}, k_r), \tag{10}$$

$$R_r = L_{r-1}. \tag{11}$$

Po 6. a 12. a 18. iterácií vykonáme [16]:

$$L'_r = R_{r-1} \oplus F(L_{r-1}, k_r) \tag{12}$$

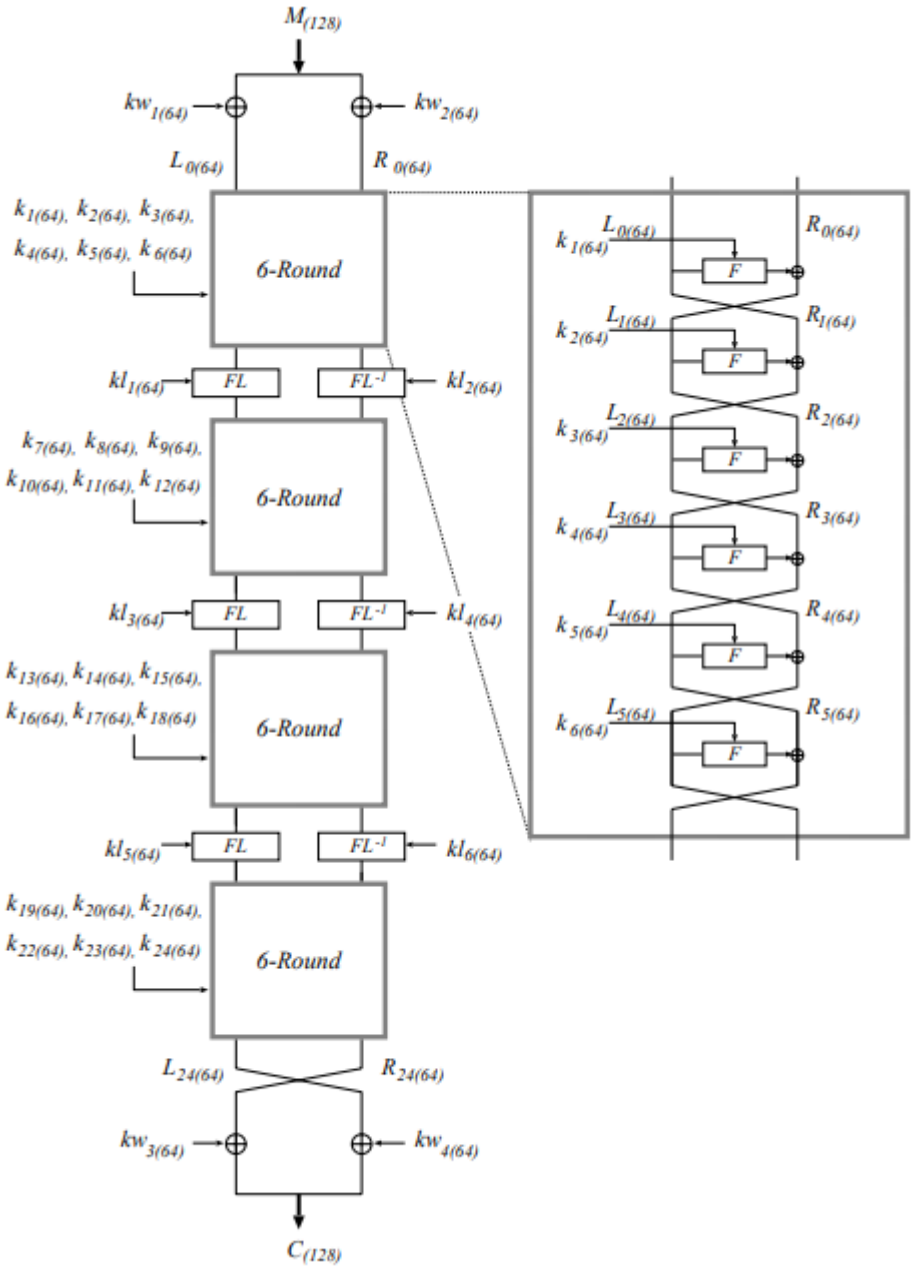
$$R'_r = L_{r-1}\,, \tag{13}$$

$$L_r = FL\left(L'_r, kl_{\frac{2r}{6}-1}\right), \tag{14}$$

$$R_r = FL^{-1}\left(R'_r, kl_{\frac{2r}{6}}\right). \tag{15}$$

ŠT je nakoniec výsledok funkcie XOR medzi zreťazenou ľavou a pravou stranou z 24. iterácie a zreťazenými kľúčmi $kw_{3(64)}$ a $kw_{4(64)}$ teda platí [16]:

$$S_{128} = (R_{24(64)} \parallel L_{24(64)}) \oplus \left(kw_{3(64)} \parallel kw_{4(64)}\right) \tag{16}$$



Obrázok 6 – Šifrovací proces Camellia pre 192 a 256 – bitový kľúč [16]

3.1.3.2 Dešifrovanie

Obrázok 7 zobrazuje dešifrovací proces pre 128 – bitový kľúč. Ako môžeme vidieť dešifrovací proces pozostáva z 18 iterácií rozdelených do troch blokov po šesť iterácií s využitím Feistelovej štruktúry. V každom bloku je potom šesťkrát po sebe volaná funkcia *F* s využitím šiestich 64-bitových podkľúčov, každým pre jednu z iterácií. Ďalej môžeme vidieť, že po 6. a 12. iterácií, čiže po prvom a druhom bloku po šesť iterácií, je opäť volaná funkcia *FL* a FL^{-1}. Okrem toho je pred prvou iteráciou a po poslednej iterácií volaná funkcia

XOR. V princípe sa teda jedná sa o proces zhodný s procesom šifrovacím, s tým rozdielom, že je nutné použiť podkľúče v reverznom poradí. Pred začiatkom samotného dešifrovacieho procesu sú teda z kľúča *K* vygenerované podkľúče $kw_{t(64)}$ (t = 1, 2, 3, 4), $k_{u(64)}$ (u = 1, 2, 3, ..., 18) a $kl_{v(64)}$ (v = 1, 2, 3, 4) – viac o tom akým spôsobom sú kľúče generované v kapitole **3.1.3.3**. Pred prvou iteráciou je na prvý blok šifrových dát a dva zreťazené podkľúče $kw_{3(64)}$ a $kw_{4(64)}$ použitá funkcia XOR a výsledok je rozdelený na pravú a ľavú časť, ktoré následne vstupujú do prvej iterácie šifry, teda platí [16]:

$$S_{(128)} \oplus \left(kw_{3(64)} \parallel kw_{4(64)}\right) = L_{18(64)} \parallel R_{18(64)} \tag{17}$$

Následne pre každú z osemnástich iterácií (v zostupnom poradí) vykonáme [16]:

$$R_{r-1} = L_r \oplus F(R_r, k_r), \tag{18}$$

$$L_{r-1} = R_r. \tag{19}$$

Po 13. a 7. iterácií vykonáme [16]:

$$R'_{r-1} = L_r \oplus F(R_r, k_r) \tag{20}$$

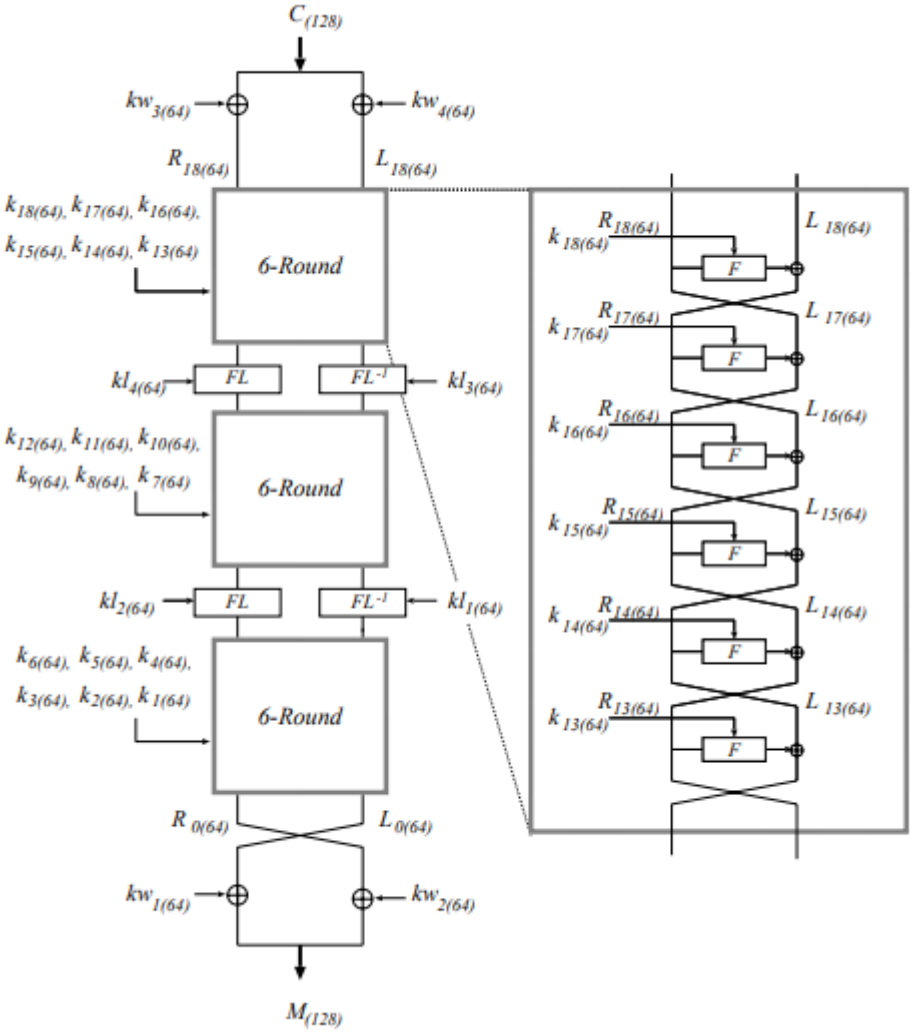
$$L'_{r-1} = R_r, \tag{21}$$

$$R_{r-1} = FL\left(R'_{r-1}, kl_{\frac{2(r-1)}{6}}\right), \tag{22}$$

$$L_{r-1} = FL^{-1}\left(L'_{r-1}, kl_{\frac{2(r-1)}{6-1}}\right). \tag{23}$$

OT je nakoniec výsledok funkcie XOR medzi zreťazenou ľavou a pravou stranou z poslednej resp. prvej iterácie a zreťazenými kľúčmi $kw_{1(64)}$ a $kw_{2(64)}$, teda platí [16]:

$$V_{128} = (R_{0(64)} \parallel L_{0(64)}) \oplus \left(kw_{1(64)} \parallel kw_{2(64)}\right) \tag{24}$$



Obrázok 7 – dešifrovací proces Camellia pre 128 – bitový kľúč [16]

Obrázok 8 zobrazuje dešifrovací proces pre 192 a 256 – bitový kľúč. Ako môžeme vidieť dešifrovací proces pozostáva z 24 iterácií rozdelených do štyroch blokov po šesť iterácií s využitím Feistelovej štruktúry. V každom bloku je potom šesťkrát po sebe volaná funkcia *F* s využitím šiestich 64-bitových podkľúčov, každým pre jednu z iterácií. Ďalej môžeme vidieť, že po 6., 12. a 18. iterácií, čiže po prvom, druhom a treťom bloku po šesť iterácií, je volaná funkcia *FL* a FL^{-1}. Okrem toho je pred prvou iteráciou a po poslednej iterácií volaná funkcia XOR. V princípe sa teda jedná sa o proces zhodný s procesom šifrovacím, s tým rozdielom, že je nutné použiť podkľúče v reverznom poradí. Pred začiatkom samotného dešifrovacieho procesu sú teda z kľúča *K* vygenerované podkľúče $kw_{t(64)}$ ($t = 1, 2, 3, 4$), $k_{u(64)}$ ($u = 1, 2, 3, ..., 24$) a $kl_{v(64)}$ ($v = 1, 2, 3, 4, 5, 6$) – viac o tom akým spôsobom sú kľúče generované v kapitole **3.1.3.3**. Pred prvou iteráciou je na prvý blok šifrových dát a dva

zreťazené podkľúče $kw_{3(64)}$ a $kw_{4(64)}$ použitá funkcia XOR a výsledok je rozdelený na pravú a ľavú časť, ktoré následne vstupujú do prvej iterácie šifry, teda platí [16]:

$$S_{(128)} \oplus \left(kw_{3(64)} \parallel kw_{4(64)}\right) = L_{24(64)} \parallel R_{24(64)} \tag{25}$$

Následne pre každú z dvadsiatich štyroch iterácií (v zostupnom poradí vykonáme [16]:

$$R_{r-1} = L_r \oplus F(R_r, k_r), \tag{26}$$

$$L_{r-1} = R_r. \tag{27}$$

Po 19., 13. a 7. iterácií vykonáme [16]:

$$R'_{r-1} = L_r \oplus F(R_r, k_r) \tag{28}$$

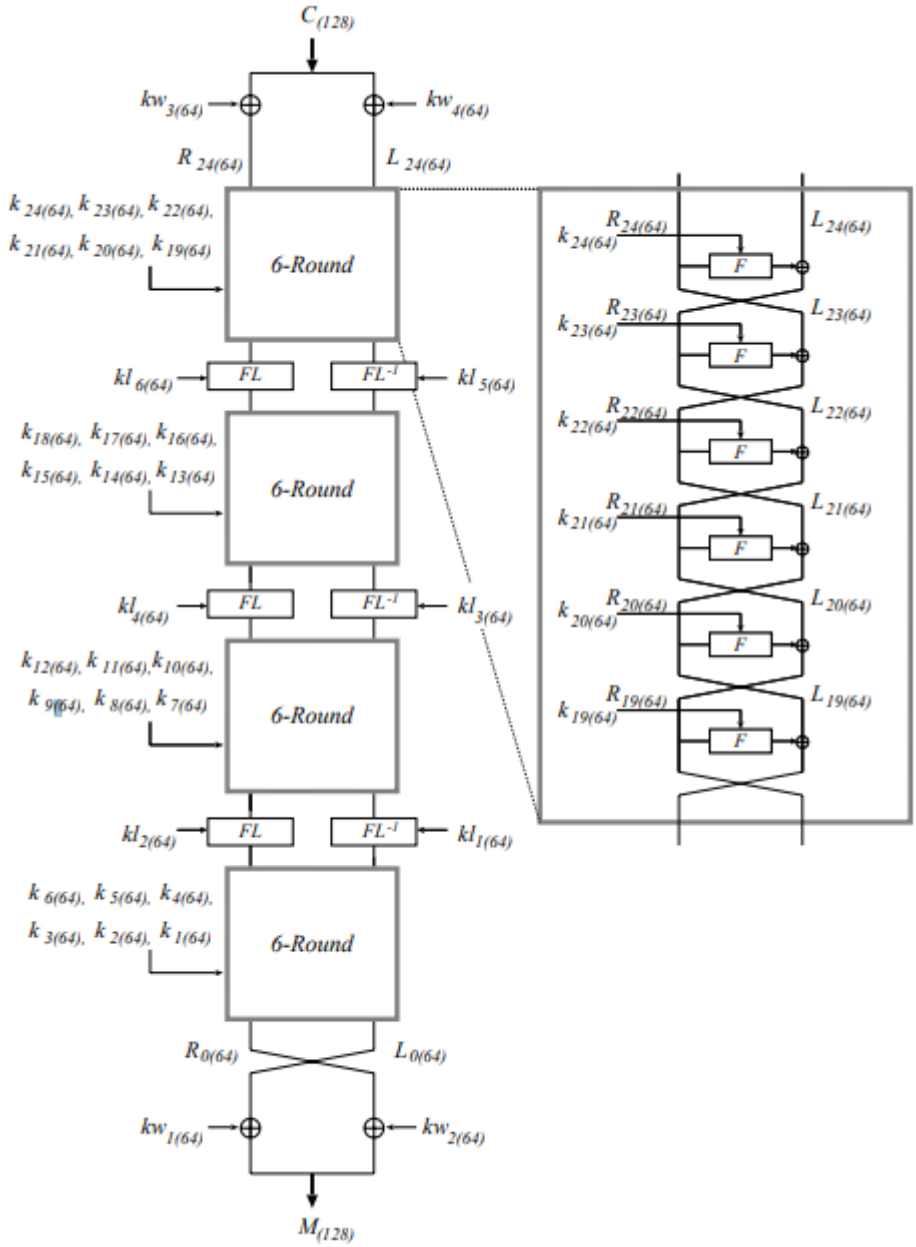
$$L'_{r-1} = R_r, \tag{29}$$

$$R_{r-1} = FL\left(R'_{r-1}, kl_{\frac{2(r-1)}{6}}\right), \tag{30}$$

$$L_{r-1} = FL^{-1}\left(L'_{r-1}, kl_{\frac{2(r-1)}{6-1}}\right). \tag{31}$$

OT je nakoniec výsledok funkcie XOR medzi zreťazenou ľavou a pravou stranou z poslednej resp. prvej iterácie a zreťazenými kľúčmi $kw_{1(64)}$ a $kw_{2(64)}$, teda platí [16]:

$$V_{128} = (R_{0(64)} \parallel L_{0(64)}) \oplus \left(kw_{1(64)} \parallel kw_{2(64)}\right) \tag{32}$$



Obrázok 8 – dešifrovací proces Camellia pre 192 a 256 – bitový kľúč [16]

3.1.3.3 Proces generovania kľúčov

Na **Obrázok 9** môžeme vidieť proces generovania kľúčov. Pre potreby jeho interpretácie budeme potrebovať niekoľko nových premenných. Dve 128 – bitové premenné $K_{L(128)}$ a $K_{R(128)}$ a štyri 64 – bitové premenné $K_{LL(64)}$, $K_{LR(64)}$, $K_{RL(64)}$ a $K_{RR(64)}$. Medzi týmito premennými potom existujú vzťahy ktoré zobrazuje **Tabuľka 1**. Taktiež budeme potrebovať niekoľko konštánt ktoré zobrazuje **Tabuľka 2**. Teraz môžeme prejsť k samotnej interpretácii. Ako môžeme vidieť na už spomínanom **Obrázok 9** na začiatku procesu generovania kľúčov sú hodnoty $K_{L(128)}$ a $K_{R(128)}$ XOR-nuté a následne "zašifrované" v dvoch

iteráciách pomocou funkcie F a konštánt $\sum_{1(64)}$ a $\sum_{2(64)}$, ktoré sú použité ako šifrovacie kľúče. Výstup je potom opätovane XOR-nutý s $K_{L(128)}$ a "zašifrovaný" v dvoch iteráciách pomocou funkcie F a konštánt $\sum_{3(64)}$ a $\sum_{4(64)}$, ktoré sú opäť použité ako šifrovacie kľúče. Výsledkom je hodnota K_A. Túto hodnotu potom XOR-neme s hodnotou $K_{R(128)}$ a "zašifrujeme" v dvoch iteráciách pomocou funkcie F a konštánt $\sum_{5(64)}$ a $\sum_{6(64)}$. Výsledkom je hodnota K_B, ktorá je použitá v len v prípade, že dĺžka kľúča je 192 alebo 256 – bitov. Hodnoty podkľúčov $kw_{t(64)}$, $ku_{(64)}$ a $kl_{v(64)}$ sú vygenerované pomocou cyklického posunu z ľavej alebo pravej časti $K_{L(128)}$, $K_{R(128)}$, $K_{A(128)}$ alebo $K_{B(128)}$. To znamená, že rovnosť $K_{RR(64)} = \overline{K_{RL(64)}}$ umožňuje kompatibilitu 192 a 256 – bitovej verzie šifry. [16]

Presné hodnoty posunu ako aj konkrétna premenná z ktorej bol daný kľúč vygenerovaný zobrazuje **Obrázok 10** v prípade použitia 128 – bitového kľúča a **Obrázok 11** v prípade použitia 192 alebo 256 – bitového kľúča.

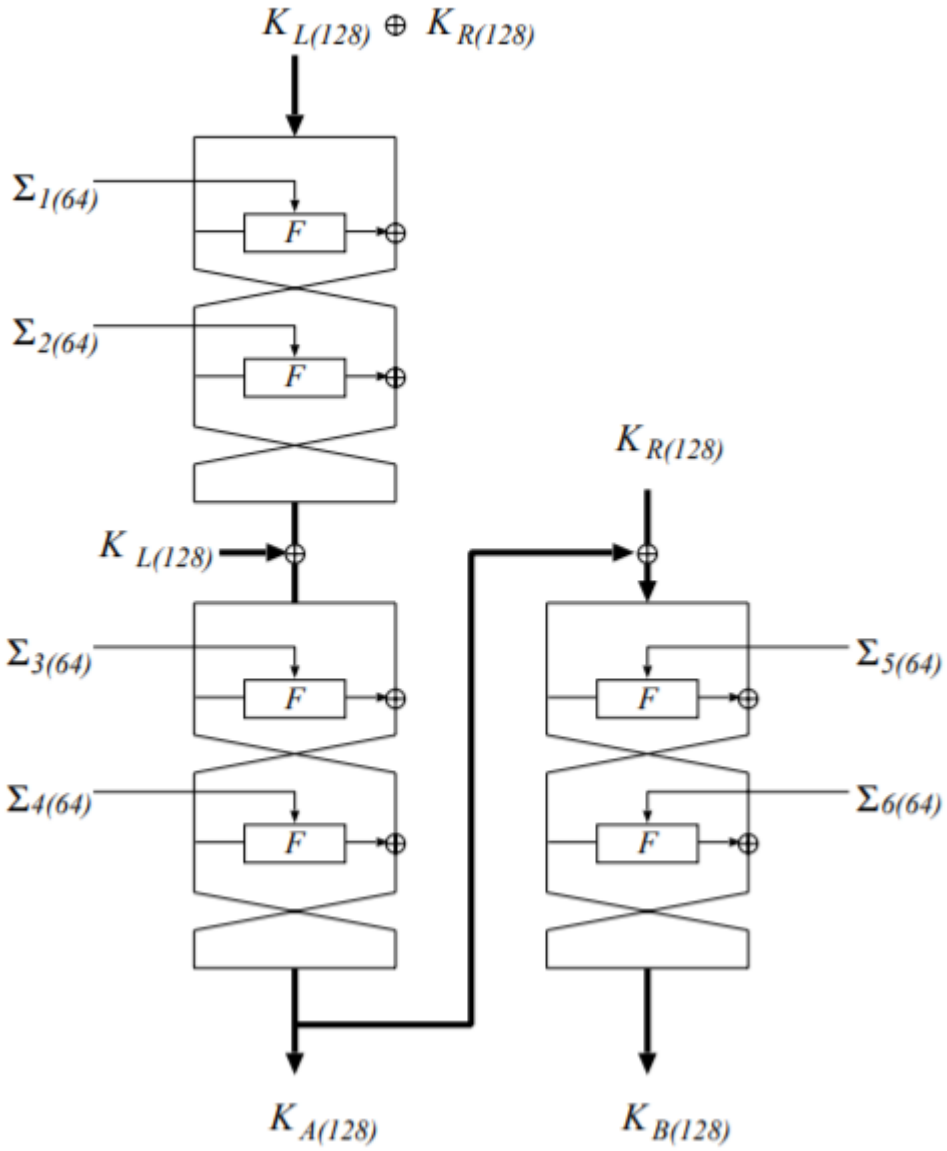
Pojem "Prewhitening" označuje aplikáciu funkcie XOR na 128 – bitový blok vstupných dát (OT) a zreťazených podkľúčov $kw_{1(64)}$ a $kw_{2(64)}$. Pojem "Postwhitening" označuje aplikáciu funkcie XOR na 128 – bitový blok vstupných dát (OT) vystupujúcich z poslednej iterácie šifry a zreťazených podkľúčov $kw_{3(64)}$ a $kw_{4(64)}$.

$K_{(128)}$	=	$K_{L(128)}$,	$K_{R(128)}$	=	0	pre 128 – bitový kľúč,
$K_{(128)}$	=	$K_{L(128)} \parallel K_{RL(64)}$,	$K_{RR(64)} = \overline{K_{RL(64)}}$	=	0	pre 192 – bitový kľúč,
$K_{(128)}$	=	$K_{L(128)} \parallel K_{R(128)}$;				pre 256 – bitový kľúč.
$K_{L(128)}$	=	$K_{LL(64)} \parallel K_{LR(64)}$,				pre akúkoľvek dĺžku kľúča,
$K_{R(128)}$	=	$K_{RL(64)} \parallel K_{RR(64)}$;				

Tabuľka 1 – Vzťahy premenných v procese generovanie kľúčov [16]

$\sum_{1(64)}$	0xA09E667F3BCC908B
$\sum_{2(64)}$	0xB67AE8584CAA73BA
$\sum_{3(64)}$	0xC6EF372FE94F82BE
$\sum_{4(64)}$	0x54FF53A5F1D36FC1
$\sum_{5(64)}$	0x10E527FADE682D1D
$\sum_{6(64)}$	0xB05688C2B3E6C1FD

Tabuľka 2 – Konštanty pre proces generovania kľúčov [16]



Obrázok 9 – proces generovania kľúčov [16]

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll 0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll 0)_{R(64)}$
F (Round 1)	$k_{1(64)}$	$(K_A \lll 0)_{L(64)}$
F (Round 2)	$k_{2(64)}$	$(K_A \lll 0)_{R(64)}$
F (Round 3)	$k_{3(64)}$	$(K_L \lll 15)_{L(64)}$
F (Round 4)	$k_{4(64)}$	$(K_L \lll 15)_{R(64)}$
F (Round 5)	$k_{5(64)}$	$(K_A \lll 15)_{L(64)}$
F (Round 6)	$k_{6(64)}$	$(K_A \lll 15)_{R(64)}$
FL	$kl_{1(64)}$	$(K_A \lll 30)_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_A \lll 30)_{R(64)}$
F (Round 7)	$k_{7(64)}$	$(K_L \lll 45)_{L(64)}$
F (Round 8)	$k_{8(64)}$	$(K_L \lll 45)_{R(64)}$
F (Round 9)	$k_{9(64)}$	$(K_A \lll 45)_{L(64)}$
F (Round 10)	$k_{10(64)}$	$(K_L \lll 60)_{R(64)}$
F (Round 11)	$k_{11(64)}$	$(K_A \lll 60)_{L(64)}$
F (Round 12)	$k_{12(64)}$	$(K_A \lll 60)_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll 77)_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll 77)_{R(64)}$
F (Round 13)	$k_{13(64)}$	$(K_L \lll 94)_{L(64)}$
F (Round 14)	$k_{14(64)}$	$(K_L \lll 94)_{R(64)}$
F (Round 15)	$k_{15(64)}$	$(K_A \lll 94)_{L(64)}$
F (Round 16)	$k_{16(64)}$	$(K_A \lll 94)_{R(64)}$
F (Round 17)	$k_{17(64)}$	$(K_L \lll 111)_{L(64)}$
F (Round 18)	$k_{18(64)}$	$(K_L \lll 111)_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_A \lll 111)_{L(64)}$
	$kw_{4(64)}$	$(K_A \lll 111)_{R(64)}$

Obrázok 10 – tabuľka podkľúčov pre 128 – bitový kľúč [16]

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll 0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll 0)_{R(64)}$
F (Round 1)	$k_{1(64)}$	$(K_B \lll 0)_{L(64)}$
F (Round 2)	$k_{2(64)}$	$(K_B \lll 0)_{R(64)}$
F (Round 3)	$k_{3(64)}$	$(K_R \lll 15)_{L(64)}$
F (Round 4)	$k_{4(64)}$	$(K_R \lll 15)_{R(64)}$
F (Round 5)	$k_{5(64)}$	$(K_A \lll 15)_{L(64)}$
F (Round 6)	$k_{6(64)}$	$(K_A \lll 15)_{R(64)}$
FL	$kl_{1(64)}$	$(K_R \lll 30)_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_R \lll 30)_{R(64)}$
F (Round 7)	$k_{7(64)}$	$(K_B \lll 30)_{L(64)}$
F (Round 8)	$k_{8(64)}$	$(K_B \lll 30)_{R(64)}$
F (Round 9)	$k_{9(64)}$	$(K_L \lll 45)_{L(64)}$
F (Round 10)	$k_{10(64)}$	$(K_L \lll 45)_{R(64)}$
F (Round 11)	$k_{11(64)}$	$(K_A \lll 45)_{L(64)}$
F (Round 12)	$k_{12(64)}$	$(K_A \lll 45)_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll 60)_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll 60)_{R(64)}$
F (Round 13)	$k_{13(64)}$	$(K_R \lll 60)_{L(64)}$
F (Round 14)	$k_{14(64)}$	$(K_R \lll 60)_{R(64)}$
F (Round 15)	$k_{15(64)}$	$(K_B \lll 60)_{L(64)}$
F (Round 16)	$k_{16(64)}$	$(K_B \lll 60)_{R(64)}$
F (Round 17)	$k_{17(64)}$	$(K_L \lll 77)_{L(64)}$
F (Round 18)	$k_{18(64)}$	$(K_L \lll 77)_{R(64)}$
FL	$kl_{5(64)}$	$(K_A \lll 77)_{L(64)}$
FL^{-1}	$kl_{6(64)}$	$(K_A \lll 77)_{R(64)}$
F (Round 19)	$k_{19(64)}$	$(K_R \lll 94)_{L(64)}$
F (Round 20)	$k_{20(64)}$	$(K_R \lll 94)_{R(64)}$
F (Round 21)	$k_{21(64)}$	$(K_A \lll 94)_{L(64)}$
F (Round 22)	$k_{22(64)}$	$(K_A \lll 94)_{R(64)}$
F (Round 23)	$k_{23(64)}$	$(K_L \lll 111)_{L(64)}$
F (Round 24)	$k_{24(64)}$	$(K_L \lll 111)_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_B \lll 111)_{L(64)}$
	$kw_{4(64)}$	$(K_B \lll 111)_{R(64)}$

Obrázok 11 – tabuľka podkľúčov pre 192 a 256 – bitový kľúč [16]

3.1.3.4 Funkcie

Nasledujúca kapitola stručne opisuje princíp funkcií použitých v rámci šifrovacieho a dešifrovacieho procesu ako aj v rámci procesu tvorby podkľúčov.

