# Návrh a implementace profesionální kryptografické kalkulačky

Marek Souček

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Marek Souček** |
| Osobní číslo: | **A20319** |
| Studijní program: | **B0613A140020 Softwarové inženýrství** |
| Forma studia: | **Prezenční** |
| Téma práce: | **Návrh a implementace profesionální kryptografické kalkulačky** |
| Téma práce anglicky: | **Design and Implementation of a Professional Cryptographic Calculator** |

## Zásady pro vypracování

1. Nastudujte si kryptografickou problematiku.
2. Identifikujte kryptografické algoritmy, které mají být implementovány.
3. Navrhněte vstupní jazyk.
4. Přiřaďte jména jednotlivým algoritmům, tato jména budou použita ve vstupním jazyku.
5. Zdůvodněte výběr kryptografických algoritmů.
6. Vyberte programovací jazyk a knihovny s příslušnými algoritmy.
7. Navrhněte a implementujte jazyk výrazů pro zadání výpočtu a pro textový záznam činnosti.
8. Zkompletujte aplikaci kalkulačky a otestujte ji.

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

Seznam doporučené literatury:


1. MENEZES, Alfred J, OORSCHOT, Paul C a VANSTONE, Scott A. Handbook of applied cryptography. Vyd. 5. Boca Raton : CRC Press, 2001. str. 816. http://cacr.uwaterloo.ca/hac/. ISBN 08-493-8523-7.
2. BURDA, Karel. Úvod do kryptografie. Brno: Akademické nakladatelství CERM, 2015. ISBN 978-807-2049-257.
3. BONEH, Dan. A Graduate Course in Applied Cryptography. Stanford, 2015.
4. SCHNEIER, Bruce. Applied cryptography: protocols, algorithms, and source code in C. 2nd ed. New York: Wiley, 1996. ISBN 9780471117094.
5. SINGH, Simon. Kniha kódů a šifer: tajná komunikace od starého Egypta po kvantovou kryptografii. Praha: Dokořán, 2003. Aliter (Argo: Dokořán): Dokořán). ISBN 80-865-6918-7.


Vedoucí bakalářské práce: **Ing. Pavel Drábek, Ph.D.**
Ústav automatizace a řídicí techniky

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

**doc. Ing. Jiří Vojtěšek, Ph.D.** v.r.
děkan

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

Ve Zlíně dne 5. ledna 2024

**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 9. 5. 2024

Marek Souček v. r.
podpis studenta

## ABSTRAKT

Bakalářská práce se zaměřuje na vytvoření kryptografické kalkulačky, kterou může uživatel používat pomocí vlastního grafického uživatelského rozhraní (GUI). Výsledek umožňuje využívat funkce šifrování, dešifrování, hashování, generování klíčů a dalších, dle požadavků jednotlivých šifer. Kalkulačka je vytvořena v jazyce Python. Cílem práce je také nastínit základní problematiku kryptologie s důrazem na obor kryptografie. To zahrnuje historické šifry, problematiku jednotlivých použitých šifer a důvody jejich použití.



Klíčová slova: Kryptologie, Kryptografie, AES, DES, Hash funkce



## ABSTRACT

The bachelor's thesis focuses on creating a cryptographic calculator that users can utilize through a custom graphical user interface (GUI). The outcome enables users to utilize functions such as encryption, decryption, hashing, key generation, and others, according to the requirements of individual ciphers. The calculator is developed using the Python programming language. The aim of the thesis is also to outline the fundamental aspects of cryptology with an emphasis on the field of cryptography. This includes historical ciphers, the issues surrounding the individual ciphers used, and the reasons for their usage.

Abstrakt ve světovém jazyce



Keywords: Cryptology, Cryptography, AES, DES, Hash function

Chtěl bych poděkovat vedoucímu práce Ing. Pavlu Drábkovi, Ph.D. za pomoc a správné nasměrování cesty bakalářské práce. Taktéž bych chtěl poděkovat svým rodičům za podporu při vypracovávání bakalářské práce.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

**ÚVOD** .......... **9**
**I TEORETICKÁ ČÁST** .......... **10**
**1 KRYPTOLOGIE** .......... **11**
1.1 KRYPTOGRAFIE .......... 11
1.2 KRYPTOANALÝZA .......... 11
1.3 STEGANOGRAFIE .......... 12
1.4 ZÁKLADNÍ POJMY .......... 12
**2 KLASICKÁ KRYPTOGRAFIE** .......... **14**
2.1 SUBSTITUČNÍ ŠIFRY .......... 14
2.1.1 Mono-alfabetické substituční šifry .......... 15
2.1.1.1 Caesarova šifra .......... 15
2.1.1.2 Pevný posun .......... 16
2.1.1.3 ATBASH .......... 16
2.1.1.4 Afinní šifra .......... 16
2.1.2 Poly-alfabetické substituční šifry .......... 17
2.1.2.1 Vigenerova šifra .......... 17
2.1.3 Polygrafické substituční šifry .......... 19
2.1.3.1 Playfair šifra .......... 19
2.1.4 Homofonní substituční šifry .......... 21
2.2 TRANSPOZIČNÍ ŠIFRY .......... 21
2.2.1 Skytale .......... 21
2.2.2 Rail-fence .......... 22
2.2.3 Transpozice s klíčem .......... 22
2.2.4 Zubatka .......... 23
2.3 HYBRIDNÍ ŠIFRY .......... 24
2.3.1 ADFGX, ADFGVX .......... 24
**3 MODERNÍ KRYPTOGRAFIE** .......... **27**
3.1 SYMETRICKÉ ŠIFROVACÍ ALGORITMY .......... 27
3.1.1 Blokové symetrické šifrovací algoritmy .......... 28
3.1.1.1 DES .......... 28
3.1.1.2 3DES .......... 29
3.1.1.3 AES .......... 29
3.1.1.4 Blowfish .......... 30
3.1.2 Proudové symetrické šifrovací algoritmy .......... 31
3.1.2.1 ChaCha20 .......... 31
3.2 ASYMETRICKÉ ŠIFROVACÍ ALGORITMY .......... 31
3.2.1 RSA .......... 31
3.2.2 DSA .......... 32
3.3 HASHOVACÍ ALGORITMY .......... 33
3.3.1 MD5 .......... 33
3.3.2 SHA-1 .......... 33
3.3.3 SHA-256 .......... 33
3.3.4 SHA-3 .......... 34

**II PRAKTICKÁ ČÁST** ... **35**
**4 SPECIFIKACE A IMPLEMENTACE ALGORITMŮ** ... **36**
4.1 AES ... 36
4.2 DES, 3DES ... 38
4.3 BLOWFISH ... 42
4.4 RSA ... 44
4.5 CHACHA20 ... 47
4.6 DSA ... 49
4.7 HASHOVACÍ FUNKCE ... 50
4.8 GENEROVÁNÍ KLÍČŮ A DALŠÍCH PARAMETRŮ ... 52
4.8.1 Generování AES klíče a dalších parametrů ... 53
4.8.2 Generování DES, 3DES, BF a ChaCha20 klíčů a dalších parametrů ... 54
4.8.3 Generování soukromého a veřejného klíče u RSA ... 55
4.8.4 Generování soukromého a veřejného klíče u DSA ... 57
**5 POUŽITÉ TECHNOLOGIE** ... **59**
5.1 PYTHON ... 59
5.2 VÝVOJOVÉ PROSTŘEDÍ ... 59
5.3 GUI V QT DESIGNER ... 59
**6 POPIS A STRUKTURA UŽIVATELSKÉHO ROZHRANÍ** ... **61**
6.1 STRUKTURA PRVKŮ V UŽIVATELSKÉM ROZHRANÍ ... 61
6.2 POPIS UŽIVATELSKÉHO ROZHRANÍ ... 62
6.2.1 Popis stránky AES ... 62
6.2.2 Popis stránky DES ... 64
6.2.3 Popis stránky RSA ... 65
6.2.4 Popis stránky ChaCha20 ... 66
6.2.5 Popis stránky DSA ... 67
6.2.6 Popis stránky pro Hashovací algoritmy ... 68
**7 IMPLEMENTACE FUNKCÍ K UŽIVATELSKÉMU ROZHRANÍ** ... **70**
7.1 AES K UŽIVATELSKÉMU ROZHRANÍ ... 70
7.2 DES, 3DES A BLOWFISH K UŽIVATELSKÉMU ROZHRANÍ ... 73
7.3 CHACHA20 K UŽIVATELSKÉMU PROSTŘEDÍ ... 76
7.4 RSA K UŽIVATELSKÉMU ROZHRANÍ ... 78
7.5 DSA K UŽIVATELSKÉMU ROZHRANÍ ... 79
7.6 HASHOVACÍCH ALGORITMŮ K UŽIVATELSKÉMU ROZHRANÍ ... 81
7.7 DALŠÍ FUNKCE K UŽIVATELSKÉMU ROZHRANÍ ... 82
7.7.1 Kopírování v AES ... 82
7.7.2 Načtení souboru skrze GUI ... 83
7.7.3 Přepínání stránek pomocí radioboxu ... 83
7.7.4 Funkce pro přepínání výběrových polí ... 84
7.7.5 Přepínání menu ... 85
7.7.6 Volání jednotlivých funkcí do GUI ... 85
**8 TESTOVÁNÍ APLIKACE** ... **87**

8.1 Šifrování, dešifrování textu a generování textového klíče s IV u AES ....... 87
8.2 Šifrování, dešifrování souboru a generování klíče v souboru s IV u AES ....... 89
8.3 Šifrování, dešifrování textu a generování textového klíče s IV u DES ....... 91
8.4 Šifrování, dešifrování souboru a generování klíče v souboru s IV u DES ....... 91
8.5 Šifrování, dešifrování textu a generování textového klíče s IV u 3DES ....... 92
8.6 Šifrování, dešifrování souboru a generování klíče v souboru s IV u 3DES ....... 92
8.7 Šifrování, dešifrování textu a generování textového klíče s IV u Blowfish ....... 93
8.8 Šifrování, dešifrování souboru a generování klíče v souboru s IV u Blowfish ....... 93
8.9 Šifrování, dešifrování textu a generování textových klíču u RSA ....... 94
8.10 Šifrování, dešifrování souboru a generování klíčů v souboru u RSA ....... 94
8.11 Šifrování, dešifrování textu a generování textového klíče a nonce u ChaCha20 ....... 95
8.12 Šifrování, dešifrování souboru a generování klíče a nonce v souboru u ChaCha20 ....... 95
8.13 Vytváření podpisu, ověřování a generování klíčů v textové formě u DSA ....... 96
8.14 Vytváření podpisu, ověřování a generování klíčů ve formě souboru u DSA ....... 97
8.15 Vytvoření hashe jako text ze souboru u SHA-1, SHA-256, SHA-3 a MD5 ....... 98
8.16 Vytvoření hashe jako soubor z textu u SHA-1, SHA-256, SHA-3 a MD5 ....... 99
**ZÁVĚR ....... 100**
**SEZNAM POUŽITÉ LITERATURY ....... 101**
**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ....... 105**
**SEZNAM OBRÁZKŮ ....... 106**
**SEZNAM TABULEK ....... 109**
**SEZNAM PŘÍLOH ....... 110**

# ÚVOD

V současné době, kdy se technologie vyvíjí stále rychlejším tempem se ochrana informací a dat stává mnohem důležitějším aspektem našich životů jak kdy předtím. Z důvodu neustálého technologického posunu je třeba si uvědomovat základy, na kterých stojí pro využití a možnosti dalšího vývoje. To samé platí i co se týče kryptologie, která se musí také neustále posouvat, aby nám ze světa nezmizelo zabezpečení informací. Neustálý vývoj nás přibližuje k realizaci kvantového počítače, což je nebezpečná představa na poli kryptografie. Taková realizace by totiž mohla ze současných, dneska považovaných bezpečných šifer udělat šifry nepotřebné a již zbytečné. Vytváření obtížnějších kryptografických algoritmů způsobuje obtížnější manipulaci s nimi. Tahle práce by měla pomoci pochopit problém kryptografie a vysvětlit některé ze známých šifer, které se používaly jak v minulosti, tak dnes.

První část práce se soustředí na teoretickou část, která začíná obecným vyjádřením oboru kryptologie, jeho rozdělení na kryptografií, kryptoanalýzu a steganografií. Dále se soustředím na základní rozdělení klasických šifer se zaměřením na jejich princip a na moderní kryptografií se zaměřením na kryptografické algoritmy, které využívám v praktické části.

Druhá část práce se zaměřuje na praktickou část, konkrétně specifikaci jednotlivých algoritmů využitých v programu s nějakým vysvětlením, popisu technologií použitých pro implementaci. Dále ukázka uživatelského rozhraní a jakým principem funguje, propojování jednotlivých algoritmů s uživatelským rozhraním a další různé funkce využité v programu a základní manuální testování, které skvěle ukazuje jak s programem lze pracovat.

# I. TEORETICKÁ ČÁST

# 1 KRYPTOLOGIE

Kryptologie je velice obsáhlý vědní obor, který se skládá z kryptografie, kryptoanalýzy a steganografie. Když se bavíme o kryptologií, mluvíme tedy o pojmu, který zastřešuje tyto tři podobory, které se navzájem doplňují a společně tvoří samotnou kryptologii. Kryptografie by se dala popsat jako obor zabývající se upravováním zprávy, dat či informace do podoby, aby nebyla čitelná nikým jiným, než tím komu je tato informace určena. Z toho vyplývá že se nezabývá nejen procesem šifrování, ale i vytvářením samotných šifer. Naopak kryptoanalýza se zabývá procesem prolomování, nebo dešifrováním této informace z pohledu útočníka. To znamená bez znalosti klíče nebo šifry která je použita. Posledním oborem je steganografie, která se od kryptografie liší tím, že se zabývá skrýváním existence dané informace, cílem je aby útočník o této informaci nevěděl. Pokud neví o její existenci, tak se nepokusí informaci dále zkoumat. [1][2]

## 1.1 Kryptografie

Kryptografie vznikla z řeckého slova *kryptos,* neboli skrytý a *graphein* neboli psát. Hlavní cíl je schovat význam dané zprávy, nikoliv zprávu. K tomu se využívá šifrování, které zajistí, aby význam zprávy zůstal pouze mezi odesílatelem a příjemcem dané zprávy. Aby k tomuto mohlo dojít, musí se daná zpráva změnit dle nějakých stanovených pravidel, aby byla pro kohokoliv jiného nečitelná. Dle složitosti daných pravidel je pro útočníka možné tuto zprávu dešifrovat buď s opravdu velkým úsilím nebo také vůbec. Tato věda se zabývá nejen tímto procesem ale i dalšími, jako je například generování klíčů, digitálním podpisem a hashovacími funkcemi. Všechny tyto algoritmy se dají také nazvat jako kryptografické funkce, kryptografie se pak zabývá jak samotným procesem zašifrování, úpravy otevřeného textu v zašifrovaný text, tak i vytvářením těchto funkcí. [2][3]

## 1.2 Kryptoanalýza

Kryptoanalýza, neboli nauka, či věda o tom jak dešifrovat zprávu bez znalosti klíče nebo samotná pravidla pro dešifrování. Zkoumá zašifrovanou zprávu, šifry nebo systémy s cílem odhalení jejich slabin a vytvoření technik pro jejich snadnější prolomování a oslabení. Kryptoanalýza zkoumá veškeré kryptografické funkce, nikoliv jen šifry. Odhalení této slabiny může fungovat buď jako pomůcka pro útočníky nebo také jako prevence k odhalení slabiny šifry, za účelem zjištění zda se šifra vyplatí používat. Tohle se dá rozhodnout dle složitosti

dešifrování a tento proces může o dané šifře říct další docela podstatné informace pro vytvoření jiných bezpečnostních opatřeních. [2][4]

## 1.3 Steganografie

Steganografie, zkráceně se dá popsat jako utajená komunikace pomocí ukrytí zprávy. Vznikla z řeckých slov *steganos* neboli schovaný a *graphein* neboli také psát. Hlavním cílem této vědy je schovat danou zprávu, aby nevzbudila žádnou pozornost. Ta však dále se zprávou nijak nepracuje, tudíž pokud se útočníkovi podaří odhalit existenci dané zprávy, tak je odhalena celá a útočník má přístup k celé zprávě v plném rozsahu. Z toho vyplývá, že pokud útočník dokáže zprávu zachytit, nic jiného mu nebrání v tom si ji přečíst. I přesto, že zajišťuje určitou vrstvu bezpečnosti, tak není sama o sobě tak efektivní jako, kdybychom zprávu zašifrovali. Právě z tohoto důvodu se mnohdy používá kombinace těchto dvou věd. [2]

## 1.4 Základní pojmy

**Šifrování/Zašifrování**

Je to proces, který přemění informaci nebo zprávu do nečitelné podoby a tím schová pravý význam této informace, k tomu využívá šifry. Taktéž se dá říct, že převádí otevřený text do podoby šifrovaného textu pomocí této šifry nebo šifrovacího algoritmu. [5]

**Dešifrování**

Jedná se o proces, díky kterému převádíme zašifrovaný text do formy otevřeného textu. Oficiálně se tento proces týká příjemce, ten zná klíč a šifru, ale dá se také použít na útočníka, který nemusí znát ani klíč a ani šifru. [2]

**Otevřený text**

Jedná se o zprávu, nebo text před procesem šifrování, tedy původní zpráva. [2][5]

**Zašifrovaný text**

Jedná se o zprávu nebo text, který je zašifrován do nečitelné formy. Ta se pak dá dešifrovat zpátky na otevřený text. [2][5]

**Šifrovací klíč**

Někdy také nazýváno čistě jako klíč, je jistá hodnota u šifer, která změní proces šifrování v jistý specifický postup. Taktéž se dá nazvat jako parametr dané šifry, pomocí kterého

šifrování provádíme. Bez znalosti klíče je obtížné zašifrovaný text dešifrovat i při znalosti použité šifry. [2][5]

**Soukromý klíč**

Někdy také nazývaný jako symetrický klíč. V symetrické kryptografií funguje tak, že tento klíč mezi sebou sdílí příjemce a odesílatel a pomocí něj odesílatel zašifruje otevřený text a odešle příjemci v podobě zašifrovaného textu. Odesílatel na to tento text dešifruje a dostane zpátky otevřený text. V případě asymetrické kryptografie funguje pouze pro příjemce, který soukromý klíč použije pouze k dešifrování. [7][2]

**Veřejný klíč**

Tento klíč je používaný pouze odesílatelem pro zašifrování otevřeného textu. Tento klíč je volně veřejný a dostupný veřejnosti, takže tento proces může provést kdokoliv, ale slouží pouze k zašifrování. [2][7]

**Šifra**

Jedná se o určitý systém, kterým se řídíme abychom dostali zašifrovaný text z otevřeného textu. V moderní kryptografií se jedná o kryptografické algoritmy, které provedeme, abychom tuto informaci zašifrovali do nečitelné podoby. [6][2]

**Kód**

Jedná se o systém, jehož úkol je schovat význam otevřeného textu tak, že nahrazuje každé slovo, frázi či znak v otevřeném textu jiným znakem, nebo skupinou znaků. Tento proces se řídi dle takzvané kódové knihy. Dalo by se také říct, že kód je šifra s jediným daným klíčem, čímž je samotná kódová kniha. [2]

**Kódová kniha**

Je to seznam znaků, frází či slov, které mají jasně definované zástupce v podobě jiného znaku, či skupinou znaků. [2]

# 2 KLASICKÁ KRYPTOGRAFIE

Co se týče klasické kryptografie, myslíme tím specificky klasické šifry a kódy. Kódy jako takové jsou velice jednoduché a jsou vázané na kódovou knihu, co se však týče klasických šifer, tak ty už jsou poněkud zajímavější a fungují na určitých pravidlech s větší možností kombinací. Tyto šifry jsou však na současné poměry nedostatečné a dají se častokrát vyřešit ručně. Kdybychom do tohoto odvětví zapojili moderní technologií, tak by pravděpodobně neobstály. Patří mezi ně veškeré jednoduché systémy, které používali jak staří Řekové a Římani až po období druhé světové války. Vývojem těchto šifer také vznikaly různé metody, které předcházely jednoduchému rozluštění těchto šifer. Například co se týče mono-alfabetických substitučních šifer, pro zvýšení bezpečnosti se používaly takzvané klamače a nomenklátory.

Klasické šifry se dělí dle techniky, kterou využívají při manipulaci s abecedou. Dělí se na substituční, transpoziční šifry a jejich kombinací taktéž takzvané hybridní. Substituční šifry fungují tak, že nahrazují znak, slovo či sekvenci znaků otevřeného textu za jiný znak, slovo či sekvenci znaků. Abeceda tohoto otevřeného textu je tedy nahrazena jinou a dá se upravovat buď systematickým posunem, nebo klíčem. Ty jsou například mono-alfabetické, poly-alfabetické, polygrafické, homofonní a další. Co se týče transpoziční šifry, tak ta jednotlivé znaky nenahrazuje, abeceda otevřeného textu zůstává stejná, avšak mění pořadí jednotlivých znaků v otevřeném textu. To se provádí buď systematicky, dle nějakého pravidla, nebo s kombinací klíče. Dalo by se říct, že pro mnohé tyto šifry stačila pouze její znalost pro její úspěšné dešifrování, což z nich automaticky na dnešní dobu dělá šifry neefektivní. Existuje však i třetí typ klasických šifer, které fungovaly na kombinaci substituční metody a transpoziční metody. To znamená, že provedli substituci znaků v otevřeném textu a poté uskutečnili transpozici těchto znaků. [2][1]

## 2.1 Substituční šifry

Jedná se o typ šifer, kdy zašifrování probíhá tak, že znak, slovo či sekvence znaků otevřeného textu jsou zaměněny za jiný znak, slovo či sekvenci znaků, dle předdefinovaných pravidel. Dešifrování pak probíhá velice podobně, akorát se u tohoto procesu obrací celá substituce, abychom získali zpátky otevřený text. Dělí se na šifry mono-alfabetické, poly-alfabetické, polygrafické, homofonní a další. Mezi nejznámější substituční šifry můžeme řadit například Caesarovu šifru, ATBASH, Afinní šifru, Vigenerovu šifru, Playfair, nebo

například upravenou verzi Caesarovi šifry jako je Pevný posun, známé také jako ROT13, která je daleko univerzálnější z pohledu využití a kombinací. [8][2]

### 2.1.1 Mono-alfabetické substituční šifry

U tohoto typu substitučních šifer je jednoznačné fixní pravidlo substituce aplikováno po celou délku otevřeného textu na všechny znaky stejně. To znamená, že pravidla u šifer tohoto typu jsou většinou jednoduché a jednoznačné. Je také charakteristická tím, že používá pouze jednu šifrovou abecedu, která je tvořena písmeny, čísly, symboly nebo jejich směsí. Negativní věcí u tohoto typu šifer je však to, že svou jednoduchostí jsou zranitelné vůči frekvenční analýze, která počítá frekvenci určitých znaků v zašifrovaném textu a tím lze snadněji analyzovat a odhadovat, jaké písmeno by mohlo zastupovat. Proto u tohoto typu šifer se začaly hojně používat takzvané nomenklátory a klamače. Klamače, někdy také nuly, mají za cíl ztížit samotnou šifru, tyto znaky nemají žádnou hodnotu ani význam co se týče zprávy jako takové v otevřeném textu, ale používají se pro zmatení útočníka při pokusu o dešifrování. Dalšími jmenovanými jsou nomenklátory, které mají podobný význam, akorát že fungují tak, že rozšíří samotnou šifrovou abecedu o jistý seznam kódových slov. Tyto kroky sice ztíží prolomení šifry, avšak nedá se říct že by dělali mono-alfabetické šifry bezpečnými. [2][8]

#### *2.1.1.1 Caesarova šifra*

Je známá také pod názvem Caesarova posunová šifra, kvůli jejímu principu. Celý princip této šifry je, že šifrová abeceda je posunutá o tři pozice dále oproti abecedě otevřeného textu. V praxi to fungovalo tak, že písmeno A je nahrazeno písmenem D, písmeno B je tedy nahrazeno písmenem E a tak dále. Poslední písmena jsou poté nahrazena písmeny ze začátku abecedy tedy písmeno Z by bylo písmenem C. Je pojmenována po Juliovi Caesarovi, který tuto substituční šifru vytvořil a také používal při psaní tajných dopisů s členy jeho armády.