***F*-funkcia** [16]

F-funkcia je definovaná ako :

$$F : \mathbf{L} \times \mathbf{L} \rightarrow \mathbf{L}$$

$$(X_{(64)}, k_{(64)}) \mapsto Y_{(64)} = P(S(X_{(64)} \oplus k_{(64)})). \quad (33)$$

***FL*-funkcia** [16]

FL-funkcia je definovaná ako :

$$FL : \mathbf{L} \times \mathbf{L} \rightarrow \mathbf{L}$$

$$(X_{L(32)} \| X_{R(32)}, kl_{L(32)} \| kl_{R(32)}) \mapsto Y_{L(32)} \| Y_{R(32)}, \quad (34)$$

kde

$$Y_{R(32)} = ((X_{L(32)} \cap kl_{L(32)}) \lll_1) \oplus X_{R(32)}, \quad (35)$$

$$Y_{L(32)} = (Y_{R(32)} \cup kl_{R(32)}) \oplus X_{L(32)}. \quad (36)$$

***FL*$^{-1}$-funkcia** [16]

FL-funkcia je definovaná ako :

$$FL^{-1} : \mathbf{L} \times \mathbf{L} \rightarrow \mathbf{L} \quad (37)$$

$$(Y_{L(32)} \| Y_{R(32)}, kl_{L(32)} \| kl_{R(32)}) \mapsto X_{L(32)} \| X_{R(32)}, \quad (38)$$

kde

$$X_{L(32)} = (Y_{R(32)} \cup kl_{R(32)}) \oplus Y_{L(32)}, \quad (39)$$

$$X_{R(32)} = ((X_{L(32)} \cap kl_{L(32)}) \lll_1) \oplus Y_{R(32)}. \quad (40)$$

***S*-funkcia** [16]

S-funkcia je definovaná ako :

$$S : \mathbf{L} \rightarrow \mathbf{L} \quad (41)$$

$$l_{1(8)}\|l_{2(8)}\|l_{3(8)}\|l_{4(8)}\|l_{5(8)}\|l_{6(8)}\|l_{7(8)}\|l_{8(8)} \mapsto l'_{1(8)}\| l'_{2(8)}\| l'_{3(8)}\| l'_{4(8)}\| l'_{5(8)}\| l'_{6(8)}\| l'_{7(8)}\| l'_{8(8)} \quad (42)$$

$$l'_{1(8)} = s_1(l_{1(8)}), \quad (43)$$

$$l'_{2(8)} = s_2(l_{2(8)}), \quad (44)$$

$$l'_{3(8)} = s_3(l_{3(8)}), \quad (45)$$

$$l'_{4(8)} = s_4(l_{4(8)}), \quad (46)$$

$$l'_{5(8)} = s_2(l_{5(8)}), \quad (47)$$

$$l'_{6(8)} = s_3(l_{6(8)}), \quad (48)$$

$$l'_{7(8)} = s_4(l_{1(8)}), \quad (49)$$

$$l'_{8(8)} = s_1(l_{1(8)}), \quad (50)$$

kde s-boxy (s_1, s_2, s_3, s_4) sú objasnené v nasledujúcej kapitole.

3.1.3.5 S-boxy

Všetky štyri s-boxy sú afinným ekvivalentom inverznej funkcie cez GF(2^8) a sú zobrazené na **Obrázok 12**, **Obrázok 13**, **Obrázok 14**, **Obrázok 15**. Algebrická reprezentácia s-boxov potom je [16]:

$$s_1 : \mathbf{L} \rightarrow \mathbf{L}$$

$$x_{(8)} = \mathbf{h}(\mathbf{g}(\mathbf{f}(0\text{xc5} \oplus x_{(8)}))) \oplus 0\text{x6e}, \quad (51)$$

$$s_2 : \mathbf{L} \rightarrow \mathbf{L}$$

$$x_{(8)} = s_1(x_{(8)}) \lll_1, \quad (52)$$

$$s_3 : \mathbf{L} \rightarrow \mathbf{L}$$

$$x_{(8)} = s_1(x_{(8)}) \ggg_1, \quad (53)$$

$$s_4 : \mathbf{L} \rightarrow \mathbf{L}$$

$$x_{(8)} = s_1(x_{(8)} \lll_1), \quad (54)$$

kde funkcie **f**, **g** a **h** sú definované nasledovne [16]:

$$\mathrm{f} : \mathbf{B} \rightarrow \mathbf{B}$$

$$a_{1(1)} \| a_{2(1)} \| a_{3(1)} \| a_{4(1)} \| a_{5(1)} \| a_{6(1)} \| a_{7(1)} \| a_{8(1)} \mapsto b_{1(1)} \| b_{2(1)} \| b_{3(1)} \| b_{4(1)} \| b_{5(1)} \| b_{6(1)} \| b_{7(1)} \| b_{8(1)}, \quad (55)$$

kde

$$b_1 = a_6 \oplus a_2, \quad (56)$$

$$b_2 = a_7 \oplus a_1, \quad (57)$$

$$b_3 = a_8 \oplus a_5 \oplus a_3, \quad (58)$$

$$b_4 = a_8 \oplus a_3, \quad (59)$$

$$b_5 = a_7 \oplus a_4, \quad (60)$$

$$b_6 = a_5 \oplus a_2, \quad (61)$$

$$b_7 = a_8 \oplus a_1, \quad (62)$$

$$b_8 = a_6 \oplus a_4. \quad (63)$$

$$g : \mathbf{B} \rightarrow \mathbf{B}$$

$$a_{1(1)} \| a_{2(1)} \| a_{3(1)} \| a_{4(1)} \| a_{5(1)} \| a_{6(1)} \| a_{7(1)} \| a_{8(1)} \mapsto b_{1(1)} \| b_{2(1)} \| b_{3(1)} \| b_{4(1)} \| b_{5(1)} \| b_{6(1)} \| b_{7(1)} \| b_{8(1)}, \quad (64)$$

kde

$$(b_8 + b_7\alpha + b_6\alpha^2 + b_5\alpha^3) + (b_4 + b_3\alpha + b_2\alpha^2 + b_1\alpha^3)\,\beta = 1 / ((a_8 + a_7\alpha + a_6\alpha^2 + a_5\alpha^3) + (a_4 + a_3\alpha + a_2\alpha^2 + a_1\alpha^3)\,\beta). \quad (65)$$

$$h : \mathbf{B} \rightarrow \mathbf{B}$$

$$a_{1(1)} \| a_{2(1)} \| a_{3(1)} \| a_{4(1)} \| a_{5(1)} \| a_{6(1)} \| a_{7(1)} \| a_{8(1)} \mapsto b_{1(1)} \| b_{2(1)} \| b_{3(1)} \| b_{4(1)} \| b_{5(1)} \| b_{6(1)} \| b_{7(1)} \| b_{8(1)}, \quad (66)$$

kde

$$b_1 = a_5 \oplus a_6 \oplus a_2, \quad (67)$$

$$b_2 = a_6 \oplus a_2, \quad (68)$$

$$b_3 = a_7 \oplus a_4, \quad (69)$$

$$b_4 = a_8 \oplus a_2, \quad (70)$$

$$b_5 = a_7 \oplus a_3, \quad (71)$$

$$b_6 = a_8 \oplus a_1, \quad (72)$$

$$b_7 = a_5 \oplus a_1, \quad (73)$$

$$b_8 = a_6 \oplus a_3. \quad (74)$$

112	130	44	236	179	39	192	229	228	133	87	53	234	12	174	65
35	239	107	147	69	25	165	33	237	14	79	78	29	101	146	189
134	184	175	143	124	235	31	206	62	48	220	95	94	197	11	26
166	225	57	202	213	71	93	61	217	1	90	214	81	86	108	77
139	13	154	102	251	204	176	45	116	18	43	32	240	177	132	153
223	76	203	194	52	126	118	5	109	183	169	49	209	23	4	215
20	88	58	97	222	27	17	28	50	15	156	22	83	24	242	34
254	68	207	178	195	181	122	145	36	8	232	168	96	252	105	80
170	208	160	125	161	137	98	151	84	91	30	149	224	255	100	210
16	196	0	72	163	247	117	219	138	3	230	218	9	63	221	148
135	92	131	2	205	74	144	51	115	103	246	243	157	127	191	226
82	155	216	38	200	55	198	59	129	150	111	75	19	190	99	46
233	121	167	140	159	110	188	142	41	245	249	182	47	253	180	89
120	152	6	106	231	70	113	186	212	37	171	66	136	162	141	250
114	7	185	85	248	238	172	10	54	73	42	104	60	56	241	164
64	40	211	123	187	201	67	193	21	227	173	244	119	199	128	158

Obrázok 12 – s-box **s1** [16]

224	5	88	217	103	78	129	203	201	11	174	106	213	24	93	130
70	223	214	39	138	50	75	66	219	28	158	156	58	202	37	123
13	113	95	31	248	215	62	157	124	96	185	190	188	139	22	52
77	195	114	149	171	142	186	122	179	2	180	173	162	172	216	154
23	26	53	204	247	153	97	90	232	36	86	64	225	99	9	51
191	152	151	133	104	252	236	10	218	111	83	98	163	46	8	175
40	176	116	194	189	54	34	56	100	30	57	44	166	48	229	68
253	136	159	101	135	107	244	35	72	16	209	81	192	249	210	160
85	161	65	250	67	19	196	47	168	182	60	43	193	255	200	165
32	137	0	144	71	239	234	183	21	6	205	181	18	126	187	41
15	184	7	4	155	148	33	102	230	206	237	231	59	254	127	197
164	55	177	76	145	110	141	118	3	45	222	150	38	125	198	92
211	242	79	25	63	220	121	29	82	235	243	109	94	251	105	178
240	49	12	212	207	140	226	117	169	74	87	132	17	69	27	245
228	14	115	170	241	221	89	20	108	146	84	208	120	112	227	73
128	80	167	246	119	147	134	131	42	199	91	233	238	143	1	61

Obrázok 13 – s-box **s2** [16]

56	65	22	118	217	147	96	242	114	194	171	154	117	6	87	160
145	247	181	201	162	140	210	144	246	7	167	39	142	178	73	222
67	92	215	199	62	245	143	103	31	24	110	175	47	226	133	13
83	240	156	101	234	163	174	158	236	128	45	107	168	43	54	166
197	134	77	51	253	102	88	150	58	9	149	16	120	216	66	204
239	38	229	97	26	63	59	130	182	219	212	152	232	139	2	235
10	44	29	176	111	141	136	14	25	135	78	11	169	12	121	17
127	34	231	89	225	218	61	200	18	4	116	84	48	126	180	40
85	104	80	190	208	196	49	203	42	173	15	202	112	255	50	105
8	98	0	36	209	251	186	237	69	129	115	109	132	159	238	74
195	46	193	1	230	37	72	153	185	179	123	249	206	191	223	113
41	205	108	19	100	155	99	157	192	75	183	165	137	95	177	23
244	188	211	70	207	55	94	71	148	250	252	91	151	254	90	172
60	76	3	53	243	35	184	93	106	146	213	33	68	81	198	125
57	131	220	170	124	119	86	5	27	164	21	52	30	28	248	82
32	20	233	189	221	228	161	224	138	241	214	122	187	227	64	79

Obrázok 14 – s-box **s_3** [16]

112	44	179	192	228	87	234	174	35	107	69	165	237	79	29	146
134	175	124	31	62	220	94	11	166	57	213	93	217	90	81	108
139	154	251	176	116	43	240	132	223	203	52	118	109	169	209	4
20	58	222	17	50	156	83	242	254	207	195	122	36	232	96	105
170	160	161	98	84	30	224	100	16	0	163	117	138	230	9	221
135	131	205	144	115	246	157	191	82	216	200	198	129	111	19	99
233	167	159	188	41	249	47	180	120	6	231	113	212	171	136	141
114	185	248	172	54	42	60	241	64	211	187	67	21	173	119	128
130	236	39	229	133	53	12	65	239	147	25	33	14	78	101	189
184	143	235	206	48	95	197	26	225	202	71	61	1	214	86	77
13	102	204	45	18	32	177	153	76	194	126	5	183	49	23	215
88	97	27	28	15	22	24	34	68	178	181	145	8	168	252	80
208	125	137	151	91	149	255	210	196	72	247	219	3	218	63	148
92	2	74	51	103	243	127	226	155	38	55	59	150	75	190	46
121	140	110	142	245	182	253	89	152	106	70	186	37	66	162	250
7	85	238	10	73	104	56	164	40	123	201	193	227	244	199	158

Obrázok 15 – s-box **s_4** [16]

3.1.4 Kuznyechik

Šifra Kuznyechik je symetrická bloková šifra, vyvinutá v rámci spolupráce "Centra pre ochranu informácií a špeciálnu komunikáciu federálnej bezpečnostnej služby Ruskej federácie" (z angl. Center for Information Protection and Special Communications of the Federal Security Service of the Russian Federation) a verejnej akciovej spoločnosti InfoTeCS JSC v roku 2015. Šifra bola vydaná v rámci ruského federálneho štandardu GOST R 34.12-2015. Kuznyechik využíva dĺžku blokov 128 bitov v kombinácií s 256 bitovým šifrovacím kľúčom. [17]

Podobne ako v prípade šifry Camellia bude pre jednoduchší zápis algoritmov potrebné definovať niekoľko symbolov [17]:

V^* - označuje všetky binárne reťazce konečnej dĺžky, vrátane prázdneho reťazca

V_S - označuje všetky binárne reťazce dĺžky s

$U \times X$ - označuje Cartézsky súčin U a X

$|A|$ - označuje dĺžku A

$A \| B$ - označuje zreťazenie A a B

$A \lll_x$ - označuje kruhovú rotáciu operandu o x-bitov doľava

$\oplus$ - označuje bitovú XOR operáciu

$\mathbb{Z}_2^s$ - označuje zostatkový kruh celých čísel po operácií modulo 2^S

$\boxplus$ - označuje operáciu sčítania v $\mathbb{Z}_2^{32}$

$\mathbb{F}$ - označuje konečné pole GF(2), kde $p(x) = x^8 + x^7 + x^6 + x + 1 \in GF(2)[x]$

$\text{Vec}_S: \mathbb{Z}_2^s \rightarrow V_S$ - označuje bijektívne mapovanie, ktoré element zo $\mathbb{Z}_2^s$ mapuje na jeho odpovedajúcu binárnu reprezentáciu

$\text{Int}_S: V_S \rightarrow \mathbb{Z}_2^s$ - označuje mapovanie inverzné k Vec_S, teda $Int_S = Vec_S^{-1}$

$\Delta: V_8 \rightarrow \mathbb{F}$ - označuje bijektívne mapovanie, ktoré mapuje binárny reťazec z V_8 do jeho reprezentácie z $\mathbb{F}$ nasledovne :

reťazec $Z_7 \| ... \| Z_1 \| Z_0$, $Z_i \in \{0, 1\}$, $i = 0, 1, \ldots, 7$, zodpovedá prvku $Z_0 + Z_1 \cdot \theta + \ldots + Z_7 \cdot \theta^7 \in \mathbb{F}$;

$\nabla: \mathbb{F} \rightarrow V_8$ - označuje mapovanie inverzné k Δ, teda $\nabla = \Delta^{-1}$

$\Phi\Psi$ - označuje kompozíciu mapovaní, kde mapovanie Ψ je aplikované jako prvé

Φ^S - označuje kompozíciu mapovaní Φ^{S-1} and Φ, kde $\Phi^1 = \Phi$.

Taktiež bude nutné definovať niekoľko transformácií [17]:

$X[k]: V_{128} \rightarrow V_{128}$	$X[k](a) = k \oplus a$, kde $k, a \in V_{128;}$
$S: V_{128} \rightarrow V_{128}$	$S(a) = S(a_{15}\|\|...\|\|a_0) = \pi(a_{15})\|\|...\|\|\pi(a_0)$, kde
	$a = a_{15}\|\|...\|\|a_0 \in V_{128,}\ a_i \in V_8$, $i = 0, 1, \ldots, 15$;
$S^{-1}: V_{128} \rightarrow V_{128}$	inverzná transformácia k S;
$R: V_{128} \rightarrow V_{128}$	$R(a) = R(a_{15}\|\|...\|\|a_0) = 1(a_{15}, ..., a_0)\|\|a_{15}\|\|...\|\|a_1$, kde
	$a = a_{15}\|\|\ldots\|\|a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, 1, \ldots, 15$;
$L: V_{128} \rightarrow V_{128}$	$L(a) = R^{16}(a)$, kde $a \in V_{128}$;
$R^{-1}: V_{128} \rightarrow V_{128}$	inverzná transformácia k R;
$L^{-1}: V_{128} \rightarrow V_{128}$	$L^{-1}(a) = (R^{-1})^{16}(a)$, kde $a \in V_{128}$;
$F[k]: V_{128} \times V_{128} \rightarrow V_{128} \times V_{128}$	$F[k](a_1,a_0) = (LSX[k](a_1) \oplus a_0, a_1)$, kde $k, a_0, a_1 \in V_{128}$

Nakoniec definujeme mapovania používané v šifrovacom resp. dešifrovanom procese šifry. Prvým mapovaním je nelineárne bijektívne mapovanie, ktoré je definované ako pole π' = (π'(0), π'(1), ... , π'(255)) a je zobrazené na **Obrázok 16**. [18]

π' = (252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77, 233, 119, 240, 219, 147, 46, 153, 186, 23, 54, 241. 187, 20, 205, 95, 193, 249, 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142, 79, 5, 132, 2, 174, 227, 106, 143, 160, 6, 11, 237, 152, 127, 212, 211, 31, 235, 52, 44, 81, 234, 200, 72, 171, 242, 42, 104, 162, 253, 58, 206, 204, 181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156, 183, 93, 135, 21, 161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178, 177, 50, 117, 25, 61, 255, 53, 138, 126, 109, 84, 198, 128, 195, 189, 13, 87, 223, 245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3, 224, 15, 236, 222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74, 167, 151, 96, 115, 30, 0, 98, 68, 26, 184, 56, 130, 100, 159, 38, 65, 173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165, 125, 105, 213, 149, 59, 7, 88, 179, 64, 134, 172, 29, 247, 48, 55, 107, 228, 136, 217, 231, 137, 225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144, 202, 216, 133, 97, 32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82, 89, 166, 116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182).

Obrázok 16 – Kuznyechik nelineárne bijektívne mapovanie [18]

Druhým používaným mapovaním je lineárne mapovanie definované ako transformácia l: $V_8^{16} \rightarrow V_8$, nasledovne [18]:

$$l(a_{15}, \ldots, a_0) = \nabla(148 \cdot \Delta(a_{15}) + 32 \cdot \Delta(a_{14}) + 133 \cdot \Delta(a_{13}) + 16 \cdot \Delta(a_{12}) + 194 \cdot \Delta(a_{11}) + 192 \cdot \Delta(a_{10}) + 1 \cdot \Delta(a_9) + 251 \cdot \Delta(a_8) + 1 \cdot \Delta(a_7) + 192 \cdot \Delta(a_6) + 194 \cdot \Delta(a_5) + 16 \cdot \Delta(a_4) + 133 \cdot \Delta(a_3) + 32 \cdot \Delta(a_2) + 148 \cdot \Delta(a_1) + 1 \cdot \Delta(a_0)) \text{ pre každé } a_i \in V_8, \; i = 0, 1, \ldots, 15. \quad (75)$$

3.1.4.1 Proces generovania kľúčov

V procese generovania kľúčov sú použité konštanty $C_i \in V_{128}$, $i = 1, 2, ..., 32$, definované ako [18]:

$$C_i = L\big(Vec_{128}(i)\big), i = 1, 2, \ldots, 32. \quad (76)$$

Z toho vyplýva, konštánt je celkovo 32, pričom každá z nich je vygenerovaná pomocou transformácie Vec_{128}, čiže bijektívneho nelineárneho mapovania na ktoré následne aplikujeme transformáciu L, teda lineárne mapovanie.

Kľúče pre jednotlivé iterácie $K_i \in V_{128}$, $i = 1, 2, ..., 10$, sú odvodené z kľúča K [18]:

$$K = k_{255} \parallel \cdots \parallel k_0 \in V_{256}, k_i \in V_1, i = 0, 1, \ldots, 255 \quad (77)$$

nasledovne

$$K_1 = k_{255} \parallel \cdots \parallel k_{128}; \quad (78)$$

$$K_2 = k_{127} \parallel \cdots \parallel k_0 \quad (79)$$

$$(K_{2i+1}, K_{2i+2}) = F\big[C_{8(i-1)+8}\big] \ldots F\big[C_{8(i-1)+1}\big](K_{2i-1}, K_{2i}), i = 1, 2, 3, 4. \quad (80)$$

Teda podkľúče K_1 a K_2 sú odvodené z kľúča K tak, že kľúč 256 bitový kľúč K je rozdelený na dve polovice a prvých 128 bitov (zľava) tvorí kľúč K_1 a zvyšných 128 bitov tvorí kľúč K_2, tak ako to možeme vidieť na rovniciach *(78)* a *(79)*. Rovnica *(80)* potom popisuje tvorbu kľúčov K_3 až K_{10}. Tieto sú vytvorené po dvojiciach ako súčin výstupov ôsmich F transformácií, kde ako vstupné parametre používame príslušné konštanty C (vždy 8 pre každú dvojicu podkľúčov) a dvojicu podkľúčov z predchádzajúcej iterácie. Napríklad teda pre podkľúče K_7 a K_8 a teda tretiu iteráciu ($i = 3$) by rovnica vyzerala nasledovne :

$$(K_7, K_8) = F[C_{24}]F[C_{23}]F[C_{22}]F[C_{21}]F[C_{20}]F[C_{19}]F[C_{18}]F[C_{17}](K_5, K_6) \quad (81)$$

3.1.4.2 Šifrovanie

Šifrovací proces je aplikovaný na blok vstupných dát (OT) v desiatich iteráciách s použitím desiatich podkľúčov K_1, K_2, ..., K_{10}. Princíp generovania jednotlivých podkľúčov opisuje predchádzajúca kapitola **3.1.4.1.**

Jedná sa o substitučný proces využívajúci súčin transformácií L, S, X v deviatich iteráciách s podkľúčom príslušným pre danú iteráciu alebo len transformáciu X s príslušným podkľúčom v závislosti na aktuálnom podkľúči, tak ako to opisuje rovnica [18]:

$$E_{K_1,\ldots,K_{10}}(a) = X[K_{10}]LSX[K_9] \ldots LSX[K_2]LSX[K_1](a), \quad (82)$$

kde $a \in V_{128}$.

Pre ktorýkoľvek z podkľúčov K_1, K_2, ..., K_9 vykonáme (v príklade využijeme K_7):

$$E_{K_7}(a) = LSX[K_7] \quad (83)$$

Pre posledný podkľúč K_{10} (a teda poslednú iteráciu) vykonáme:

$$E_{K_{10}}(a) = X[K_{10}] \quad (84)$$

3.1.4.3 Dešifrovanie

Dešifrovací proces je v princípe reverzným šifrovacím procesom. Rovnako ako v prípade šifrovacieho procesu teda pozostáva z desiatich iterácií a využíva buď súčin funkcií $S^{-1}L^{-1}X$ alebo len funkciu X v závislosti na aktuálnom podkľúči tak ako to opisuje rovnica [18]:

$$D_{K_1,\ldots,K_{10}}(a) = X[K_1]S^{-1}L^{-1}X[K_2] \ldots S^{-1}L^{-1}X[K_9]S^{-1}L^{-1}X[K_{10}](a), \quad (85)$$

kde $a \in V_{128}$.

Pre prvý podkľúč K_1 (a teda prvú iteráciu) vykonáme:

$$D_{K_1}(a) = X[K_1] \quad (86)$$

Pre ktorýkoľvek zo zvyšných podkľúčov K_1, K_2, ..., K_9 vykonáme (v príklade využijeme K_3):

$$D_{K_3}(a) = S^{-1}L^{-1}X[K_3] \quad (87)$$

3.2 Symetrické prúdové šifrovacie systémy

Prúdové šifrovacie systémy šifrujú vstupnú bitovú sekvenciu dát bit po bite. Výsledok šifrovania daného bitu zväčša nezávisí od ostatných bitov v sekvencií. Šifrovanie spravidla realizujeme pomocou binárnych operácií medzi jednotlivými bitmi OT a kľúča. Podľa toho tieto šifry ďalej delíme na synchrónne, kedy sekvencia kľúča použitého na šifrovanie nie je závislá na výstupe (ŠT) a asynchrónne kedy sekvencia kľúča závisí od kľúča samotného, ale taktiež od výstupu zo šifry (ŠT). [2] [3]

3.2.1 ChaCha20

ChaCha20 je prúdová šifra vyvinutá D. J. Bernstein-om v roku 2008. Jedná sa v princípe o modifikáciu šifry, ktorú publikoval v roku 2005 pod názvom Salsa za účelom zvýšenia "rozptýlenia" dát v rámci každej iterácie, čím sa zvýšila bezpečnosť, avšak bez nežiadúceho inkrementu času potrebného na ich zašifrovanie. Najpoužívanejším variantom je ChaCha20/20 kedy šifrovací proces pozostáva z dvadsiatich iterácií. Avšak existuje aj variant so šestnástimi iteráciami (ChaCha20/16) a variant s ôsmimi iteráciami (ChaCha20/8), kedy sa so znižujúcim počtom iterácií zvyšuje rýchlosť šifrovania avšak znižuje sa bezpečnosť šifrovania. [19] [20]

3.2.1.1 Matrix

ChaCha operuje na šestnástich 32 – bitových blokov (spolu teda 512 bitov), ktoré sú uložené do matice 4x4 nasledovne [19] [20]:

$$\begin{pmatrix} KONŠTANTA & KONŠTANTA & KONŠTANTA & KONŠTANTA \\ KĽÚČ & KĽÚČ & KĽÚČ & KĽÚČ \\ KĽÚČ & KĽÚČ & KĽÚČ & KĽÚČ \\ VSTUP & VSTUP & VSTUP & VSTUP \end{pmatrix}$$

Konštanty sú *expand 32-byte* k a *expand 16-byte k* v ASCII. Kľúč predstavuje tajný kľúč zadaný užívateľom a má veľkosť 256 bitov (6 x 32 bitov). Vstupmi rozumieme kryptografickú noncu a číslo bloku. Kryprografická nonca v princípe predstavuje náhodné číslo, ktoré je použité vždy len raz. To znamená, že aj pokiaľ použijeme rovnaký kľúč a konštanty, kedže tie sa nemenia, na zašifrovanie rovnakých vstupných dát zmenou nonce docielime rozdielne výstupy. Nonca zvyčajne zaberá 64 bitov a čislo bloku 32 bitov teda výsledná matica vyzerá nasledovne [19] [20]:

$$\begin{pmatrix} KONŠTANTA & KONŠTANTA & KONŠTANTA & KONŠTANTA \\ KĽÚČ & KĽÚČ & KĽÚČ & KĽÚČ \\ KĽÚČ & KĽÚČ & KĽÚČ & KĽÚČ \\ ČÍSLO\ BLOKU & NONCA & NONCA & NONCA \end{pmatrix}$$

Takto zostavená matica následne prechádza cez r iterácií kde $r \in \{8, 16, 20\}$. Pre ChaCha20/20 je $r = 20$. Tieto iterácie sú následne rozdelené na stĺpcové (z angl. column) a diagonálne (z angl. diagonal) podľa toho akým spôsobom je matica upravovaná (s akými indexmi v matici pracujeme) a teda akú funkciu na jej úpravu voláme. Môže sa jednať o *columnround* funkciu, kedy maticu upravujeme po stĺpcoch alebo *diagonalround* funkciu, kedy maticu upravujeme diagonálne. Columnround funkcia a diagonalround funkcia sa počas iterácií striedajú, teda v prvej iterácií je volaná columnround funkcia, v druhej diagonalround funkcia, v tretej opäť columnround funkcia, v štvrtej opäť diagonalround funkcia atď. až do požadovaného počtu iterácií. [19] [20]

Po takejto úprave je všetkých šestnásť slov matice (1 slovo = 32 bitov) sčítaných s pôvodnými šesťnástimi slovami (sčítanie predstavuje logickú operáciu ADD mod 32). Finálnych 64 bajtov je následne zreťazených a po slovách XOR-nutých so vstupnými dátami (OT) čim vzniká ŠT. Pokiaľ je bitová dĺžka OT dlhšia ako 512 bitov čo je bitová dĺžka výstupného bloku šifry je čislo bloku inkrementované, nonca pozmenená a šifrovací proces sa opakuje až pokým vygenerujeme dostatočný počet bajtov resp. bitov na zašifrovanie všetkých bitov vstupných dát (OT). Šifra samotná teda slúži len ako generátor kľúčovej sekvencie. Dešifrovací proces je reverzným procesom šifrovania pokiaľ sú teda vstupné dáta ŠT a nie OT, po zreťazení výstupu šifry a XOR-nutí so vstupnými dátami (ŠT) vzniká OT. [21]

3.2.1.2 Quaterround function

Quaterround funkcia upravuje sekvenciu štyroch vstupných slov na 4-slovný výstup nasledovne [20]:

$$a = a + b;\ \ d = \ d \oplus a;\ \ \ d = d \lll_{16};\tag{88}$$

$$c = c + d;\ \ b = \ b \oplus c;\ \ \ b = d \lll_{12};\tag{89}$$

$$a = a + b;\ \ d = \ d \oplus a;\ \ \ d = d \lll_{8}\tag{90}$$

$$c = c + d;\ \ b = \ b \oplus c;\ \ \ b = b \lll_7; \tag{91}$$

kde každý do znakov *a*, *b*, *c* a *d* predstavuje jedno slovo z matice.

3.2.1.3 Columnround funkcia

Columnround funkcia pozostáva zo štyroch volaní quaterround funkcie, teda upravuje sekvenciu šestnástich vstupných slov na 16-slovný výstup. Ak *x* predstavuje súbor všetkých šestnástich slov matice s ich príslušnou pozíciou v matici v podobe spodných indexov teda $x = \{x_0, x_1, \ldots, x_{15}\}$ a *y* predstavuje 16-slovný výstup z columnround funkcie s ich príslušnou pozíciou v matici v podobe spodných indexov teda $y = \{y_0, y_1, \ldots, y_{15}\}$ potom *columnround(x) = y*, kde [19] [21]:

$$(y_0, y_4, y_8, y_{12}) = quaterround(x_0, x_4, x_8, x_{12}) \tag{92}$$

$$(y_1, y_5, y_9, y_{13}) = quaterround(x_1, x_5, x_9, x_{13}) \tag{93}$$

$$(y_2, y_6, y_{10}, y_{14}) = quaterround(x_2, x_6, x_{10}, x_{14}) \tag{94}$$

$$(y_3, y_7, y_{11}, y_{15}) = quaterdround(x_3, x_7, x_{11}, x_{15}) \tag{95}$$

3.2.1.4 Diagonalround funkcia

Diagonalround funkcia rovnako ako columnround funkcia pozostáva zo štyroch volaní quaterround funkcie, teda upravuje sekvenciu šestnástich vstupných slov na 16-slovný výstup avšak prvky matice vstupujú do quaterround funkcie v inom poradí. Ak *x* predstavuje súbor všetkých šestnástich slov matice s ich príslušnou pozíciou v matici v podobe spodných indexov teda $x = \{x_0, x_1, \ldots, x_{15}\}$ a *y* predstavuje 16-slovný výstup z diagonalround funkcie s ich príslušnou pozíciou v matici v podobe spodných indexov teda $y = \{y_0, y_1, \ldots, y_{15}\}$ potom *diagonalround(x) = y* kde [19] [21]:

$$(y_0, y_5, y_{10}, y_{15}) = quaterround(x_0, x_5, x_{10}, x_{15}) \tag{96}$$

$$(y_1, y_6, y_{11}, y_{12}) = quaterround(x_1, x_6, x_{11}, x_{12}) \tag{97}$$

$$(y_2, y_7, y_8, y_{13}) = quaterround(x_2, x_7, x_8, x_{13}) \tag{98}$$

$$(y_3, y_4, y_9, y_{14}) = quaterround(x_3, x_4, x_9, x_{14}) \tag{99}$$

3.3 Vernamova šifra

Aj napriek tomu, že Vernamova šifra bola patentovaná už v roku 1918 Gilbertom Vernamom a v princípe sa jedná o modifikáciu Vigenerovej šifry, ktorá využíva kľúč o rovnakej dĺžke ako je dĺžka OT, jedná sa o jedinú šifru, ktorá preukázateľne poskytuje bezpodmienečnú bezpečnosť – teda je preukázateľne neprelomiteľná – za splnenia určitých podmienok. Medzi tieto podmienky patrí to, že kľúč musí byť skutočne náhodný (pseudonáhodné generátory teda nepripadajú do úvahy), ďalej by mali by existovať presne dve kópie náhodného kľúča (jedna na dešifrovanie, jedna na šifrovanie) a nakoniec, že každá z týchto kópií by mala byť použitá len raz a ideálne by mala byť zničená po tom ako bola použitá. V ďalšom odseku si len veľmi stručne priblížime fungovanie One-time Pad verzie tejto šifry ako aj to prečo je za splnenia vyššie uvedených podmienok neprelomiteľná. [22]

Princípom šifrovania je že vstupné dáta (OT) ako aj kľúč sú prevedené do binárnej podoby a kedže OT je rovnako dlhý ako kľúč dáta sú zašifrované pomocou funkcie XOR bit po bite. Fakt, že sekvencia bitov kľúča je dokonale náhodná spôsobí, že aj výsledný ŠT bude dokonale náhodná sekvencia bitov a teda diferenciálna kryptoanalýza nemá v tomto prípade zmysel. Rovnako nemá zmysel ani útok hrubou silou a to ani v prípade kedy by mal útočník k dispozícií nekonečný (a teda nereálny) výpočetný výkon, proti ktorému neobstojí v princípe žiadna iná šifra. A to preto, že aj keby útočník dokázal vyskúšať všetky kombinácie kľúča binárnej dĺžky *n* získal by len všetky možné OT dĺžky *n* a kedže binárna dĺžka aj relatívne krátkeho textu je pomerne veľká bolo by extrémne ťažké zistiť, ktorá správa je tá "správna". Pokiaľ budeme uvažovať, že jeden znak textu zodpovedná ôsmim bitom a chceli by sme zašifrovať text : "Toto je super tajná sprava", dostaneme bitový reťazec o dĺžke 176 bitov. Prípadný útočník by teda pri úspešnom útoku hrubou silou už pri takto krátkom texte dostal 176 možných ŠT. [22] [23]

Avšak aj napriek tomu, že One-time Pad prestavuje veľmi silnú šifrovaciu metódu jeho nedostatky znemožňujú jeho použitie v praxi. Jednou z nevýhod je generovanie samotného kľúča, kedže vygenerovať skutočne náhodný (nie pseudonáhodný) kľúč nie je jednoduché, ďalej je to potreba tento kľúč predom zdieľať, kedy opätovne vznikajú isté bezpečnostné riziká a v neposlednom rade je to dĺžka kľúča. Keby sme pomocou šifrovacej metódy One-Time Pad chceli zašifrovať dáta o veľkosti napr. 1 GB potrebovali by sme kľúč rovnakej veľkosti.

3.4 Asymetrické šifrovacie systémy

3.4.1 Jednocestné funkcie

V kapitole **1.3.2** bolo povedané, že algoritmy asymetrickej kryptografie vo veľkej miere využívajú a spoliehajú sa na tzv. jednocestné funkcie – teda funkcie o ktorých sa vo všeobecnosti verí, že sú pomerne ľahko vykonateľné v jednom smere, ale je pomerne ťažké ich vykonať v reverznom smere. Pokiaľ teda uvažujeme funkciu f s doménou X a kodoménou Y môžeme povedať, že funkcia f je jednocestná funkcia ak hodnota $f(x)$ je ľahko spočítateľná, teda spočítateľná v polynomiálnom čase, pre každé $x \in X$. Avšak pre všetky prvky $y \in Y$, je síce teoreticky výpočtovo možné, ale nepraktické nájsť (spočítať) také y pre ktoré platí $f(x) = y$. Za veľmi jednoduchý príklad môžeme pokladať napríklad miešanie farieb, kedy zmiešať dve rozdielne farby je veľmi jednoduchý proces, avšak je takmer nemožné zo zlúčeniny vzniknutej zmiešaním týchto farieb následne získať pôvodné farby. Za ďalšie príklady môžeme považovať násobenie a spätnú faktorizáciu prvočísel (ktorú využíva napríklad šifra RSA), problematiku diskrétneho logaritmu (na ktorej je založený napríklad Diffie-Helman protokol výmeny kľúčov), problematiku eliptických kriviek (ktorú je použiteľná v širokej škále už existujúcich kryptologických systémov na zvýšenie ich bezpečnosti), komplexnú modulárnu aritmetiku a isté princípy jednocestných funkcií môžeme vidieť aj v prípade HASH funkcií. [24]

3.4.2 HASH funkcie

Hash funkcia sú funkcie použiteľné v prípade, že potrebujeme dáta zašifrovať, ale už ich nepotrebujeme dešifrovať. Príkladom môže byť uloženie hesiel užívateľov do systému, kedy heslo užívateľa zašifrujeme HASH funkciou a do systému uložíme tento hash nie heslo samotné. Vzhľadom k tomu, že proces šifrovania je pevne daný a nemenný v prípade, že sa užívateľ chce prihlásiť do systému a zadá heslo systém opätovne vygeneruje hash a porovná ho s hash-om v systéme. [25]

Aby sme funkciu vôbec mohli pokladať za HASH funkciu, musí spĺňať niekoľko podmienok. Funkciu h považujeme za HASH funkciu, ak spĺňa aspoň nasledujúce podmienky [26]:

- *kompresiu* – funkcia h mapuje vstup x variabilnej bitovej veľkosti n na výstup $h(x)$ fixnej bitovej veľkosti n – v závislosti na použitom algoritme,

- *výpočtovú jednoduchosť* – výstup funkcie h pre dané x teda $h(x)$ je ľahko spočítateľný,
- *lavínovosť* – výstup h pre dané x musí natoľko závisieť na každom bite x, že aj zmena jedného bitu v x spôsobí výraznú zmenu výstupu $h(x)$.

HASH funkcie ďalej delíme na HASH funkcie bez tajného kľúča a HASH funkcie s tajným kľúčom. Ako príklad na HASH funkciu bez tajného kľúča použijeme tzv. *MDC* alebo *manipulation detection codes*, ktoré primárne (často aj v kombinácií s inými funkciami a postupmi) slúžia na detekciu zmien vo vstupných dátach, resp. na uistenie sa, že dáta zmenené neboli. Príkladom môže byť, vygenerovanie hash-u z veľkého dátového súboru, ktorý následne odošleme nechráneným komunikačným kanálom a hash vygenerovaný z tohto súboru, ktorý je výrazne menší ako pôvodný súbor, kanálom chráneným. Následne z dát ktoré obdržal príjemca opätovne vygeneruje hash (pomocou rovnakého algoritmu) a porovná ho s hash-om, ktorý obdržal prostredníctvom chráneného komunikačného kanálu. Kedže ako sme povedali, aj minimálna zmena vstupných dát sa vo veľkej miere odrazí na vygenerovanom hash-y, môžeme sa takto uistiť, že ak sa príjemcom vygenerovaný hash a hash, ktorý príjemca obdržal zhodujú, s dátami sa počas ich prenosu od odosielateľa k príjemcovi nemanipulovalo. Avšak aby sme mohli HASH funkciu považovať za *MDC* musí spĺňať dodatočné podmienky okrem podmienok, ktoré sme už definovali pre HASH funkcie vo všeobecnosti a to [25] [26]:

- *jednocestnosť* – musí platiť, že je jednoduché pre doménu X a kodoménu Y spočítať pre dané $x \in X$ výstup $h(x)$ avšak je extrémne náročné a nepraktické nájsť (spočítať) také $y \in Y$, pre ktoré platí, že $h(x) = y$,
- *slabú bezkolíznosť* – musí platiť, že pre funkciu h s doménou X nie je možné v rozumnom čase nájsť pre dané x_1 také $x_2 \neq x_1$ pre ktoré platí $h(x_1) = h(x_2)$. Teda nie je možné nájsť pre daný známy vstup iný, rozdielny vstup pre ktorý vygenerujeme rovnaký hash,
- *silnú bezkolíznosť* – musí platiť, že pre funkciu h s doménou X nie je možné v rozumnom čase nájsť také x_1 a $x_2 \neq x_1$ pre ktoré platí $h(x_1) = h(x_2)$. Teda nie je možné nájsť dve rozdielne vstupné dáta pre ktoré vygenerujeme rovnaký hash.

Ako príklad na funkciu s tajným kľúčom použijeme tzv. *MAC* alebo *message authentication code*, *MAC* a *MDC* algoritmy v princípe zdieľajú svoje primárne použitie a to uistenie sa, že s dátami sa pred ich doručením adresátovi nemanipulovalo. V prípade *MDC* algoritmov boli

jediným vstupným parametrom potrebným na vygenerovanie hash-u pre dané vstupné samotné vstupné dáta. *MAC* algoritmy pri tvorbe hash-u pre dané vstupné dáta taktiež vyžadujú ďalší vstupný parameter a to tajný kľúč, na ktorom sa účastníci komunikácie predom dohodli, napríklad pomocou protokolov výmeny kľúčov. *MAC* algoritmy teda pozostávajú z troch častí [27]:

1. *algoritmu generujúceho tajný kľuč* – prevažne sa jedná o bitový reťazec dĺžky *n*,
2. *algoritmu generujúceho "MAC" (hash) pre dané vstupné dáta* – v princípe sa jedná o HASH algoritmus generujúci hash fixnej bitovej dĺžky pre dvojicu daných vstupných dát variabilnej bitovej dĺžky a kľúča. Pokiaľ teda uvažujeme MAC algoritmus h s kľúčom k a vstupnými dátami x výstupné dáta y (hash) získame ako $y = h_k(x)$,
3. *algoritmu verifikujúceho "MAC" (hash) dát* – v praxi sa jedná o proces opätovného vygenerovania hash-u pre dvojicu vstupných dát a tajného kľuča a následného porovnania vygenerovaného hash-u a hash-u, ktorý bol adresátovi doručený spolu s dátami.

Pri *MDC* algoritmoch teda mohol ktokoľvek s prístupom k vstupným dátam a znalosťou použitého hash-ovacieho algoritmu vygenerovať hash pre dané vstupné dáta. V prípade *MAC* algoritmov znalosť "len" týchto dvoch premenných nie je postačujúca, kedže je nutné poznať aj tajný kľúč.

3.4.3 Protokoly výmeny kľúčov

Ako sme uvideli v kapitole **1.3.2** asymetrické šifrovacie systémy využívajú dvojicu kľúčov tzv. kľúčový pár, pozostávajúci z VK a PK, na ktorom sa účastníci komunikácie predom dohodli a medzi kľúčmi existuje isté spojenie. Rovnako sme ako jednu (a diskutabilne možno hlavnú) z výhod tohto prístupu uviedli, že nie je nutné kľuč zdieľať predom. Logicky teda vyvstáva otázka akým spôsobom by sme mohli čo možno najviac zabezpečiť proces zdieľania kľúča pred zahájením komunikácie v prípade symetrického kľúča. Kedže ako sme uviedli symetrické algoritmy majú v porovnaní s asymetrickými algoritmami nižšiu výpočtovú náročnosť (a teda aj časovú náročnosť) a preto sú vhodnejšie na prenosy väčších objemov dát. Odpoveďou na túto otázku sú *protokoly výmeny kľúča* (z angl. Key-exchange protocols) niekedy taktiež nazývané *protokoly dohody o kľúči* (z angl. Key-agreement protocols). Tieto protokoly sa vo väčšine prípadov snažia o vygenerovanie a výmenu

zdieľaného kľúča tak, aby sa účastníci komunikácie mohli na kľúči dohodnúť predom, ale bez nutnosti zdieľania kľúča samotného prostredníctvom komunikačných kanálov. [26] [28]

3.4.3.1 Diffie-Hellman Key Agreement

Skutočnosť, že dvaja účastníci komunikácie by mohli bez nutnosti stretnutia alebo zdieľania tajného kľúča, prípadne iného tajného obsahu dospieť k spoločnému tajnému kľúču bola prvýkrát navrhnutá Diffie-m a Hellman-om v roku 1976. Pôvodný protokol umožňoval komunikáciu dvoch účastníkov prostredníctvom nezabezpečeného komunikačného kanálu za účelom vytvorenia spoločného symetrického kľúča pričom bol (a je) založený na jednocestnej funkcií diskrétneho logaritmu. Nevýhodou bola náchylnosť tohto postupu na tzv. *Man-in-the-Middle attack*, tento problém bol odvtedy vyriešený pridaním autentizácie do protokolu a teda je aj napriek svojmu veku stále frekventovane využívaným protokolom. V súčasnosti už vo veľkej miere v spojení s eliptickými krivkami, ktorým je venovaná kapitola **3.4.6**. V ktorej sa taktiež nachádza príklad vytvorenia zdieľaného kľúča pomocou Diffie-Hellman protokolu využívajúceho eliptické krivky. Teraz si popíšeme kroky potrebné na vytvorenie zdieľaného kľúča pomocou "klasickej" verzie Diffie-Hellman protokolu. Na začiatku sa účastníci komunikácie (Alica a Bob), ktorí chcú vygenerovať spoločný kľúč K, dohodnú na spoločných, verejne známych číslach [28]:

- prvočísle p,
- čísle α takom, že platí $1 \leq \alpha \leq p - 1$ a súčasne $\alpha^{p-1} \equiv 1 \pmod{p}$.

Následne Alica a Bob postupujú v rámci protokolu nasledovne :

1. Alica vygeneruje náhodné čislo x také, že platí $1 \leq x \leq p - 2$ a spočíta $a = \alpha^x \bmod p$ a odošle a Bobovi,
2. Bob vygeneruje náhodné čislo y také, že platí $1 \leq y \leq p - 2$ a spočíta $b = \alpha^y \bmod p$ a odošle b Alici,
3. Alica po obdržaní b od Boba spočíta kľúč $K_A = b^x \bmod p$ $(\alpha^{yx} \bmod p)$,
4. Bob po obdržaní a od Alice spočíta kľúč $K_B = a^y \bmod p$ $(\alpha^{xy} \bmod p)$.

Teda platí $K_A = K_B$ a teda Alica a Bob majú svoj spoločný, zdieľaný kľúč K bez nutnosti jeho zdieľania predom. Aj napriek tomu, že účastníci komunikácie generujú symetrický kľúč nachádzame tu isté prvky asymetrickej kryptografie kedy x a y predstavujú v podstate PK oboch účastníkov a a a b predstavujú akoby VK.

Už spomínaný Man-in-the-Middle attack na tento protokol spočíva v tom, že Natália odpočúva verejnú komunikáciu medzi Alicou a Bobom. Najskôr zachytí ich vzájomnú dohodu na p a α, následne si zvolí vlastné číslo n a zachytí prenos $a = \alpha^x \bmod p$ od Alice k Bobovi a Alici následne odošle $n = \alpha^n \bmod p$ predstierajúc, že je Bob. Rovnako potom odošle $n = \alpha^n \bmod p$ Bobovi predstierajúc, že je Alica a zachytí prenos $b = \alpha^y \bmod p$ od Boba k Alici. Alica spočíta $K_A = n^x \bmod p$ ($\alpha^{nx} \bmod p$) a Natália $K_{AN} = a^n \bmod p$ ($\alpha^{xn} \bmod p$). Bob spočíta $K_B = n^y \bmod p$ ($\alpha^{ny} \bmod p$) a Natália $K_{BN} = b^n \bmod p$ ($\alpha^{yn} \bmod p$). Natália teda bude mať spoločný zdieľaný kľúč s Alicou kedže $K_{AN} = K_A$ a spoločný zdieľaný kľúč s Bobom kedže $K_{BN} = K_B$, pričom $K_{AN} \neq K_{BN}$. Pokiaľ teda Alica zašifruje správu kľúčom K_A a pošle ju Bobovi, Natália ju zachytí a dešifruje pomocou K_{AN}, čím logicky získa jej obsah. Následne správu zašifruje pomocou K_{BN} a odošle ju Bobovi, ktorý ju dešifruje pomocou K_B. Takto môže Alica s Bobom komunikovať bez akéhokoľvek tušenia o tom, že ich komunikáciu zachytáva Natália. Zo vzťahu $K_{AN} \neq K_{BN}$ ale vyplýva, že Alica a Bob spolu nemôžu komunikovať bez toho aby Natália zachytávala a prešifrovávala ich správy.

3.4.3.2 "Shamir's Three Pass Protocol"

Ďalšou možnosťou ako by mohli dvaja účastníci komunikácie dospieť k tajnému, zdieľanému kľúču je Shamir-ov algoritmus, ktorý na rozdiel od algoritmu navrhnutého Diffie-m a Hellman-om nevyžaduje aby účastníci komunikácie vlastnili kľúčový pár (VK a PK), každý z účastníkov vlastní len svoj symetrický kľúč. Rovnako ako Diffie-Hellman protokol Shamir-ov neposkytuje autentifikáciu – overenie pôvodu kľúča. V prípade Diffie-Hellman protokolu Alica a Bob zdieľaný kľúč vygenerovali každý nezávisle bez transferu samotného kľúča prostredníctvom nezabezpečeného komunikačného kanála. Naopak v prípade Shamir-ovho algoritmu je cez nezabezpečený komunikačný kanál bezpečne prenesený samotný kľúč. Na začiatku sa Alica a Bob dohodnú na spoločnom prvočísle p, Alica si následne zvolí číslo a a Bob číslo b, nesúdelné s p - 1 tak aby platilo [26]:

$$1 \leq a, b \leq p - 2 \qquad (100)$$

Následne pre vytvorenie každého nového zdieľaného kľúča vykonáme :

1. Alica si zvolí kľúč K tak aby platilo $1 \leq K \leq p - 1$. Spočíta $K^a \bmod p$ a výsledok odošle Bobov-i.
2. Bob spočíta $(K^a)^b \bmod p$ a odošle výsledok späť Alici.

3. Alica spočíta $(K^{ab})^{a-1} \bmod p$, čím akoby zneguje úpravu kľúča, ktorú vykonala v 1. kroku a výsledok $(K^b \bmod p)$ odošle Bobovi.
4. Bob následne spočíta $(K^b)^{b-1} \bmod p$. Čím získava spoločný zdieľaný kľúč $K \bmod p$.

Z tretieho kroku môže vidieť, že algoritmus je založený na komutatívnosti funkcií. Teda na predpoklade, že nezáleží na poradí operandov. Ak predpokladáme, že umocnenie kľúča na a alebo b je akoby šifrovací proces (S) a umocnenie kľúča na a^{-1} a b^{-1} (D) je teda proces dešifrovací môžeme kroky algoritmu zapísať v zjednodušenej podobe nasledovne :

1. Alica zašifruje kľúč pomocou a, teda spočíta $S_a(K)$ a výsledok odošle Bobovi.
2. Bob zašifruje už zašifrovaný kľúč od Alice pomocou b, teda spočíta $S_b(S_a(K))$ a výsledok odošle späť Alici.
3. Alica dešifruje dvakrát zašifrovaný kľúč pomocou a^{-1}, teda spočíta $D_{a-1}(S_b(S_a(K)))$ a tým získa $S_b(K)$, ktoré odošle Bobovi.
4. Ten dešifruje $S_b(K)$ pomocou b^{-1}, teda spočíta $D_{b-1}(S_b(K))$, čím získa zdieľaný kľúč K.