Otevřená abeceda abcdefghijklmnopqrstuvwxyz
šifrová abeceda DEFGHIJKLMNOPQRSTUVWXYZABC
Otevřený text Veni, Vidi, Vici
šifrovýtext YHQL. YLGL. YLFL

Obrázek 1. Ukázka Caesarovy šifry [2]

Můžeme zde vidět seznam abecedy otevřeného textu, která je běžná anglická abeceda. Šifrová abeceda je však posunutá o tři místa dále a tyto abecedy zde pomáhají snadnějšímu

procesu šifrování. Pod touto abecedou můžeme také vidět otevřený text, který chceme zašifrovat a nakonec samotný zašifrovaný text, který využil šifrové abecedy během procesu šifrování. [2][8]

#### *2.1.1.2 Pevný posun*

Jedná se o další mono-alfabetickou šifru, která vznikla z principu Caesarovy šifry, někdy také nazývána jako Upravená Caesarova šifra, zde se písmeno neposouvá o tři písmena dále v šifrovací abecedě, ale je zde tohoto principu využito a umožněno tento posun libovolně měnit o libovolný počet míst v rozptylu 1-25. Tento rozptyl je v této formě šifry také známý jako klíč a dalo by se tedy říct, že Caesarova šifra je vlastně Pevný posun s klíčem tři. Je taky více univerzální díky vícero možnostem šifrování. [8][2]

#### *2.1.1.3 ATBASH*

ATBASH, také známé jako Reverzní abeceda, byla původně používána u hebrejské abecedy, ale samozřejmě ji můžeme použít u jakékoliv jiné abecedy, podobně jako Caesarovu šifru. Stejně jako Caesarova šifra, obsahuje velice jednoduchý princip, jehož vlastním klíčem je tedy přehozená abeceda. Vypadá nakonec tak, že přehazuje písmena z otevřeného textu první za poslední, druhé za předposlední a tak dále. Zašifrovaný text by tedy vypadal nakonec tak, že písmeno A by bylo tedy písmeno Z, písmeno B by pak bylo písmeno Y a tak dále. [8][2]

#### *2.1.1.4 Afinní šifra*

Jedná se o další mono-alfabetickou substituční šifru, která při zašifrování nahrazuje každý znak otevřeného textu jiným, dle stanovené hodnoty kterou dostaneme po provedení určité matematické operace dle následující funkce: $(ax + b)\ mod\ (26)$, kde $a$ násobí $x$, což představuje pořadí znaku z otevřeného textu v abecedě, $b$ je zde hodnota posunu a operátor *mod* zde slouží pro zajištění toho, že číslo bude vždy v rozsahu abecedy. Pro dešifrování poté použijeme opačnou funkci: $(x-b) * a^{-1}\ mod\ (26)$. Zde je písmeno $x$ pořadí znaku zašifrovaného textu, $b$ je tedy ten stejný posun a hodnotou pod písmenem $a$ se v tomto případě dělí a nakonec operátor *mod* opět zařídí to, aby výsledná hodnota bylo v rozsahu abecedy. Jako jednoduchý příklad si můžeme určit $a$ jako 1, $b$ můžeme také určit jako 1 a $x$ bude písmeno B, tedy v pořadí abecedy číslo 2. Když tyto hodnoty použijeme v naší matematické funkci, výsledná hodnota bude 3, tedy písmeno B z otevřeného textu bude v zašifrovaném textu

písmenem C. Při dešifrování můžeme hodnotu 3 použít tentokrát do opačné funkce a konečný výsledek bude opět číslo 2, tedy písmeno B. [9][8]

### 2.1.2 Poly-alfabetické substituční šifry

Tento typ šifer je velice podobný principu mono-alfabetických šifer, protože taky z nich velkým způsobem vychází. Jediný rozdíl mezi mono-alfabetickou šifrou a poly-alfabetickou šifrou je počet šifrovacích abeced, které využívá. U mono-alfabetické to byla vždycky jedna, akorát se měnila pravidla šifry, co se týče poly-alfabetických šifer, tak tam vstupuje více než jedna šifrovací abeceda. Úplně první poly-alfabetická šifra vznikla na myšlence, že první znak otevřeného textu by mohl být zašifrovaný první šifrovací abecedou, druhý znak zase druhou šifrovací abecedou, třetí by pak mohl být zašifrovaný zase pomocí první a tak dále. Tento princip se však samotným tvůrcem dále nerozvíjel a postupné pokusy o vývoj nebyly zdaleka tak úspěšné, jako šifra která v současné době zastupuje poly-alfabetické substituční šifry a tím je Vigenerova šifra, jejíž síla spočívá v kombinacích 26 různých šifrovacích abeced. [2][10]

#### *2.1.2.1 Vigenerova šifra*

Jedná se o poly-alfabetickou substituční šifru, kterou vytvořil francouzský diplomat Blaise de Vigenère na základě nesporně významných příspěvků Albertiho, Trithemia a Porty, nakonec byla pojmenována právě tvůrci této šifry. K šifrování pomocí Vigenerovy šifry sloužil seznam šifrovacích abeced dle Pevného posunu, kde každý řádek byl posunut o jednu pozici doprava a klíčové slovo, které určovalo kterou abecedu by měl odesílatel k zašifrování použít. Síla této šifry spočívá v kombinaci různých šifrovacích abeced, proto kdybychom použili právě pouze jedné abecedy, jednalo by se o šifrování jednoduchým Pevným posunem a nikoliv pomocí Vigenerovy šifry. Tento seznam šifrovacích abeced se také nazýval jako Vigenerův čtverec, kde součástí tohoto čtverce byl první řádek, který reprezentoval znaky otevřeného textu a první sloupec, který reprezentoval klíč šifry. Tento klíč určoval jakou abecedu použijeme při šifrování otevřeného textu. [2][10]

| | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| 2 | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| 3 | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| 4 | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| 5 | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| 6 | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| 7 | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| 8 | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| 9 | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| 10 | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| 11 | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| 12 | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| 13 | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| 14 | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| 15 | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 16 | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| 17 | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| 18 | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| 19 | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| 20 | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| 21 | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| 22 | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| 23 | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| 24 | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| 25 | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
| 26 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

Obrázek 2. Ukázka Vigenerova čtverce [2]

Kdybychom tedy chtěli zašifrovat otevřený text AHOJTE pomocí klíče TEXT, probíhalo by to následujícím způsobem. Pro lepší zorientování bychom si mohli napsat náš otevřený text a klíč pod sebe s tím, že první si můžeme všimnout, že otevřený text má více znaků jak klíč. Právě proto budeme při šifrování v tomto případě opakovat naše klíčové slovo, což způsobí, že ve výsledku náš klíč bude vypadat jako TEXTTE. Takže, první si najdeme v prvním řádku A, dále ve sloupci najdeme T a zašifrujeme dle průsečíku těchto písmen. V našem případě by to bylo písmeno T. Dále v řádku najdeme H a budeme hledat ve sloupci E, tudíž další písmeno bude L. A tak dále bychom pokračovali a dostali výsledný zašifrovaný text ve formě TLLCMI, dle našeho výše ukázaného Vigenerova čtverce. Kdybychom chtěli slovo dešifrovat, stačilo by akorát dle klíčového slova a průsečíku ve formě znaků šifrového textu. Dle umístění těchto znaků by nám stačilo použít první řádek Vigenerova čtverce a postupným procesem bychom se dopracovali k úspěšnému dešifrování. I přesto jak geniální tato šifra byla na tehdejší dobu, právě kvůli dlouhému procesu šifrování se od ní často utíkalo a začala se používat tehdy, kdy mono-alfabetické šifry byly známé jakožto nepoužitelné, za to se nedá Vigenerově šifře upřít, že byla pro tehdejší poměry opravdu neprolomitelná. [2][10]

### 2.1.3 Polygrafické substituční šifry

Další typ substitučních šifer jsou šifry polygrafické. Ty jsou charakteristické tím, že otevřený text rozdělují na skupiny písmen a poté šifrují po těchto skupinách znaků postupně. Výsledek, tedy zašifrovaný text pak dostává formy jednoho znaku, nebo jiné kombinaci vícero znaků. Tento typ šifer vzniká na principu jednoduchých substitučních šifer a vylepšuje se právě v počtu znaků, které šifrujeme zároveň. Pro dešifrování se pak proces šifrování zrcadlí. Různé polygrafické šifry se šifrují v jiných počtech znaků zároveň, avšak mezi nejznámější takové šifry můžeme řadit šifru Playfair, která je šifra digramová, to znamená že se šifrují vždy 2 znaky. [2][11]

#### *2.1.3.1 Playfair šifra*

I přesto, že název šifry je Playfair, jako jeho opravdového tvůrce, bychom spíše mohli označit Charlese Wheatstona. Baron Lyon Playfair byl přítelem Charlese Wheatstona a zároveň ten, kdo samotnou šifru proslavil a značně pomohl pro větší využití této šifry britskou armádou, i přesto že nebyl přímo jejím tvůrcem. Jedná se o digramovou polygrafickou substituční šifru, která seskupí abecedu do tabulky 5x5, to napovídá že jeden znak musí být vynechán. Vynechání se většinou praktikovalo pro písmena, která se málo používala, jako by mohlo být X a Q. Nejpopulárnější verzí je spojení písmen I a J jakožto jedno, kvůli své podobnosti. Při vytváření této tabulky se vybere slovo jakožto klíč a tím se vyplní daná tabulka jako první, kdy znaky v tabulce musí být unikátní. Poté se doplní zbytkem abecedy a slouží nám jak pro šifrování, tak dešifrování. Dále se otevřený text rozdělí po dvou znacích, kdy by dvojice neměly být stejnými znaky. Pokud tomu tak je, tak se přidává nějaký z méně používaných písmen, jak již výše uvedené X nebo Q a přidává se k této dvojici. Tyto znaky se taktéž přidávají, pokud je počet znaků lichý, protože potřebujeme znaky šifrovat po dvou. Pokud máme připravené dvojice, tak je šifrujeme dle následujících pravidel. [12][2]

Tabulka 1. Playfair pravidlo řádkové

| P | L | A | Y | F |
|---|---|---|---|---|
| I | R | E | X | M |
| B | C | D | G | H |
| K | N | O | Q | S |
| T | U | V | W | Z |

U tabulky 1. můžeme vidět první pravidlo, zeleně vidíme původní text a červeně nahrazený text z BG dostaneme CH. Pokud máme znaky naší dvojice na stejném řádku, každý tento znak nahradíme znakem, který se nachází o jednu pozici napravo.[12][2]

Tabulka 2. Playfair pravidlo sloupcové

| P | L | A | Y | F |
|---|---|---|---|---|
| I | R | E | X | M |
| B | C | D | G | H |
| K | N | O | Q | S |
| T | U | V | W | Z |

U tabulky 2. můžeme vidět druhé pravidlo, zeleně vidíme původní text a červeně nahrazený text z RN dostaneme CU. Jestliže je písmeno na konci řádku, posouvá se na začátek tohoto řádku. Dále pokud naše znaky máme ve stejném sloupci, tak ho nahradíme znakem ležícím o jedno níže, opět pokud jsme na konci sloupce, bereme písmeno na začátku sloupce.[12][2]

Tabulka 3. Playfair pravidlo křížové

| P | L | A | Y | F |
|---|---|---|---|---|
| I | R | E | X | M |
| B | C | D | G | H |
| K | N | O | Q | S |
| T | U | V | W | Z |

U tabulky 3. můžeme vidět třetí pravidlo, zeleně vidíme původní text a červeně nahrazený text z RS dostaneme NM. Pokud se naše znaky nachází na různých řádcích i sloupcích, tak se nahradí znakem, který leží na stejném řádku, jako ten náš jeden znak, ale na sloupci druhého znaku. To znamená, že vytvoříme pomyslný obdélník z těchto znaků a použijeme opačné rohy na tentýž řádku. Pro dešifrování probíhá stejný proces, akorát opačně. To znamená že se nám posouvají doleva v prvním případě, nahoru v druhém případě a na opačné rohy pomyslného obdélníku v třetím případě. [12][2]

Tento proces opakujeme pro každou dvojici a dostaneme požadovaný výsledek. Využití této šifry trvalo až do doby první a druhé světové války a přestala se používat právě kvůli vynálezu prvního počítače společně s většinou klasických šifer. [12][2]

### 2.1.4 Homofonní substituční šifry

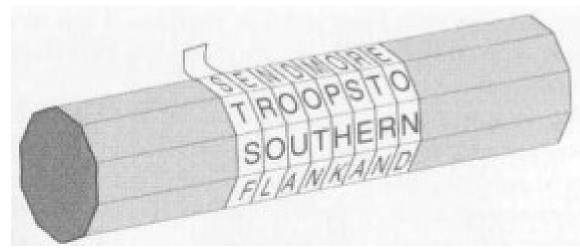
Jedná se o šifry, které vznikly jako reakce na problém, který kryptografy tehdy trápil. Monoalfabetické byly příliš snadné na prolomení, zatímco poly-alfabetické byly příliš složité na použití. Právě kvůli tomuto problému hledali zlatou střední cestu a mezi polygrafickými, vznikl také nápad na homofonní, kde hlavní myšlenka homofonní šifry je, že znak v otevřeném textu, bychom mohli nahradit jistou řadou znaků, které by se mohly měnit jak by se nám to hodilo. To znamená, že písmeno A bychom mohli nahradit například znaky B, 2, či otazníkem a podobné. Takže jeden znak otevřeného textu by měl například až 8 znaků v šifrovacím textu, které by ho mohly nahrazovat. Také jedna taková šifra u homofonního typu rozdělovala počet znaků, dle procent výskytu jistého znaku v psaném textu, a právě dle toho by se mohl určit počet nahrazovacích znaků. Tento krok měl zaručit, aby každý znak v textu měl stejné procento zastoupení v zašifrovacím textu. Toho se snaží dosáhnout tím, že se nahrazovací znaky náhodně střídají. Na základě tohoto kroku by se dalo tvrdit, že homofonní šifry dosáhly toho co měly, jelikož jsou opravdu daleko bezpečnější než mono-alfabetické a nejsou tak složité jako poly-alfabetické. Avšak nemůžeme tvrdit, že by byly nerozluštitelné, nebo že by byly bezpečnější jako poly-alfabetické. [2]

## 2.2 Transpoziční šifry

Jedná se o další typ šifer, které řadíme do klasické kryptografie. U tohoto typu šifer se znaky původní abecedy otevřeného textu nenahrazují, ale pouze přehazují pořadí znaků tak, aby v zašifrovaném textu byly pro třetí stranu nečitelné. Mezi první transpoziční šifry řadíme například Skytale, který používali Sparťané pomocí tyče a pruhu kůže nebo pergamenu. [2]

### 2.2.1 Skytale

Jedná se o velice starou šifru a také se jedná o první vojenské šifrovací zařízení, pravděpodobně také jednu z prvních transpozičních šifer. Princip této šifry je velice jednoduchý, kolem dřevěné tyče se omotá proužek kůže, či pergamenu. Poté odesílatel napíše otevřený text podél dřevěné tyče a pergamen či kůži odmotá. Samotný proužek nedává smysl a útočník potřebuje správný poloměr tyče, aby tuto zprávu přečetl. [1][2]




Obrázek 3. Ukázka principu Skytale [2]

### 2.2.2 Rail-fence

Jedná se o principově velice jednoduchou šifru, otevřený text rozdělíme po znacích které napíšeme do sloupců od vrchu dolů postupně po počtu řádků, kolik potřebujeme. Nejjednodušší by tedy byly 2 řádky. Poté můžeme text spojit do sebe jako kdybychom to četli normálně a dostaneme zašifrovaný text. Pro přehlednost se tato šifra často píše šikmo pod sebe a pak šikmo nad sebe, aby byla snadnější na pochopení a přečtení pro odesílatele. [13]

Tabulka 4. Ukázka Rail-fence

| D | E | K | M | M | H | Z | Y | E |
|---|---|---|---|---|---|---|---|---|
| N | S | A | A | E | E | K | D | N |

### 2.2.3 Transpozice s klíčem

Jedná se o další docela jednoduchou transpoziční šifru, kde opět seřadíme otevřený text jistým způsobem do tabulky, tentokrát normálně jak ho čteme, tedy zleva doprava, musíme si vybrat klíčové slovo, dle kterého budeme šifrovat. Musíme si ale uvědomit, že počet sloupců je dle velikosti klíčového slova. Tudíž kdybychom si vybrali klíčové slovo například TAJNY, museli bychom mít 5 sloupců. Dále napíšeme klíčové slovo nad naši nachystanou tabulku a seřadíme sloupce dle klíčového slova seřazené podle abecedy. V našem případě by se klíčové slovo seřadilo jako AJNTY, nesmíme však zapomenout seřadit náš klíč společně se sloupci, které pod něj spadají. Kdyby nám nevycházelo slovo do tabulky, můžeme samozřejmě písmena doplnit dle libosti. Pro větší bezpečnost se tento proces dělá ještě jednou, aby zvýšil bezpečnost naší šifry, avšak principově funguje stejně akorát s jiným klíčem. Příjemce pak musí znát klíč a celý zašifrovaný text, aby mohl text seřadit úplně stejně a přehodit sloupce dle klíčového slova, aby se dopracoval zpátky na otevřený text. U tabulek níže můžeme vidět proces zašifrování a přehozením sloupců. [13]

Tabulka 5. Ukázka otevřeného textu u Transpozice s klíčem

| T | A | J | N | Y |
| --- | --- | --- | --- | --- |
| D | N | E | S | K |
| A | M | A | M | E |
| P | E | K | N | Y |
| D | E | N | A | B |

Tabulka 6. Ukázka zašifrovaného textu u Transpozice s klíčem.

| A | J | N | T | Y |
| --- | --- | --- | --- | --- |
| N | E | S | D | K |
| M | A | M | A | E |
| E | K | N | P | Y |
| E | N | A | D | B |

### 2.2.4 Zubatka

Jedná se o transpoziční šifru, která se provádí opět pomocí tabulky. Princip funguje tak, že vybereme klíč, nejlépe dostatečně dlouhý pro náš otevřený text. Toto slovo nahradíme čísly, začínáme od 1 a počítáme dál, které rozřadíme dle pořadí písmen v abecedě. V případě opakujícího znaku přidáme hodnotě další číslo. Dále zapíšeme náš otevřený text do tabulky zleva doprava tak, aby počet sloupců které využijí odpovídal dle čísla hodnoty klíče v našem znaku, nakonec tabulku doplníme náhodnými znaky. Níže můžeme vidět ukázku Zubatky, kde používáme klíč STONOZKA a šifrujeme otevřený text DNESKA JE VENKU VELICE NADHERNE POCASI, které máme zvýrazněné zelenou barvou a klíč s číselnými hodnotami znaků zase barvou oranžovou.[14]

Tabulka 7. Ukázka Zubatky

| S | T | O | N | O | Z | K | A |
|---|---|---|---|---|---|---|---|
| 6 | 7 | 4 | 3 | 5 | 8 | 2 | 1 |
| D | N | E | S | K | A | J | E |
| V | E | N | K | U | V | E | J |
| L | I | C | E | N | A | O | K |
| D | H | E | C | R | N | M | P |
| E | P | N | V | O | C | A | R |
| A | S | Y | I | E | I | E | D |
| G | C | L | J | P | X | Z | T |
| F | S | L | T | C | B | G | V |

## 2.3 Hybridní šifry

Jedná se o další kategorií šifer, která je umístěna na stejné úrovni jako substituční a transpoziční šifry. Základní princip této kategorie je, že kombinuje prvky substitučních a transpozičních šifer do sebe. Mezi nejznámější hybridní šifru můžeme řadit například ADFGX společně s její modifikací ADFGVX. [2]

### 2.3.1 ADFGX, ADFGVX

Jedná se o takzvanou polní vojenskou šifru, byla používána německou armádou během první světové války. První vytvoříme tabulku znaků 5x5, kde v ADFGX se používají písmena abecedy. Ta se náhodně uspořádá a jako klíč tabulky nám budou sloužit samotná písmena ADFGX, které umístíme jakožto řádky a sloupce. Náš otevřený text zapíšeme pomocí těchto souřadnic, poté vybereme klíč ve formě řetězce znaků. Naše slovo zapsané pomocí souřadnic sepíšeme zleva doprava na délku našeho klíče a poté pokračujeme novým řádkem. Následně seřadíme sloupce dle klíče podle abecedy, také musíme u této šifry dávat pozor, aby se nám neopakovaly znaky. Poté sepíšeme na jeden řádek dle sloupců od vrchu dolů a rozdělíme po znacích po pěti. Druhá verze, tedy ADFGVX je rozšířená o jiné symboly a čísla, to znamená že tabulku zvětšíme na 6x6. Z toho vyplývá, že hlavní rozdíly mezi ADFGX a ADFGVX je, že se nám počet znaků, které můžeme používat zvýší z 25 na 36. Hlavní důvod použití znaků

ADFGVX je, že se tato šifra využívala na bojišti a nejčastěji se domlouvali skrze morseovku pomocí radiové komunikace. Takže se vybíraly znaky, které jsou srozumitelné i během špatného příjmu. [15][16]

Tabulka 8. Ukázka tabulky ADFGVX šifry

| | A | D | F | G | V | X |
|---|---|---|---|---|---|---|
| A | I | W | O | U | L | D |
| D | E | F | R | Y | 0 | A |
| F | 9 | B | C | 1 | G | H |
| G | J | K | 2 | M | N | 7 |
| V | P | 3 | Q | S | 6 | T |
| X | 4 | V | X | 5 | Z | 8 |

Po tom, co bychom si nachystali tabulku, tak je čas abychom náš otevřený text přepsali dle těchto souřadnic. Vybereme si třeba otevřený text DNESKA JE PEKNY DEN. Kdybychom postupovali dle tabulky 8 tak dostaneme řetězec znaků AX GV DA VG GD DX GA DA VA DA GD GV DG AX DA GV, ten pak seřadíme zleva doprava a postupně dolů pod náš klíč, což je slovo HYBRIDNI a dostaneme tabulku 9.

Tabulka 9. ADFGVX s klíčem HYBRIDNI před transpozicí

| H | Y | B | R | I | D | N |
|---|---|---|---|---|---|---|
| A | X | G | V | D | A | V |
| G | G | D | D | X | G | A |
| D | A | V | A | D | G | D |
| G | V | D | G | A | X | D |
| A | G | V | | | | |

Tabulka 10. ADFGVX s klíčem HYBRIDNI po transpozici

| B | D | H | I | N | R | Y |
|---|---|---|---|---|---|---|
| G | A | A | D | V | V | X |
| D | G | G | X | A | D | G |
| V | G | D | D | D | A | A |
| D | X | G | A | D | G | V |
| V | | A | | | | G |

Dále můžeme zapsat sloupce od vrchu dolů a rozdělit po 5 znacích. To znamená že dle tabulky 10. dostaneme zašifrovaný text GDVDV AGGXA GDGAD XDAVA DDVDA GXGAV G.

# 3 MODERNÍ KRYPTOGRAFIE

Vývoj kryptografie na moderní kryptologií můžeme určit převážně kvůli rozvoji počítačů, právě tento rozvoj vytvořil druhou éru kryptografie. Tato éra měla za příčinu nejen vytváření jednotlivých šifer, ale zdokonalování jejich existence a snížení chybovosti během provádění. Tento vynález taky umožňoval to, že se veškeré operace dělaly daleko snadněji a bez chyb.

Hlavně největším rozdílem oproti klasické kryptografií je ten, že se týká téměř všech lidí u jakékoliv komunikace, která je provedena přes internet. Dříve byla kryptografie používána pouze opravdovými specialisty, nadšenci nebo také měla svůj prostor ve vojenské komunikaci. I přesto že se tato skutečnost nemění, její užití se daleko více zvýšilo. Nemyslíme tím však, že běžní lidé používají kryptografií ke své komunikaci, ale spíš systémy počítačů které tito jedinci používají, různé typy softwarů nebo internetové stránky.

Vytvoření moderní kryptografie samozřejmě neznamená, že všichni běžní lidé o ni věděli hned s příchodem prvního počítače. Nějakou dobu trvalo, než si lidi mohli přečíst z veřejných akademických článků o reálných průlomech, avšak je opravdu důležité si uvědomit důležitost této vědy v našich současných životech.

Do této vědy spadá spousta kryptografických algoritmů, které používáme. Kdybychom však měli říct hlavní, tak bychom samozřejmě nemohli zapomenout moderní šifry, neboli šifrovací algoritmy. Ty dělíme na asymetrické a symetrické hlavně podle klíče. Často se v praxi používá spíše jejich kombinace. Dále můžeme například dělit již zmíněné symetrické šifry na blokové a proudové, dle toho jakým způsobem pracují s daty během šifrování. Můžeme sem také řadit hashovací funkce (v angličtině hash). Ty se často používají spíše jako otisk, tedy pro ověření pravosti souboru, někdy společně s digitálními podpisy. Do této vědy můžeme také řadit například generování klíčů pro jednotlivé šifrovací algoritmy a podobné. [7][3][17]

## 3.1 Symetrické šifrovací algoritmy

Symetrické šifrování je nejvíce charakteristické tím, že používá jeden klíč, také nazývaný jako soukromý klíč. Tento klíč používají pro zašifrování otevřeného textu do formy zašifrovaného textu a naopak. Tento typ šifer se pak dále dělí na blokové a proudové šifry.

Mezi nejznámější symetrické šifrovací algoritmy řadíme DES, její vylepšená verze 3DES, AES a také například IDEA. Symetrické šifry jsou docela rychlé a jednoduché na používání. Dnešní moderní symetrické šifry se považují za šifry bezpečné, avšak jejich nevýhoda

spočívá v tom, že existence pouze jednoho klíče je daleko rizikovější. Stejný klíč totiž používá jak odesílatel, tak příjemce. V případě že by klíč byl odhalen, vzniká obrovské riziko dešifrování útočníkem. [17][7]

### 3.1.1 Blokové symetrické šifrovací algoritmy

Jedná se o skupinu symetrických šifrovacích algoritmů, které rozdělí otevřený text na bloky bitů a šifruje každý tento blok zvlášť. [17]

#### *3.1.1.1 DES*

Jedná se o zkratku z anglického jazyka Data Encryption Standard (v češtině bychom mohli přeložit jako Standard pro Šifrování Dat), byla vytvořena v 70. letech IBM s pomocí NSA (National Security Agency). V současné době již není považována jako bezpečná kvůli nedostatečné velikosti klíče. Dnes se spíše používá její vylepšená verze 3DES.

Princip tohoto algoritmu je vlastně takzvaná Feistelova struktura provedena v 16 kolech uvnitř šifry, která využívá substituční a permutační síť nebo také S-boxy a P-boxy. Na začátku dostaneme blok otevřeného textu v 64 bitech a klíč v 64 bitech. Klíč se využívá z 56 bitů, proto je každý 8. bit vynechán. Dále pro každé kolo Feistelovy struktury je použitý podklíč, který má 48 bitů a je odvozen z původního klíče pro každé kolo tedy jiný. Díky využití této struktury nám je umožněno téměř stejný proces šifrování a dešifrování se změnou v obráceném plánování klíče. Poprvé otevřený text o velikosti 64 bitů vložíme do bloku, kde přehodíme pozice jednotlivých bitů. Poté začíná samotné kolo Feistelovy struktury. Uvnitř každého kola to funguje tak, že se blok rozdělí na půlku, tedy levou a pravou. Je jedno kterou nazýváme jako levá a pravá, hlavní v tomto bodě je že pracuje s jednou stranou a tu můžeme nazývat pravá. Dále tedy pravá strana se stane levou do dalšího kola a zároveň vstupuje do funkce, díky které zvětšíme velikost tohoto bloku z 32 bitů na 48 bitů. Následně provedeme operaci XOR s podklíčem, který je také o velikosti 48 bitů a dostaneme se do S-boxů. Specificky se jedná o 8 těchto S-boxů, proto musíme naši upravenou verzi bloku rozdělit po 6 bitech do každého z S-boxů. Na výstupu pak dle tabulky S-boxu dostaneme výstupné 4 bity od každého S-boxu. Tím dosáhneme jisté substituce, výstupy pak spojíme do 32 bitů které dostaneme do P-boxu, ten zase přehodí jednotlivé pořadí bitů. Ke konci tohoto bloku provedeme XOR s původním levým blokem a konečný výstup dostaneme do pravé strany bloku. Tento proces se pak provede po 16 kolech. Po výstupu z těchto kol v bloku přehodí pozici bitů z otevřeného textu tak, že přehodí levou a pravou stranu a ke konci opět vstoupí do

bloku, kde přehodíme pozice jednotlivých bitů. Nakonec vznikne samotný blok zašifrovaného textu o velikosti 64 bitů. I přesto že nemluvíme o nejlepší současné šifře, právě tato šifra tvořila velký základ pro velkou část současných modernějších šifer. [7][17]

#### *3.1.1.2 3DES*

Proces 3DES je stejný jako DES s rozdílem, že 3DES je akorát třikrát provedené šifrování DES. Jistě je více náročné, avšak bezpečnější a dá se považovat za bezpečné. I přes velkou náročnost původní DES, je její nová bezpečnější verze daleko náročnější. Je sice bezpečná a používaná, za to se nedá říct, že by se jednalo o zrovna výsledkově nejlepší šifru. [7]

#### *3.1.1.3 AES*

Po nějaké době se zdálo DES poněkud neefektivní a vznikala potřeba najít nového vhodného následovníka, který bude pod označením AES tedy Advanced Encryption Standard (v češtině také jako Pokročilý Šifrovací Standard). Právě proto NIST, neboli v angličtině National Institute of Standards and Technology of the United States ohlásil soutěž, ta měla za úkol najít šifru která by mohla nahradit tehdy používanou DES. Nakonec zvítězil algoritmus pod jménem Rijndael který byl vytvořen John Daemenem a Vincentem Rijmenem, kteří pocházeli z Belgie v roce 2000. Od té doby je AES velice hojně používána a do dnes nebyla nalezena žádná vážná bezpečnostní chyba. Avšak i přesto nemůžeme říct že Rijndael je stejné jako AES, jelikož AES je specifický standard. Takže i přesto že z Rijndael vychází, nejedná se o totéž věc. U AES jsou bloky bitů u otevřeného textu limitované na velikosti 128 bitů a u klíče se jedná o hodnoty 128, 192 nebo 256 bitů.

S těmito bloky bitů, můžeme pracovat jakožto 16 bytů. Stejně jako u DES tak i AES se skládá z několika kol, počet kol se zvětšuje dle velikosti klíče z 10 kol, na 12 až 14. Každé kolo obsahuje jednotlivé funkce které provádí. Konkrétněji se jedná o AddRoundKey, SubBytes, ShiftRows a MixColumn s tím že v posledním kole se MixColumn nahrazuje další AddRoundKey funkcí.

AddRoundKey – Vstupuje sem pole bytů z otevřeného textu ve velikosti 16 bytů a pole podklíče o stejné velikosti. To se každým kolem mění podobně jako u DES. Poté se provede XOR těchto dvou polí.

SubBytes – V tomto kroku naše pole bytů z otevřeného znaku se postupně, byte po bytu nahrazuje z předem vytvořené substituční tabulky, hlavním cílem této tabulky je, aby

substituce bytů nebyla stejná jako byte který nahrazuje, nebo aby se nejednalo o opačné hodnoty.

ShiftRows – Zde se přehází pozice bytů v poli tak, že první řádek se neposune vůbec, druhý se posune o krok doleva s tím že žádný z nich nepřeteče, ale objeví se na konci řádku. Další řádek se posouvá podobně o 2 místa a poslední o 3 místa.

MixColumn – Zde se každý sloupce v poli zamíchá pomocí invertibilní lineární transformace. Tu můžeme pro zjednodušení chápat jako násobení dvou matic dle sloupců, což způsobí přeházení pozic bytů v poli.

Nakonec již není třeba dělat MixColumn, jelikož tato funkce nám vlastně připravuje pole na další kolo a přidá se AddRoundKey, čímž dostáváme konečně náš zašifrovaný blok.[7][18]

#### *3.1.1.4 Blowfish*

Jedná se o další šifrovací algoritmus, byl vytvořen roku 1993 Brucem Schneierem a měl původně sloužit jako nová náhrada za DES. Největší slabina, kvůli které byla považována Blowfish jako nedostatečně bezpečná je kvůli velikosti bloků. Zde se bloky pohybují ve velikosti 64 bitů. I přesto nebylo do dnešního dne potvrzené žádné prolomení této šifry a k tomu všemu tuto slabinu nakonec vylepšili v jeho novější verzi známé jako Twofish.

Zde se pracuje s blokem bitů otevřeného textu ve velikosti 64 bitů, kde během šifrování se dělí na 2 půlky podobně jako u DES a klíčem od velikosti 32 bitů až po 448 bitů. Proces šifrování se skládá z vytváření podklíčů a samotného šifrování dat pomocí klíče. Celkem se prochází 16 kol a používá se 18 různých podklíčů. Ty jsou v procesu ve stejné velikosti jako blok, který šifrujeme. Taktéž zde pracujeme s Feistelovou strukturou, kterou jsme popisovali u šifry DES, zde však máme 4 S-boxy a ne 6. V našem P-boxu se nachází určitá prvočísla, která pak prochází operací XOR s těmito klíči. Následně je u P-boxu a jedné strany bloku provedena operace XOR. Výsledek je pak zaměněn do podoby, aby mohl vejít do S-boxů, tedy je výsledek rozdělen na 4 byty. Tyto S-boxy mezi sebou provedou operaci XOR dle specifického systému. Pak nadchází operace XOR se společným výsledkem těchto S-boxů a druhou stranou bloku. Dále tedy tento upravený blok vchází do levé strany a hodnota z P-boxu na začátku nahrazuje pravou stranu. Tento proces se opakuje 16krát. Na konci 16 kol se naše 2 klíče, které jsme doteď nevyužili použijí každý na jednu stranu a provede se s nimi opět XOR. Strany se spojí a dostaneme zašifrovaný blok o velikosti 64 bitů. Hlavní síla této šifry je nespočetné generování podklíčů, které využívá v každém kroku i po nich. [19]

### 3.1.2 Proudové symetrické šifrovací algoritmy

Jedná se o skupinu symetrických šifrovacích algoritmů, které nedělí otevřený text na pevné bloky, ale šifruje data po jednom bitu, či bytu. V rozdílu blokový a proudových šifer je třeba dodat, že proudové jsou známé jakožto více náchylné na útoky. Může to být i tím, že nejsou standardizované jak blokové, kde AES a DES jsou standardy šifrování. [17][7]

#### *3.1.2.1 ChaCha20*

Symetrická proudová šifra, vytvořena Daniel J. Bernsteinem v roce 2008, založena na předešlé šifře Salsa20. V kombinaci s Poly1305 doporučena pro TLS, anglicky Transport Layer Security, následovník možná známější SSL, neboli Secure Sockets Layer. Dále se využívá u HTTPS připojení, šifrování souborů nebo je také hojně využíván u VPN a chatovacích aplikací. Největší výhodou je rychlost, jelikož patří mezi nejrychlejší šifrovací algoritmy a jednoduchost při její implementaci. Používá klíč o velikosti 256 bitů (32 bytů) společně s nonce, což má 96 bitů (12 bytů) ke generování pseudonáhodného proudu klíčů. Ten společně s otevřeným textem prochází operací XOR. Při dešifrování opět vytvoří další pseudonáhodný proud klíčů a prochází se zašifrovaným textem operací XOR. Nonce je většinou náhodně vygenerovaný a dle kombinace s klíčem uvádí šifrovací algoritmus do daného stavu, pro generování pseudonáhodného proudu klíčů. [26][23]

## 3.2 Asymetrické šifrovací algoritmy

Jedná se o šifrovací algoritmy které využívají dva typy klíčů, soukromý a veřejný. Tedy jeden pro zašifrování a jeden pro dešifrování, nejčastěji veřejný klíč pro zašifrování a soukromý pro dešifrování. Veřejný klíč může použít kdokoliv k zašifrování otevřeného textu, avšak soukromý je známý jen tomu, kdo by měl zašifrovaný text dešifrovat. Lze je využívat nejen k šifrování, ale také k vytvoření digitálního podpisu, ten slouží pro ověření autenticity a integrity souboru, otevřeného textu nebo i uživatele. [17]

### 3.2.1 RSA

Jedná se o asymetrický algoritmus, jehož využití není jen co se týče šifrování, ale také co se týče digitálního podpisu. Poprvé vydán v roce 1977 Ronem Rivestem, Adi Shamirem a Leonardem Adlemanem z MIT. Díky RSA bylo možné využívat soukromý a veřejný klíč na šifrování a dešifrování. To způsobuje, že je možné použít jak soukromý tak veřejný klíč k šifrování a druhý by pak zašifrovaný text dešifroval. Je využíván v protokolech jako

například SSH, OpenPGP, S/MIME a SSL/TLS. Klíč generuje násobením dvou prvočísel p a q, kde dostaneme modul tedy n. Ten pak používáme jak v soukromém, tak veřejném klíči. Dále pomocí Eulerovy funkce vypočítáme hodnotu pro výběr veřejného exponentu, tedy $\phi(n) = (p - 1) \times (q - 1)$. Dle této hodnoty vybere exponent e, který musí být vzájemně prvočíselný s hodnotou Eulerovy funkce tedy ϕ(n). Soukromý exponent je pak inverzní veřejnému s modulo ϕ(n), takže $d = e^{-1}\ mod\ \phi(n)$. Šifrování je pak jednodušší, M reprezentuje převedenou zprávu na číslo, e je tedy veřejný exponent, n je modul a C zašifrovaná zpráva, $C = M^e\ mod\ n$. Dešifrování pak vypadá následovně, $M = C^d\ mod\ n$. Zde je M původní zpráva v číselné formě, C je opět zašifrovaný text v číselné formě, d je hodnota soukromého exponentu a n je modul. [27][22]

### 3.2.2 DSA

Jedná se o asymetrický algoritmus pro digitální podpis, byl vyvinutý americkými federálními orgány společně s NIST v roce 1991. Využívá se k ověření autenticity a integrity zprávy.

Klíče generujeme tak, že vybereme náhodné prvočíslo q a p kde p = kq + 1, zde je k konstanta. Ta zaručuje aby byly q a p dostatečně dlouhé a bezpečné, aby zaručily bezpečnost celého algoritmu. Dále se musí vybrat generátor grupy g, kde typicky platí $g = h^{\frac{p-1}{q}}\ mod\ p$. Zde je q a p náhodná prvočísla zmíněná výše a h je náhodné celé číslo v intervalu $(1, p - 1)$. Dále je náhodný privátní klíč náhodné číslo x v intervalu $(1, q - 1)$ a výpočet veřejného klíče je pak $y = g^x mod\ p$. Soukromý klíč obsahuje parametry p, g, q a x, podobně jako veřejný, ten obsahuje p, g, q a y. Vytvoření podpisu pak funguje tak, že vybereme náhodné číslo k z intervalu $(1, q - 1)$. Vypočítáme hodnotu, kde $r = (g^k\ mod\ p)\ mod\ q$, pomocí hodnot ze soukromého klíče a konstanty k. Nakonec je vytvoření samotného podpisu pomocí hodnoty s, kde $s = [k^{-1}(H(m) + xr)]\ mod\ q$. Zde se využívá již výše zmíněných hodnot, avšak H(m) je zde hash zprávy m vytvořený díky danému hashovacímu algoritmu a x jako hodnota soukromého klíče. Podpis se pak skládá z hodnoty H(m), s a r

U ověření podpisu je třeba vypočítat pár pomocných hodnot, tedy první $w = s^{-1}\ mod\ q$. Dále hodnotu $u_1 = (H(m) \times w)\ mod\ q$. Hodnotu $u_2 = (r \times w)\ mod\ q$, nakonec vypočítáme hodnotu v pro ověření $v = [(g^u \times y^u)\ mod\ p]\ mod\ q$. Hodnotu v porovnáme s hodnotou r a pokud $v = r$, pak je podpis ověřen. [28][29]

## 3.3 Hashovací algoritmy

Funkce hashovacích algoritmů je zpracování dat o jakékoliv délce do podoby zprávy o pevně dané délce. Vytváří tak jistou reprezentaci zprávy, či souboru z kterého vychází. [24]

### 3.3.1 MD5

MD5, neboli anglicky message-digest algorithm. Byl vyvinutý v roce 1991 Ronaldem Rivestem, vytvořen jako následovník MD4. Ten z jakéhokoliv vstupu vytváří 128bitový otisk. První rozdělí data na 512bitové bloky a inicializuje interní stav programu. Bloky jsou rozděleny na 16 slov, tedy po 32 bitech s kterými pak algoritmus jednotlivě pracuje. Ty jsou zpracovány různými matematickými operacemi jako je bitový posun, XOR, modulární aritmetika a nelineární transformace. Výsledek je pak prezentován jako řetězec znaků. Musíme však říct, že dneska je MD5 považovaný za zastaralý a není doporučován používat. Je tomu tak kvůli objevu kolizních výstupů, to znamená že ze dvou různých vstupů vytvořil jeden stejný otisk. [30]

### 3.3.2 SHA-1

SHA-1, neboli anglicky Secure Hash Algorithm 1. Byl vyvinutý v roce 1993 NIST. Přetváří jakýkoliv vstup na 160bitový otisk. Podobně jako MD5 vstupy dělí na 512bitové bloky, které dále rozebírá na bloky po 16 slovech, tedy 32 bitů. Ty nadále upravuje pomocí bitových rotací, bitových operací jako například XOR a bitového posunu. Ty jsou aplikovány iterativně až dostane otisk jako řetězec znaků. V dnešní době již není doporučený k použití, jelikož je zastaralý a také kvůli objevené možnosti kolizních výstupů. [31][33]

### 3.3.3 SHA-256

SHA-256, neboli Secure Hash Algorithm 256 z rodiny SHA-2. Byl vyvinutý v roce 2001 díky spolupráci NSA a NIST. Ten vytváří ze vstupu pevný 256bitový otisk. Rozděluje samotný vstup na bloky opět o velikosti 512 bitů. Pak s podobným principem je zpracován jako u SHA-1, kdy SHA-256 má více iterací a rozmanitější sadu logických operací které využívá. Jako například různé bitové operace jako XOR, OR a NOT, bitové posuny, bitové funkce jako je bitová suma a zrcadlení nebo také modulární aritmetiku. SHA-256 je hojně využívána v současné době a taky doporučena pro použití. [32][33]

### 3.3.4 SHA-3

SHA-3, neboli anglicky Secure Hash Algorithm 3. Byl vítězem soutěže, která byla zahájena v roce 2007, vítězný algoritmus byl zveřejněný v roce 2015. Byl vyvinutý belgickými vědci Guidem Bertoni, Joan Daemenem, Michael Peuletem a Vincentem Rijmenem pod názvem Keccak. Otisk bývá v délkách 224, 256, 384 nebo 512 bitů dle typu algoritmu. U verze 256 využívá permutaci a absorpční fázi. Zde absorbuje blok do interního stavu algoritmu, tam prochází různými fázemi stavů. Algoritmus má stavy absorpční a ždímací, těmi jednotlivé bloky probíhají až naberou finální fázi. Tento algoritmus patří mezi nejnovější a nejvíce pokročilé hashovací algoritmy vůbec. [34]

# II. PRAKTICKÁ ČÁST

# 4 SPECIFIKACE A IMPLEMENTACE ALGORITMŮ

Pro vytvoření je potřeba specifikovat jednotlivé kryptografické algoritmy, které budou využity pro implementaci samostatné aplikace. Mezi nejznámější řadíme AES, jedná se o blokovou symetrickou šifru, jejichž klíče se můžou objevovat ve velikosti 128, 192, či 256 bitů.