3.4.4 Digitálny podpis

Na rozdiel od fyzických dokumentov písaných rukou, v prípade digitálnych dokumentov a správ nemôžeme pohľadom rozoznať kto daný dokument vytvoril. Digitálne podpisy teda podobne ako podpisy fyzických dokumentov ubezpečujú príjemcu daného dokumentu, že dokument bol naozaj odoslaný legitímnou osobou od ktorej bol tento dokument očakávaný a aj o tom, že sa s dokumentom pred jeho prijatím nemanipulovalo. [29]

3.4.4.1 Digital Signature Schemes (DSS)

DSS zahŕňajú techniky používané na overenie, že istá entita videla, vytvorila a prípadne aj zaslala istý digitálny dokument. Rovnako okrem overenia pôvodu dokumentu vieme overiť aj jeho pravosť a autentickosť – či sa s dokumentom na ceste medzi odosielateľom a adresátom nemanipulovalo. Entita podpisujúca daný dokument vygeneruje digitálny podpis, ktorý je závislí na podpisovanom dokumente a jej privátnom kľúči. Ktokoľvek kto má následne prístup k danému dokumentu, verejnému kľúču danej entity a digitálnemu podpisu vygenerovanému danou entitou môže následne overiť pravosť dokumentu. Z faktu, že digitálny podpis je závislí na privátnom kľúči danej entity taktiež vyplýva fakt, že pokiaľ raz daná entita daný dokument podpíše, nemôže neskôr tento fakt poprieť, kedže

predpokladáme, že jej privátny kľúč nebol prezradený a teda len a len ona mohla daný dokument podpísať. Medzi príklady *DSS* patrí napríklad *RSA digital signature scheme* – ktorú budeme rozoberať v jednej z nasledujúcich kapitol*, ElGamal digital signature scheme, Schnorr digital signature scheme, DSA (Digital signature algorithm)* a mnohé iné. *DSS* algoritmy vo svojej podstate pozostávajú z troch hlavných častí a to [29]:

- algoritmu generujúceho kľúčový pár
- algoritmu slúžiaceho na podpis dokumentu
- algoritmu slúžiaceho na overenie dokumentu

3.4.4.2 RSA

V kapitole **1.3.2** sme uviedli, že rok 1976 by sa dal pokladať za rok zrodu asymetrickej kryptografie. O rok neskôr, v roku 1977, Ronald Rivest, Adi Shamir a Leonard Adleman navrhli asymetrickú šifru RSA podľa prvých písmen ich priezvisk. Šifra sa odvtedy stala najviac používanou šifrou na svete a to aj napriek príchodu šifier využívajúcich eliptické krivky alebo diskrétne algoritmy. [30]

Šifra je založená na princípe jednocestnej funkcie, konkrétne na už spomínanom probléme faktorizácie prvočísel. Teda na predpoklade, že je pomerne jednoduché dve veľké prvočísla medzi sebou vynásobiť, ale neexistuje dostatočne sofistikovaný (rýchly) algoritmus, ktorý by v polynomiálnom čase zo súčinu týchto prvočísel získal čísla pôvodné. Z toho vyplýva, že úroveň bezpečnosti ako aj neprelomiteľnosť (v polynomiálnom čase, s použitím doposiaľ známych algoritmov a výpočtovej techniky) tejto šifry spočíva v dostatočnej veľkosti zvolených prvočísel. Vo väčšine prípadov je dĺžka kľúča pre RSA v súčasnosti 2048 bitov, pričom táto dĺžka je považovaná za (zatiaľ) bezpečnú. Predpokladá sa, napr. na základe Moorovho zákona, že bude v istom bode potrebné prejsť na väčšie dĺžky kľúčov – 3072 a viac bitov. Tým sa zvýši bezpečnosť a zabezpečí sa, že šifra bude naďalej neprelomiteľná (za už zmienených predpokladov). Logicky sa ale so stúpajúcou veľkosťou a teda aj bitovou dĺžkou prvočísel s ktorými pracujeme, zvýši čas potrebný na vykonanie výpočtov v rámci šifrovacieho procesu. [30]

Pred začiatkom samotného šifrovacieho procesu je potrebné vygenerovať šifrovacie kľúče resp. kľúčový pár. Tento proces pozostáva z niekoľkých krokov [30]:

1. zvolíme dve náhodné veľké prvočísla p a q,
2. spočítame hodnotu $n = p * q$,

3. spočítame hodnotu Eulerovej funkcie $\varphi(n) = (p - 1)(q - 1)$,
4. zvolíme hodnotu časti verejného kľúča v tak aby platilo $v < \varphi(n)$ a súčasne $gcd(v, \varphi(n)) = 1$. Teda v musí byť menšie ako hodnota Eulerovej funkcie a súčasne s ňou musí byť nesúdelné,
5. následne nájdeme časť hodnoty privátneho kľúča s takú aby pre ňu platilo $s * v \equiv \mathrm{mod}(\varphi(n))$.

Verejný kľúč potom tvorí dvojica (n, v) a privátny kľúč tvorí dvojica (n, s).

Predtým než dáta zašifrujeme pomocou vygenerovaných kľúčov je nutné vstupné dáta (OT) upraviť do vhodnej podoby. Vstupné dáta môžeme previesť do binárnej podoby a následne spojiť do jedného veľkého binárneho čísla avšak nie väčšieho než $n - 1$ bitov. Pokiaľ teda budeme používať prvočísla o veľkosti 1024 bitov dostaneme n o veľkosti 2048 bitov, teda využívame šifrovací kľúč o veľkosti 2048 bitov. Keby sme teda uvažovali, že vstupné dáta sú text mohli by sme naraz zašifrovať 255 znakov (2048/8 – 1) alebo teda 255 bajtov. Keby sme chceli zašifrovať text o väčšej veľkosti bolo by nutné text rozdeliť do blokov o veľkosti 255 bajtov a šifrovať tieto bloky samostatne. Vzhľadom k tomu, že šifra je, z dôvodu časovej náročnosti pri šifrovaní väčších vstupných dát (MB, GB atď.) využívaná najmä na šifrovanie kľúčov pre symetrické algoritmy toto delenie vstupných dát na bloky zväčša nie je potrebné. Samotný šifrovací proces je potom veľmi jednoduchá matematická operácia [30]:

$$c = m^v \ mod \ n \qquad (101)$$

kde c označuje ŠT (výstupné dáta) a m označuje OT (vstupné dáta).

Proces dešifrovania je rovnako veľmi jednoduchá matematická operácia :

$$m = c^s \ mod \ n \qquad (102)$$

kde c označuje ŠT (vstupné dáta) a m označuje OT (výstupné dáta).

3.4.4.3 RSA Digital signature scheme

V dvoch predchádzajúcich kapitolách sme uviedli základné princípy fungovania *DSS* a priblížili princíp fungovania šifry RSA. V tejto kapitole len v stručnosti konkrétnejšie rozoberieme proces tvorby digitálne podpisu. Na konci kapitoly **3.4.4.1** sme uviedli tri základné časti z ktorých *DSS* pozostávajú. Prvou časťou bol algoritmus generujúci kľúčový pár. Odosielateľ teda pred podpisom musí vygenerovať v tomto prípade pomocou algoritmu

RSA svoj kľúčový pár (K_P, K_V) kde K_P označuje privátny kľúč odosielateľa, teda $K_P = (n, s)$ a K_V označuje verejný kľúč odosielateľa, teda $K_V = (n, v)$ – pokiaľ odosielateľ nemal daný kľúčový pár vygenerovaný predom. Druhou častou bol algoritmus slúžiaci na podpis dokumentu. Tu využijeme kombináciu HASH-ovacieho algoritmu (napr. SHA-512 a iné.) a šifry *RSA*. Najskôr z dokumentu, ktorý chceme podpísať vygenerujeme hash pomocou zvoleného HASH-ovacieho algoritmu, takto vzniknutý hash dokumentu označíme *H*. Následne pomocou šifry *RSA* a privátneho kľúča K_V odosielateľa zašifrujeme hash *H*, takto zašifrovaný hash správy označíme H_d. Teda platí $H_d = H^s \bmod n$. Samotný dokument by sme rovnako mohli zašifrovať danou zvolenou šifrou, ale pre jednoduchosť tento krok vynecháme. Následne dokument spolu s zašifrovaným hash-om (H_d) odošleme adresátovi. Treťou a poslednou častou bol algoritmus overujúci pravosť dokumentu. Adresát teda z obdržaného dokumentu opätovne vygeneruje hash rovnakým postupom a s použitím rovnakého HASH-ovacieho algoritmu ako odosielateľ, tento hash označíme H_A. Následne adresát dešifruje hash, ktorý obdržal od odosielateľa pomocou verejného kľúča odosielateľa teda získa $H = (H_d)^v \bmod n$. Potom už len zostáva porovnať či platí rovnosť $H_A = H$. Ak daná rovnosť platí vieme, že dokument pochádza od odosielateľa a nebol po ceste medzi odosielateľom a adresátom pozmenený. [31]

3.4.5 Homomorfné šifrovanie

V dobe kedy sa prostredníctvom výpočtovej techniky a hlavne internetu zdieľa snáď najväčšie množstvo osobných (a aj iných) údajov vôbec, čo zo sebou logicky prinieslo aj oveľa väčšie nároky na ochranu týchto dát, sa objavuje otázka ako pracovať so zašifrovanými dátami (ktoré obsahujú napríklad citlivé osobné informácie) bez nutnosti ich dešifrovania, tak aby sa zmeny v dátach po dešifrovaní prejavili rovnako ako keby sme zmeny vykonali na dátach dešifrovaných. Odpoveďou na túto otázku by malo byť práve homomorfné šifrovanie resp. homomorfné šifrovacie systémy, ktorých základné vlastnosti, využitie a to kedy vlastne šifrovací systém (alebo algoritmus) môžeme za homomorfný považovať si priblížime v tejto kapitole. Homomorfné vlastnosti asymetrických šifrovacích systémov a široké možnosti ich využitia si kryptografici všimli už začiatkom 70. rokov minulého storočia. Odvtedy sa, ako bolo spomenuté, otázka ochrany citlivých informácií viac a viac stupňovala. Homomorfné šifrovanie teda našlo svoje uplatnenie v širokej škále aplikácií v rôznych sektoroch. V medicínskom sektore napr. v prípade, že potrebujeme pracovať s údajmi o pacientovi (meno, vek, váha, zdravotný stav atď.) za účelom štatistickej analýzy náchylnosti daného pacienta na isté choroby alebo len vo všeobecnosti pri

pravidelnej kontrole jeho zdravotného stavu. Pri použití homomorfných techník môžeme vykonať analýzu na základe údajov z rôznych zdrojov, bez toho, aby sme týmto zdrojom museli poskytnúť dáta v dešifrovanej podobe. Ďalším sektorom môže byť sektor finančný, kedy jedna spoločnosť chce použiť algoritmus druhej spoločnosti napr. algoritmus odhadujúci vývoj akcií na burze, ale nechce druhej spoločnosti prezradiť svoje akciové portfólio a rovnako druhá spoločnosť nechce prvej spoločnosti prezradiť samotný algoritmus. Riešením je opäť využitie homomorfných techník, kedy tieto spoločnosti môžu medzi sebou zdieľať dáta obsahujúce informácie o akciovom portfóliu prvej spoločnosti a dáta obsahujúce algoritmus druhej spoločnosti v zašifrovanej podobe bez toho aby jedna strana zistila čokoľvek o obsahu dát protistrany. Využitie homomorfných techník nachádzame aj v oblasti *hĺbkovej analýzy dát* (z angl. *Data mining*), tajných volieb alebo aukcií, ale aj v reklamnom sektore. Kedy homomorfné techniky umožňujú pracovať s dátami o užívateľovi, na základe ktorých mu je zobrazená reklama upravená a cielená konkrétne na neho, bez toho aby sme jeho osobné údaje museli dešifrovať a tým ich vystaviť riziku. [32] [33]

Kedy teda môžeme označiť šifrovací algoritmus za homomorfný ? Za homomorfný šifrovací algoritmus vo všeobecnosti označujeme algoritmus ktorý medzi doménou OT a príslušnou kodoménou šifrového textu ponecháva po zašifrovaní istú matematickú spojitosť. Napríklad v prípade že súčin dvoch ŠT sa rovná ŠT, ktorý vznikol zo súčtu týchto ŠT pred zašifrovaním [33]:

$$S(m_1) * S(m_2) = S(m_1 + m_2) \qquad (103)$$

kde *S* označuje šifrovací algoritmus, *m* označuje OT a predpokladáme, že všetky šifrovacie procesy využívajú rovnaký kľúč. Pojem *homomorfné šifrovanie* je taktiež často priamo spájaný s matematickým pojmom homomorfizmu grúp (grupa = usporiadaná dvojica množiny a matematickej operácie na tejto množine, ktorá spĺňa tri podmienky – je asociatívna, existuje neutrálny a inverzný prvok), ktorý označuje mapovanie medzi grúpami v ktorom sa zachováva istá matematická spojitosť. [33]

3.4.6 Kryptografia založená na eliptických krivkách

Kryptografia založená na princípe eliptických kriviek bola nezávisle navrhnutá Nealom Koblitzom a Victorom Saulom Millerom v roku 1985 avšak jej implementácie do technickej praxe sa dočkala až okolo roku 2005. V kapitole **3.4.4.2** o šifre RSA sme uvideli, že v súčasnosti sú za bezpečné považované šifrovacie kľúče o veľkosti aspoň 2048 bitov

a predpokladá sa (na základe Moorovho zákona), že bude nutné prejsť na ešte dlhšie kľúče aby sa zachovala bezpečnosť a neprelomiteľnosť šifry. S takto dlhými šifrovacími kľúčmi avšak prichádza aj zvýšená výpočtová náročnosť a čas potrebný na zašifrovane dát či generovanie kľúčov (resp. kľúčových párov). Nehovoriac o tom, že 2048 bitový asymetrický kľúč poskytuje rovnakú úroveň zabezpečenia ako 112 bitový symetrický kľúč a 3072 bitový asymetrický kľúč poskytuje rovnakú úroveň zabezpečenia ako 128 bitový symetrický kľúč. Ako vhodné riešenie sa ukázalo práve využitie eliptických kriviek, aj napriek tomu, že v prípade kryptografie založenej na eliptických krivkách sa nevychádza z čisto číselného poňatia operácií, ale je založená na matematických operáciách (výpočtoch), ktoré sú na krivku aplikované pomocou geometrických operácií. Pri použití eliptických kriviek nám pre 2048 bitový asymetrický kľúč, ktorý ako bolo povedané zodpovedá približne 112 bitovému symetrickému kľúču, postačí 224 bitový kľúč a pre 3072 bitový asymetrický kľúč, ktorý teda zodpovedá približne 128 bitovému symetrickému kľúču, postačí 256 bitový kľúč. Môžeme teda vidieť, že bitové dĺžky kľúčov pre algoritmy založené na eliptických krivkách sú dvojnásobkom bitovej dĺžky kľúčov symetrických (pre rovnakú úroveň zabezpečenia), avšak stále sú tieto bitové dĺžky oveľa kratšie ako v prípade asymetrických kľúčov. [30]

V nasledujúcich častiach si zbežne priblížime rovnicu z ktorej eliptické krivky vychádzajú, princípy zostrojenia eliptických kriviek, uvedieme kedy krivku považujeme resp. nepovažujeme za eliptickú a ukážeme si jednoduchý príklad výmeny spoločného kľúča bez toho aby sme príliš zachádzali do "matematiky veci".

3.4.6.1 Rovnica eliptickej krivky

Eliptickú krivku *E* nad množinou *K* vo všeobecnosti definuje *Weierstrassova rovnica* [30]:

$$E/K: \; y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (104)$$

kde $a_1, a_2, a_3, a_4, a_6, x, y \in K$.

Na aplikáciu v praxi sa ale všeobecná *Weierstrassova rovnica* príliš nehodí pretože výpočty vykonávané pomocou tejto rovnice sa ukázali ako zbytočné náročné. Preto sa v praxi uplatnila tzv. *zjednodušená forma Weierstrassovej rovnice* [30]:

$$y^2 = x^3 + a * x + b \qquad (105)$$

kde $a, b \in K$.

3.4.6.2 Zostrojenie eliptických kriviek

Na ľavej strane *zjednodušenej formy Weierstrassovej rovnice*, sa nachádza kvadratický člen y^2. Teda logicky pre každé riešenie tejto rovnice dostaneme dve možné riešenia *y* a *-y*, ktoré spočítame na základe uvedených vzorcov [30]:

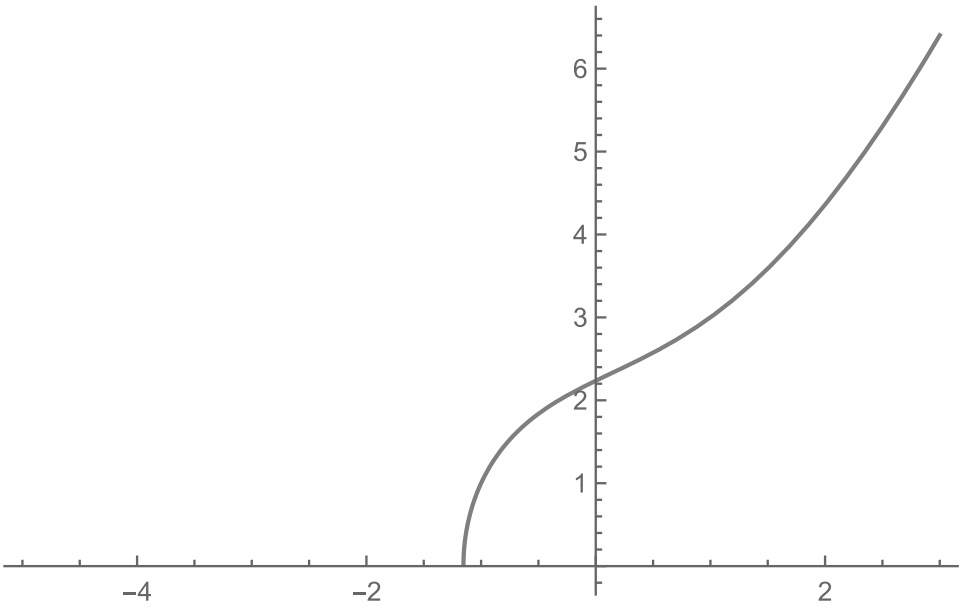
$$y = \sqrt{x^3 + a * x + b} \tag{106}$$

$$-y = -\sqrt{x^3 + a * x + b} \tag{107}$$

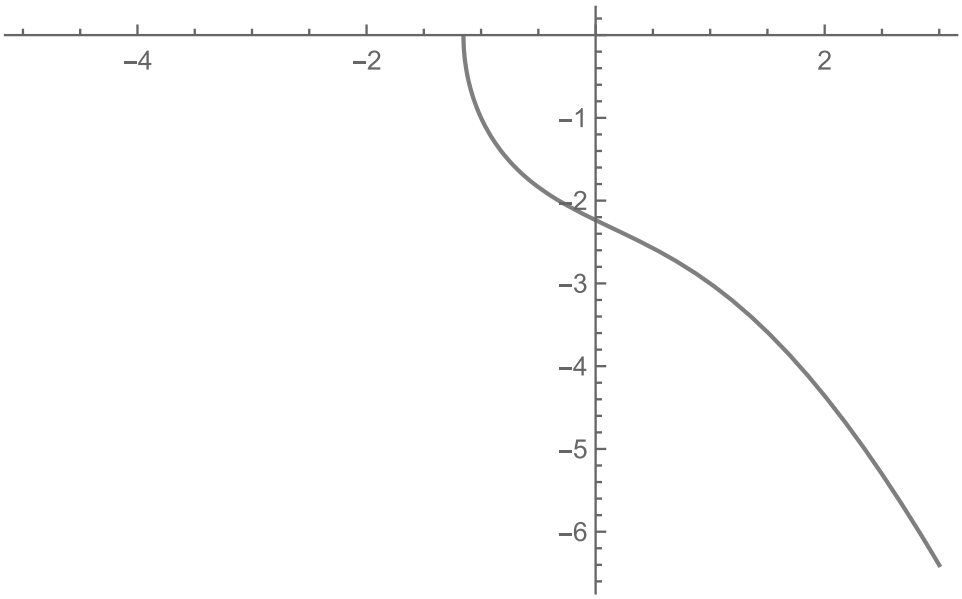
Neopomenuteľnou podmienkou zostrojenia eliptickej krivky je nutnosť voliť premenné *a*, *b* tak, aby diskriminant *D* rovnice nebol nulový. Diskriminant pre *zjednodušenú formu Weierstrassovej rovnice* spočítame ako $D = -16(4a^3 + 27b^2)$ a teda musí platiť [30]:

$$D = -16(4a^3 + 27b^2) \neq 0 \tag{108}$$

Za predpokladu vhodne zvolených premenných dostaneme pre *y* a *-y* grafy :

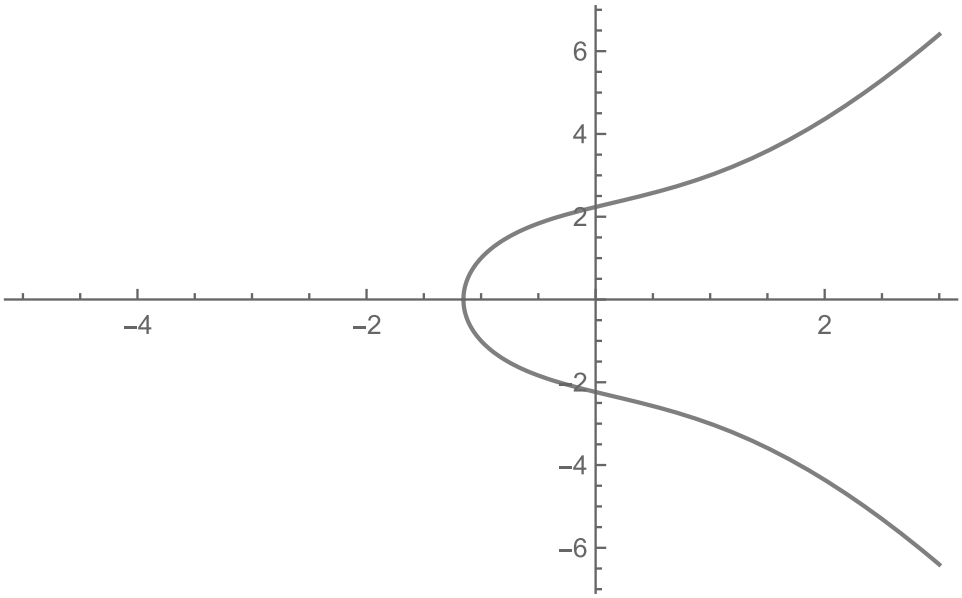


Obrázok 17 – eliptická krivka $y = \sqrt{x^3 + 3 * x + 5}$, Zdroj: vlastný



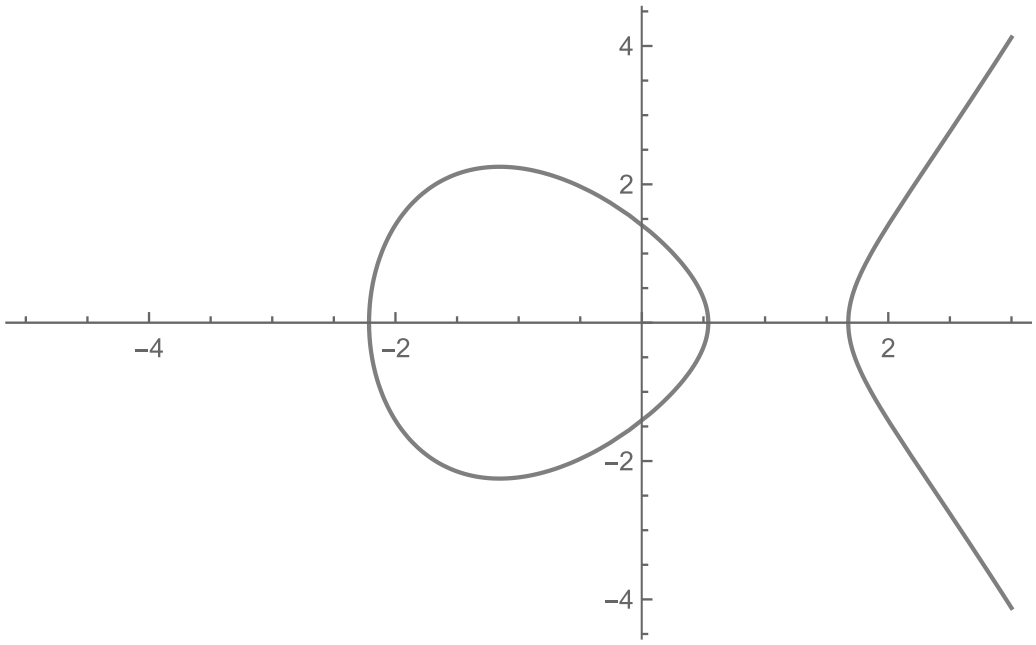
Obrázok 18 – eliptická krivka $-y = -\sqrt{x^3 + 3 * x + 5}$, Zdroj: vlastný

Výsledná krivka s ktorou budeme ďalej pracovať zobrazená na **Obrázok 19** je spojením kriviek na **Obrázok 17** a **Obrázok 18** a toto spojenie teda budeme ďalej chápať ako jednu krivku.



Obrázok 19 – spojenie grafov rovníc *y* a *-y,* Zdroj: vlastný

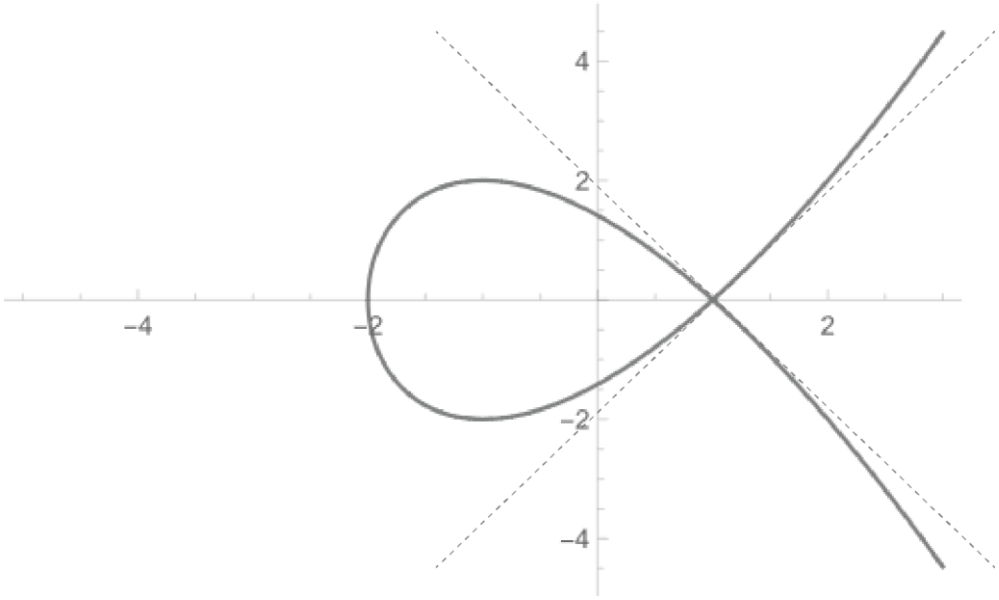
V niektorých prípadoch môže nastať situácia kedy je graf eliptickej krivky rozdelený na 2 samostatné časti ako to zobrazuje **Obrázok 20**. Toto nastáva z dôvodu, že krivka je definovaná len na intervaloch kedy výraz pod odmocninou rovníc $y = \sqrt{x^3 - 4x + 2}$ a $-y = -\sqrt{x^3 - 4x + 2}$ nadobúda kladných hodnôt. Riešenia pre hodnoty kedy výraz pod odmocninou rovníc nadobúda záporných hodnôt nie sú v obore reálnych čísel a preto nie sú na grafe zobrazené. Toto rozdelenie krivky, ale neznamená, že na nich nie je možné vykonávať algebrické operácie a teda, že nie sú vhodné na použitie. Ako môžeme vidieť krivky zostrojené pomocou rovníc so záporným ako aj kladným diskriminantom sú eliptickými krivkami a teda sú vhodné na použitie v rámci kryptografie založenej na princípe eliptických kriviek. Pričom v prípade kladného diskriminantu je eliptická krivka rozdelená na dve samostatné časti ako to zobrazuje napr. **Obrázok 20** a v prípade záporného diskriminantu je eliptická krivka tvorená jedinou spojitou častou ako to zobrazuje napr. **Obrázok 19**. [30]



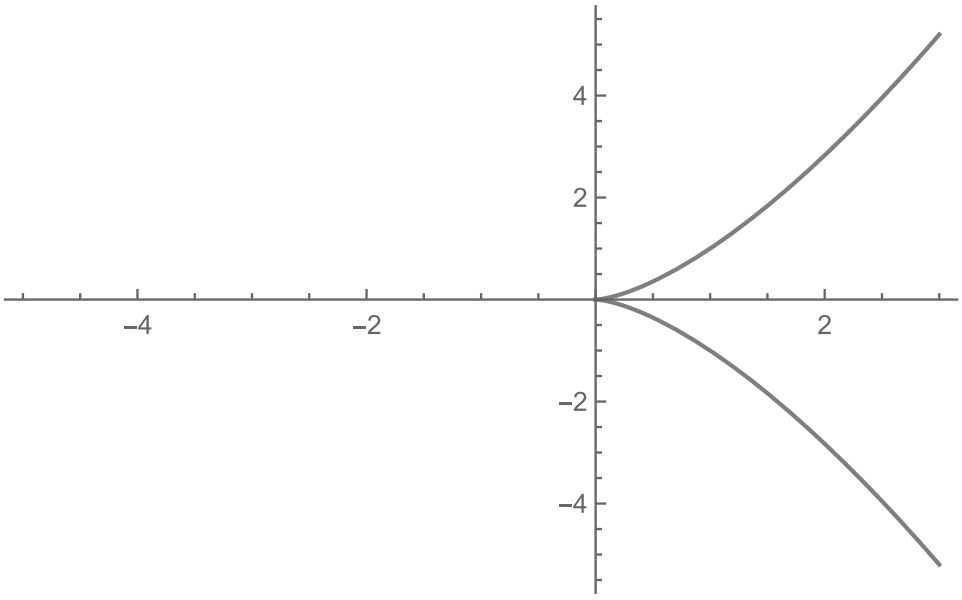
Obrázok 20 – eliptická krivka $y^2 = x^3 - 4x + 2$, Zdroj: vlastný