## 4.1 AES

AES, by se dal považovat za nejen neznámější, ale taktéž nejpoužívanější současný šifrovací algoritmus, který se stal dnešním standardem. Pro šifrování je využito IV, neboli Initialization Vector, který je používán jako náhodná proměnná. Ta přináší do celého procesu větší náhodnost. IV je taktéž využíván při použití GCM režimu při zašifrování. GCM, neboli Galois/Counter Mode, jedná se o režim který se využívá u blokových šifer, přidává celému procesu prvek bezpečnosti navíc a autentifikaci při šifrování. Pomocí tohoto se získá Tag, neboli autentifikační proměnná která plní funkci ověřování integrity při dešifrování. [20][17]

```
def encrypt_aes(message, key, iv=None):
    # If IV is not provided or its length is not within the required range, generate a new IV
    if iv is None or len(iv) != 12:
        iv = os.urandom(12)

    # Encrypt the message
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(message) + encryptor.finalize()

    # Get the authentication tag
    tag = encryptor.tag

    return (iv, ciphertext, tag)

def decrypt_aes(iv, ciphertext, tag, key):
    #Just decryption part
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag), backend=default_backend())
    decryptor = cipher.decryptor()
    message = decryptor.update(ciphertext) + decryptor.finalize()

    return message
```

Obrázek 4. Ukázka implementace funkcí encrypt_aes a decrypt_aes

V obrázku číslo 4 je prezentována ukázka implementace funkcí pro šifrování a dešifrování pomocí AES s režimem GCM. V funkci encrypt_aes je zpracováván otevřený text v podobě zprávy, klíč a volitelně i inicializační vektor IV. Na začátku funkce se provádí kontrola velikosti vstupního IV, pokud není v požadovaném rozsahu, je vygenerován nový. Pomocí balíčku knihoven cryptography je realizován samotný šifrovací algoritmus, který vytváří zašifrovaný text pod ciphertext, autentifikační tag jako Tag a nový IV. Funkce decrypt_aes

pracuje s parametry iv, ciphertext, tag a key pro dešifrování dat. Na výstupu poskytuje otevřený text ve formě zprávy.

```
def encrypt_file_aes(file_path, key, iv = None):
    #open file
    with open(file_path, "rb") as file:
        plaintext = file.read()
    #encrypt and get output values
    iv, ciphertext, tag = encrypt_aes(plaintext, key, iv)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    #Add .enc
    encrypted_file_path = file_path + b".enc"

    # Save the IV, ciphertext, tag, file extension length, and file extension to a file
    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(iv + tag + file_extension_length + file_extension + ciphertext)
```

Obrázek 5. Ukázka implementace funkce encrypt_file_aes

Na obrázku číslo 5 je zobrazena šifrovací funkce AES pro soubory. Ve funkci encrypt_file_aes jsou definovány vstupní hodnoty jako file_path, který identifikuje cestu k souboru který má být zašifrován, klíč a iv jako inicializační vektor. Nejprve je soubor otevřen pomocí funkce open s režimem rb pro čtení souboru v bytech. Obsah je uložen do proměnné plaintext. Následně je použita funkce encrypt_aes pro provedení šifrování, jak bylo popsáno v obrázku číslo 4 a původní přípona souboru z file_path je zachována. Dále je uložena délka přípony a k zašifrovanému souboru je přidána přípona .enc. Nakonec je opět použita funkce open s režimem wb pro uložení souboru v bytech. Soubor obsahuje iv, tag, délku přípony, samotnou příponu a zašifrovaný obsah a je uložen do původní složky.

```python
def decrypt_file_aes(file_path, key):
    #open file
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract the IV, tag, file extension length, and file extension from the encrypted data
    iv = encrypted_data[:12]  # IV size is 12 bytes
    tag = encrypted_data[12:28]  # Tag size is 16 bytes
    file_extension_length = int.from_bytes(encrypted_data[28:32], 'big')  # Decode file extension length
    file_extension = encrypted_data[32:32+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[32+file_extension_length:]  # Extract ciphertext

    # Decrypt and get plaintext back
    plaintext = decrypt_aes(iv, ciphertext, tag, key)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 6. Ukázka implementace funkce decrypt_file_aes

V obrázku číslo 6 je zobrazena funkce pro dešifrování AES pro soubory. Funkce decrypt_file_aes má vstupní hodnoty file_path, což je cesta k zašifrovanému souboru a klíč ve formě key. Nejprve je soubor načten pomocí funkce open a jeho obsah je uložen do proměnné. Poté je zašifrovaný soubor rozdělen do jednotlivých proměnných podle bytů. První částí je inicializační vektor IV, který obsahuje prvních 12 bytů souboru. Druhou částí je autentifikační tag, který zabírá dalších 16 bytů. Třetí částí je velikost přípony souboru, která je reprezentována 4 byty. Následuje samotná přípona souboru, jejíž délka odpovídá velikosti přípony. Zbytek souboru pak představuje zašifrovaný obsah, který spolu s ostatními proměnnými slouží k dešifrování pomocí funkce decrypt_aes. Poté je odstraněna přípona .enc a přidána původní přípona souboru. V posledním kroku je původní soubor uložen jako nový soubor.

## 4.2 DES, 3DES

DES a 3DES, jedná se o předchůdce a následovníka i přesto že byl nahrazen AES. Jedná se o nedílnou součást historie šifrovacích algoritmů a stále jedna z největších blokových symetrických šifrovacích algoritmů. Pro šifrování je využíván CBC režim, jedná se o jeden z režimů pro blokové šifry. Každý blok otevřeného textu je pak kombinován s předchozím blokem zašifrovaného textu pomocí operace XOR, první blok pak pracuje s IV místo zašifrovaného bloku a pak se celý proces řetězí. [7][21]

```
def encrypt_des(message, key, iv):
    if iv is None or len(iv) != 8:
        iv = os.urandom(8)

    # Ensure the key is 8 bytes (DES block size)
    key = key[:8].ljust(8, b'\0')

    # Pad the message
    padded_message = pad(message, 8)

    # Create a DES cipher object
    cipher = DES.new(key, DES.MODE_CBC, iv = iv)

    # Encrypt the message
    ciphertext = cipher.encrypt(padded_message)

    return iv, ciphertext

def decrypt_des(ciphertext, key, iv):
    # Ensure the key is 8 bytes (DES block size)
    key = key[:8].ljust(8, b'\0')

    # Create a DES cipher object
    cipher = DES.new(key, DES.MODE_CBC, iv=iv)

    # Decrypt the ciphertext
    decrypted_message = cipher.decrypt(ciphertext)

    # Unpad the decrypted message using PKCS7 padding
    unpadded_message = unpad(decrypted_message, 8)

    return unpadded_message
```

Obrázek 7. Ukázka implementace encrypt_des a decrypt_des

V obrázku číslo 7 je zobrazena funkce encrypt_des a decrypt_des. Ve funkci encrypt_des jsou vstupními proměnnými message, představující otevřený text k šifrování, key jako klíč a IV. Funkce na počátku ověřuje, zda má inicializační vektor 8 bytů. V případě nedostatečné délky je generován nový. Algoritmus se ujišťuje, že vstupní klíč má přesně 8 bytů, v případě delšího klíče jsou akceptovány pouze prvních 8 bytů. Tato úprava usnadňuje práci uživatele s funkcí DES. Funkce dále využívá padding z knihovny Pycryptodome pro doplnění dat na konec zprávy, což je nezbytné pro správnou funkci šifrovacího algoritmu. Následně je provedeno šifrování a vráceny jsou inicializační vektor IV a zašifrovaný text jako ciphertext.

Funkce decrypt_des přijímá stejná vstupní data a pracuje s nimi podobně, avšak provádí dešifrování a následně odstraňuje vyplňovací data pomocí funkce unpad. Nakonec je vrácen původní otevřený text.

```
def encrypt_file_des(file_path, key, iv = None):
    with open(file_path, "rb") as file:
        plaintext = file.read()

    iv, ciphertext = encrypt_des(plaintext, key, iv)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    # Save the IV, ciphertext, file extension length, and file extension to a file
    encrypted_file_path = file_path + b".enc"

    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(iv + file_extension_length + file_extension + ciphertext)
```

Obrázek 8. Ukázka implementace funkce encrypt_file_des

V obrázku číslo 8 je zobrazena funkce encrypt_file_des pro šifrování souborů pomocí DES, fungující podobně jako AES. Funkce přijímá cestu k souboru, který má být zašifrován file_path, klíč key a inicializační vektor IV. Nejprve je soubor otevřen a jeho obsah je uložen do proměnné. Tato proměnná je poté využita společně s dalšími vstupy k provedení funkce encrypt_des, jak bylo vysvětleno v obrázku číslo 7. Stejně jako u AES je získána přípona souboru a jeho délka, po odstranění původní přípony je přidána nová s příponou .enc. Nakonec je celá funkce uložena pod novým souborem, přičemž na začátek je přidán inicializační vektor, následuje délka přípony, samotná přípona a zašifrovaný obsah souboru.

```
def decrypt_file_des(file_path, key):
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract the IV, file extension length, and file extension from the encrypted data
    iv = encrypted_data[:8]  # IV size is 8 bytes
    file_extension_length = int.from_bytes(encrypted_data[8:12], 'big')  # Decode file extension length
    file_extension = encrypted_data[12:12+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[12+file_extension_length:]  # Extract ciphertext

    # Decrypt the file contents
    plaintext = decrypt_des(ciphertext, key, iv)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 9. Ukázka implementace funkce decrypt_file_des

Na obrázku číslo 9 je zobrazena funkce decrypt_file_des pro dešifrování souborů pomocí DES, ta je opět velmi podobná verzi pro AES. Úprava spočívá pouze ve velikosti jednotlivých částí zašifrovaného souboru. Na vstupu jsou přijímány proměnné file_path jako cesta k souboru, který má být dešifrován a klíč pod key. Funkce načte obsah souboru do proměnné a rozdělí ho na jednotlivé části podle bytů, jak byly zašifrovány. Prvních 8 bytů je určeno pro inicializační vektor IV, následují 4 byty pro velikost přípony souboru, která je poté použita jako samotná přípona souboru a zbytek bytů je převzat jako zašifrovaný obsah. Tato funkce využívá decrypt_des a tyto proměnné k provedení dešifrování souboru, následně upraví příponu a uloží nový dešifrovaný soubor.

```
def encrypt_3des(message, key, iv):
    if iv is None or len(iv) != 8:
        iv = os.urandom(8)
    # Ensure the key is 24 bytes (3 blocks for Triple DES)
    key = key[:24].ljust(24, b'\0')

    # Pad the message
    padded_message = pad(message, 8)

    # Create a 3DES cipher object
    cipher = DES3.new(key, DES3.MODE_CBC, iv = iv)

    # Encrypt the message
    ciphertext = cipher.encrypt(padded_message)

    return iv, ciphertext

def decrypt_3des(ciphertext, key, iv):
    # Ensure the key is 24 bytes (3 blocks for Triple DES)
    key = key[:24].ljust(24, b'\0')

    # Create a 3DES cipher object
    cipher = DES3.new(key, DES3.MODE_CBC, iv = iv)

    # Decrypt the ciphertext
    decrypted_message = cipher.decrypt(ciphertext)

    # Unpad the decrypted message
    unpadded_message = unpad(decrypted_message, 8)

    return unpadded_message
```

Obrázek 10. Ukázka implementace encrypt_3des a decrypt_3des

Na obrázku číslo 10 jsou zobrazeny funkce encrypt_3des a decrypt_3des, které fungují podobně jako předchozí funkce encrypt_des a decrypt_des. Hlavním rozdílem je velikost klíče, který je nyní 24 bytů, což odpovídá třem DES klíčům. To je z důvodu použití 3DES algoritmu. Tyto funkce využívají metodu DES3 místo DES z knihovny Pycryptodome a režim šifrování je CBC.

```
def encrypt_file_3des(file_path, key, iv = None):
    with open(file_path, "rb") as file:
        plaintext = file.read()

    iv, ciphertext = encrypt_3des(plaintext, key, iv)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    # Save the IV, ciphertext, file extension length, and file extension to a file
    encrypted_file_path = file_path + b".enc"

    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(iv + file_extension_length + file_extension + ciphertext)

def decrypt_file_3des(file_path, key):
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract the IV, file extension length, and file extension from the encrypted data
    iv = encrypted_data[:8]  # IV size is 8 bytes
    file_extension_length = int.from_bytes(encrypted_data[8:12], 'big')  # Decode file extension length
    file_extension = encrypted_data[12:12+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[12+file_extension_length:]  # Extract ciphertext

    # Decrypt the file contents
    plaintext = decrypt_3des(ciphertext, key, iv)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 11. Ukázka implementace funkcí encrypt_file_3des a decrypt_file_3des

Na obrázku číslo 11 jsou zobrazeny funkce encrypt_file_3des a decrypt_file_3des, které fungují velmi podobně jako funkce encrypt_file_des a decrypt_file_des popsané u obrázků číslo 8 a 9, s rozdílem v použití jiných funkcí pro šifrování a dešifrování. Taktéž si lze všimnout, že velikost klíče neovlivňuje implementaci těchto funkcí.

## 4.3 Blowfish

Ohledně Blowfish, jedná se o zajímavou blokovou symetrickou šifru, která je stále používána kvůli systémům, které jsou na ni závislé. Zajímavostí této šifry jsou velikosti klíčů, jelikož lze používat klíče o velikost od 32 bitů až po 448 bitů. Podobně jako u 3DES a DES pracuje s režimem CBC, který byl vysvětlen v části DES, 3DES. [19]

```python
def encrypt_BF(message, key, iv):
    if iv is None or len(iv) != 8:
        iv = os.urandom(8)
    # Ensure the key is either 16, 24, or 56 bytes
    key = key[:56].ljust(56, b'\0')

    # Pad the message
    padded_message = pad(message, Blowfish.block_size)

    # Create a Blowfish cipher object
    cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv = iv)

    # Encrypt the message
    ciphertext = cipher.encrypt(padded_message)

    return iv, ciphertext

def decrypt_BF(ciphertext, key, iv):
    # Ensure the key is either 16, 24, or 56 bytes
    key = key[:56].ljust(56, b'\0')

    # Create a Blowfish cipher object
    cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv = iv)

    # Decrypt the ciphertext
    decrypted_message = cipher.decrypt(ciphertext)

    # Unpad the decrypted message
    unpadded_message = unpad(decrypted_message, Blowfish.block_size)

    return unpadded_message
```

Obrázek 12. Ukázka implementace funkcí encrypt_BF a decrypt_BF

Na obrázku číslo 12 jsou zobrazeny funkce encrypt_BF a decrypt_BF, které jsou implementovány velmi podobně jako funkce encrypt_des a decrypt_des. Oba algoritmy pracují s otevřeným textem pod message, klíčem key a inicializačním vektorem IV. Hlavní rozdíl spočívá v maximální velikosti klíče, která u algoritmu Blowfish činí 56 bytů. Dále jsou zde použity funkce pad a unpad, které pracují s velikostí bloku odpovídající velikosti algoritmu Blowfish, tudíž 8 bytů. Tato proměnná je zde použita pro jistotu. Samotné funkce pro šifrování a dešifrování encrypt_BF a decrypt_BF využívají algoritmus Blowfish z knihovny Pycryptodome.

```
def encrypt_file_bf(file_path, key, iv = None):
    with open(file_path, "rb") as file:
        plaintext = file.read()

    iv, ciphertext = encrypt_BF(plaintext, key, iv)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    # Save the IV, ciphertext, file extension length, and file extension to a file
    encrypted_file_path = file_path + b".enc"

    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(iv + file_extension_length + file_extension + ciphertext)

def decrypt_file_bf(file_path, key):
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract the IV, file extension length, and file extension from the encrypted data
    iv = encrypted_data[:8]  # IV size is 8 bytes
    file_extension_length = int.from_bytes(encrypted_data[8:12], 'big')  # Decode file extension length
    file_extension = encrypted_data[12:12+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[12+file_extension_length:]  # Extract ciphertext

    # Decrypt the file contents
    plaintext = decrypt_BF(ciphertext, key, iv)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 13. Ukázka implementace funkcí encrypt_file_bf a decrypt_file_bf

Na obrázku číslo 13 jsou zobrazeny funkce pro šifrování a dešifrování souborů pomocí Blowfish. Funkce encrypt_file_bf přijímá cestu k souboru, který má být zašifrován jako file_path, a klíč key. Nejprve je otevřen požadovaný soubor podle zadané cesty a jeho obsah je načten do proměnné. Následně je zavolána funkce encrypt_BF k provedení šifrování dat. Poté jsou získány přípona a délka přípony původního souboru. Původní přípona je nahrazena příponou .enc a zašifrovaný soubor je uložen do stejné cesty. Inicializační vektor IV, délka přípony, přípona souboru a zašifrovaný obsah jsou skládány do nového souboru.

Soubor dešifruje velice podobně, načte soubor dle cesty, rozdělí dle bytů na proměnné. Poté dešifruje a seskládá přípony dle proměnných a uloží jako nový soubor, již dešifrovaný.

## 4.4 RSA

RSA, jedná se o velice populární asymetrickou šifru a nachází využití nejen co se týče šifrování a dešifrování, ale také například co se týká digitálního podpisu. Pro šifrování jsou použity klíče v PEM formátu. Je zde taktéž využito OAEP, neboli Optimal Asymmetric Encryption Padding, to zajišťuje zvýšení bezpečnosti a náhodnou výplň do zašifrovaného textu.

Dále je zde využito MGF1, neboli Mask Generation Function 1, ta zde plní podobnou funkci. Vygeneruje náhodná data jakožto masku což znamená, že tyto data budou vložena na začátek zašifrovaného textu. OAEP se pak postará o zbytek společně s hashovacím algoritmem SHA-256, který je zde využíván jako část MGF a také výplni náhodných dat uvnitř zašifrovaného textu. Kombinace OAEP s RSA snižuje velikost souboru, která je možná zašifrovat kvůli výplni, ta je však převážně závislá na velikosti klíčů. Jelikož mé klíče jsou o velikosti 2048 bitů, tak maximální velikost je něco kolem 200 bytů. [7][22]

```
def encrypt_RSA(message_bytes, public_key_pem):
    try:
        public_key = serialization.load_pem_public_key(public_key_pem, backend=default_backend())

        # Encrypt the message
        ciphertext = public_key.encrypt(
            message_bytes,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        return ciphertext  # Return bytes instead of a string
    except Exception as e:
        return str(e)

def decrypt_RSA(ciphertext, private_key_pem):
    try:
        private_key = serialization.load_pem_private_key(private_key_pem, password=None, backend=default_backend())

        # Decrypt the ciphertext
        plaintext = private_key.decrypt(
            ciphertext,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        return plaintext  # Decode bytes to string
    except Exception as e:
        return str(e)
```

Obrázek 14. Ukázka implementace funkcí encrypt_RSA a decrypt_RSA

V Na obrázku číslo 14 jsou zobrazeny funkce encrypt_RSA a decrypt_RSA. Funkce encrypt_RSA pracuje s otevřeným textem a veřejným klíčem, oba v bytech. Jelikož se jedná o šifrování, pracuje s veřejným klíčem uloženým v PEM formátu pod názvem public_key_pem. Tento klíč je načten pomocí serializace a funkce load_pem_public_key. Následně je veřejný klíč použit k zašifrování otevřeného textu s využitím parametrů MGF1 a OAEP společně s hashovacím algoritmem SHA-256. Funkce pak vrací výsledek jako zašifrovaný text.

Při dešifrování ve funkci decrypt_RSA je pracováno se zašifrovaným vstupem v bytech pod názvem ciphertext a soukromým klíčem v PEM formátu pod názvem private_key_pem. Tento klíč je také načten pomocí serializace a funkce load_pem_private_key. Následně je

provedeno dešifrování s využitím stejných parametrů jako při šifrování a funkce vrací dešifrovaný otevřený text.

```
# encrypt file with RSA
def encrypt_file_RSA(file_path, public_key):
    with open(file_path, "rb") as file:
        plaintext = file.read()

    ciphertext = encrypt_RSA(plaintext, public_key)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    # Save the ciphertext, file extension length, and file extension to a file
    encrypted_file_path = file_path + b".enc"

    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(file_extension_length + file_extension + ciphertext)

# decrypt file with RSA
def decrypt_file_RSA(file_path, private_key):
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract file extension length, and file extension from the encrypted data
    file_extension_length = int.from_bytes(encrypted_data[:4], 'big')  # Decode file extension length
    file_extension = encrypted_data[4:4+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[4+file_extension_length:]  # Extract ciphertext

    # Decrypt the file contents
    plaintext = decrypt_RSA(ciphertext, private_key)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 15. Ukázka implementace funkcí encrypt_file_RSA a decrypt_file_RSA

Na obrázku číslo 15 jsou zobrazeny funkce encrypt_file_RSA a decrypt_file_RSA. Funkce encrypt_file_RSA přijímá vstupní argumenty file_path pro cestu k souboru, který má být šifrován a public_key jako klíč pro šifrování. Nejprve je načten cílový soubor do proměnné podle zadané cesty. Tento soubor je následně zašifrován pomocí poskytnutého veřejného klíče. Dále je získána přípona původního souboru a jeho velikost. Původní přípona je nahrazena příponou .enc a zašifrovaný soubor je uložen do stejné složky.

Funkce decrypt_file_RSA poté načte zašifrovaný soubor podle zadané cesty a použije soukromý klíč z private_key. Obsah souboru je rozdělen na proměnné podle bytů a následně je provedeno dešifrování. Nakonec je upravena přípona souboru stejným způsobem jako u funkce encrypt_file_RSA a dešifrovaný soubor je uložen na stejném umístění.

## 4.5 ChaCha20

ChaCha20, jedná se o celkem populární šifrovací algoritmus, který řadíme mezi symetrické proudové šifrovací algoritmy, což je také důvod jeho výběru. V implementaci je využito nonce, což znamená number used once. To společně s klíčem a otevřeným textem vstupuje do algoritmu a šifruje je v zašifrovaný text. [23]

```
def encrypt_chacha20(message, key, nonce):
    # Ensure the key is 256 bits (32 bytes)
    key = key[:32].ljust(32, b'\0')

    nonce = nonce[:16].ljust(16, b'\0')
    # Create a ChaCha20 cipher object
    cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None, backend=default_backend())
    encryptor = cipher.encryptor()

    # Encrypt the message
    ciphertext = encryptor.update(message) + encryptor.finalize()

    return ciphertext

def decrypt_chacha20(ciphertext, key, nonce):
    # Ensure the key is 256 bits (32 bytes)
    key = key[:32].ljust(32, b'\0')

    # Ensure the nonce is 128 bits (16 bytes)
    nonce = nonce[:16].ljust(16, b'\0')

    # Create a ChaCha20 cipher object
    cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None, backend=default_backend())
    decryptor = cipher.decryptor()

    # Decrypt the ciphertext
    decrypted_message = decryptor.update(ciphertext) + decryptor.finalize()

    return decrypted_message
```

Obrázek 16. Ukázka implementace funkcí encrypt_chacha20 a decrypt_chacha20

Na obrázku číslo 16 jsou zobrazeny funkce encrypt_chacha20 a decrypt_chacha20. Funkce encrypt_chacha20 přijímá vstupní proměnné otevřený text jako message, klíč key a nonce. Velikost klíče je zde 32 bytů a nonce má 16 bytů. Tyto proměnné jsou pak předány do funkce Cipher pro šifrování pomocí algoritmu ChaCha20, která vrací zašifrovaný text. Pro implementaci byl opět použit balíček knihoven cryptography.

Funkce decrypt_chacha20 pracuje podobně, s tím rozdílem, že provádí dešifrování zašifrovaného textu zpět na otevřený text, který následně vrací jako výstup funkce.

```python
def encrypt_file_chacha20(file_path, key, nonce):
    with open(file_path, "rb") as file:
        plaintext = file.read()

    ciphertext = encrypt_chacha20(plaintext, key, nonce)

    # Extract the original file extension
    _, file_extension = os.path.splitext(file_path)

    # Encode file extension and its length
    file_extension_length = len(file_extension).to_bytes(4, 'big')  # Encode as 4-byte

    file_path = file_path[:-len(file_extension)]
    # Save the ciphertext, file extension length, and file extension to a file
    encrypted_file_path = file_path + b".enc"

    with open(encrypted_file_path, "wb") as encrypted_file:
        encrypted_file.write(file_extension_length + file_extension + ciphertext)
```

Obrázek 17. Ukázka implementace funkce encrypt_file_chacha20

V obrázku číslo 17 je zobrazena funkce encrypt_file_chacha20, která přijímá cestu k souboru, který má být zašifrován jako file_path, klíč key a nonce. Soubor je načten a zašifrován, poté je získána koncovka souboru a její velikost. Tato koncovka je odstraněna a nahrazena novou příponou .enc. Nakonec je nový zašifrovaný soubor uložen ve stejné složce jako původní soubor. Zde jsou do souboru zapsány pouze délka přípony, tvar přípony a zašifrovaný soubor, neboť ostatní argumenty jsou již k dispozici na vstupu.

```python
def decrypt_file_chacha20(file_path, key, nonce):
    with open(file_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Extract file extension length, and file extension from the encrypted data
    file_extension_length = int.from_bytes(encrypted_data[:4], 'big')  # Decode file extension length
    file_extension = encrypted_data[4:4+file_extension_length]  # Decode the file extension
    ciphertext = encrypted_data[4+file_extension_length:]  # Extract ciphertext

    # Decrypt the file contents
    plaintext = decrypt_chacha20(ciphertext, key, nonce)

    # Remove the ".enc" suffix from the input file name
    decrypted_file_path = file_path[:-4] + file_extension.decode()

    # Write the decrypted data to a new file
    with open(decrypted_file_path, "wb") as decrypted_file:
        decrypted_file.write(plaintext)
```

Obrázek 18. Ukázka implementace funkce decrypt_file_chacha20

U obrázku číslo 18 je zobrazena funkce decrypt_file_chacha20, která přijímá vstupní argumenty file_path jako cestu k souboru, klíč key a nonce. Nejprve je načten cílový soubor do proměnné. Tato proměnná je následně rozdělena na jednotlivé části podle bytů, zahrnující velikost přípony, tvar přípony a samotný zašifrovaný soubor. Následuje dešifrování souboru, přičemž je odstraněna přípona .enc a nahrazena příponou z původního souboru. Nakonec je dešifrovaný soubor uložen do složky, ze které byl původně načten zašifrovaný soubor.

## 4.6 DSA

DSA, nebo také Digital Signature Algorithm je používán jakožto elektronický podpis a splňuje funkci ověřování autenticity textu nebo souboru. Je zde využito hashovací algoritmus SHA-256 a privátní klíč k vytvoření podpisu. Poté je ověřena autenticitu textu, či souboru pomocí veřejného klíče a SHA-256. Klíč je vytvořen podobně jako u RSA v takzvaném PEM formátu. [7]

```
def sign_dsa(message, private_key):
    signature = private_key.sign(
        message,
        hashes.SHA256()
    )
    return signature

def verify_dsa(message, signature, public_key):
    try:
        public_key.verify(
            signature,
            message,
            hashes.SHA256()
        )
        return True
    except InvalidSignature:
        return False
```

Obrázek 19. Ukázka implementace funkcí sign_dsa a verify_dsa

U obrázku číslo 19 je zobrazena implementace funkcí sign_dsa a verify_dsa, které slouží k vytváření a ověřování digitálních podpisů. Funkce sign_dsa přijímá otevřený text ze kterého má být vytvořen podpis jako message a soukromý klíč private_key. Tato funkce vytváří podpis podle daných parametrů, kterými jsou soukromý klíč, otevřený text a hashovací algoritmus SHA-256. Vytvořený podpis je poté vrácen funkcí.

Funkce verify_dsa pak přijímá stejný otevřený text jako message, vytvořený podpis jako signature a veřejný klíč v podobě public_key. S těmito parametry funkce určuje, zda je podpis platný nebo ne, dle čehož vrací hodnotu True nebo False podle výsledku ověření.

```
def sign_DSA_file(file_path, private_key, signature_path):
    with open(file_path, "rb") as file:
        message = file.read()

    signature = sign_dsa(message, private_key)

    # Write the signature to a binary file
    with open(signature_path, "wb") as signature_file:
        signature_file.write(signature)

def verify_DSA_file(file_path, signature_path, public_key):
    with open(file_path, "rb") as file:
        message = file.read()

    with open(signature_path, "rb") as signature_file:
        signature = signature_file.read()

    if verify_dsa(message, signature, public_key):
        return True
    else:
        return False
```

Obrázek 20. Ukázka implementace funkcí sign_DSA_file a verify_DSA_file

Na obrázku číslo 20 jsou zobrazeny implementace funkcí pro práci se soubory, konkrétně sign_DSA_file a verify_DSA_file. Funkce sign_DSA_file přijímá cestu k souboru, ze kterého má být vytvořen podpis jako file_path, soukromý klíč private_key a cestu pro uložení podpisu signature_path. Tato funkce načte soubor a vytvoří podpis pomocí funkce sign_dsa, která byla popsána u obrázku číslo 19. Vytvořený podpis je uložen na zvolené cestě, obvykle s příponou .bin.

Funkce verify_DSA_file přijímá cestu k souboru ze kterého byl vytvořen podpis file_path, cestu k uloženému podpisu signature_path a veřejný klíč public_key. Tato funkce načte soubor a podpis do proměnných a poté využije funkci verify_dsa, která byla popsána u obrázku číslo 19.

## 4.7 Hashovací funkce

Specificky se jedná o SHA-3, SHA-256, SHA-1 a MD5. V implementaci pracuje s textem, či souborem z kterého vytváří zahashovaný soubor, či text. Jedná se o velice používané a známé hashovací algoritmy, což je také důvod jejich implementace. Dnešní nejnovější SHA-3 vznikl z původní SHA-1 a SHA-256, který řadíme mezi jedny z nejpoužívanějších hashovacích algoritmů. I přesto že MD5 již není bezpečný společně s SHA-1, tak jejich existence je nedílnou součástí vývoje hashovacích funkcí, což je také důvod jejich implementace. [7][24]

```
def calculate_md5(input_data, output_file=None):
    md5 = hashlib.md5()
    if isinstance(input_data, str):
        md5.update(input_data.encode())
    elif isinstance(input_data, bytes):
        md5.update(input_data)

    hashed_text = md5.hexdigest()

    if output_file:
        with open(output_file, 'w') as f:
            f.write(hashed_text)
    else:
        return hashed_text
```

Obrázek 21. Ukázka implementace funkce calculate_md5

Na obrázku číslo 21 je zobrazena funkce calculate_md5, která vytváří hash pomocí metody MD5. Tato funkce přijímá vstupní data buď ve formě souboru nebo textu. To již řídí funkce, která spojuje uživatelské rozhraní s funkcí pro hashování souborů ve formě vstupních dat. Vstupní data jsou označena jako input_data. Existence druhého argumentu, tedy output_file, určuje, zda-li se má výsledek zapsat jako soubor na zadané místo. Funkce využívá knihovnu hashlib a upravuje text do formy vstupu. Pokud uživatel nezvolí výstupní soubor, funkce vrací hash ve formě textu.

```
def calculate_sha3_256(input_data, output_file=None):
    sha3_256 = hashlib.sha3_256()
    if isinstance(input_data, str):
        sha3_256.update(input_data.encode())
    elif isinstance(input_data, bytes):
        sha3_256.update(input_data)

    hashed_text = sha3_256.hexdigest()

    if output_file:
        with open(output_file, 'w') as f:
            f.write(hashed_text)
    else:
        return hashed_text
```

Obrázek 22. Ukázka implementace funkce calculate_sha3_256

Na obrázku číslo 22 je zobrazena funkce calculate_sha3_256, která funguje podobně jako calculate_md5. Přijímá vstupní data jako input_data a cestu k výstupnímu souboru jako output_file, kde má být hashovací soubor uložen. Tato funkce vytváří hashový kód ze vstupního souboru podle jeho typu a vrací ho podle volby uživatele.

```
def calculate_sha256(input_data, output_file=None):
    sha256 = hashlib.sha256()
    if isinstance(input_data, str):
        sha256.update(input_data.encode())
    elif isinstance(input_data, bytes):
        sha256.update(input_data)

    hashed_text = sha256.hexdigest()

    if output_file:
        with open(output_file, 'w') as f:
            f.write(hashed_text)
    else:
        return hashed_text

def calculate_sha1(input_data, output_file=None):
    sha1 = hashlib.sha1()
    if isinstance(input_data, str):
        sha1.update(input_data.encode())
    elif isinstance(input_data, bytes):
        sha1.update(input_data)

    hashed_text = sha1.hexdigest()

    if output_file:
        with open(output_file, 'w') as f:
            f.write(hashed_text)
    else:
        return hashed_text
```

Obrázek 23. Ukázka implementace funkcí calculate_sha256 a calculate_sha1

Na obrázku číslo 23 jsou funkce pro implementaci SHA256 a SHA1. Tyto funkce fungují podobně jako předchozí funkce popsané u obrázků 21 a 22. Rozdíl spočívá v tom, že volají jinou funkci z knihovny hashlib, v závislosti na vybraném hashovacím algoritmu.

## 4.8 Generování klíčů a dalších parametrů

Velikost klíče, nebo jiných proměnných které daný algoritmus používá ovlivňuje jak je tento klíč generován. Obvykle jsou zde využité knihovny os a secret, co se týče generování asymetrických klíčů je zde využívána knihovna cryptography. Některé klíče mají vlastní dedikovanou funkci, některé jsou generované přímo ve funkci která pracuje s GUI. Rozdíl v nich je pouze dle jednotlivého algoritmu který používá z dané knihovny, avšak stále se jedná o formát náhodného klíče v požadované velikosti. Rozdílnost implementace náhodnosti jednotlivých využitých algoritmů by neměla hrát roli na funkci klíčů.

### 4.8.1 Generování AES klíče a dalších parametrů

První prezentovanou funkcí je funkce pro generování klíče a inicializačního vektoru podle obrázku číslo 24. Tato funkce využívá knihovnu os pro generování inicializačního vektoru. Celé grafické uživatelské rozhraní pracuje s výběrovým polem, které určuje jak bude klíč vygenerován. Tato funkce pracuje pouze s klíčem v textové podobě, to znamená že umísťuje hodnoty proměnných a funkcí do požadovaných textových polí. Nakonec upozorňuje uživatele o úspěšném vygenerování klíče, včetně inicializačního vektoru. AES je rozděleno do dvou oddělených oken pro šifrování a dešifrování, a proto pracuje s odlišnými textovými poli. To je důvodem, proč jsou funkce rozděleny na "enc" pro šifrování a "dec" pro dešifrování. I když možnost generování klíčů nemá moc smyslu pro okno dešifrování, protože je zapotřebí také samotný tag a generování nových klíčů není vhodné, je zde ponecháno pro větší flexibilitu uživatele.

```
def generate_key_AESenc(self):
    iv = os.urandom(12)
    if(self.ui.key_AEScombo.currentIndex() == 0):
        set_Text(self.ui.key_text_AES, generate_aes_key)
        self.ui.iv_AES_encrypt.setText(bytes_to_string(iv))
    else:
        self.ui.key_AEScombo.setCurrentIndex(0)
        set_Text(self.ui.key_text_AES, generate_aes_key)
        self.ui.iv_AES_encrypt.setText(bytes_to_string(iv))
    self.ui.info_label_AESenc.setText("Key as a text was successfully generated")
```

Obrázek 24. Ukázka implementace generování klíče a IV pro AES v GUI

Na obrázku číslo 25 je zobrazena funkce generate_aes_key, ta pracuje s výchozí hodnotou klíče o velikosti 32 bytů. Poté generuje pomocí knihovny secrets klíč v požadované velikosti a vrací ve formě textu, který poté vkládá do GUI.

```
def generate_aes_key(key_length=32):
    # default value set on key_length=16
    if key_length not in [16, 24, 32]:
            raise ValueError("Key length must be 16, 24, or 32 bytes (128, 192, or 256 bits)")

        # Generate a random key of the specified length
    key = secrets.token_bytes(key_length)
    key_string = bytes_to_string(key)
    return key_string
```

Obrázek 25. Ukázka implementace generování klíče pro AES

Na obrázku číslo 26 je zobrazena funkce generate_key_file_AESenc, která opět generuje inicializační vektor. Ten je následně vložen do textové formy, i když klíč je ve formě souboru. Nejprve se uživatele zeptá, kam chce tento soubor uložit, a získá tak cestu, kam vygeneruje tento klíč jako soubor pomocí funkce generate_file_key. Poté uloží cestu do

příslušného textového pole a inicializační vektor do dalšího textového pole. V případě, že uživatel nevybral typ souboru ve výběrovém poli, tak se pole změní na požadovaný výběr. Nakonec uživateli oznámí o úspěšném vygenerování klíče v souboru. Stejně jako u dešifrování, i zde je funkce identická, pouze s jinými názvy prvků.

```
def generate_key_file_AESenc(self):
    iv = os.urandom(12)
    file_path, _ = QFileDialog.getSaveFileName(self, "Save File", "", "Text Files (*.txt)")
    generate_file_key(32, file_path)
    if(self.ui.key_AEScombo.currentIndex() == 1):
        self.ui.key_file_AES.setText(file_path)
        self.ui.iv_AES_encrypt.setText(bytes_to_string(iv))
    else:
        self.ui.key_AEScombo.setCurrentIndex(1)
        self.ui.key_file_AES.setText(file_path)
        self.ui.iv_AES_encrypt.setText(bytes_to_string(iv))
    self.ui.info_label_AESenc.setText("Key as a file was successfully generated")
```

Obrázek 26. Ukázka implementace generování klíče v souboru pro AES v GUI

Na obrázku číslo 27 je zobrazena funkce generate_file_key, která na vstupu přijímá velikost klíče s výchozí hodnotou 8 a cestu k souboru s výchozí hodnotou key.txt. S klíči v souboru pracuje ve formě textového souboru, tedy pro .txt, ale bylo by možné tuto možnost změnit a v programu upravit pro práci s .bin soubory. Funkce poté generuje klíč pomocí knihovny secrets a v textové formě ho ukládá do nového souboru na cestu, kterou zvolil uživatel.

```
def generate_file_key(key_length=8, file_path="key.txt"):
    # Generate a random key of the specified length
    key = secrets.token_bytes(key_length)
    key_string = bytes_to_string(key)
    with open(file_path, "w") as key_file:
        key_file.write(key_string)
```

Obrázek 27. Ukázka implementace funkce pro generování klíče v souboru

### 4.8.2 Generování DES, 3DES, BF a ChaCha20 klíčů a dalších parametrů

U obrázku číslo 28 je zobrazena funkce generate_key_DES, která generuje klíč a inicializační vektor v textové formě pomocí knihovny os. Ty pak vkládá do textových polí, včetně kontroly výběrového pole, které mění v závislosti na typu generování. Nakonec uživateli dává vědět o úspěšném generování klíče a IV. Co se týče 3DES, BF a ChaCha20 jsou funkce velice podobné s rozdíly ve velikostech klíčů a vlastními názvy textových polí, do kterých tyto hodnoty vkládají. U 3DES je velikost klíče 24 bytů a u BF je zde 56 bytů, co se týče ChaCha20 generuje klíč o velikosti 32 bytů a nonce v 16 bytech.

```
def generate_key_DES(self):
    key_bytes = base64.b64encode(os.urandom(8))
    iv_bytes = base64.b64encode(os.urandom(8))
    set_Text(self.ui.iv_text_DES, lambda: iv_bytes.decode('utf-8'))
    if(self.ui.key_DEScombo.currentIndex() == 0):
        set_Text(self.ui.key_text_DES, lambda: key_bytes.decode('utf-8'))
    else:
        self.ui.key_DEScombo.setCurrentIndex(0)
        set_Text(self.ui.key_text_DES, lambda: key_bytes.decode('utf-8'))
    self.ui.info_label_DES.setText("Key and IV as a text were successfully generated")
```

Obrázek 28. Ukázka implementace funkce pro generování klíče a IV pro DES v GUI

U obrázku číslo 29 je zobrazena funkce generate_key_file_DES, která se nejprve ptá uživatele, kam chce klíč uložit v textové formě i když v souboru. Poté tuto cestu použije pro vygenerování klíče v souboru pomocí funkce generate_file_key s parametrem 8 bytů. Následně generuje inicializační vektor v textové formě pomocí knihovny os. Poté uloží jak IV, tak cestu k souboru klíče do textových polí. V případě, že uživatel zvolil jinou hodnotu ve výběrovém poli, funkce tuto hodnotu změní a uživateli dá vědět o úspěšném generování klíče a IV. U ChaCha20 se jedná rozdílově o nonce.

```
def generate_key_file_DES(self):
    file_path, _ = QFileDialog.getSaveFileName(self, "Save File", "", "Text Files (*.txt)")
    generate_file_key(8, file_path)
    iv_bytes = base64.b64encode(os.urandom(8))
    set_Text(self.ui.iv_text_DES, lambda: iv_bytes.decode('utf-8'))
    if(self.ui.key_DEScombo.currentIndex() == 1):
        self.ui.key_file_DES.setText(file_path)
    else:
        self.ui.key_DEScombo.setCurrentIndex(1)
        self.ui.key_file_DES.setText(file_path)
    self.ui.info_label_DES.setText("Key as a file and IV as a text were successfully generated")
```

Obrázek 29. Ukázka implementace funkce pro generování klíče v souboru pro DES v GUI

### 4.8.3 Generování soukromého a veřejného klíče u RSA

U obrázku číslo 30 je zobrazena funkce generate_keyRSA, která pomocí funkce generate_key_pair vygeneruje soukromý a veřejný klíč. Poté dle hodnoty výběrového pole toto pole změní nebo ponechá v textovém typu. Dále tyto hodnoty vloží do požadovaných textových polí v GUI. Nakonec je uživateli oznámeno o úspěšném generování klíčů.

```
def generate_keyRSA(self):
    private_key_pem, public_key_pem = generate_key_pair()
    private_key_text = private_key_pem.decode('utf-8')
    public_key_text = public_key_pem.decode('utf-8')
    if(self.ui.key_RSAcombo.currentIndex() == 0):
        # Setting the keys into the UI text lines
        set_Text(self.ui.private_key_text_RSA, lambda: private_key_text)
        set_Text(self.ui.public_key_text_RSA, lambda: public_key_text)
    else:
        self.ui.key_RSAcombo.setCurrentIndex(0)
        set_Text(self.ui.private_key_text_RSA, lambda: private_key_text)
        set_Text(self.ui.public_key_text_RSA, lambda: public_key_text)
    self.ui.info_label_RSA.setText("Public and Private keys as a text were successfully generated")
```

Obrázek 30. Ukázka implementace generování klíčů pro RSA v GUI

U obrázku číslo 31 je zobrazena funkce generate_key_pair, která prvně generuje soukromý klíč ve velikosti 2048 bytů pomocí funkce rsa z knihovny cryptography. Následně dle hodnoty soukromého klíče vygeneruje veřejný a oba tyto klíče jsou serializovány do PEM formátu. Nakonec funkce vrátí tyto klíče.