Pokiaľ sme teda pokryli prípad kladného ako aj záporného diskriminantu, poslednou možnosťou sú krivky tvorené rovnicami s nulovým diskriminantom o ktorých sme si v úvode tejto podkapitoly povedali, že nie sú krivkami eliptickými a teda nie sú vhodné na použitie v rámci kryptografie založenej na princípe eliptických kriviek a to preto, že

v prípade týchto kriviek sa vyskytujú tzv. *singulárne body*. Teda body v ktorých možeme zostrojiť dve dotyčnice ako to zobrazuje bod [1,0] na **Obrázok 21.** Naopak pre isté body na týchto krivkách nemôžeme zostrojiť dotyčnice vôbec napr. pre bod [0,0] na **Obrázok 22**. Ďalším problémom je, že pre isté body týchto rovníc neexistuje derivácia teda hovoríme, že tieto body sú tzv. *nediferencovateľné*. [30]



Obrázok 21 – eliptická krivka $y^2 = x^3 - 3x + 2$, Zdroj: vlastný



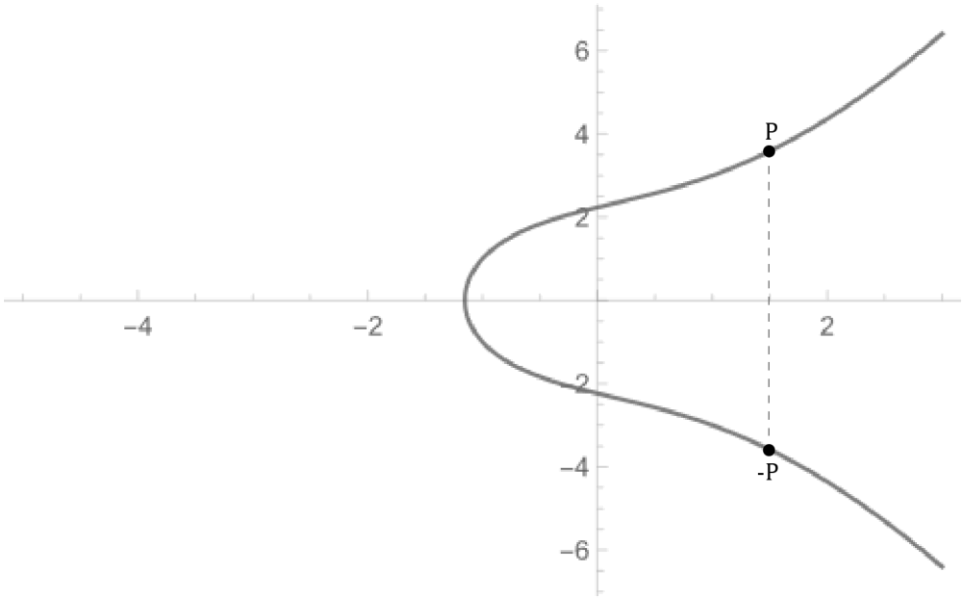
Obrázok 22 – eliptická krivka $y^2 = x^3$, Zdroj: vlastný

3.4.6.3 Algebrické operácie na eliptických krivkách

Negácia bodu je prvou algebrickou operáciou, ktorú definujeme a to z dôvodu, že sa v podstatne jedná o elementárnu operáciu, ktorú budeme využívať ako súčasť iných algebrických operácií a taktiež v rámci praktického príkladu (kapitola **3.4.6.4**) využitia eliptických kriviek pri tvorbe kľúča pomocou upraveného Diffie-Hellman protokolu z kapitoly **3.4.3.1**. Každý bod *P* ležiaci eliptickej krivke *E*, ktorý neleží v nekonečne *O*, môžeme definovať ako ustálenú dvojicu *P*[*x*, *y*] a môžeme k nemu vytvoriť opačný (negovaný) bod -*P*[*x*, -*y*] teda platí:

$$\forall P[x,y] \in E \wedge P \neq O \quad \exists \quad -P[x,-y] \tag{109}$$

Z definície teda jasne vyplýva, že negáciu bodu vykonáme ako jednoduché otočenie znamienka *y*-ovej súradnice daného bodu ako to zobrazuje **Obrázok 23**. [30]

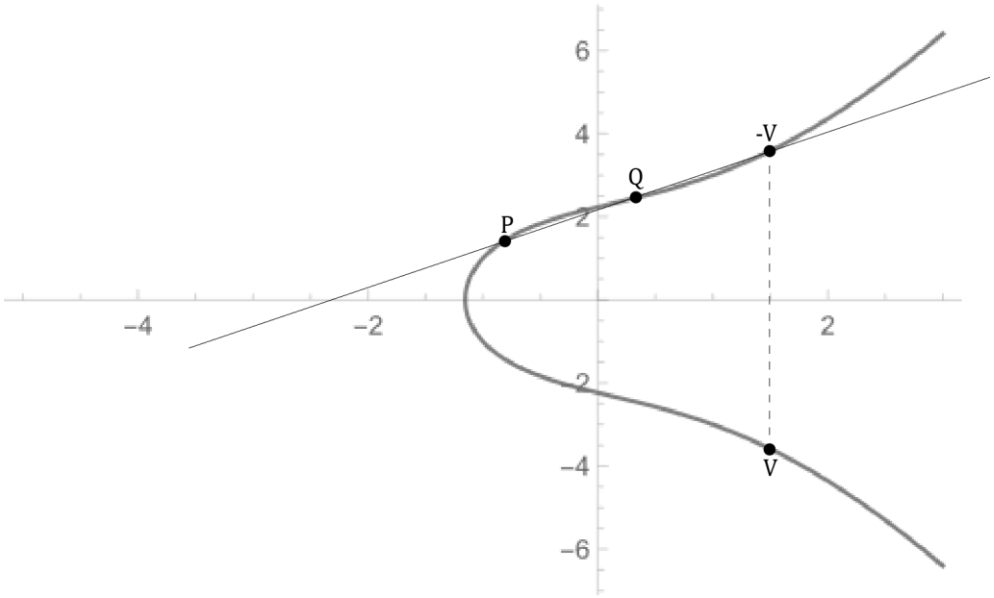


Obrázok 23 – negácia bodu *P* na eliptickej krivke $y^2 = x^3 + 3 * x + 5$, Zdroj: vlastný

Sčítanie bodov je ďalšou operáciou, ktorú si na eliptickej krivke definujeme. Budeme sčitovať body $P[x_1, y_1]$ a $Q[x_2, y_2]$, ktoré nie sú opačné a výsledkom bude bod *V*. Možno najintuitívnejším riešením by bolo vytvoriť bod *V* pomocou sčítania x-ových a y-ových súradníc bodov *P* a *Q*. Avšak toto riešenie je nesprávne teda platí [30]:

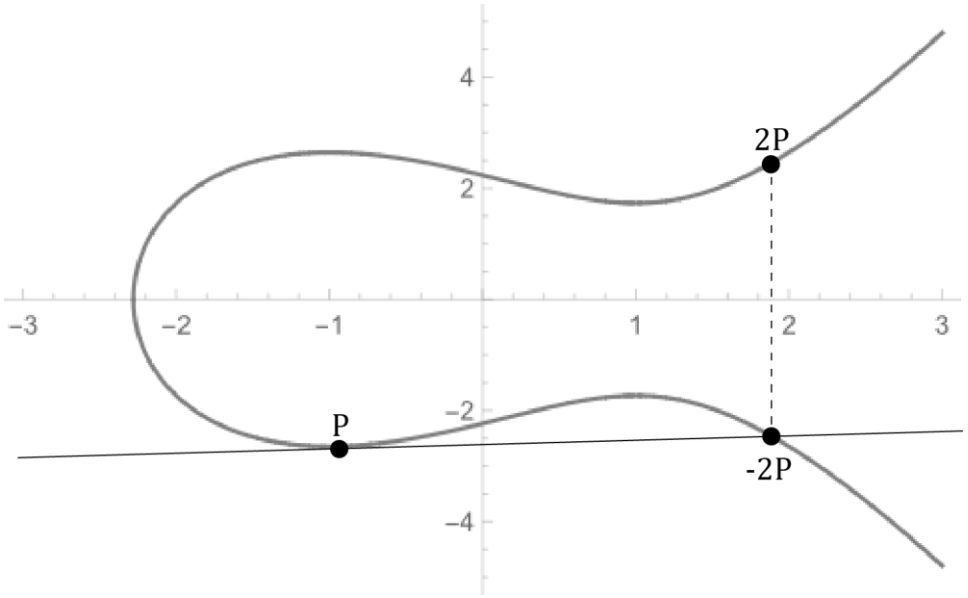
$$P[x_1, y_1] + Q[x_2, y_2] \neq V[x_1 + x_2, y_1 + y_2] \qquad (110)$$

Sčítanie dvoch bodov vykonáme tak, že oba tieto body preložíme priamkou a tam kde táto priamka pretne eliptickú krivku dostaneme bod -*V* následne vykonáme negáciu bodu -*V*, podľa postupu, ktorý sme definovali v prechádzajúcom odseku a dostávame bod *V* ako to zobrazuje **Obrázok 24**. [30]



Obrázok 24 – sčítanie bodov na eliptickej krivke $y^2 = x^3 + 3 * x + 5$, Zdroj: vlastný

Doubling je špeciálnym prípadom sčítania dvoch bodov, kedy sčitujeme dvakrát ten istý bod. Teda sčitujeme bod $P[x, y]$ a ten istý bod $P[x, y]$ a dostávame výsledok $2P = V$. Kedže, ako sme povedali, sa jedná len o špeciálny prípade sčitovania dvoch rozdielnych bodov, môžeme vychádzať z predpokladu, že sčitujeme dva body *P* a *Q*, pričom *P* = *Q*. Teda postupujeme rovnako v prípade kedy sme sčitovali dva rozdielne body. Preložíme cez tieto body priamku (kedže body *P* a *Q* ležia na tom istom mieste na eliptickej krivke, bude sa v tomto prípade jednať o dotyčnicu týchto bodov respektíve bodu *P*) a tam kde táto priamka pretne eliptický krivku dostávame bod $-2P = -V$. Vykonáme teda negáciu bodu $-2P = -V$ a dostávame bod $2P = V$. Tento postup zobrazuje **Obrázok 25**. [30]



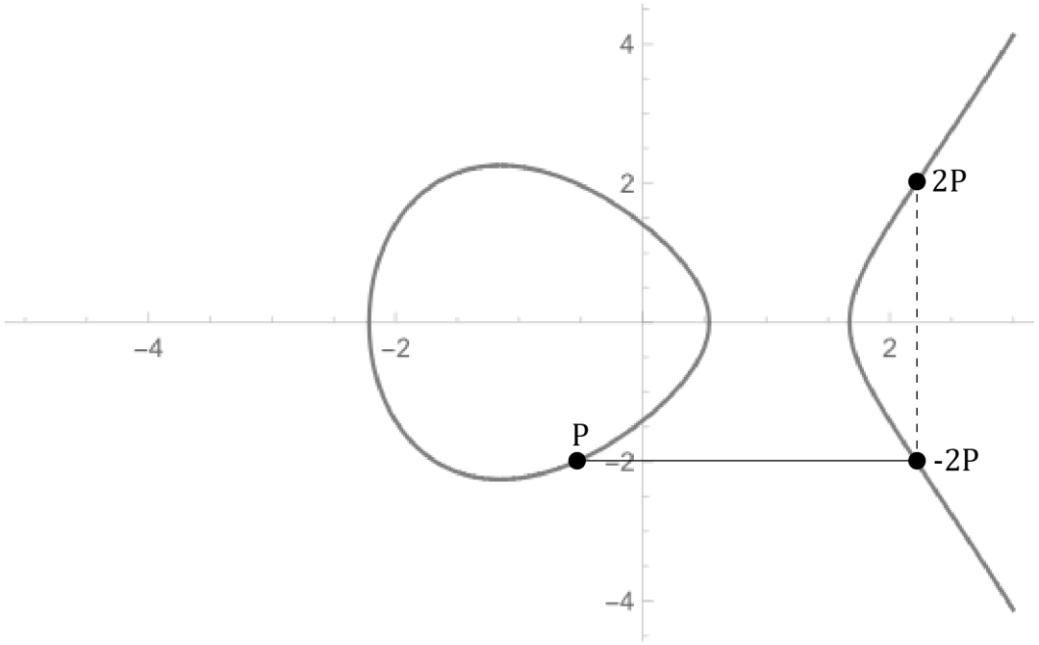
Obrázok 25 – doubling bodu *P* na eliptickej krivke $y^2 = x^3 - 3 * x + 5$, Zdroj: vlastný

Násobenie bodu skalárom je poslednou operáciou, ktorú definujeme. Jedná sa vlastne o násobenie bodu $P[x, y]$ ležiaceho na eliptickej krivke číslom *n*, väčším ako 1. Výsledkom je potom bod $V = n * P$. Bod *V* dostanem tak, že jednoducho *n*-krát sčítame bod *P*. Pokiaľ by sme teda uvažovali $n = 3$, jednoducho trikrát sčítame hodnotu *P* podľa postupu, ktorý sme popísali v odseku o sčitovaní bodov na eliptických krivkách [30]:

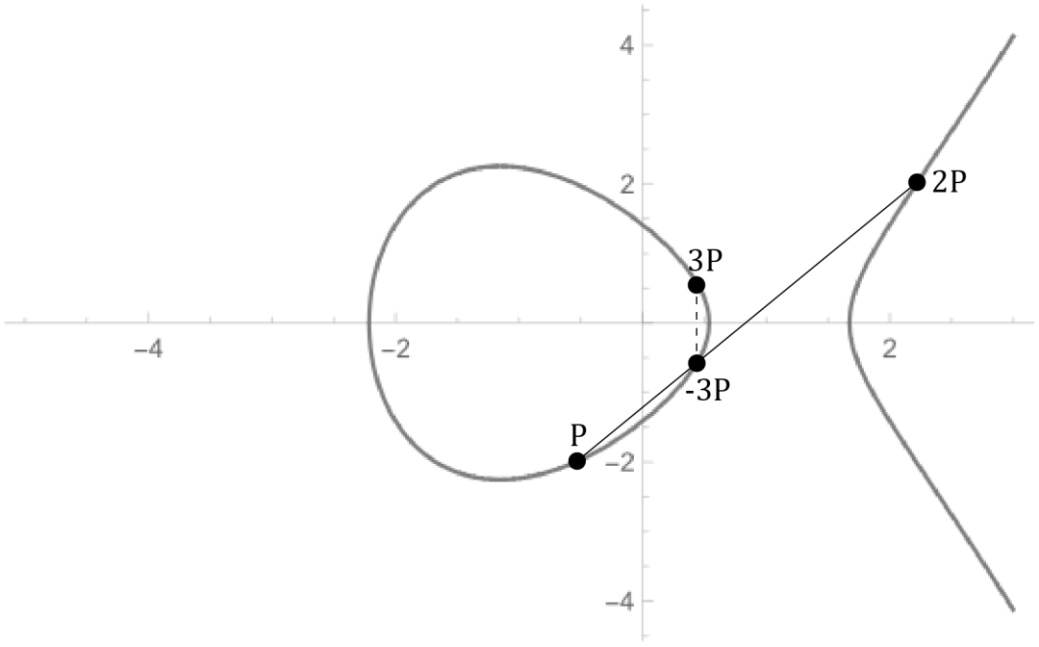
$$P + P + P = 3P = V \tag{113}$$

Sčítanie bodov na eliptických krivkách je asociatívne. Teda k výsledku $3P$ môžeme dospieť vyššie uvedeným postupom, ale nie je to jediný možný postup výpočtu. Jedným z ďalších možných riešení je napríklad doubling bodu 2P (zobrazený na **Obrázok 27**) [30]:

$$2P + P = 3P = V \tag{114}$$



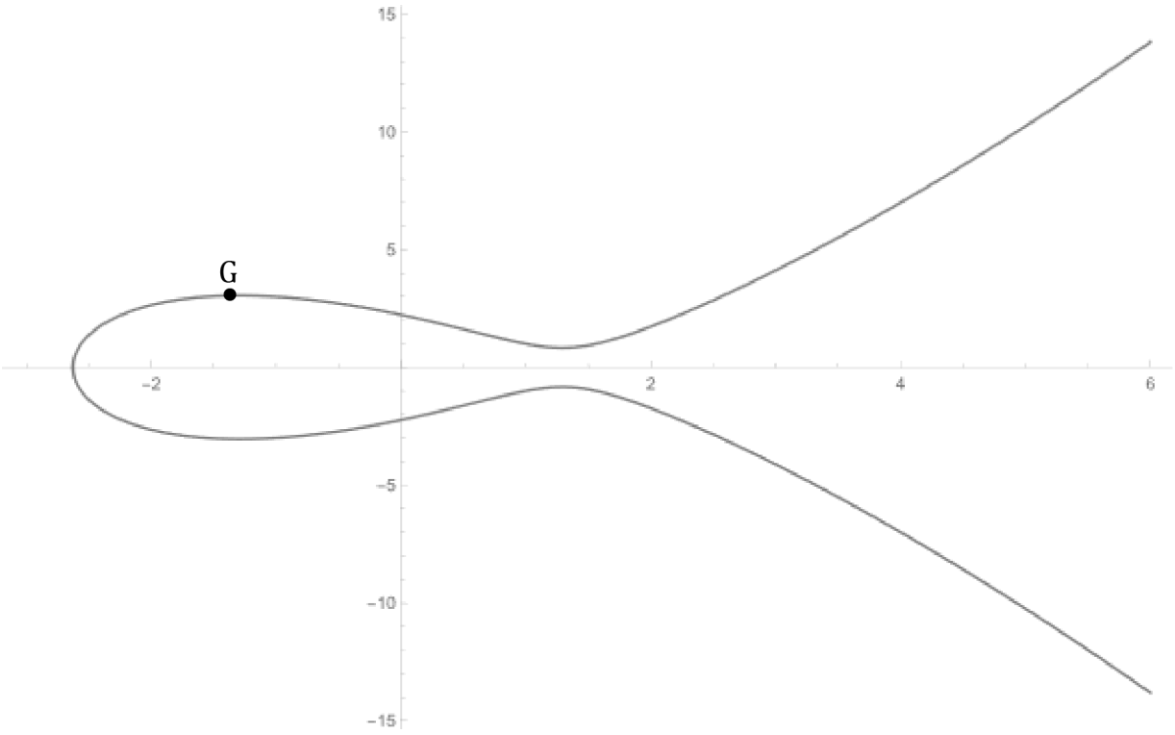
Obrázok 26 – doubling bodu *P* na eliptickej krivke $y^2 = x^3 - 4x + 2$, Zdroj: vlastný



Obrázok 27 – sčítanie bodu 2*P* a *P* na eliptickej krivke $y^2 = x^3 - 4x + 2$, Zdroj: vlastný

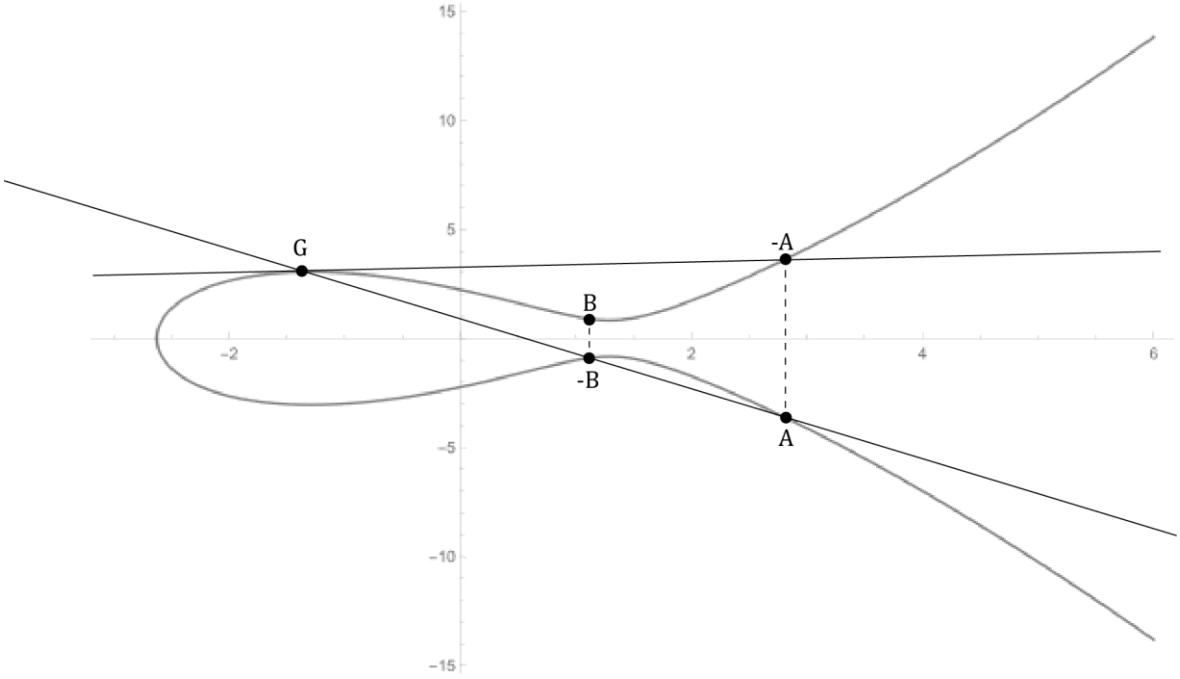
3.4.6.4 Praktické využitie

V nasledujúcej kapitole si ukážeme modelovú situáciu kedy chcú Alica a Bob komunikovať prostredníctvom nezabezpečeného kanála, s využitím symetrických šifrovacích systémov, kvôli ich rýchlosti aj pri prenose väčších objemov dát. Alica a Bob si teda musia pred začiatkom komunikácie vytvoriť ich zdieľaný šifrovací kľúč prostredníctvom nezabezpečeného komunikačného kanála, tak aby nedošlo k jeho prezradeniu. Na toto použijú upravenú formu Diffie-Hellman protokolu s využitím eliptických kriviek. Alica a Bob sa zhodnú na predpise eliptickej krivky, ktorú použijú a počiatočnom bode *G*, túto situáciu zobrazuje **Obrázok 28**. Eliptická krivka ako aj bod G je teda Alici aj Bobovi známy. [30]

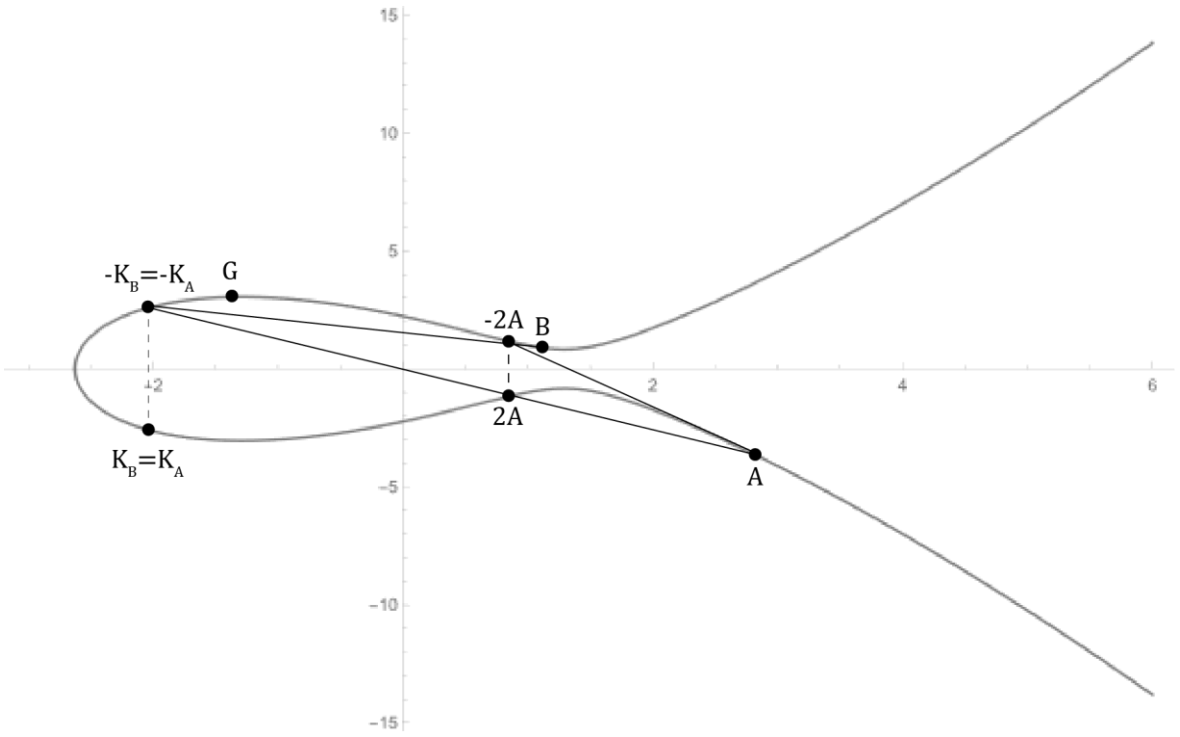


Obrázok 28 – eliptická krivka $y^2 = x^3 - 5x + 5$ s bodom *G,* Zdroj: vlastný

Alica si zvolí svoj privátny kľúč $P_A = 2$ a Bob si zvolí svoj privátny kľúč $P_B = 3$. Alica spočíta svoj verejný kľúč (bod A) ako $A = P_A * G = 2G$. Bob taktiež spočíta svoj verejný kľúč (bod B) ako $B = P_B * G = 3G$ ako môžeme vidieť na **Obrázok 29**. Alica následne odošle svoj verejný kľúč (bod $A = 2G$) Bobovi a Bob odošle svoj verejný kľúč (bod $B = 3G$) Alici. Alica následne spočíta hodnotu kľúča $K_A = P_A * B = 2B$. Bob spočíta hodnotu kľúča $K_B = P_B * A = 3A$. Ako možeme vidieť na **Obrázok 30** $K_A = K_B$ a teda Alica a Bob majú svoj zdieľaný spoločný kľúč. [30]



Obrázok 29 – eliptická krivka $y^2 = x^3 - 5x + 5$ s verejnými kľúčmi, Zdroj: vlastný



Obrázok 30 – eliptická krivka $y^2 = x^3 - 5x + 5$ so spoločným kľúčom, Zdroj: vlastný

3.5 Kvantová kryptografia

Kvantová kryptografia implementuje princípy kvantovej fyziky v šifrovacích systémoch. Koncept kvantovej kryptografie sa prvýkrát objavil už v roku 1970, kedy Stephen Weisner napísal svoju vedeckú prácu *Conjugate Coding*. Práca bola, ale publikovaná až v roku 1983, čiže až o celých 13 rokov neskôr. Počas tejto doby Charles H. Bennett, ktorý poznal Weisnerov koncept a Gilles Brassard publikovali niekoľko vedeckých prác, ktorými potvrdili technickú realizovateľnosť tohto konceptu. Kvantová kryptografia využíva základný princíp kvantovej fyziku a to tzv. *Heisenbergov princíp neurčitosti*, ktorý hovorí, že meranie kvantového systému spôsobí zmeny v tomto systéme. Výsledkom toho merania teda budú skreslené alebo neúplné informácie o stave tohto systému a čo viac, zmena v tomto systéme upozorní legitímnych užívateľov o prítomnosti narušiteľa. Kvantová kryptografia teda ponúka možnosti ako vytvoriť kryptografický systém na tvorbu zdieľaného kľúča bez toho aby komunikujúce strany museli predom zdieľať informácie o tomto kľúči, čo je možné samozrejme docieliť aj bez použitia kvantovej kryptografie. Avšak ponúka taktiež možnosť detekovať prítomnosť narušiteľa pri tvorbe tohto kľúča. Pokiaľ by sme teda predpokladali, že narušiteľ má dostupný nekonečný výpočetný výkon, nedokázal by ho využiť, pretože akonáhle by sa pokúsil zistiť akékoľvek informácie o stave kvantového systému legitímny užívatelia by detekovali jeho prítomnosť. Kvantová kryptografia by teda mohla eliminovať problémy iných šifrovacích systémov spôsobené neustále sa zväčšujúcim výpočtovým výkonom moderných zariadení. Aj napriek tomu, že sa jedná o veľmi mladú časť kryptografie jej popularita neustále narastá a s ňou aj jej efektívnosť a rýchlosť. Prvý prototyp využitý na kvantovú výmenu kľúča postavený v roku 1989 pracoval na vzdialenosť 32 cm, dnes už sa jedná o desiatky kilometrov – avšak stále sa jedná o pokusy v laboratórnych podmienkach než o plošné, každodenné nasadenie v praxi. Princípy fungovania kvantovej tvorby (výmeny) spoločného kľúča si bližšie priblížime na dvoch protokoloch typoch protokolov [34]:

- Prepare and measure protocols,
- Entanglement based protocols.