```
def generate_key_pair():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    public_key = private_key.public_key()

    # Serialize keys to PEM format
    private_key_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    public_key_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    return private_key_pem, public_key_pem
```

Obrázek 31. Ukázka implementace generování klíčů pro RSA

U obrázku číslo 32 je zobrazena funkce generate_key_file_RSA, která se stará o generování klíčů ve formě souborů. Nejprve uživatele požádá o vyběr cesty pro uložení těchto klíčů. Poté zavolá funkci generate_key_file_pair, která se stará o samotné generování klíčů v PEM formátu. Následně, podle hodnoty ve výběrovém poli, buď změní hodnotu nebo ji nechá beze změny. Poté vloží cesty k těmto souborům do požadovaných textových polí. Nakonec uživatele informuje o úspěšném generování klíčů.

```
def generate_key_file_RSA(self):
    file_path_private,_ = QFileDialog.getSaveFileName(self, "Save private key", "", "PEM Files (*.pem)")
    file_path_public,_ = QFileDialog.getSaveFileName(self, "Save public key", "", "PEM Files (*.pem)")
    generate_key_file_pair(file_path_private, file_path_public)
    if(self.ui.key_RSAcombo.currentIndex() == 1):
        self.ui.private_key_file_RSA.setText(file_path_private)
        self.ui.public_key_file_RSA.setText(file_path_public)
    else:
        self.ui.key_RSAcombo.setCurrentIndex(1)
        self.ui.private_key_file_RSA.setText(file_path_private)
        self.ui.public_key_file_RSA.setText(file_path_public)
    self.ui.info_label_RSA.setText("Public and Private keys as a file were successfully generated")
```

Obrázek 32. Ukázka implementace generování klíčů v souboru pro RSA v GUI

Na obrázku číslo 33 je zobrazena funkce generate_key_file_pair, která na vstupních parametrech bere samotné cesty soukromého a veřejného klíče, které si uživatel zvolil. Dále pomocí funkce generate_key_pair vygeneruje klíče v PEM formátu, následně na to tyto hodnoty uloží do souborů dle cest, které si uživatel zvolil ve funkci generate_key_file_RSA.

```
def generate_key_file_pair(private_key_path, public_key_path):
    private_key, public_key = generate_key_pair()

    with open(private_key_path, "wb") as private_key_file:
        private_key_file.write(private_key)

    with open(public_key_path, "wb") as public_key_file:
        public_key_file.write(public_key)
```

Obrázek 33. Ukázka implementace generování klíčů v souboru pro RSA

### 4.8.4 Generování soukromého a veřejného klíče u DSA

Na obrázku číslo 34 je zobrazena funkce generate_keyDSA, která funguje podobně jako funkce generate_keyRSA. Na začátku je volána funkce generate_dsa_keypair, která vygeneruje jak soukromý, tak veřejný klíč. Dále dle hodnoty výběrového pole tyto hodnoty buď změní, nebo je nechá. Nakonec jsou tyto klíče vloženy v textové formě do příslušných textových polí a uživatel je informován o úspěšném generování klíčů.

```
def generate_keyDSA(self):
    private_key_pem, public_key_pem = generate_dsa_keypair()
    private_key_text = private_key_pem.decode('utf-8')
    public_key_text = public_key_pem.decode('utf-8')
    if(self.ui.key_DSAcombo.currentIndex() == 0):
       # Setting the keys into the UI text lines
       set_Text(self.ui.private_key_text_DSA, lambda: private_key_text)
       set_Text(self.ui.public_key_text_DSA, lambda: public_key_text)
    else:
       self.ui.key_DSAcombo.setCurrentIndex(0)
       set_Text(self.ui.private_key_text_DSA, lambda: private_key_text)
       set_Text(self.ui.public_key_text_DSA, lambda: public_key_text)
    self.ui.sign_verify_DSA.setText("Public and Private keys as a text were successfully generated")
```

Obrázek 34. Ukázka implementace generování klíčů pro DSA v GUI

Na obrázku číslo 35 je zobrazena funkce generate_dsa_keypair, která první vygeneruje soukromý klíč pomocí dsa z knihovny cryptography. Dále ze soukromého klíče vygeneruje klíč veřejný a oba tyto klíče jsou serializovány do PEM formátu. Oba tyto klíče nakonec funkce vrací.

```
def generate_dsa_keypair():
    private_key = dsa.generate_private_key(key_size=2048)
    public_key = private_key.public_key()

    private_key_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
        )
    public_key_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
    return private_key_pem, public_key_pem
```

Obrázek 35. Ukázka a implementace generování klíčů u DSA

Na obrázku číslo 36 je zobrazena funkce generate_key_file_DSA, která se nejprve zeptá uživatele, kam chce uložit klíče a tyto cesty uloží do proměnných. Poté volá funkci generate_dsa_file_keypair s těmito cestami jako vstupními argumenty. Dále podle hodnoty výběrového pole tuto hodnotu buď změní, nebo nechá beze změny a uloží cesty klíčů do příslušných textových polí. Nakonec uživatele informuje o úspěšném generování souborů.

```
def generate_key_file_DSA(self):
    file_path_private,_ = QFileDialog.getSaveFileName(self, "Save private key", "", "PEM Files (*.pem)")
    file_path_public,_ = QFileDialog.getSaveFileName(self, "Save public key", "", "PEM Files (*.pem)")
    generate_dsa_file_keypair(file_path_private, file_path_public)
    if(self.ui.key_DSAcombo.currentIndex() == 1):
        self.ui.private_key_file_DSA.setText(file_path_private)
        self.ui.public_key_file_DSA.setText(file_path_public)
    else:
        self.ui.key_DSAcombo.setCurrentIndex(1)
        self.ui.private_key_file_DSA.setText(file_path_private)
        self.ui.public_key_file_DSA.setText(file_path_public)
    self.ui.sign_verify_DSA.setText("Public and Private keys as a file were successfully generated")
```

Obrázek 36. Ukázka a implementace generování klíčů v souboru pro RSA v GUI

Na obrázku číslo 37 je zobrazena funkce generate_dsa_file_keypair, která přijímá vstupní argumenty cesty pro soukromý a veřejný klíč. Následně jsou klíče vygenerovány v PEM formátu pomocí funkce generate_dsa_keypair a nakonec tyto hodnoty jsou uloženy do dvou souborů jako veřejný a soukromý klíč dle zadaných cest.

```
def generate_dsa_file_keypair(private_key_path, public_key_path):
    private_key, public_key = generate_dsa_keypair()

    with open(private_key_path, "wb") as private_key_file:
        private_key_file.write(private_key)

    with open(public_key_path, "wb") as public_key_file:
        public_key_file.write(public_key)
```

Obrázek 37. Ukázka implementace generování klíčů v souboru pro RSA

# 5 POUŽITÉ TECHNOLOGIE

Pro zpracování programu byl vybrán programovací jazyk Python, který je ve verzi 3.11. Během práce bylo využíváno vývojové prostředí Spyder a grafické rozhraní pro uživatele bylo vytvořeno pomocí QT Designer.

## 5.1 Python

Programovací jazyk Python byl vybrán převážně kvůli osobní preferenci a předchozí zkušenosti s aplikováním jednotlivých šifrovacích algoritmů pomocí tohoto jazyka. Avšak velkou výhodou jsou i různé knihovny které ulehčily práci s těmito algoritmy, jednoduchost kterou samotný jazyk nabízí a počet zdrojů pro ulehčení řešení různých problémů které se během práce vyskytly. K práci byly využity knihovny Pycryptodome společně s Cryptography, které mi posloužily pro samotné kryptografické algoritmy jako je AES, DES, DES3, Blowfish a podobné. Taktéž os pro vytváření náhodných klíčů, Base64 pro úpravu textu pro práci s jednotlivými algoritmy a samotným GUI. Hashlib pak pro jednotlivé hashovací funkce jako je SHA-3, MD5, SHA-256 a SHA-1 a v nepochybné řadě i PyQt5, díky kterému jsem mohl funkce spojovat se samotným GUI.

## 5.2 Vývojové prostředí

Jak již bylo zmíněno, využito bylo IDE Spyder, který je open source, neboli zdarma ke stažení a použití. Spyder je prezentován jakožto vývojové prostředí pro datovou analýzu se spousty dalšími prvky. Ten byl však vybrán převážně kvůli osobní přívětivosti. [25]

## 5.3 GUI v QT Designer

Jedná se o program, který byl využit pro vytvoření GUI. Díky čemuž byl vytvořen vzhled a základní struktury prvků v samotném GUI v programu. Velké výhody nabízí v jednoduchosti tvoření bez potřeby psaní samotných prvků v kódu, čímž urychlí spoustu času. Dalo

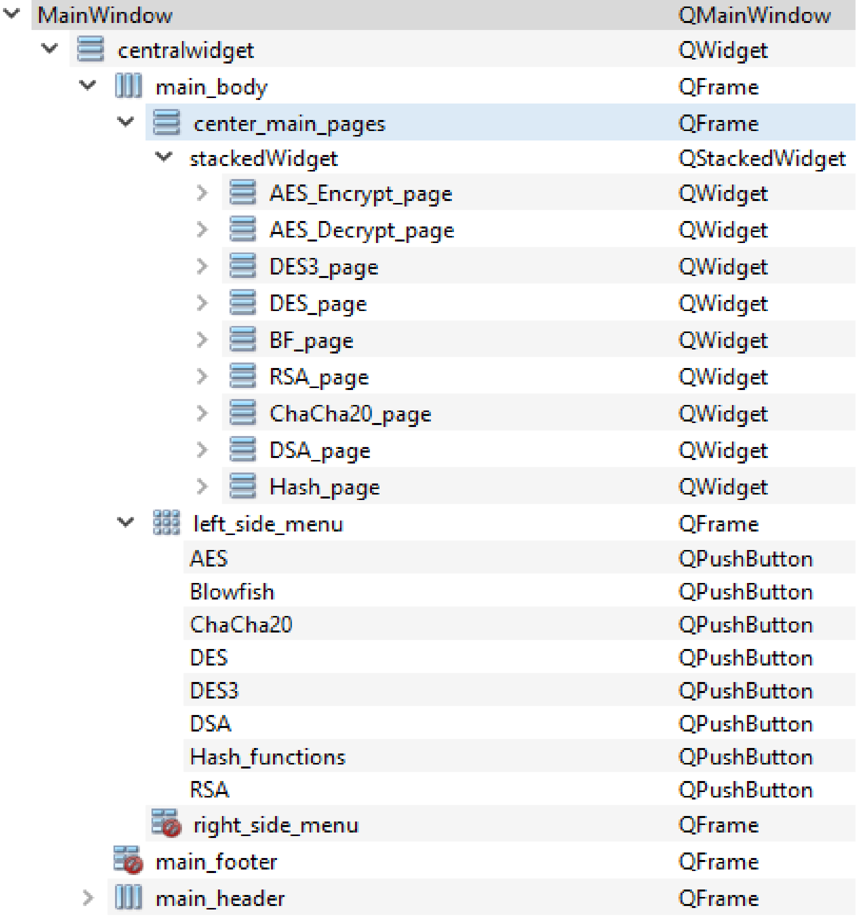
by se říct, že po zvyknutí si na tento program jde práce na tvoření GUI velice rychle a naučit se práci s rozvržením různých prvků udělá veškerou práci mnohem jednodušší. Program nabízí mnoho částí, které lze využít v různých GUI. Veškeré prvky se nachází pod QMainWindow, kde lze vkládat cokoliv a je možné si z něj vytvořit vlastní GUI dle libosti. Lze upravovat styly jednotlivých prvků, tedy například barvy jak textu tak prvků, jejich atributy co se týče minimální šířky či velikost písma a podobné.

# 6 POPIS A STRUKTURA UŽIVATELSKÉHO ROZHRANÍ

Tato kapitola se zabývá popisem celého GUI, jeho strukturou a taktéž funkčností všech tlačítek, textových a výběrových polí.

## 6.1 Struktura prvků v uživatelském rozhraní

Celé GUI spadá do objektu QMainWindow pod kterým se nachází seskupené prvky dle rozložení. Zde se nachází main_header nahoře, centralwidget uprostřed a main_footer dole. Dále je rozdělen main_body na center_main_pages, který je umístěn uprostřed, následuje left_side_menu na levé straně a right_side_menu na pravé straně. Prvek left_side_menu se skládá z tlačítek, ty přepínají stránky v stackedWidget dle názvů. Také je třeba zmínit účel ikonek před jednotlivými prvky. Ty určují jakým stylem jsou prvky pod tímto prvkem seskládány, převážně buď vodorovně nebo svisle, avšak jsou zde i například tabulkově jako u left_side_menu nebo vůbec jako například u main_footer.



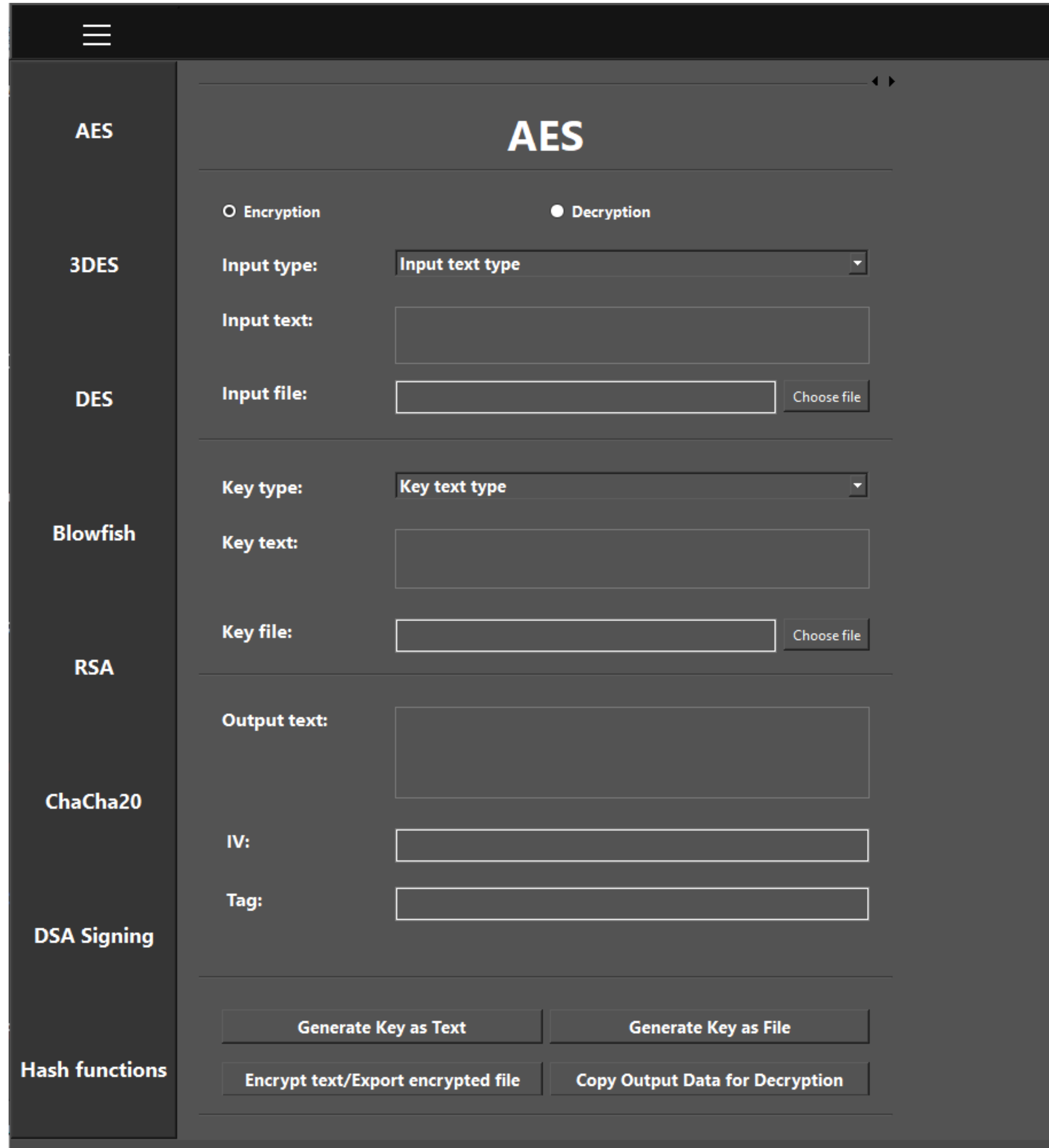

Obrázek 38. Struktura prvků v GUI

## 6.2 Popis uživatelského rozhraní

Tato část kapitoly se rozděluje dle jednotlivých stran, které jsou rozděleny dle jednotlivých algoritmů. Zaměřuje se na popis tlačítek a textových polí, které do těchto stránek spadají.

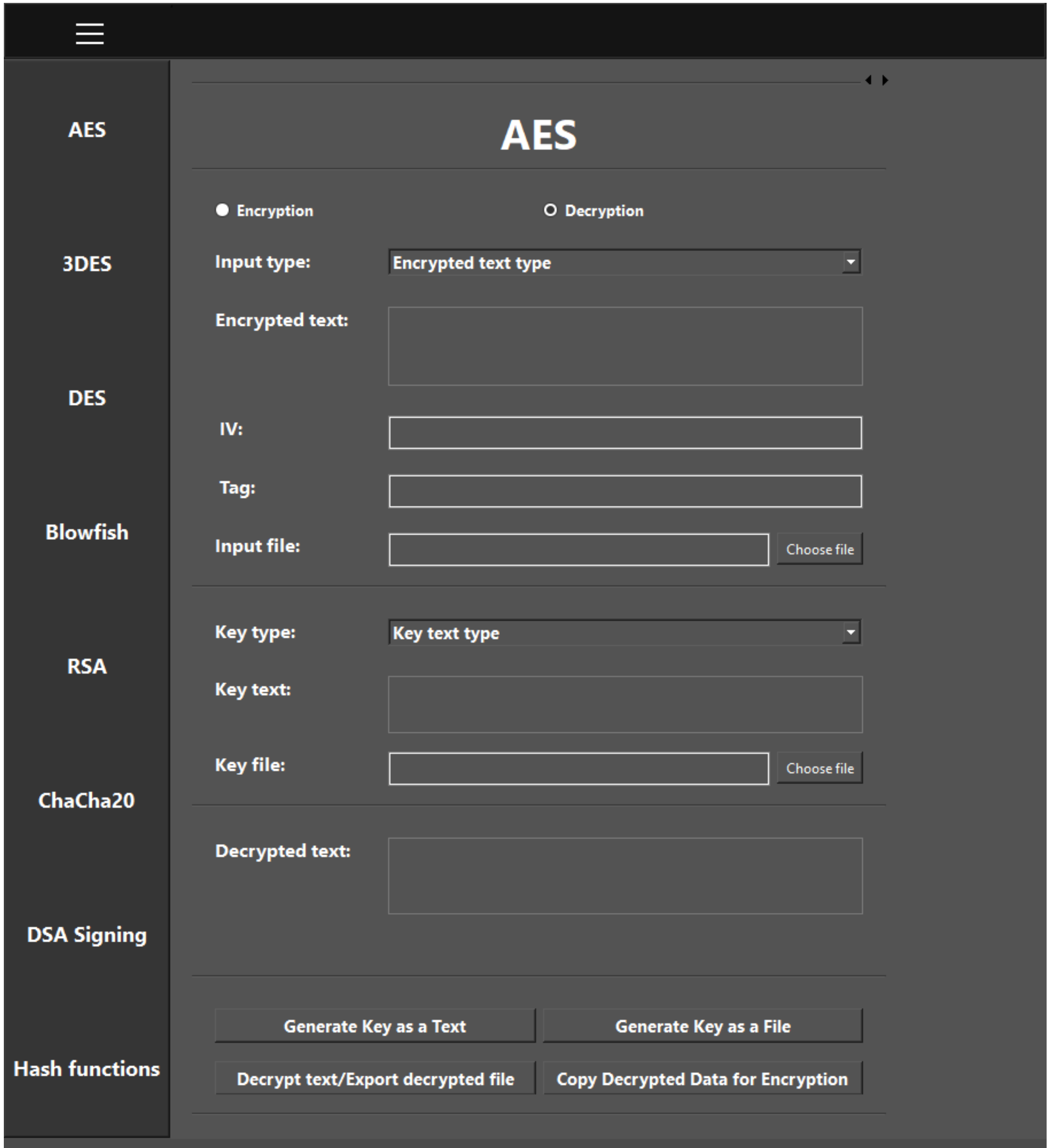
### 6.2.1 Popis stránky AES

Na obrázku číslo 39 je zobrazena ukázka stránky AES. Na vrchu stránky je nadpis a pod ním je možnost volby mezi šifrováním pod Encryption a dešifrováním pod Decryption, což přepíná stránky podle zvoleného režimu. Následuje rozdělení stránky na části Vstup jako Input, Klíč jako Key, Výstup jako Output, IV a Tag, a nakonec část pro tlačítka. V části Input a Key je první výběrové pole, kde si uživatel může vybrat typ vstupu – textový nebo souborový. Volba typu vstupu buď zamkne tlačítko pro výběr souboru a textové pole pro cestu k souboru, nebo textové pole pro vstup. Po kliknutí na tlačítko Choose file si uživatel může vybrat cestu k souboru, kterou pak uloží do vedlejšího textového pole. Část Output slouží pro zobrazení zašifrovaného textu, část IV pro zobrazení inicializačního vektoru a Tag pro zobrazení hodnoty Tag, který uživatel obdrží jako výstup po zašifrování. Níže se pak nacházejí tlačítka Generate Key as Text, které vygeneruje klíč a IV do příslušných polí a Generate Key as File, které vygeneruje klíč jako soubor a IV do textového pole. Pod nimi jsou tlačítka Encrypt text/Export encrypted file, která na základě vstupu zašifrují text nebo soubor a Copy Output Data for Decryption, které zkopíruje výstupy a potřebné informace pro dešifrování do stránky pro dešifrování a současně uživatele přepnou na tuto stránku.




Obrázek 39. Ukázka stránky AES pro šifrování

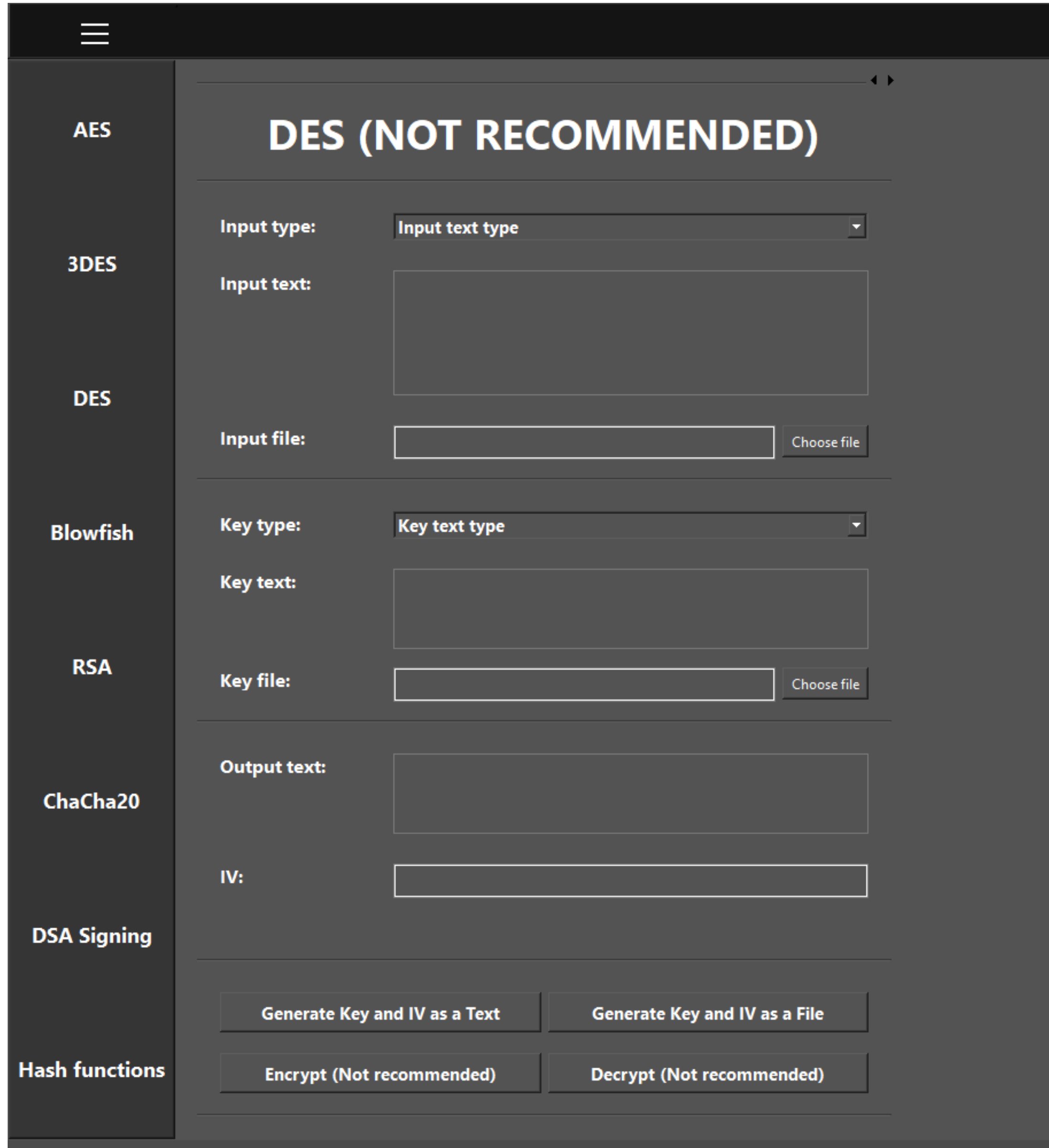
Na obrázku číslo 40 je zobrazeno okno pro dešifrování pomocí AES. Zde je opět možnost přepnout se zpět na stránku pro šifrování. Uživatel má možnost volby mezi prací se souborem nebo textem, přičemž volba souboru uzamkne pole pro zašifrovaný text Encrypted text, IV a Tag, zatímco volba textu uzamkne pole pro vstupní soubor a tlačítko Choose file. Pokud uživatel pracuje se souborem, IV a Tag jsou zahrnuty v samotném souboru a není třeba je zadávat ručně v textové formě. Klíč funguje stejně jako při šifrování. V části Decrypted text se zobrazuje dešifrovaný text. Dále jsou zde tlačítka Generate Key as Text a Generate Key as File, která fungují stejně jako u šifrování. Tlačítko Decrypt text/Export decrypted file slouží k dešifrování textu nebo souboru. Tlačítko Copy Decrypted Data for Encryption kopíruje dešifrovaný výstup, klíč a IV zpět na stránku pro šifrování.




Obrázek 40. Ukázka stránky AES pro dešifrování

### 6.2.2 Popis stránky DES

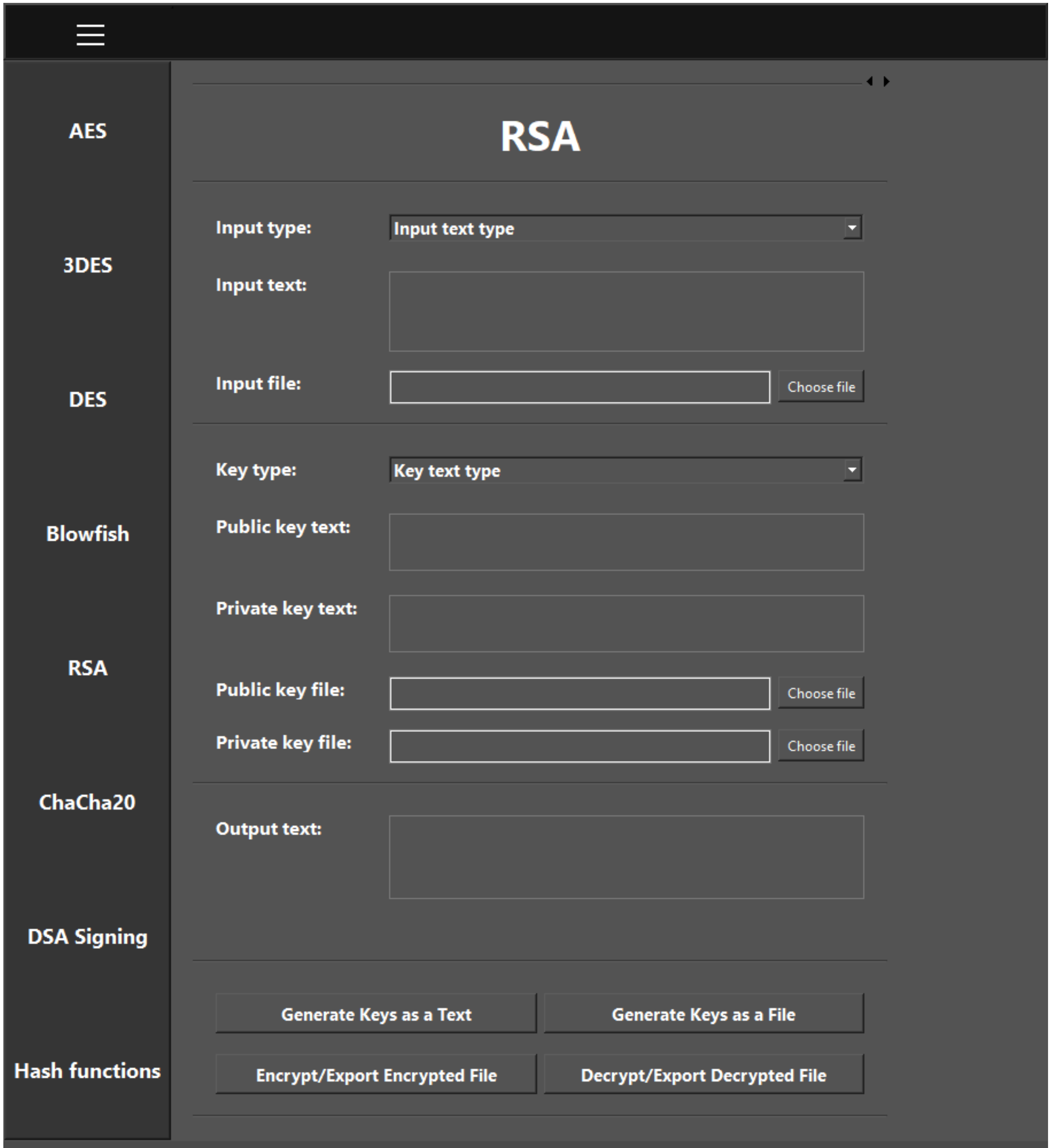
Na obrázku číslo 41 je zobrazeno okno pro DES, to je rozděleno na Input, Key, Output a tlačítka. Opět výběrové pole u Input a Key určuje typ vstupu pro klíč a otevřený text. Output pro výstup zašifrovaného textu a IV pro inicializační vektor. Tlačítka Generate Key and IV as a Text a Generate Key and IV as a File mají funkci generování klíče a IV dle typu buď textové nebo v souboru a IV v textovém. Nakonec Encrypt pro šifrování a Decrypt pro dešifrování. Je stejně strukturované jako 3DES a Blowfish.




Obrázek 41. Ukázka stránky DES

### 6.2.3 Popis stránky RSA

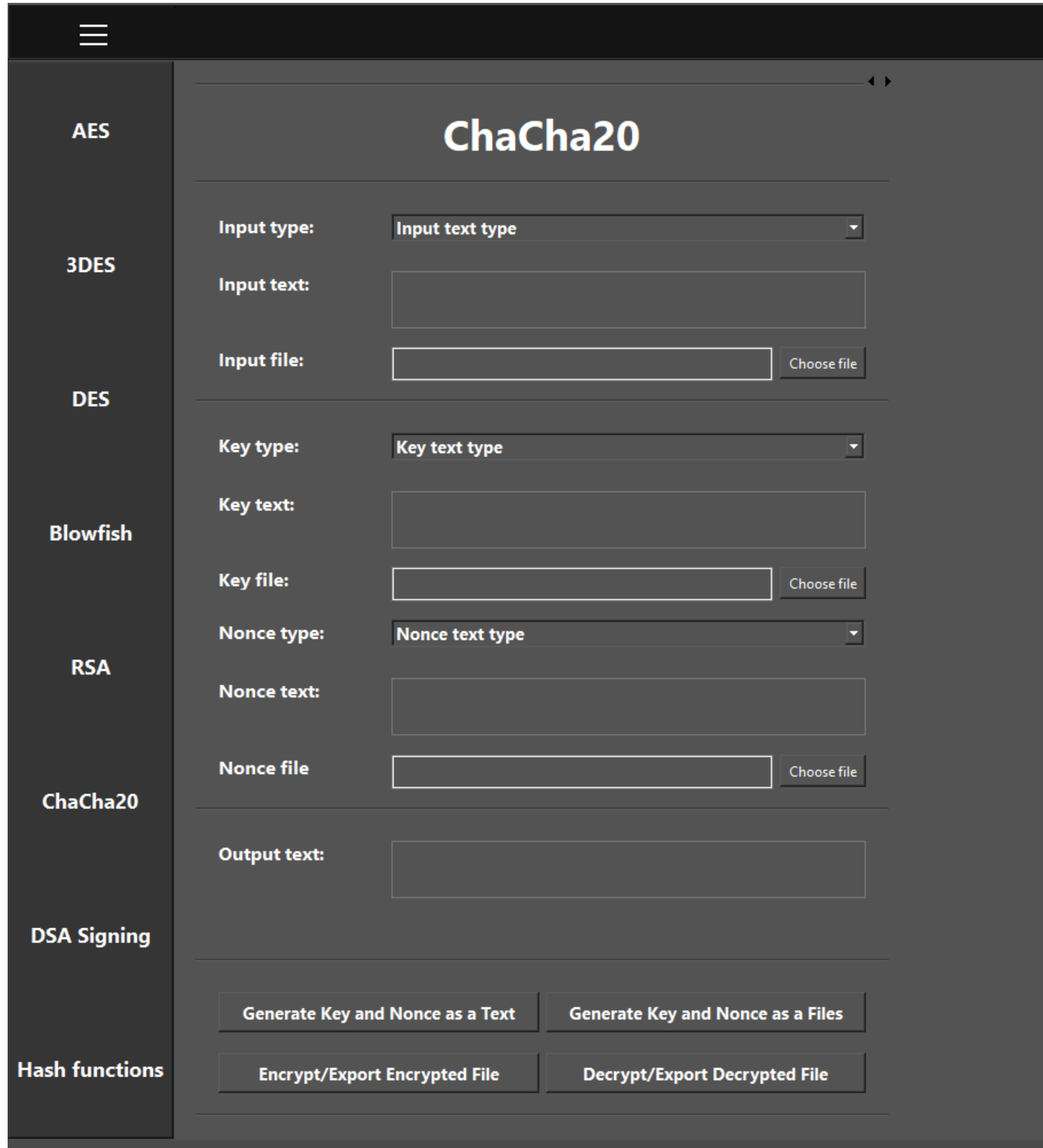
Na obrázku číslo 42 je zobrazena stránka RSA, zde máme opět rozdělenou stránku na Input, Key, Output a tlačítka. Input a Key funguje stejně s rozdílem dvou klíčů, tedy veřejný a soukromý. Výstup funkce je zapsán do textového pole u Output. Tlačítka fungují stejně jako u DES, 3DES a Blowfish, takže vrchní generují klíče a spodní šifrují a dešifrují.




Obrázek 42. Ukázka stránky RSA

### 6.2.4 Popis stránky ChaCha20

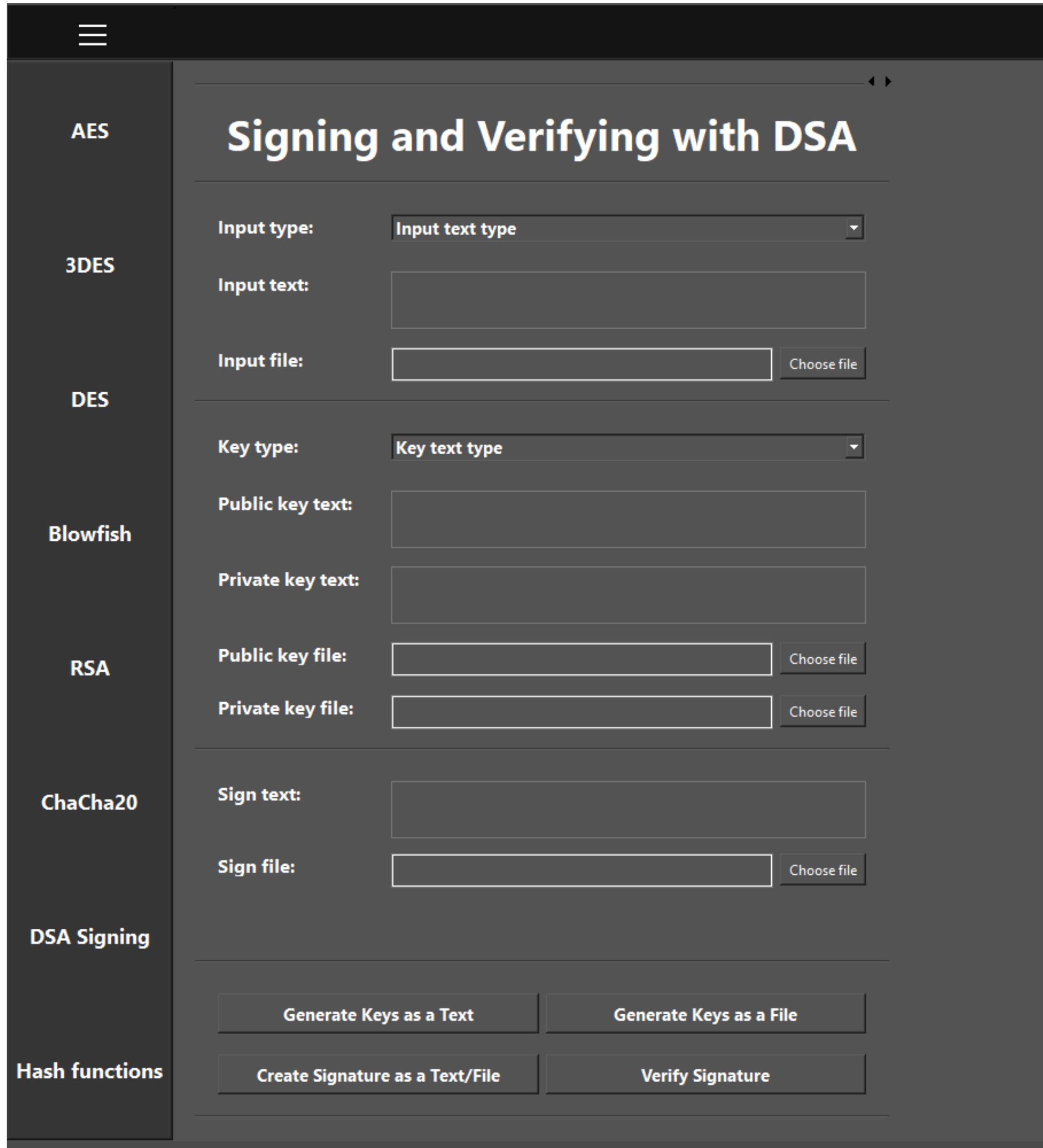
Na obrázku číslo 43 je zobrazena stránka ChaCha20, ta obsahuje části Input, Key, Nonce, Output a tlačítka. Zde Input, Key a Nonce funguje velice stejně s výběrovým polem pro typ vstupu. Vrchní tlačítka generují klíč a nonce pro ChaCha20 jedno v textové formě a druhé ve formě souborů. Spodní tlačítka pak šifrují a dešifrují vstup dle šifrovacího algoritmu.




Obrázek 43. Ukázka stránky ChaCha20

### 6.2.5 Popis stránky DSA

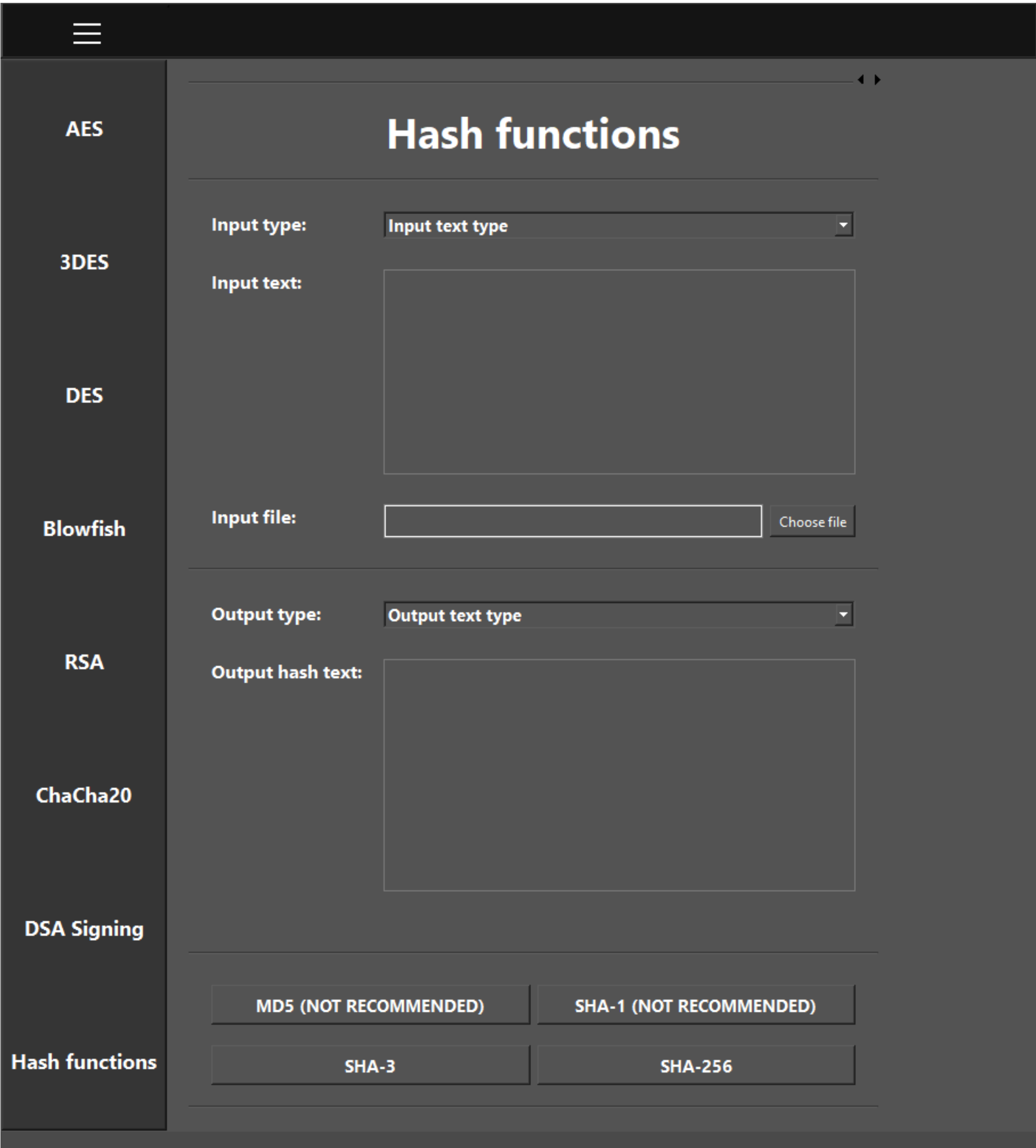
Na obrázku číslo 44 je zobrazena stránka pro DSA, ta vypadá a funguje velice podobně jako RSA. Takže část Input, Key s rozdílem výstupu jako Sign pro samotný podpis jak v textové tak ve formě souboru. Dále část pro tlačítka, kde vrchní tlačítka opět generují klíče jak v textové formě, tak ve formě souboru. Create Signature as a Text/File pak vytváří podpis dle vstupu. A nakonec Verify Signature slouži pro potvrzení zda je podpis dle vstupu pravý.




Obrázek 44. Ukázka stránky DSA

### 6.2.6 Popis stránky pro Hashovací algoritmy

Na obrázku číslo 45 je zobrazena stránka Hash functions, ta je rozdělena na Input, Output a tlačítka. Typ vstupu jde vybrat pomocí výběrového pole stejně tak jako typ výstupu. Tlačítka jsou pak specifická pro každý hashovací algoritmus zvlášť.




Obrázek 45. Ukázka stránky Hash functions

# 7 IMPLEMENTACE FUNKCÍ K UŽIVATELSKÉMU ROZHRANÍ

Tato kapitola se zabývá implementací funkcí, které přímo komunikují s uživatelským rozhraním s výjimkou těch již popsaných do nichž řadíme generování klíčů. To znamená že jsou zde popsány implementace jednotlivých algoritmů přímo ke GUI, různých funkcí které s GUI přímo komunikují a jejich využití přímo v kódu.

## 7.1 AES k uživatelskému rozhraní

První část kódu pro šifrování se zabývá s manipulací s klíčem jak je zobrazeno na obrázku číslo 46. Je zde zavedena hodnota pro chybové hlášky error_messages, která se rozšiřuje dle chyb ve vstupu. Podle výběrového pole u klíče rozhoduje jestli vezme klíč z textového pole, nebo ze souboru. Mezitím validuje vstup tohoto klíče v obou případech. Využívá funkce load_key_from_file která pouze otevře a načte soubor pro proměnné, kterou dále vrací.