Prepare and measure protocols sú protokoly založené na vysielaní fotónou s rôznym spinom – priestorovou orientáciou. Táto orientácia potom predstavuje stavy týchto fotónov pričom ortogonálne orientované (orientované kolmo na seba) fotóny ([0°, 90°],[45°, 135°]) sú jasne odlíšiteľné a neortogonálne orientované fotóny ([0°, 45°],[45°, 90°]) nie. Definujme

teda štyri stavy 0°, 45°, 90° a 135° fotónov na základe ich priestorovej orientácie a stavom 0° a 45° prideľme binárnu hodnotu 0 a stavom 90° a 135° binárnu hodnotu 1. Alica následne začne vysielať lúč týchto fotónov Bobovi, pričom tieto fotóny budú mať náhodne zvolený stav. Bob sa pri prijatí každého z fotónov musí rozhodnúť či daný fotón zmeria ako (0°, 90°) alebo (45°, 135°). Ak zvolí správne získa presnú informáciu o stave daného fotónu a tým aj správnu binárnu hodnotu 1 alebo 0, pokiaľ zvolí nesprávne získa náhodný stav a tým náhodnú binárnu hodnotu 1 alebo 0. Po ukončení prenosu Bob informuje Alicu v akom poradí použil filtre v rámci prenosu, ale nepovie jej získané binárne hodnoty. Alica Bobovi povie kedy filter zvolil správne a kedy nie. Výmena týchto informácií by teda mohla prebiehať aj prostredníctvom nezabezpečeného kanála kedy by prípadný narušiteľ zistil len to, kedy Bob zvolil správne, ale nie získanú binárnu hodnotu. Binárne hodnoty kedy Bob zvolil správne budú použité ako ich spoločný zdieľaný kľúč a hodnoty kedy Bob nezvolil správne zahodia. V ideálnom prípade by Bob mal zvoliť správne v polovici prípadov, čo znamená že 50% bitov bude použitých ako zdieľaný kľúč a 50% bitov bude zahodených. V realite, ale toto rozdelenie nie je takto presné a to z dôvodu prirodzeného šumu – teda skreslenia, čo znamená, že malá časť správne zmeraných bitov bude vyhodnotená nesprávne. Mieru tohoto skreslenia môže zvýšiť prípadný narušiteľ, ktorý zmeraním daných fotónov opätovne zmení ich orientáciu. Prirodzená a narušiteľom spôsobená miera skreslenia sa nedá rozoznať. V princípe, ale narušiteľ spôsobí výrazne väčšiu mieru skreslenia než je prirodzená miera skreslenia spôsobená šumom a teda je možné narušiteľa detekovať. Existuje množstvo variácií tohto typu protokolov napr. protokol využívajúci len dva stavy namiesto štyroch, na ktorých sme demonštrovali princíp fungovania. [34]

Entanglement based protocols sú protokoly využívajúce kvantové previazanie párov fotónov. To znamená, že tieto časti spolu tvoria jeden kvantový systém a nie je možné ich stav popísať samostatne. Jeden z fotónov páru potom príjme Alica a druhý Bob. V momente zmeranie jedného z fotónu z daného páru nadobudne tento fotón stav a rovnaký stav nadobudne aj druhý fotón z tohto páru, kedže ako sme povedali sa jedná o jeden a ten istý systém. Alica aj Bob teda získajú rovnaký stav a teda aj rovnakú binárny hodnotu. Prípadný narušiteľ by spôsobil narušenie korelácie medzi previazaným fotónovým párom a teda je ho možné detekovať podobne ako v prípade *prepare and measure protokolov*. [34]

II. PRAKTICKÁ ČASŤ

4 APLIKÁCIA VIZUALIZUJÚCA ŠIFROVACIE ALGORITMY

Praktická časť práce sa zameriava na vytvorenie webovej aplikácie, ktorá slúži na vizualizáciu vybraných šifrovacích algoritmov Camellia (viz. kapitola **3.1.3**), Kuznyechik (viz. kapitola **3.1.4**) a ChaCha20 (viz. kapitola **3.2.1**), popísaných v teoretickej časti tejto práce. Na vývoj bol použitý framework Django a voľne dostupná bootstrap šablóna, upravená podľa špecifických potrieb práce. Frontend práce je teda implementovaný pomocou jazykov Java, JavaScript, CSS a HTML. Implementácia šifrovacích algoritmov je realizovaná pomocou jazyka Python. Počas vývoja bolo na editovanie zdrojového kódu použité vývojové prostredie Visual Studio Code. Aplikácia je dostupná na webe na adrese zdroja [35].

4.1 Prostriedky využité k návrhu a vývoju aplikácie

Predtým než prejdeme k štruktúre samotnej aplikácie a implementácií šifrovacích algoritmov, si v nasledujúcej kapitole stručne priblížime prostriedky, ktoré boli využité pri jej tvorbe.

4.1.1 Django

Django je voľne dostupný framework, využívajúci jazyk Python, slúžiaci na tvorbu webových stránok. Často je označovaný za jeden z najlepších, ak nie najlepší framework na tvorbu webových aplikácií v tomto jazyku. Je okrem iného pomerne ľahko škálovateľný, automaticky ponúka zabezpečenie webov, funguje s väčšinou databáz, neustále sa vyvíja vďaka svojej širokej komunite. Tento framework sme preto zvolili na vývoj našej webovej aplikácie. V nasledujúcich kapitolách si popíšeme inštaláciu tohto framework-u a počiatočné nastavenie a prípravu prostredia pred začiatkom tvorby samotnej webovej aplikácie. Inštalácia frameworku Django predpokladá, že v systéme je nainštalovaný jazyk Python a nástroj na správu balíkov pip, preto sme ako jazyk Python, tak aj pip nainštalovali predom.

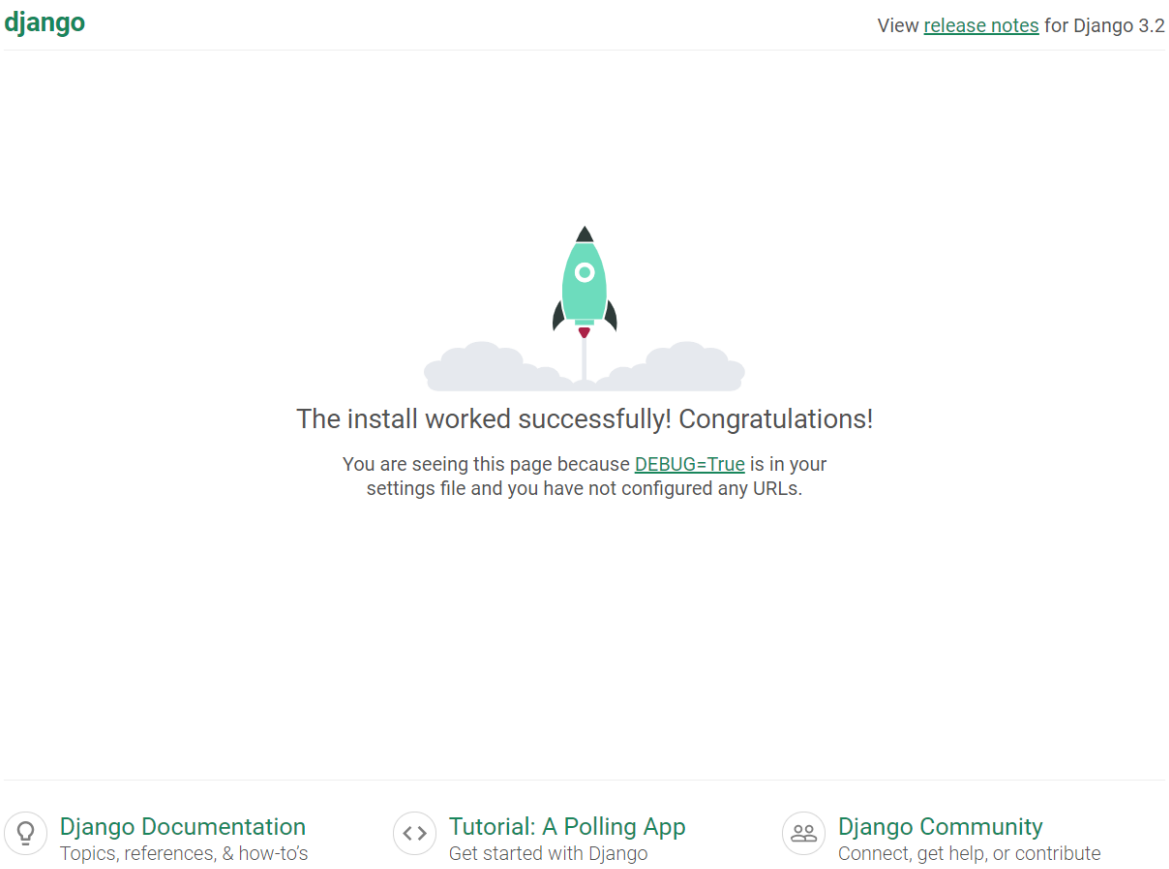
4.1.1.1 Inštalácia

Predtým ako nainštalujeme Django musíme vytvoriť zložku do ktorej budeme Django inštalovať. Túto zložku pomenujeme *BP_WEB_VIZUALIZACIA_SA* a v konzole sa do nej presunieme. Následne v tejto zložke vytvoríme virtuálne prostredie pomocou príkazu python -m -venv virtual a inicializujeme ho pomocou príkazu .\virtual\Scripts\activate. Teraz

zadáme príkaz pip install django a počkáme na dokončenie sťahovania potrebných balíčkov a ich inštaláciu. Stav po dokončení sťahovania a inštalácie zobrazuje **Obrázok 31**. Ďalším krokom je vytvorenie samotného Django projektu pomocou príkazu django-admin startproject a názvu projektu, ktorý pomenujeme rovnako ako zložku v ktorej ho vytvárame teda BP_WEB_VIZUALIZACIA_SA. Nakoniec spustíme projekt na lokálnom servery pomocou príkazu python manage.py runserver aby sme sa uistili, že inštalácia prebehla úspešne. V prípade úspešnej inštalácie sa nám na adrese localhost (defaultý port 8000) zobrazí webová stránka s informáciou o úspešnej inštalácií, ktorú zobrazuje **Obrázok 32**.

```
(virtual) PS C:\BP_WEB_VIZUALIZACIA_SA> pip install django
Collecting django
  Downloading Django-3.2-py3-none-any.whl (7.9 MB)
     |████████████████████████████████| 7.9 MB 1.3 MB/s
Collecting asgiref<4,>=3.3.2
  Downloading asgiref-3.3.4-py3-none-any.whl (22 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.1-py3-none-any.whl (42 kB)
Collecting pytz
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Installing collected packages: asgiref, sqlparse, pytz, django
Successfully installed asgiref-3.3.4 django-3.2 pytz-2021.1 sqlparse-0.4.1
```

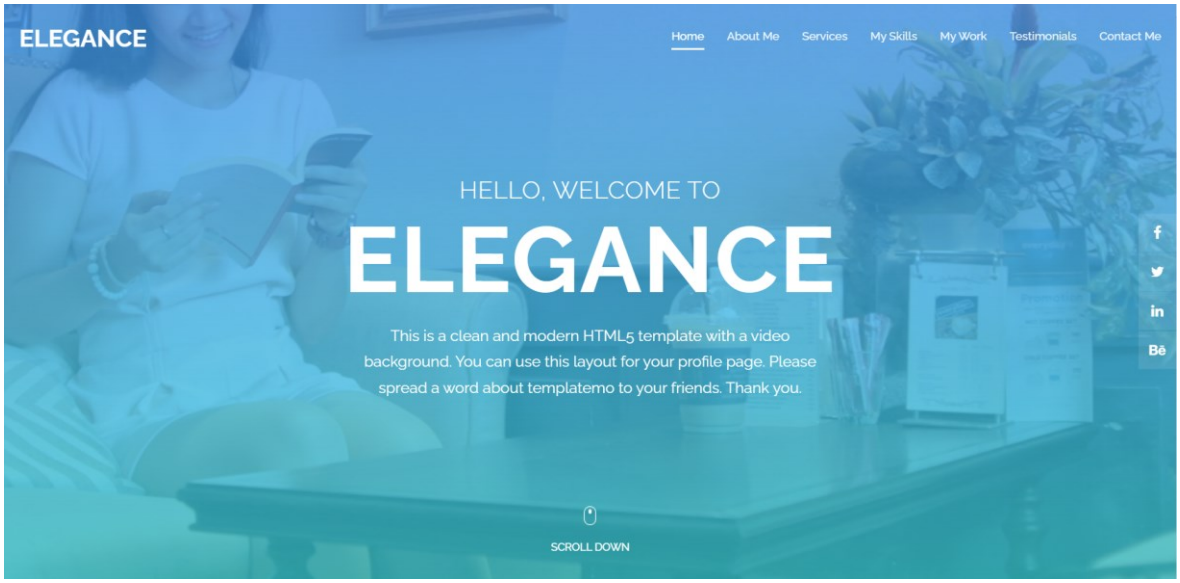
Obrázok 31 – Django inštalácia, Zdroj: vlastný



Obrázok 32 – Django úspěšná inštalácia a spustenie servera, Zdroj: vlastný

4.1.2 HTML šablóna

Ako základ užívateľského rozhrania bola použitá voľne dostupná HTML šablóna Elegance od spoločnosti Templatemo (dostupná na adrese zdroja [36]), ktorú v priebehu vývoja práce upravíme podľa špecifických potrieb nami vyvíjanej webovej aplikácie. Dôležitosť jazyka HTML pri vývoji webových aplikácií, nie je nutné zdôrazňovať. Samotná šablóna po stiahnutí obsahuje súbory CSS, JavaScript, Font-y a obrázky využité v príslušnom HTML súbore, ktoré sú tiež dostupné po stiahnutí a výslednú šablónu zobrazuje **Obrázok 33**.



Obrázok 33 – základný stav Elegance šablóny po stiahnutí, Zdroj: vlastný

4.1.3 Visual Studio Code

Podstatnou súčasťou vývoja každej aplikácie je vývojové prostredie. Na vývoj našej webovej aplikácie využijeme voľne dostupný a veľmi obľúbený editor od spoločnosti Microsoft – Visual Studio Code. Voľba vhodného vývojového prostredia je v mnohom subjektívna. Visual Studio Code sme zvolili pretože, okrem iného ponúka jednoduché a intuitívne rozhranie, automatické dopĺňanie, inštaláciu veľkého množstva rozšírení a podporu synchronizácie s GitHub-om atď.

5 POPIS A FUN KCIE APLIKÁCIE

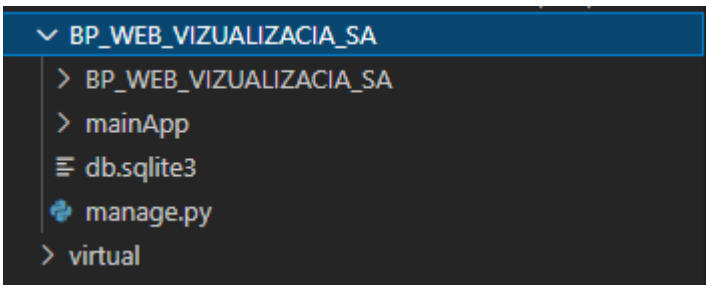
Nasledujúca kapitola popisuje ako elementárnu štruktúru projektu, tak aj najpodstatnejšie zložky napr. zložky HTML šablón, konfiguračných súborov a zdrojových kódov. V rámci súborov zdrojových kódov taktiež stručne popisuje funkcie a triedy, ktoré obsahujú, v závislosti na danej šifre.

5.1 Štruktúra

Samotná aplikácia pozostáva z niekoľkých častí, ktoré opäť pozostávajú z ďalších častí. Opisovať všetky tieto časti by bolo veľmi časovo náročné, pričom mnohé z nich sú vytvorené automaticky ako základ každého Django projektu a preto sa budeme venovať len najpodstatnejším z nich. Konkrétne dvom "aplikáciám" (z angl. "apps", podľa dokumentácie framework-u Django):

1. **BP_WEB_VIZUALIZÁCIA_SA** – jedná sa o Django aplikáciu, ktorá bola vytvorená automaticky pri inštalácií projektu v kapitole **4.1.1.1**,
2. **mainApp** – opäť sa jedná sa o Django aplikáciu, ktorá bola vytvorená ručne pre súbory nášho projektu.

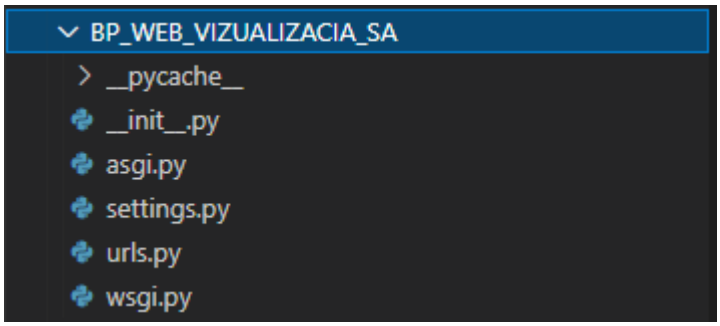
Celú štruktúru projektu zobrazuje nasledujúci obrázok:



Obrázok 34 – štruktúra projektu, Zdroj: vlastný

5.1.1 BP_WEB_VIZUALIZÁCIA_SA

Štruktúru tejto aplikácie zobrazuje **Obrázok 35** nižšie:



Obrázok 35 – štruktúra aplikácie BP_WEB_VIZUALIZÁCIA_SA, Zdroj: vlastný

V predchádzajúcej kapitole sme povedali, že súbory nášho projektu budú uložené v Django aplikácií *mainApp*, práve preto sú pre nás veľmi dôležité súbory *settings.py* a *urls.py*. V súbore *setting.py* musíme najskôr našu Django aplikáciu zaregistrovať (zobrazuje **Obrázok 36**) a taktiež definovať cestu k zložke obsahujúcej statické súbory nášho projektu – CSS, JavaScript a pod. (zobrazuje **Obrázok 37**). V súbore *urls.py* musíme definovať, cestu k súboru obsahujúcemu vzory ciest (z angl. "Route patterns") pre navigáciu v rámci našej aplikácie pre dané url (zobrazuje **Obrázok 38**).

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'mainApp',
]
```

Obrázok 36 – registrácia mainApp aplikácie, Zdroj: vlastný

```
STATIC_URL = '/static/'
```

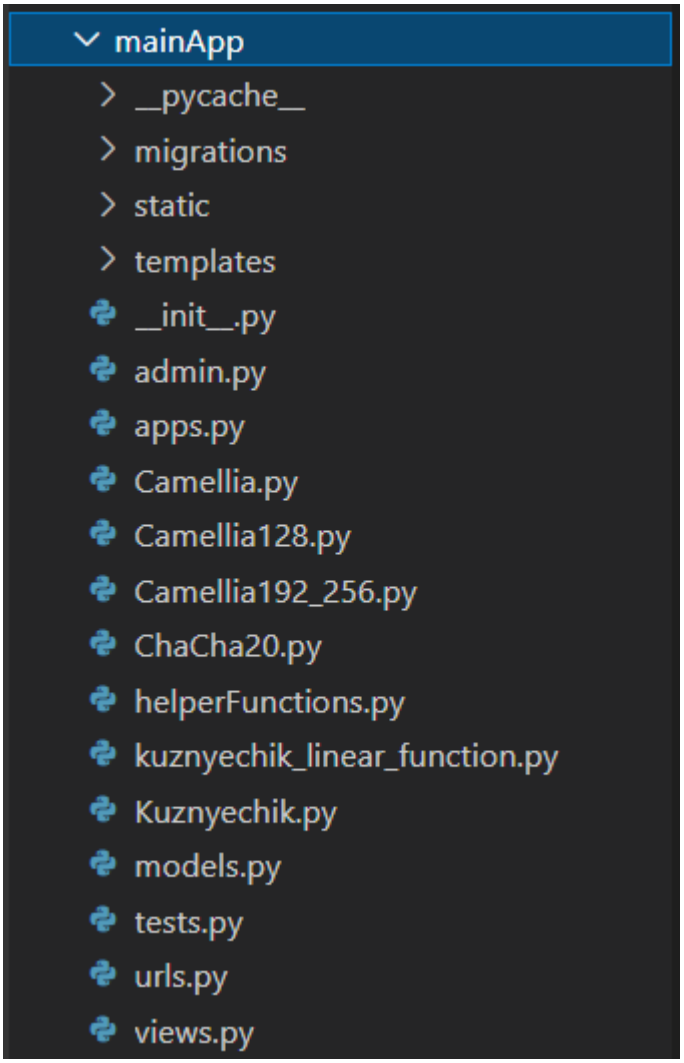
Obrázok 37 – cesta k zložke obsahujúcej statické súbory, Zdroj: vlastný

```
urlpatterns = [
    path('', include('mainApp.urls')),
    path('admin/', admin.site.urls),
]
```

Obrázok 38 – cesta k navigačným vzorcom mainApp aplikácie, Zdroj: vlastný

5.1.2 mainApp

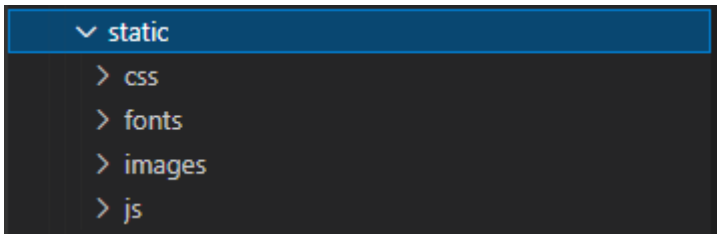
Štruktúru tejto aplikácie zobrazuje **Obrázok 39**Obrázok 35 nižšie:



Obrázok 39 – štruktúra aplikácie mainApp, Zdroj: vlastný

Ako môžeme vidieť, štruktúra mainApp aplikácie je v porovnaní s predchádzajúcou aplikáciou o niečo obšírnejšia. Okrem iného obsahuje už spomínanú zložku statických súborov *static*, ktorá ďalej obsahuje 4 pod zložky (zobrazuje **Obrázok 40**):

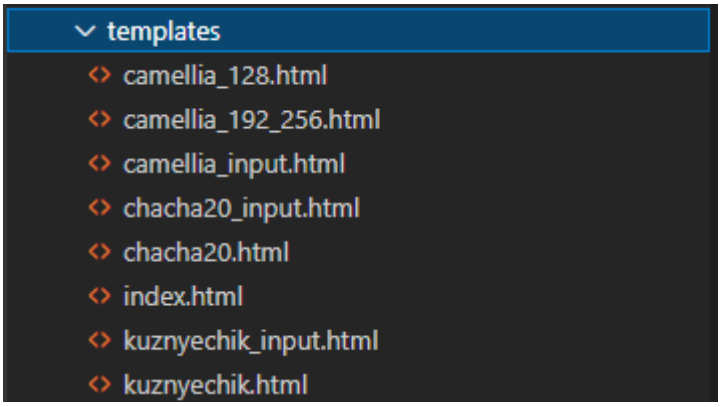
1. css – obsahuje CSS súbory projektu,
2. fonts – obsahuje fonty využité v projekte,
3. images – obsahuje obrázky použité v projekte,
4. js – obsahuje JavaScript-ové zdrojové kódy projektu.



Obrázok 40 – zložka static, Zdroj: vlastný

Ďalej zložku **templates** (zobrazuje **Obrázok 41**) obsahujúcu HTML súbory projektu :

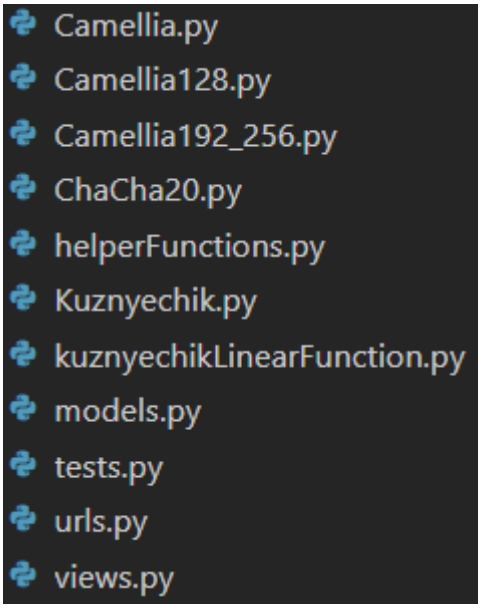
1. camellia_128.html – HTML súbor pre šifru Camellia s 128 bitovým kľúčom
2. camellia_192_256.html – HTML súbor pre šifru Camellia s 192 alebo 256 bitovým kľúčom
3. camellia_input.html – HTML súbor pre šifru Camellia na zadanie vstupných dát
4. chacha20_input.html – HTML súbor pre šifru ChaCha na zadanie vstupných dát
5. index.html – HTML súbor pre úvodnú obrazovku
6. kuznyechik_input.html – HTML súbor pre šifru Kuznyechik na zadanie vstupných dát
7. kuznyechik.html – HTML súbor pre šifru Kuznyechik



Obrázok 41 – zložka templates, Zdroj: vlastný

V neposlednom rade taktiež tieto súbory (zobrazuje **Obrázok 42**):

1. Camellia.py – súbor so zdrojovým kódom spoločným pre šifru Camellia pre 128, 192 aj 256 bitový kľúč,
2. Camellia128.py – súbor so zdrojovým kódom pre šifru Camellia špecificky pre 128 bitový kľúč,
3. Camellia192_256.py – súbor so zdrojovým kódom pre šifru Camellia špecificky pre 192 a 256 bitový kľúč,
4. ChaCha20.py – súbor so zdrojovým kódom pre šifru ChaCha,
5. helperFunctions.py – súbor so zdrojovým kódom pomocných funkcií pre všetky 3 šifry,
6. Kuzneychik.py – súbor so zdrojovým kódom pre šifru Kuznyechik,
7. kuznyechikLinearFunction.py – súbor s funkciami pre lineárnu substitúciu šifry Kuznyechik,
8. urls.py – súbor obsahujúci už spomínané vzory ciest (viz. kapitola **5.1.1**)
9. views.py – súbor obsahujúci zdrojový kód slúžiaci na vyrendrovanie a predanie dát do HTML šablón.



Obrázok 42 – súbory zdrojových kódov, Zdroj: vlastný

5.1.3 urls.py

Súbor urls.py slúži na navigáciu v rámci celej aplikácie pre dané url ako môžeme vidieť na **Obrázok 43** nižšie. Teda napríklad pre url končiace "\camellia" (riadok 8) vráti príslušnú funkciu zo súboru *views.py* teda *views.camellia*.

urls.py

BP_WEB_VIZUALIZACIA_SA > mainApp > urls.py > {} admin

```
1  from django.contrib import admin
2  from django.urls import path, include
3  from . import views
4
5  urlpatterns = [
6      path('', views.index, name="index"),
7      path('camellia_input', views.camellia_input, name="camellia_input"),
8      path('camellia', views.camellia, name="camellia"),
9      path('chacha20_input', views.chacha20_input, name="chacha20_input"),
10     path('chacha20', views.chacha20, name="chacha20"),
11     path('kuznyechik_input', views.kuznyechik_input, name="kuznyechik_input"),
12     path('kuznyechik', views.kuznyechik, name="kuznyechik")
13 ]
```

Obrázok 43 – súbor urls.py, Zdroj: vlastný

5.1.4 views.py

Ako sme spomenuli v predchádzajúcej kapitole súbor views.py obsahuje funkcie, ktoré preberú informácie o danej šifre z HTML **POST** metódy (formulára), inicializujú objekt (triedu) danej šifry, spracujú vstupné dáta, vygenerujú pomocou tohto objektu všetky premenné potrebné pre vizualizáciu ako aj šifrovanie (šifrovacie kľúče, matice atď.) a následne dáta zašifrujú. Všetky dáta potom predajú funkcií render, slúžiacej na renderovanie HTML kódu spolu s názvom HTML šablóny zo zložky *templates* (**5.1.2)**.

5.1.5 helperFunctions.py

Predtým než prejdeme k podrobnejšiemu opisu úvodnej obrazovky a každej šifry v samostatných kapitolách, bližšie rozoberieme súbor helperFunctions.py, obsahujúci pomocné funkcie pre všetky tri šifry a to :

1. pad – slúži na doplnenie vstupného bitového reťazca dĺžky *n* na požadovanú dĺžku *x* pomocou pridania požadovaného počtu núl na začiatok reťazca *n*,
2. processInput – slúži na "preklad" vstupného reťazca znakov na reťazec binárnych reprezentácií (hodnôt) každého znaku na základe jeho číselnej hodnoty v tabuľke ASCII,
3. rotateLeft – slúži na rotáciu binárneho reťazca o *n* znakov doľava a jeho doplnenie na chcený počet znakov pomocou funkcie *pad*. Rotácia je realizovaná v kruhovej forme teda pokiaľ budeme rotovať napr. pre reťazec 10111001 o tri znaky dostaneme reťazec 11001101,
4. getCharacters – slúži na "preklad" binárneho reťazca na reťazec znakov pomocou prevodu binárnych hodnôt na číselné hodnoty jednotlivých znakov v ASCII tabuľke.
5. getCharactersKuznyechik – slúži na "preklad" binárneho reťazca na reťazec znakov pomocou prevodu binárnych hodnôt na číselné hodnoty jednotlivých znakov v ASCII tabuľke pre šifru Kuznyechik.

Jednotlivé funkcie a ich funkcia v kóde je rovnako okomentovaná v kóde, spolu so vstupným a výstupnými dátovými typmi pre danú funkciu ako to zobrazuje **Obrázok 44**.

```
def pad(inputToPad, padTo):

    """ Function that pads given bit string to desired length by adding 0
        to beggining of the string

    Parameters:
    inputToPad (str): Bit value to be padded
    padTO (int): Length to pad the input to (in bit length)

    Returns:
    str: The input after padding

    """

    inputToPad = str(inputToPad)
    while(len(inputToPad) != padTo):
        inputToPad = '0' + inputToPad

    return inputToPad
```

Obrázok 44 – funkcia pad v súbore helperFunctions.py, Zdroj: vlastný

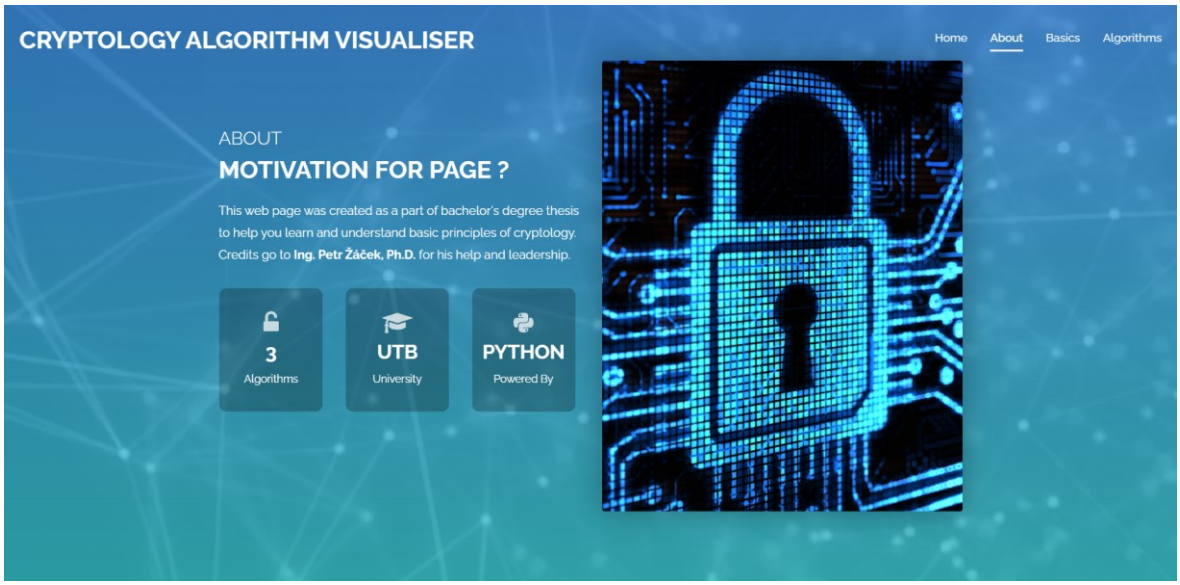
5.2 Úvodná obrazovka aplikácie

Úvodná obrazovka aplikácie zahŕňa jednu HTML šablónu a jednoduchú funkciu *index* v súbore *views.py*, ktorá vracia funkciu *render*, s názvom HTML šablóny, ktorú chceme renderovať, v tomto prípade *index.html*, ako môžeme vidieť na **Obrázok 45**. Finálny vzhľad prvej strany úvodnej obrazovky zobrazuje **Obrázok 46**. Úvodná obrazovka obsahuje ďalšie tri samostatné strany, na navigáciu medzi týmito stranami môžeme použiť koliesko myši alebo navigačné menu v pravom hornom rohu. Druhá strana ("About", **Obrázok 47**) obsahuje základné informácie o aplikácií, tretia strana ("Basics", **Obrázok 48**) veľmi stručne objasňuje najzakladenejšie kryptografické pojmy a štvrtá strana ("Algorithms", **Obrázok 49**) obsahuje stručné opisy jednotlivých šifier, pričom rovnako slúži aj na navigáciu k rozhraniu daných šifier kliknutím na boxy obsahujúce informácie o konkrétnych šifrách.

```
def index(request):
    return render(request, 'index.html', {})
```

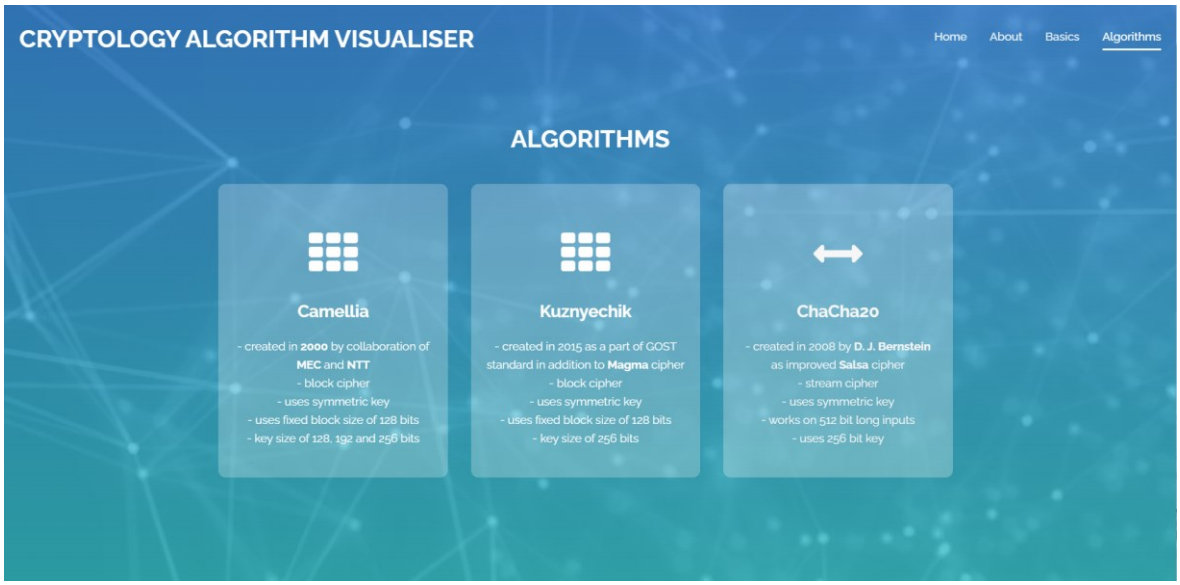
Obrázok 45 – funkcia na renderovanie šablóny úvodnej obrazovky, Zdroj: vlastný

Obrázok 46 – úvodná obrazovka, Zdroj: vlastný



Obrázok 47 – obrazovka About, Zdroj: vlastný

Obrázok 48 – obrazovka Basics, Zdroj: vlastný



Obrázok 49 – obrazovka Algorithms, Zdroj: vlastný

5.3 Camellia – popis a funkcie

V predchádzajúcich kapitolách sme zhrnuli najpodstatnejšie časti (štruktúru) celej aplikácie. V nasledujúcich kapitolách sa podrobnejšie pozrieme na konkrétne súbory (časti) a funkcie konkrétnych šifier, počínajúc šifrou Camellia. Jednotlivé funkcie každej šifry sú taktiež okomentované v zdrojovom kóde, rovnako ako v prípade pomocných funkcií z predchádzajúcej kapitoly **5.1.5**. V prípade šifry Camellia sa jedná o tri HTML šablóny, ktorých funkciu sme opísali v kapitole **5.1.2** (zložka *templates*):

1. camellia_input.html,
2. camellia_128.html,
3. camellia_192_256.html.

Dvoch funkciách slúžiacich na renderovanie týchto šablón (súbor *views.py*):

1. camellia_input – jednoduchá funkcia zhodná svojim telom s funkciou index z **Obrázok 45**, pričom samozrejme renderujeme rozdielnu šablónu, v tomto prípade *camellia_input.html*,
2. camellia – zložitejšia funkcia, ktorej všeobecnú funkciu sme priblížili v kapitole **5.1.4**, pričom v nej taktiež dochádza k jednoduchému rozhodovaniu či sa jedná o 128 alebo 192 a 256 bitovú verziu šifry (kľúča), na základe čoho potom okrem iného využívame rozdielne funkcie, meníme dĺžku šifrovacieho procesu a taktiež renderujeme rozdielnu šablónu.

A troch súboroch so zdrojovými kódmi:

1. Camellia.py,
2. Camellia128.py,
3. Camellia192_256.py.

Camellia.py obsahuje triedu *CamelliaBase*, od ktorej ďalej dedia triedy, na ktoré sa bližšie pozrieme neskôr. Trieda *CamelliaBase*, ďalej obsahuje spolu 9 funkcií spoločných ako pre 128 tak aj 192 a 256 bitovú verziu šifry (kľúča):

1. __init__ – inicializačná funkcia triedy,

2. rotateRightBy1 – funkcia rotujúca bitový reťazec v kruhovej forme o jedno miesto doprava,
3. evenThenUnevenList – funkcia vracajúca pole, ktoré vzniká z poľa, ktoré do funkcie vstupuje ako vstupný argument, pričom najskôr z tohto poľa vyberáme len prvky na párnych indexoch a vkladáme ich do nového poľa a potom tento proces opakujeme pre prvky na nepárnych indexoch,
4. generateSBoxes – funkcia vracajúca štyri S-boxy (256-prvkové polia), pričom prvý z S-box-ov je definovaný priamo v kóde a zvyšné tri sú z neho vygenerované pomocou určitých matematických operácií (viz. kapitola **3.1.3.5**).
5. sFunction – funkcia, ktorá rozdelí 64 bitov dlhý vstupný bitový reťazec na 8 bitov dlhé časti, ktoré následne konvertuje na ich číselné reprezentácie a použije ich ako substitučné indexy do S-box-ov (viz. kapitola **3.1.3.4**),
6. pFunction – funkcia, ktorá rozdelí 64 bitov dlhý vstupný bitový reťazec na 8 bitov dlhé časti, ktoré následne konvertuje na ich číselné reprezentácie a XOR-ne tieto hodnoty medzi sebou podľa špecifikácie šifry (viz. kapitola **3.1.3.4**),
7. fFunction – funkcia, ktorá prevedie vstupný bitový reťazec na číselnú (integer) hodnotu a XOR-ne ju s vstupným kľúčom resp. podkľúčom. Takto vzniknutú hodnotu potom použije ako vstup do *sFunkcie* a tento výsledok potom do *pFunkcie*, čím získame finálnu návratovú hodnotu (viz. kapitola **3.1.3.4**),
8. flFunction – funkcia, ktorá rozdelí 64 bitový vstupný bitový reťazec a 64 bitový *kl* podkľúč na dve 32 bitové polovice – ľavú a pravú. Následne vykoná operáciu AND medzi ľavou častou vstupného reťazca a ľavou častou *kl* podkľúča. Tento výsledok je potom orotovaný o 1 bit doľava v kruhovej forme a XOR-nutý s pravou častou vstupného reťazca. Výsledok týchto operácií následne tvorí pravých 32 bitov výstupu. Ľavých 32 bitov výstupu tvorí pravých 32 bitov výstupu, ktoré boli vytvorené v predchádzajúcich krokoch a pravých 32 bitov *kl* podkľúča na ktoré medzi sebou aplikujeme operáciu OR. Takto získame 32 bitový výsledok, ktorý potom XOR-neme s pôvodnými ľavými 32 bitmi vstupného reťazca. Takto vytvorenú ľavá a pravá časť je následne spojená a vrátená ako 64 bitový výstup (viz. kapitola **3.1.3.4**),
9. flminusFunction – funkcia, ktorá je veľmi podobná predchádzajúcej *flFunkcií*, ale pravá a ľavá časť vznikajú opačnými procesmi ako v prípade *flFunkcie*. Teda

rovnako rozdelíme 64 bitový vstupný reťazec a *kl* podkľúč na 32 bitové (ľavé a pravé) časti. Ľavá časť výstupu vzniká ako výsledok operácie OR medzi pravou časťou vstupu a pravou častou *kl* podkľúča, pričom tento výsledok následne XOR-neme s ľavou častou vstupného reťazca. Pravá časť výstupu vzniká ako výsledok operácie AND medzi ľavou častou vstupného reťazca a ľavou častou *kl* podkľúča, pričom tento výsledok potom orotujeme o 1 bit doľava v kruhovej forme a XOR-neme s pôvodnou pravou častou vstupného reťazca. Takto vzniknutá ľavá a pravá časť výstupu je následne spojená a vrátená ako 64 bitový výstup (viz. kapitola **3.1.3.4**).

Camellia128.py obsahuje triedu *Camellia128*, ktorá dedí od triedy *CamelliaBase* a sama obsahuje štyri ďalšie funkcie, špecifické pre použitie 128 bitového kľúča:

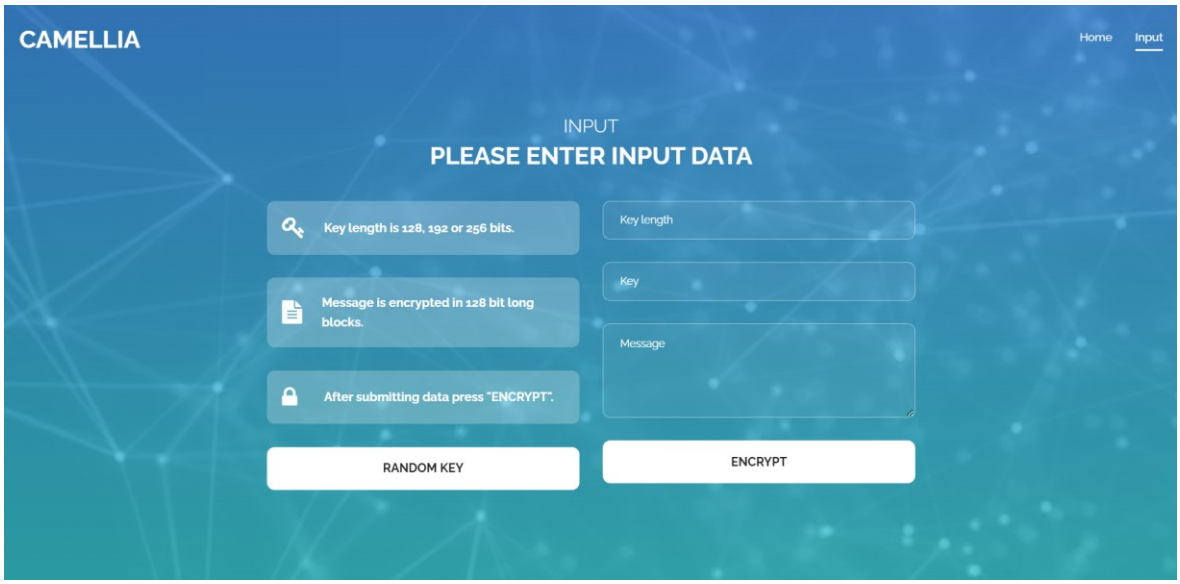
1. generateKa128Key – funkcia, ktorej vstupom je 128 bitový kľúč Pričom po prevedení krokov definovaných v dokumentácií šifry (zobrazuje **Obrázok 9**), je výstupom šifry podkľúč *Ka* a niekoľko premenných (vektorov) slúžiacich na vizualizačné účely (*KeyTemps, KeyTempsXORed, KeyLs, KeyRs*),
2. generateSubKeys128 – funkcia, slúžiaca na vytvorenie vektora *k* obsahujúceho 18 šifrovacích podkľúčov, vektora *kw* obsahujúceho 4 šifrovacie podkľúče a vektora *kl,* ktorý rovnako obsahuje 4 šifrovacie podkľúče. Všetky tieto vektory sú vytvorené zo vstupného 128 bitového kľúča a *Ka* podkľúča, ktorý bol vytvorený predom, pomocou predchádzajúcej funkcie (*generateKa128Key*), tak ako to definuje dokumentácia (zobrazuje **Obrázok 10**),
3. encryptCamellia128 – funkcia slúžiaca na zašifrovanie vstupu pomocou vygenerovaných podkľúčov s využitím algoritmu šifry *Camellia*, pre 128 bitový kľúč. Funkcia okrem zašifrovaného vstupu vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*PTL_init, PTR_init, PTRs, PTLs*),
4. decryptCamellia128 – funkcia slúžiaca na dešifrovanie vstupu pomocou vygenerovaných podkľúčov s využitím algoritmu šifry *Camellia*, pre 128 bitový kľúč. Funkcia rovnako okrem dešifrovaného vstupu vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*PTL_init_decipher*, *PTR_init_decipher*, *PTLs_decipher*, *PTRs_decipher*, *PTL_final_decipher*, *PTR_final_decipher*).

Camellia192_256.py obsahuje triedu *Camellia192_256*, rovnako dedí od triedy *CamelliaBase* a sama obsahuje päť ďalších funkcií, špecifických pre použitie 192 alebo 256 bitového kľúča, vzhľadom k tomu, že 192 a 256 bitové verzie šifry sú kompatibilné (viz. kapitola **3.1.3.3** a **Tabuľka 1**):

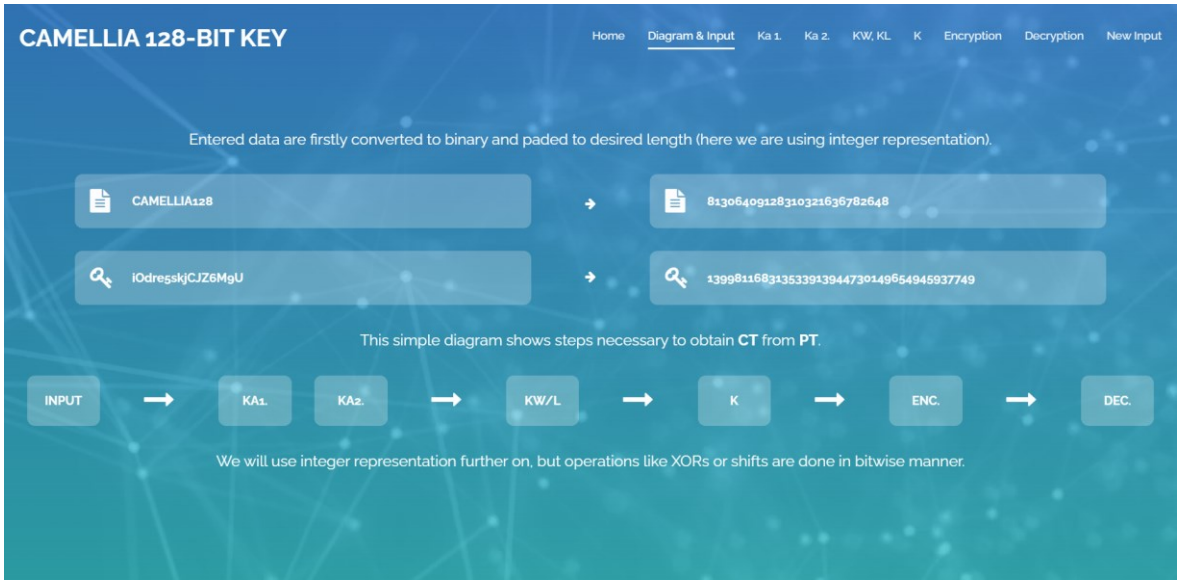
1. switchNullsAndOnes – funkcia špecifická pre 192 bitovú verziu kľúča, kedy "dopočítame" zvyšných 64 bitov a pridáme ich k 192 bitovému vstupnému kľúču aby sme získali 256 bitov dlhý kľúč. Keby sme 192 bitový reťazec rozdelili na 128 bitovú ľavú polovicu a 64 bitovú pravú časť a túto pravú časť potom označili ako ľavú stranu pravej časti, pravú stranu pravej časti získame ako binárnu negáciu ľavej strany pravej časti – teda zameníme všetky jednotky za nuly a všetky nuly za jednotky. Následne spojíme ľavú a pravú stranu pravej časti a získame 128 bitovú pravú polovicu (viz. kapitola **3.1.3.3** a **Tabuľka 1**),
2. generateKaKb192_256Key – funkcia, ktorej vstupom je 192 alebo 256 bitový kľúč. Pričom po prevedení krokov definovaných v dokumentácií šifry (zobrazuje **Obrázok 9**), sú výstupom šifry podkľúče *Ka*, *KB* a dva vektory slúžiace na vizualizačné účely (*KLLs*, *KLRs*),
3. generateSubKeys192_256 – funkcia, slúžiaca na vytvorenie vektora *k* obsahujúceho 24 šifrovacích podkľúčov, vektora *kw* obsahujúceho 4 šifrovacie podkľúče a vektora *kl,* ktorý rovnako obsahuje 4 šifrovacie podkľúče. Všetky tieto vektory sú vytvorené zo vstupného 192 alebo 256 bitového kľúča a *Ka*, *Kb* podkľúčov, ktoré boli vytvorené predom, pomocou predchádzajúcej funkcie (*generateKaKb192_256Key*), tak ako to definuje dokumentácia (zobrazuje **Obrázok 11**),
4. encryptCamellia192_256 – funkcia slúžiaca na zašifrovanie vstupu pomocou vygenerovaných podkľúčov s využitím algoritmu šifry *Camellia*, pre 192 alebo 256 bitový kľúč. Funkcia okrem zašifrovaného vstupu vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*PTL_init, PTR_init, PTRs, PTLs*),
5. decryptCamellia192_256 – funkcia slúžiaca na dešifrovanie vstupu pomocou vygenerovaných podkľúčov s využitím algoritmu šifry *Camellia*, pre 192 alebo 256 bitový kľúč. Funkcia rovnako okrem dešifrovaného vstupu vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*PTL_init_decipher*, *PTR_init_decipher*, *PTLs_decipher*, *PTRs_decipher*).

5.3.1 GUI

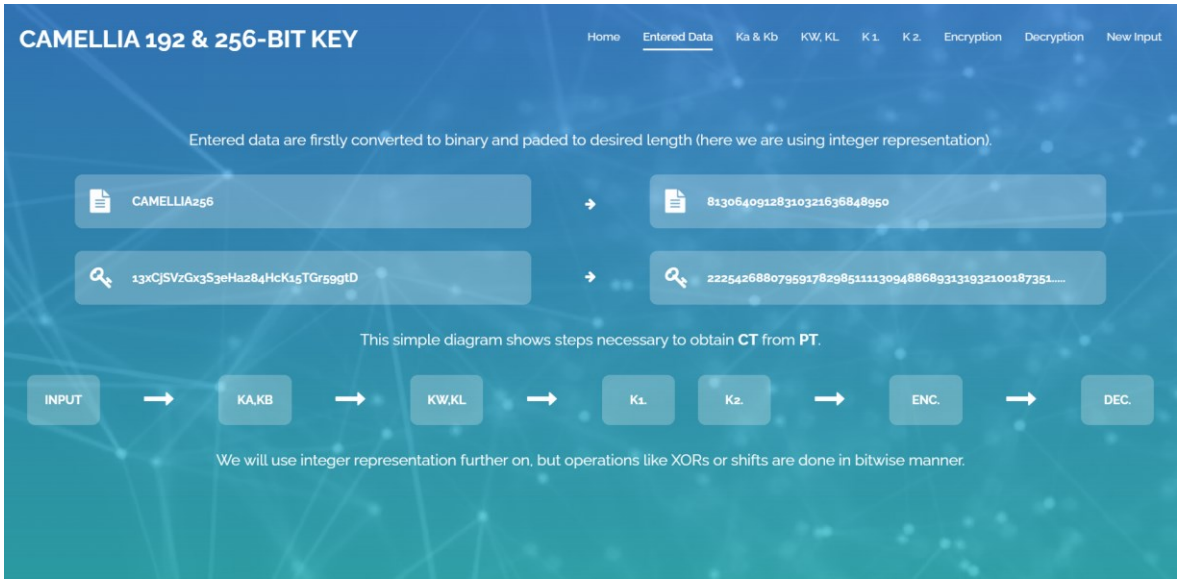
Vstupné dáta do šifry *Camellia* zadávame pomocou HTML šablóny *camellia_input.html* (zobrazuje **Obrázok 50**), kedy po stisknutí tlačidla "Encrypt", sú dáta predané funkcií *camellia* (súbor *views.py*), ktorá na základe dĺžky zadaného kľúča rozhodne, či sa jedná o 128 alebo 192 resp. 256 bitovú verziu šifry a podľa toho následne vytvorí objekt danej šifry, spracuje dáta a taktiež vráti príslušnú HTML šablónu. Každá zo šablón obsahuje niekoľko strán popisujúcich proces tvorby šifrovacích podkľúčov ako aj samotného šifrovania a dešifrovania. Na navigáciu slúži jednoduchý diagram v spodnej časti strany alebo menu v pravom hornom rohu. (zobrazuje **Obrázok 51** a **Obrázok 52**).



Obrázok 50 – HTML šablona camellia_input.html, Zdroj: vlastný



Obrázok 51 – úvodná strana šifry Camellia pre 128 bitový náhodný kľúč a správu "CAMELLIA128", Zdroj: vlastný



Obrázok 52 – úvodná strana šifry Camellia pre 256 bitový náhodný kľúč a správu "CAMELLIA256", Zdroj: vlastný

5.4 ChaCha20 – popis a funkcie

Funkcionalitu šifry ChaCha zabezpečujú dve HTML šablóny, ktorých funkciu sme opísali v kapitole **5.1.2** (zložka *templates*):

1. chacha20_input.html,
2. chacha20.

Dve funkcie slúžiace na renderovanie týchto šablón (súbor *views.py*):

1. chacha20_input – jednoduchá funkcia zhodná svojim telom s funkciou index z **Obrázok 45**, pričom samozrejme renderujeme rozdielnu šablónu, v tomto prípade *chacha20_input.html*,
2. chacha20 – zložitejšia funkcia, ktorej všeobecnú funkciu sme priblížili v kapitole **5.1.4**

A jeden súbor so zdrojovým kódom:

1. ChaCha20.py.

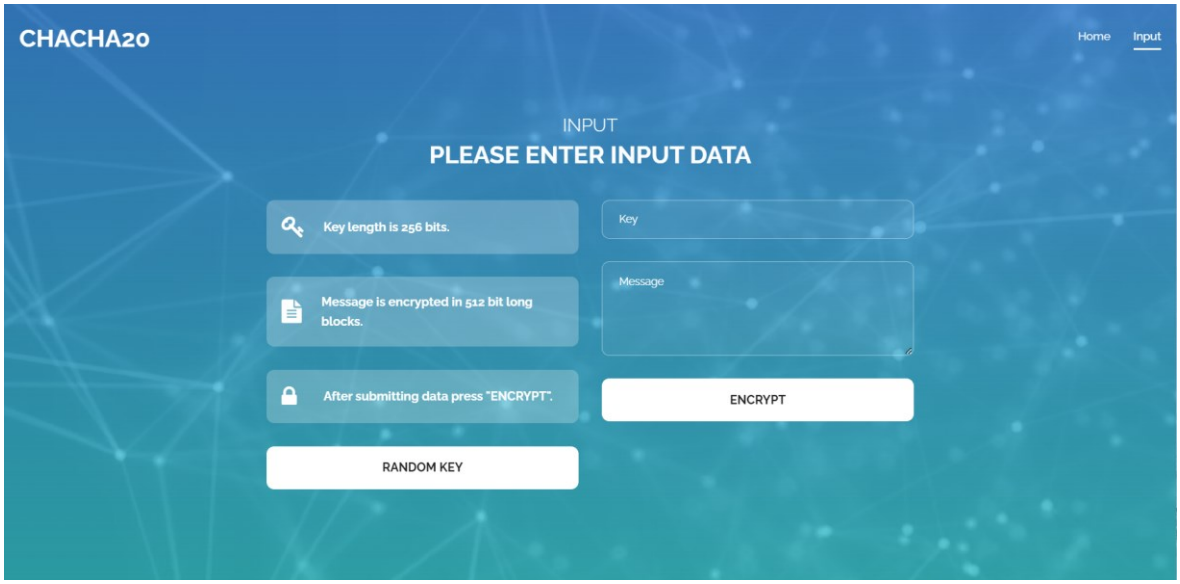
ChaCha20.py obsahuje triedu *ChaCha20*, ktorá obsahuje spolu 12 funkcií:

1. __init__ – inicializačná funkcia triedy,
2. processConstant – funkcia, ktorá prevedie každý zo 16 znakov konštanty na jeho binárnu reprezentáciu pomocou jeho číselnej hodnoty v tabuľke ASCII,
3. generateConstantParts – funkcia, ktorá spojí každé štyri znaky z konštanty v ich binárnych reprezentáciách a spojí ich a prevedie na číselnú (integer) hodnotu. Teda vracia vektor štyroch číselných hodnôt,
4. splitKey – funkcia, ktorá rozdelí 256 bitov dlhý vstupný kľúč na 32 bitov dlhé časti, ktoré následne konvertuje na ich číselnú (integer) reprezentáciu a vráti vektor týchto hodnôt,
5. getRandomNumber96Bits – funkcia, ktorá vygeneruje 12 pseudonáhodných číselných hodnôt, ktoré konvertuje na ich binárnu reprezentáciu a spojí, čím získame 96 bitov dlhý, výstupný reťazec,

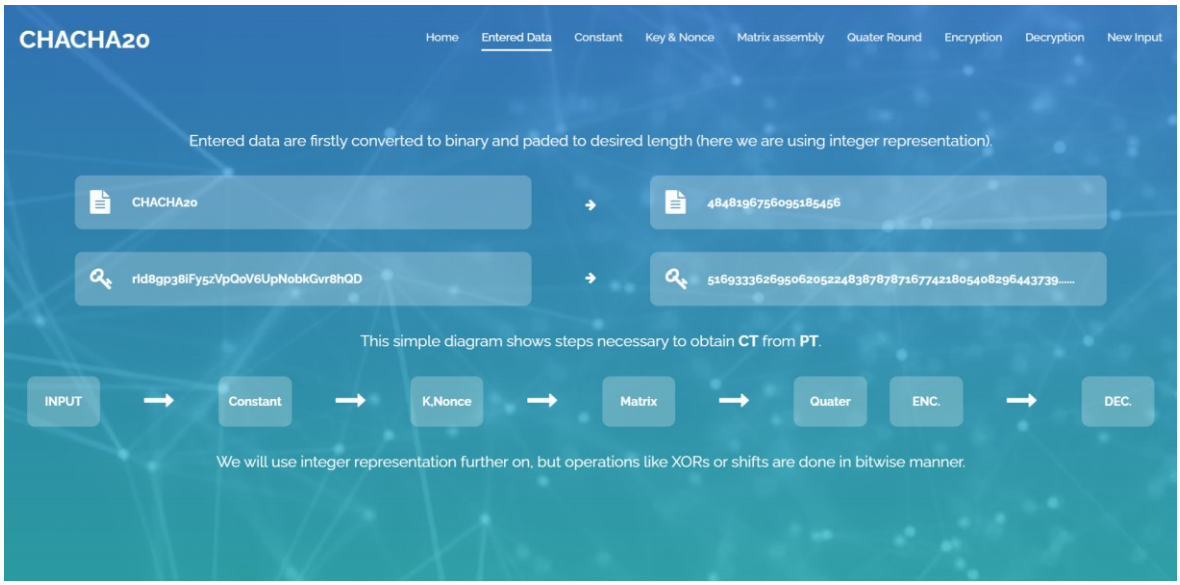
6. splitNonce – funkcia, ktorá rozdelí 96 bitov dlhú vstupnú noncu na 32 bitov dlhé časti, ktoré následne konvertuje na ich číselnú (integer) reprezentáciu a vráti vektor týchto hodnôt,
7. generateMatrix – funkcia, ktorá vygeneruje maticu, z hodnôt vytvorených pomocou predchádzajúcich funkcií, tak ako je definovaná v dokumentácií šifry ChaCha20 (viz. kapitola **3.2.1.1**),
8. quaterRoundFunction – funkcia, slúžiaca na zmenu štyroch vstupných číselných hodnôt z matice pomocou operácií sčitovania, modulácie, logického súčtu a rotácie bitov, ako udáva dokumentácia (viz. kapitola **3.2.1.2**),
9. columnRoundFunction – funkcia, volajúca *quaterRoundFunkciu* na stĺpce matice resp. na prvky v týchto stĺpcoch (viz. kapitola **3.2.1.3**),
10. diagonalRoundFunction – funkcia, volajúca *quaterRoundFunkciu* na diagonály matice resp. na prvky v týchto diagonálach (viz. kapitola **3.2.1.4**),
11. encryptChaCha20 – funkcia slúžiaca na zašifrovanie vstupu s využitím algoritmu šifry *ChaCha20*. Funkcia okrem zašifrovaného vstupu vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*matrixFlat, PT*),
12. decryptChaCha20 – funkcia slúžiaca na dešifrovanie vstupu pomocou vygenerovaných podkľúčov s využitím algoritmu šifry *Camellia*, pre 192 alebo 256 bitový kľúč. Funkcia rovnako okrem dešifrovaného vstupu vracia aj vektor, ktorého hodnota slúži na vizualizačné účely (*CT*).

5.4.1 GUI

Vstupné dáta do šifry *ChaCha* zadávame pomocou HTML šablóny *chacha20_input.html* (zobrazuje **Obrázok 53**), kedy po stisknutí tlačidla “Encrypt”, sú dáta predané funkcií *chacha20* (súbor *views.py*), ktorá vytvorí objekt danej šifry, spracuje dáta a taktiež vráti príslušnú HTML šablónu. Každá zo šablón obsahuje niekoľko strán popisujúcich proces tvorby šifrovacích podkľúčov ako aj samotného šifrovania a dešifrovania. Na navigáciu slúži jednoduchý diagram v spodnej časti strany alebo menu v pravom hornom rohu (zobrazuje **Obrázok 54**).



Obrázok 53 – HTML šablona chacha20_input.html, Zdroj: vlastný



Obrázok 54 – úvodná strana šifry ChaCha20 pre 256 bitový náhodný kľúč a správu "CHACHA20", Zdroj: vlastný

5.5 Kuznyechik – popis a funkcie

Funkcionalitu šifry Kuznyechik zabezpečujú dve HTML šablóny, ktorých funkciu sme opísali v kapitole **5.1.2** (zložka *templates*):

1. kuznyechik_input.html,
2. kuznyechik.

Dve funkcie slúžiace na renderovanie týchto šablón (súbor *views.py*):

1. kuznyechik_input – jednoduchá funkcia zhodná svojim telom s funkciou index z **Obrázok 45**, pričom opäť renderujeme rozdielnu šablónu, v tomto prípade *kuznyechik_input.html*,
2. kuznyechik – zložitejšia funkcia, ktorej všeobecnú funkciu sme priblížili v kapitole **5.1.4**

A dva súbory so zdrojovými kódmi:

1. Kuznyechik.py,
2. kuznyechikLinearFunction.py.

Kuznyechik.py obsahuje triedu *Kuznyechik*, ktorá obsahuje spolu 11 funkcií :

1. __init__ – inicializačná funkcia triedy,
2. X – funkcia, ktorá XOR-ne dve vstupné číselné (integer) hodnoty medzi sebou a vráti výsledok tejto operácie,
3. S – funkcia, ktorá prevedie 128 bitový vstup na šestnásť osembitových hodnôt, ktoré prevedie na ich číselný reprezentáciu a využije ich ako indexy na substitúciu do vektora *Pi*. Každú zo substituovaných hodnôt následne prevedie späť na jej binárnu reprezentáciu, spojí všetky tieto hodnoty a prevedie tento výsledok na číselnú hodnotu, ktorú následne vráti,
4. S_inverse – funkcia, zhodná s predchádzajúcou *S* funkciou avšak na substitúciu využíva vektor *Pi_neg* nie vektor *Pi*,
5. R – funkcia, ktorá na vstup aplikuje funkciu *kuznyechik_linear_function* (súbor *kuznyechikLinearFunction.py*), výsledok orotuje doľava o 120 bitov a následne logicky sčíta s pôvodnou hodnotou vstupu orotovanou o 8 bitov doprava,

6. R_inverse – inverzná, funkcia k funkcií *R*,
7. L – funkcia, volajúca funkciu *R* na svoj vstup 16 krát,
8. L_inverse – funkcia, volajúca funkciu *R_inverse* na svoj vstup 16 krát,
9. generateSubKeys – funkcia, slúžiaca na tvorbu šifrovacích podkľúčov, podľa dokumentácie (viz. kapitola **3.1.4.1**),
10. encryptKuznyechik – funkcia slúžiaca na zašifrovanie vstupu s využitím algoritmu šifry *Kuznyechik*. Funkcia okrem vektora vstupu (OT) v iteráciách šifrovacieho procesu (*CT_list*) vracia aj vektory, ktorých hodnoty slúžia na vizualizačné účely (*c_list, keys*),
11. decryptKuznyechik – funkcia slúžiaca na dešifrovanie vstupu s využitím algoritmu šifry *Kuznyechik*. Funkcia vracia iba vektor vstupu (ŠT) v iteráciách dešifrovacieho procesu.

A dva vektory obsahujúce číselné hodnoty slúžiace na substitúciu v rámci šifrovacieho procesu :

1. Pi,
2. Pi_neg.

kuznyechikLinearFunction.py obsahuje spolu 5 funkcií, ktorých autorom je pán **Terry Jackson** a sú voľne dostupné na adrese zdroja [37]**:**

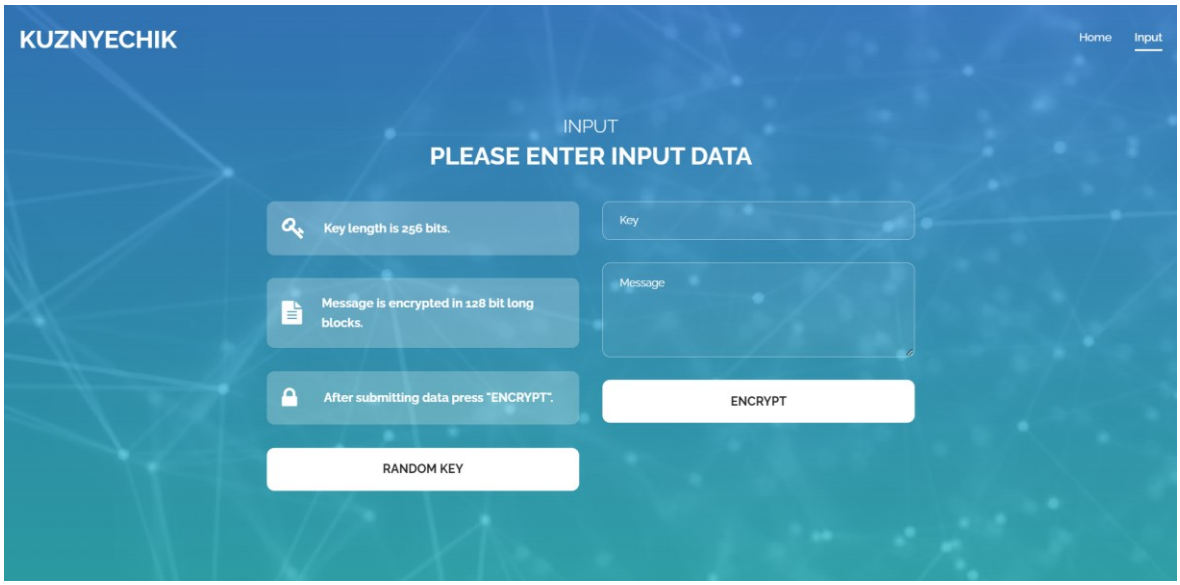
1. multiply_ints_as_polynomials – funkcia, násobiaca číselné hodnoty ako binárne mnohočleny (polynómy),
2. mod_int_as_polynomial – funkcia, modulujúca číselné hodnoty ako binárne mnohočleny (polynómy),
3. kuznyechik_multiplication – funkcia, ktorá najskôr vynásobí vstupné hodnoty medzi sebou pomocou funkcie *multiply_ints_as_polynomials* a následne tento výsledok použije ako vstup do funkcie *mod_int_as_polynomial* spolu s polynómom *m* udávajúcim príznaky obsadených miest v polynóme v dokumentácií šifry (viz. kapitola **3.1.4 -** $\mathbb{F}$),
4. kuznyechik_linear_function – funkcia, ktorá násobí, v cykle while, 16 bajtovú premennú *x* bajt po bajte spolu s vektorom číselných hodnôt (každá o veľkosti jedného bajtu) definovanom v dokumentácií (viz. kapitola **3.1.4** – tesne nad

začiatkom nasledujúcej podkapitoly **3.1.4.1**), počínajúc posledným prvkom. Táto hodnota je následne logickým súčtom (OR) sčítaná s hodnotou premennej *y*, ktorá bola inicializovaná ako 0 na začiatku cyklu,

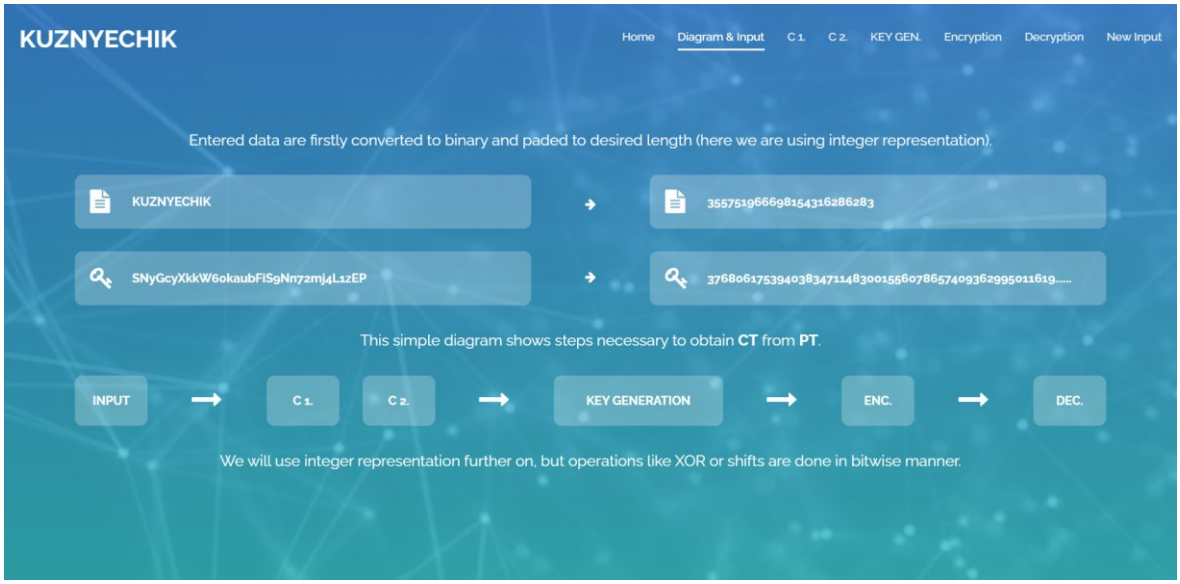
5. number_bits – jednoduchá funkcia, vracajúca dĺžku bitového reťazca,

5.5.1 GUI

Vstupné dáta do šifry *Kuznyechik* zadávame pomocou HTML šablóny *kuznyechik_input.html* (zobrazuje **Obrázok 55**), kedy po stisknutí tlačidla "Encrypt", sú dáta predané funkcií *kuznyechik* (súbor *views.py*), ktorá vytvorí objekt danej šifry, spracuje dáta a taktiež vráti príslušnú HTML šablónu. Každá zo šablón obsahuje niekoľko strán popisujúcich proces tvorby šifrovacích podkľúčov ako aj samotného šifrovania a dešifrovania. Na navigáciu slúži jednoduchý diagram v spodnej časti strany alebo menu v pravom hornom rohu rovnako ako v prípade ostatných šifier (zobrazuje **Obrázok 56**).



Obrázok 55 – HTML šablona kuznyechik_input.html, Zdroj: vlastný



Obrázok 56 – úvodná strana šifry Kuznyechik pre 256 bitový náhodný kľúč a správu “KUZNYECHIK”, Zdroj: vlastný

6 NÁVRH ÚPRAV

Ako každá aplikácia/program aj naša aplikácia má svoje slabé stránky a dala by sa v mnohých smeroch upraviť a vylepšiť. Pri snáď každej aplikácií by sme mohli optimalizovať zdrojový kód a užívateľské rozhranie aplikácie, čím by sme docielili zrýchlenie chodu aplikácie a efektívnejšie využívanie zdrojov. Ďalej by sme konkrétne pre našu aplikáciu mohli zvážiť napr. (okrem spomínanej optimalizácie už existujúceho kódu):

1. implementáciu ďalších šifrovacích algoritmov,
2. pridane šifrovania v režimoch činnosti,
3. pridanie viacerých jazykových mutácií,
4. prispôsobenie rozhrania pre zobrazenie na mobilných zariadeniach,
5. rozšírenie teoretických znalostí týkajúcich sa kryptológie obsiahnutých na stránke.

ZÁVER

Cieľom tejto bakalárskej práce bolo popísať problematiku modernej kryptológie, zvoliť vhodné algoritmy a implementačné, vizualizačné prostriedky, implementovať samotné algoritmy v podobe webovej stránky a v neposlednom rade zhodnotiť výsledky implementácie.

V teoretickej časti boli popísané základné kryptologické pojmy a princípy (viz kapitola **1**). V druhej kapitole potom práca popisuje vývoj kryptológie a jej metód (viz kapitola **2**). V tretej kapitole, ktorej bol venovaný dlhší úsek, bola popísaná problematika modernej kryptológie (viz. kapitola **3**), algoritmy zvolené na implementáciu (viz. kapitoly **3.1.3**, **3.1.4** ,**3.2.1**) a taktiež aj najnovšie metódy zabezpečenia (viz kapitoly **3.4.6, 3.5**).

Praktická časť práce v troch kapitolách najskôr opisuje prostriedky použité pri vývoji aplikácie (viz kapitola **4.1**). V druhej kapitole popisuje štruktúru aplikácie, teda definuje súbory obsahujúce šablóny, zdrojové kódy, pomocné funkcie atď. Pričom rovnako popisuje aj funkcie v nich obsiahnuté. Pri realizácií aplikácie som samozrejme narazil na niekoľko problémov, spojenými ako s backend-om tak aj frontend-om aplikácie. Vo väčšine prípadov boli tieto problémy riešiteľné preštudovaním dokumentácie (použitých jazykov, framework-u Django, platformy Azure). Pričom som často narazil aj na, nové možnosti riešenia implementácie súčastí aplikácie (algoritmov a pod.). Zdrojové kódy a stručná dokumentácia aplikácie je dostupná v prílohe alebo na adrese zdroja [38]. Aplikácia je taktiež dostupná priamo na webe na adrese zdroja [35]. Dokumentácia je dostupná priamo v zložke **mainApp** (alebo na adrese zdroja [39]) v podobe HTML súboru pre každý súbor so zdrojovým kódom (viz. kapitola **5.1.2** a **Obrázok 42**). Napríklad pre súbor *Kuznyechik.py* je to teda súbor *Kuznyechik.html.* V tretej kapitole je taktiež uvedených niekoľko návrhov na vylepšenie aplikácie.

Na záver by sme mohli povedať, že aplikácia je v praxi použiteľná a vhodná na ďalší vývoj ,pričom vo svojej terajšej podobe poslúži najmä na edukačné účely.

ZOZNAM POUžitej literatúry

[1] VONDRUŠKA, Pavel. *Kryptologie, šifrování a tajná písma 1. vydání*. Praha: Albatros, 2006, s. 340.

[2] PAAR, Christof. *Understanding Cryptography*. Berlin: Springer-Verlag GmbH Germany, 2010, s. 372.

[3] PIPER, F.C. *Kryptografie*. Praha: Dokořán, 2006, s. 157.

[4] CRAMER, Shoup. Design and Analysis of Practical Public-Key Encryption Schemes. *Https://www.shoup.net/papers/cca2.pdf*. New York, 2003. Dostupné také z: https://www.shoup.net/papers/cca2.pdf

[5] *The Ancient Cryptography History*. 2017. Dostupné také z: http://www.icits2015.net/ancient-cryptography-history/

[6] *A Brief History of Cryptography*. 2008. Dostupné také z: http://cryptozine.blogspot.com/2008/05/brief-history-of-cryptography.html

[7] *A Brief History of Cryptography*. 2013. Dostupné také z: http://www.cypher.com.au/crypto_history.htm

[8] F.L, Bauer. Cæsar Cipher. VAN TILBORG H.C.A, Jajodia. *Encyclopedia of Cryptography and Security.* Boston: Springer, 2011, s. 1500. Dostupné také z: https://doi-org.proxy.k.utb.cz/10.1007/978-1-4419-5906-5_162

[9] F.L., Bauer. Substitutions and Permutations. VAN TILBORG H.C.A., Jajodia. *Encyclopedia of Cryptography and Security.* Boston: Springer, 2011, s. 1500. Dostupné také z: https://doi-org.proxy.k.utb.cz/10.1007/978-1-4419-5906-5_176

[10] SINGH, Simon. *Kniha kódů a šifer : tajná komunikace od starého Egypta po kvantovou kryptografii.* Praha: Dokořán, 2009, s. 382.

[11] HANŽL, Tomáš,. *Šifry a hry s nimi: kolektivní outdoorové hry se šiframi.* Praha: Portál, 2007.

[12] BAUER, Friedrich. *Decrypted Secrets: Methods and Maxims of Cryptology*. Munich: Springer-Verlang Berlin, 2002, s. 401.

[13] VANĚK., Tomáš. *Moderní blokové šifry I.* Dostupné také z: rantos.cz/IBE-prezentace/P03_Blokove_sifry_v1.4.3SHORT.pdf

[14] A., Biryukov. Feistel Cipher. VAN TILBORG H.C.A., Jajodia. *Encyclopedia of Cryptography and Security.* Boston: Springer, 2011.

[15] B., Preneel. Modes of Operation of a Block Cipher. VAN TILBORG H.C.A., Jajodia. *Encyclopedia of Cryptography and Security.* Boston: Springer, 2011.

[16] AOKI KAZUMARO, ICHIKAWA. Specification of Camellia - 128-bit Block Cipher. *NTT Group*. 2000. Dostupné také z: https://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf

[17] V. DOLMATOV, Ed. *Https://www.ietf.org/*. 2016. Dostupné také z: https://tools.ietf.org/html/rfc7801#section-2

[18] TECHNICAL COMMITTEE FOR STANDARDIZATION "CRYPTOGRAPHY AND SECURITY MECHANISMS". *Https://tc26.ru/en/*. 2015. Dostupné také z: https://tc26.ru/en/standards/standards/gost-r/gost-r-34-12-2015-information-technology-cryptographic-data-security-block-ciphers.html

[19] BERNSTEIN, Daniel. Cr.yp.to. *Salsa20*. 2007. Dostupné také z: https://cr.yp.to/snuffle.html#specification

[20] BERNSTEIN, Daniel. Cr.yp.tp. *ChaCha20*. 2008. Dostupné také z: https://cr.yp.to/chacha/chacha-20080128.pdf

[21] ADAM LANGLEY, Wan-Teh. Tools.ietf.org. *The ChaCha Stream Cipher for Transport Layer Security*. 2014. Dostupné také z: https://tools.ietf.org/id/draft-mavrogiannopoulos-chacha-tls-01.html

[22] GABRIELA, Bílková. *Moderní kryptologie. Bakalárska práca.* Zlín: Univerzita Tomáše Bati ve Zlíně, 2010.

[23] Crypto Museum. *One-Time Pad*. 2016. Dostupné také z: https://www.cryptomuseum.com/crypto/otp/index.htm

[24] ROBSHAW, Matthew. One-Way Function. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[25] PRENEEL, Bart. Hash Functions. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[26] MENEZES ALFRED J., van. *Handbook of Applied Cryptography*. CRC Press, 1996.

[27] PRENEEL, Bart. MAC Algorithms. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[28] JUST, Mike. Diffie–Hellman Key Agreement. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[29] SAKO, Kazue. Digital Signature Schemes. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[30] OULEHLA, Milan. *Moderní kryptografie*. Praha: IFP Publishing s.r.o., 2017.

[31] KALISKIJR., Burt. RSA Digital Signature Scheme. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[32] FREDERIK ARMKNECHT, Colin. Eprint.iacr.org. *A Guide to Fully Homomorphic*. 2016. Dostupné také z: https://eprint.iacr.org/2015/1192

[33] SCHOENMAKERS, Berry. Homomorphic Encryption. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[34] CRÉPEAU, Gilles. Quantum Cryptography. HENK C. A. VAN TILBORG, Sushil. *Encyclopedia of Cryptography and Security*. Boston: Springer, 2011.

[35] *Https://algorithmvisualiser.azurewebsites.net/*.

[36] *Https://www.free-css.com/free-css-templates/page260/elegance*.

[37] *Https://github.com/jacksoninfosec/kuznyechik/blob/main/kuznyechik.py*.

[38] *Https://github.com/Milan-Janovic/algvisualiser*.

[39] *Https://github.com/Milan-Janovic/algvisualiser/tree/main/mainApp*.

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

OT otvorený text

PK privátny (súkromný) kľúč

ŠT šifrovaný text

VK verejný kľúč

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍHLOH

PRÍLOHA P I: OBSAH CD

Štruktúra priloženého CD:

- Adresár **Text bakalárskej práce** – obsahuje text bakalárskej práce vo formáte PDF.
- Adresár **Zdrojové kódy** – obsahuje zdrojové kódy šifier s príponou .py programovacieho jazyka Python verzie 3.x.
- Adresár **BP_WEB_VIZUALIZACIA_SA** – vo formáte .zip, obsahuje všetky súbory aplikácie. Slúži na otvorenie aplikácie ako celku vo vývojom prostredí.
- Adresár **Dokumentácia** – obsahuje HTML dokumentáciu .py súborov zdrojových kódov šifier a súborov zdrojových kódov, ktoré šifry využívajú.