```
def process_encryption_AES(self):
    # Initialize list to store error messages
    error_messages = []

    # Define key_bytes and iv_bytes outside of the try-except block
    key_bytes = None
    iv_bytes = None

    # Check if the user wants to load the key from a file
    if self.ui.key_AEScombo.currentIndex() == 0:
        # Normal key input process
        key_text = self.ui.key_text_AES.toPlainText()
        try:
            key_bytes = string_to_bytes(key_text)
        except:
            # Handle base64 decoding error
            error_messages.append("Invalid key.")
    else:
        # Load key from file
        key_path = self.ui.key_file_AES.text()
        if not validate_file_path(key_path):
            # File path is invalid
            error_messages.append("Invalid key file path.")
        else:
            key_bytes = string_to_bytes(load_key_from_file(key_path))

    # Check if key_bytes is not None before validating
    if key_bytes is not None:
        if not validate_key_AES(key_bytes):
            # Key is invalid
            error_messages.append("Invalid key.")
```

Obrázek 46. Ukázka implementace funkce pro šifrování AES část první – klíč

Druhá část je zobrazena na obrázku číslo 47, ta vezme IV, upraví a validuje ji.

```
# Convert iv_text to bytes
iv_text = self.ui.iv_AES_encrypt.text()
try:
    iv_bytes = string_to_bytes(iv_text)
except:
    error_messages.append("Invalid IV.")

# Check if iv_bytes is not None and validate IV
if iv_bytes is not None:
    if not validate_iv_AES(iv_bytes):
        # IV is invalid
        error_messages.append("Invalid IV.")
```

Obrázek 47. Ukázka implementace funkce pro šifrování AES část druhá – IV

Třetí část v obrázku číslo 48 pak pracuje s možností, že je vstup ve formě souboru, takže cesta k němu je validována a taktéž je volána funkce encrypt_file_aes která se o šifrování stará a při úspěchu informuje o této skutečnosti i uživatele.

```
# Check if the user wants to encrypt a file
if self.ui.input_AEScombo.currentIndex() == 1:
    input_path = self.ui.input_file_AES.text().encode('utf-8')
    if not validate_file_path(input_path):
        # Input file path is invalid
        error_messages.append("Invalid input file path.")
    else:
        try:
            encrypt_file_aes(input_path, key_bytes, iv_bytes)
            if not error_messages:
                self.ui.info_label_AESenc.setText("File was successfully encrypted in input file path")
        except:
            error_messages.append("Invalid input file path.")
```

Obrázek 48. Ukázka implementace funkce pro šifrování AES část třetí – šifrování souboru

Čtvrtá část v obrázku číslo 49 pak ukazuje šifrování, kdyby byl vstup ve formě textu. Načítá a kontroluje vstup pro šifrování, poté volá funkce encrypt_aes pro samotné šifrování a informuje uživatele o úspěšném šifrování. V případě že by se objevila jakákoliv chyba, vrátí chybou hlášku nebo kdyby se vyskytla chybová hláška již předtím, tuto část vynechá. Poté nakonec v případě že chybové hlášky jsou, o nich informuje uživatele.

```
else:
    input_text = self.ui.input_text_encrypt_AES.toPlainText().encode('utf-8')
    if not validate_input_data_AES(input_text):
        # Input text is invalid
        error_messages.append("Invalid input text.")
    else:
        if not error_messages:
            try:
                iv, ciphertext, tag = encrypt_aes(input_text, key_bytes, iv_bytes)
                # Set the outputs
                self.ui.iv_AES_encrypt.setText(bytes_to_string(iv))
                self.ui.output_text_AES.setText(base64.b64encode(ciphertext).decode('utf-8'))
                self.ui.tag_AES_encrypt.setText(bytes_to_string(tag))
                self.ui.info_label_AESenc.setText("Text was successfully encrypted")
            except:
                error_messages.append("Invalid encryption, check your inputs.")

# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.info_label_AESenc.setText(error_message)
```

Obrázek 49. Ukázka implementace funkce pro šifrování AES část čtvrtá – šifrování textu

Další následuje část pro stránku k dešifrování AES, zde jsou vynechány první dvě části, jelikož jsou totožné s přídavkem tagu. Takže práce s klíčem ve formě textu nebo souboru a načítání a dešifrování, včetně chybových hlášek.

Tato část v obrázku číslo 50 pracuje s možností, že je na vstupu zašifrovaný soubor, kontroluje cestu k souboru a volá funkci decrypt_file_aes která se o proces dešifrování postará. Dále dá vědět uživateli o úspěšném dešifrování, nebo posune dále chybovou hlášku.

```
# Check if the user wants to decrypt a file
if self.ui.input_AEScombo_2.currentIndex() == 1:
    input_path = self.ui.input_file_AES_2.text()
    if not validate_file_path(input_path):
        # Input file path is invalid
        error_messages.append("Invalid input file path.")
    else:
        try:
            decrypt_file_aes(input_path, key_bytes)
            self.ui.info_label_AESdec.setText("File was successfully decrypted in input file path")
            error_messages.clear()
        except:
            error_messages.append("Invalid input file path.")
```

Obrázek 50. Ukázka implementace funkce pro dešifrování AES část první – dešifrování souboru

Tato část v obrázku číslo 51 pracuje se vstupem ve formě textu, načte vstup a volá funkci decrypt_aes, ta se postará o dešifrování a vrací dešifrovaný text v message. Následně vkládá text do textového pole a oznamuje uživateli o úspěšném dešifrování. V opačném případě přidává text do chybových hlášek které na konci vypíše.

```
else:
    try:
        ciphertext = base64.b64decode(self.ui.input_text_decrypt_AES.toPlainText().encode('utf-8'))
    except:
        error_messages.append("Invalid ciphertext.")
    else:
        if not error_messages:
            try:
                message = decrypt_aes(iv_bytes, ciphertext, tag_bytes, key_bytes)
                # Set the output
                self.ui.output_text_AES_2.setText(message.decode('utf-8'))
                self.ui.info_label_AESdec.setText("Text was successfully decrypted")
            except:
                error_messages.append("Invalid decryption, check your inputs.")

# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.info_label_AESdec.setText(error_message)
```

Obrázek 51. Ukázka implementace funkce pro dešifrování AES část druhá – dešifrování textu

## 7.2 DES, 3DES a Blowfish k uživatelskému rozhraní

Jednotlivé bloky jsou popsány v jednom, jelikož obě pracují se vstupy identicky a funkce jsou dlouhé v poměru s ostatními. Rozdíly jsou zde dle toho jaké textové pole volají, algoritmy které využívají pro šifrování a dešifrování a také část GUI do které vypisují informační hlášky. Funkce je rozdělena do jednotlivých bloků dle části s kterou pracuje.

První část v obrázku číslo 52 se věnuje práci s klíčem a inicializačním vektorem. Sdílí práci s chybovými hláškami jako u AES, načítá klíč ve formě textu, nebo souboru dle výběrového pole. Klíč upravuje a validuje, při nevalidním vstupu přidá zprávu do error_messages. Dále načítá IV z textového pole, to upravuje a validuje velice podobně a následně posílá dál v případě validních vstupů.

```
def process_DES(self, value):
    # Initialize list to store error messages
    error_messages = []

    key_bytes = None
    iv_bytes = None

    # Check if the user wants to load the key from a file
    if self.ui.key_DEScombo.currentIndex() == 0:
        # Normal key input process
        key_text = self.ui.key_text_DES.toPlainText()
        try:
            key_bytes = string_to_bytes(key_text)
        except:
            # Handle base64 decoding error
            error_messages.append("Invalid key.")
    else:
        # Load key from file
        key_path = self.ui.key_file_DES.text()
        if not validate_file_path(key_path):
            # File path is invalid
            error_messages.append("Invalid key file path.")
        else:
            key_bytes = string_to_bytes(load_key_from_file(key_path))

    # Convert iv_text to bytes
    iv_text = self.ui.iv_text_DES.text()
    try:
        iv_bytes = string_to_bytes(iv_text)
    except:
        error_messages.append("Invalid IV.")

    if key_bytes is not None:
        if not validate_key_DES(key_bytes):
            # Key is invalid
            error_messages.append("Invalid key.")

    # Check if iv_bytes is not None and validate it
    if iv_bytes is not None:
        if not validate_iv_DES(iv_bytes):
            # IV is invalid
            error_messages.append("Invalid IV.")
```

Obrázek 52. Ukázka implementace funkce pro DES část první – klíč a IV

Část druhá v obrázku číslo 53 se zabývá šifrování a dešifrování souborů. Vezme cestu k souboru kterou validuje. Dále dle rozhodnutí jestli uživatel šifruje nebo dešifruje spouští encrypt_file_des nebo decrypt_file_des, v případě jiných algoritmů se tato funkce mění. Nakonec při úspěšném procesu o této skutečnosti uživatele informuje, kdyby se vyskytla nějaká chyba, přidává zprávu mezi další chybové hlášky.

```
# Check if it's encryption or decryption
if self.ui.input_DEScombo.currentIndex() == 1:
    input_path = self.ui.input_file_DES.text()
    if not validate_file_path(input_path):
        error_messages.append("Invalid input file path.")
    else:
        if not error_messages:
            try:
                if value == 0:
                    encrypt_file_des(input_path.encode('utf-8'), key_bytes, iv_bytes)
                    self.ui.info_label_DES.setText("File was successfully encrypted in input file path")
                else:
                    decrypt_file_des(input_path, key_bytes, iv_bytes)
                    self.ui.info_label_DES.setText("File was successfully decrypted in input file path")
            except:
                error_messages.append("Invalid input file path.")
```

Obrázek 53. Ukázka implementace funkce pro DES část druhá – vstup ve formě souboru

V třetí poslední části v obrázku číslo 54 se zabývá šifrováním a dešifrování textu. Dle hodnoty pracuje buď s šifrováním nebo dešifrováním. Vezme jednotlivý text a upraví ho dle šifrování nebo dešifrování, tento vstup validuje a poté pokud se žádná chyba nevyskytla volá encrypt_des nebo decrypt_des dle tlačítka. Nakonec dá uživateli vědět o úspěšném šifrování, dešifrování nebo přidá chybovou hlášku, kdyby se nějaká chyba vyskytla. Kdyby se vyskytla tak o ni nakonec uživatele informuje.

```
else:
    if value == 0:
        try:
            input_text = self.ui.input_text_DES.toPlainText().encode("utf-8")
            if not validate_input_data_AES(input_text):
                error_messages.append("Invalid input text.")
            else:
                if not error_messages:
                    iv_bytes, output_text = encrypt_des(input_text, key_bytes, iv_bytes)
                    self.ui.info_label_DES.setText("Text was successfully encrypted")
                    self.ui.output_text_DES.setText(bytes_to_string(output_text))
        except:
            error_messages.append("Invalid input text.")
    else:
        try:
            ciphertext = base64.b64decode(self.ui.input_text_DES.toPlainText().encode("utf-8"))
            if not validate_input_data_AES(ciphertext):
                error_messages.append("Invalid ciphertext.")
            else:
                if not error_messages:
                    output_text = decrypt_des(ciphertext, key_bytes, iv_bytes)
                    self.ui.info_label_DES.setText("Text was successfully decrypted")
                    self.ui.output_text_DES.setText(output_text.decode("utf-8"))
        except:
            error_messages.append("Invalid ciphertext.")

# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.info_label_DES.setText(error_message)
```

Obrázek 54. Ukázka implementace funkce pro DES část třetí – vstup ve formě textu

## 7.3 ChaCha20 k uživatelskému prostředí

V první části v obrázku číslo 55 kontroluje zda je nonce nebo klíč ve formě textu či souboru, poté tyto klíče načítá, validuje a upravuje. Mezitím se opět stará o chybové hlášky a v souborové části se stará o jejich načtení do proměnné a jejich správné validaci.

```
def process_ChaCha20(self, value):
    # Initialize list to store error messages
    error_messages = []

    # Load keys and nonce based on user selection
    if self.ui.key_ChaCha20combo.currentIndex() == 0:
        try:
            key_text = self.ui.key_text_ChaCha20.toPlainText()
            key_bytes = string_to_bytes(key_text)
            if len(key_bytes) != 32:
                error_messages.append("Invalid key.")
        except:
            error_messages.append("Invalid key.")
    else:
        try:
            key_path = self.ui.key_file_ChaCha20.text()
            key_bytes = string_to_bytes(load_key_from_file(key_path))
            if len(key_bytes) != 32:
                error_messages.append("Invalid key.")
        except:
            error_messages.append("Invalid key path.")

    if self.ui.nonce_ChaCha20combo.currentIndex() == 0:
        try:
            nonce_text = self.ui.nonce_text_ChaCha20.toPlainText()
            nonce_bytes = string_to_bytes(nonce_text)
            if len(nonce_bytes) != 16:
                error_messages.append("Invalid nonce.")
        except:
            error_messages.append("Invalid nonce.")
    else:
        try:
            nonce_path = self.ui.nonce_file_ChaCha20.text()
            nonce_bytes = string_to_bytes(load_key_from_file(nonce_path))
            if len(nonce_bytes) != 16:
                error_messages.append("Invalid nonce.")
        except:
            error_messages.append("Invalid nonce path.")
```

Obrázek 55. Ukázka implementace funkce pro ChaCha20
část první – klíč a nonce

V druhé části v obrázku číslo 56 se stará o šifrování a dešifrování ve formě souboru. První načítá cestu k souboru a mezitím ji validuje. Dále v případě že zde nejsou žádné chybové hlášky, tak postupuje buď k šifrování pomocí encrypt_file_chacha20 nebo dešifrování pomocí decrypt_file_chacha20. Nakonec uživatele informuje o úspěšném šifrování nebo dešifrování nebo přidá zprávu do chybových hlášek.

```
# Perform encryption or decryption based on user selection
if self.ui.input_ChaCha20combo.currentIndex() == 1:
    # Encryption operation
    if value == 0:
        input_path = self.ui.input_file_ChaCha20.text().encode('utf-8')
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            try:
                if not error_messages:
                    encrypt_file_chacha20(input_path, key_bytes, nonce_bytes)
                    self.ui.info_label_ChaCha20.setText("File was successfully encrypted in input file path")
            except:
                error_messages.append("Failed to encrypt file.")
    # Decryption operation
    else:
        input_path = self.ui.input_file_ChaCha20.text()
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            try:
                if not error_messages:
                    decrypt_file_chacha20(input_path, key_bytes, nonce_bytes)
                    self.ui.info_label_ChaCha20.setText("File was successfully decrypted in input file path")
            except:
                error_messages.append("Failed to decrypt file. Invalid input.")
```

Obrázek 56. Ukázka implementace funkce pro ChaCha20 část druhá – vstup ve formě souboru

Nakonec v třetí části v obrázku číslo 57 šifruje nebo dešifruje ve formě textu. Opět bere vstupní data textových polí, upravuje a validuje je. V případě že není žádná chybová hláška, tak pokračuje buď k šifrování pomocí encrypt_chacha20 nebo dešifrování pomocí decrypt_chacha20. Poté informuje uživatele o úspěšném šifrováním nebo dešifrováním v případě neúspěchu přidává zprávu do chybových hlášek. Nakonec v případě výskytu chybových hlášek o této skutečnosti informuje uživatele.

```
else:
    # Encryption or decryption based on user selection
    try:
        if value == 0:
            input_text = self.ui.input_text_ChaCha20.toPlainText().encode('utf-8')
            if not validate_input_data_AES(input_text):
                error_messages.append("Invalid input text.")
            if not error_messages:
                output_text = encrypt_chacha20(input_text, key_bytes, nonce_bytes)
                self.ui.output_text_ChaCha20.setText(bytes_to_string(output_text))
                self.ui.info_label_ChaCha20.setText("Text was successfully encrypted")
        else:
            input_text = base64.b64decode(self.ui.input_text_ChaCha20.toPlainText().encode('utf-8'))
            if not validate_input_data_AES(input_text):
                error_messages.append("Invalid input text.")
            if not error_messages:
                output_text = decrypt_chacha20(input_text, key_bytes, nonce_bytes)
                self.ui.output_text_ChaCha20.setText(output_text.decode('utf-8'))
                self.ui.info_label_ChaCha20.setText("Text was successfully decrypted")
    except:
        error_messages.append("Failed to encrypt or decrypt text. Invalid input.")
# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.info_label_ChaCha20.setText(error_message)
```

Obrázek 57. Ukázka implementace funkce pro ChaCha20 část třetí – vstup ve formě textu

## 7.4 RSA k uživatelskému rozhraní

V první části v obrázku číslo 58 se funkce stará o načítání klíčů do proměnných buď z textu nebo ze souboru. Při neúspěchu přidává chybovou hlášku.

```
def process_RSA(self, value):
    # Initialize list to store error messages
    error_messages = []

    # Load keys based on user selection
    if self.ui.key_RSAcombo.currentIndex() == 0:
        try:
            private_key_text = self.ui.private_key_text_RSA.toPlainText()
            public_key_text = self.ui.public_key_text_RSA.toPlainText()
            private_key, public_key = load_keys_from_text(private_key_text, public_key_text)
            private_key_bytes = private_key_text.encode('utf-8')
            public_key_bytes = public_key_text.encode('utf-8')
        except:
            error_messages.append("Invalid public or private key.")
    else:
        try:
            private_key_path = self.ui.private_key_file_RSA.text()
            public_key_path = self.ui.public_key_file_RSA.text()
            private_key_pem, public_key_pem = load_key_pair_from_files(private_key_path, public_key_path)
            private_key_bytes, public_key_bytes = serialize_to_pem(private_key_pem, public_key_pem)
        except:
            error_messages.append("Invalid public or private key.")
```

Obrázek 58. Ukázka implementace funkce pro RSA část první – klíče

V druhé části v obrázku číslo 59 se stará o šifrování nebo dešifrování ve formě souboru. Bere cestu kterou poté validuje včetně validace velikosti souboru, jejichž problém byl vysvětlen v části Specifikace a implementace algoritmů pod RSA. V případě že zde nejsou chybové hlášky, tak šifruje pomocí encrypt_file_RSA nebo dešifruje pomocí decrypt_file_RSA, nakonec dá uživateli vědět o úspěšném šifrování či dešifrování, nebo přidá chybovou hlášku.

```
# Perform encryption or decryption based on user selection
if self.ui.input_RSAcombo.currentIndex() == 1:
    # Encryption operation
    if value == 0:
        input_path = self.ui.input_file_RSA.text().encode('utf-8')
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            if os.path.getsize(input_path) > 200:  # Check if file size exceeds the limit
                error_messages.append("Input file size exceeds the limit for encryption.")
            else:
                try:
                    encrypt_file_RSA(input_path, public_key_bytes)
                    self.ui.info_label_RSA.setText("File was successfully encrypted in input file path")
                except:
                    error_messages.append("Failed to encrypt file.")
    # Decryption operation
    else:
        input_path = self.ui.input_file_RSA.text()
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            try:
                decrypt_file_RSA(input_path, private_key_bytes)
                self.ui.info_label_RSA.setText("File was successfully decrypted in input file path")
            except:
                error_messages.append("Failed to decrypt file. Invalid input.")
```

Obrázek 59. Ukázka implementace funkce pro RSA část druhá – vstupy ve formě souboru

V poslední třetí části v obrázku číslo 60 pracuje se šifrováním a dešifrováním ve formě textu. Načítá vstupy dle šifrování nebo dešifrování a následně je validuje opět společně s velikostí

vstupu při šifrování. Poté vstup šifruje pomocí encrypt_RSA či dešifruje pomocí decrypt_RSA a informuje o tom uživatele. V případě že se objeví jakákoliv chyba během těchto procesů, tak přidá chybovou hlášku. Nakonec vypíše všechny chybové hlášky uživateli pokud existují.

```
else:
    # Encryption operation
    if value == 0:
        input_text = self.ui.input_text_RSA.toPlainText().encode('utf-8')
        if not validate_input_data_AES(input_text):
            error_messages.append("Invalid input text.")
        else:
            if len(input_text) > 200:  # Check if input text size exceeds the limit
                error_messages.append("Input text size exceeds the limit for encryption.")
            else:
                try:
                    ciphertext = encrypt_RSA(input_text, public_key_bytes)
                    encoded_ciphertext = base64.b64encode(ciphertext)
                    self.ui.output_text_RSA.setText(encoded_ciphertext.decode('utf-8'))
                    self.ui.info_label_RSA.setText("Text was successfully encrypted")
                except:
                    error_messages.append("Failed to encrypt text.")
    # Decryption operation
    else:
        try:
            ciphertext = base64.b64decode(self.ui.input_text_RSA.toPlainText().encode('utf-8'))

            decrypted_text = decrypt_RSA(ciphertext, private_key_bytes)
            self.ui.output_text_RSA.setText(decrypted_text.decode('utf-8'))
            self.ui.info_label_RSA.setText("Text was successfully decrypted")
        except:
            error_messages.append("Failed to decrypt text. Invalid input.")

# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.info_label_RSA.setText(error_message)
```

Obrázek 60. Ukázka implementace funkce pro RSA část třetí – vstupy ve formě textu

## 7.5 DSA k uživatelskému rozhraní

V první části v obrázku číslo 61 se funkce stará o načítání klíčů buď ve formě textu nebo souborů. Při jakékoliv chybě vytvoří chybovou hlášku, jinak klíče posílá dál.

```
def process_DSA(self, value):
    # Initialize list to store error messages
    error_messages = []

    # Load keys based on user selection
    if self.ui.key_DSAcombo.currentIndex() == 0:
        try:
            private_key_text = self.ui.private_key_text_DSA.toPlainText()
            public_key_text = self.ui.public_key_text_DSA.toPlainText()
            private_key, public_key = load_keys_from_text(private_key_text, public_key_text)
        except:
            error_messages.append("Invalid public or private key.")
    else:
        try:
            private_key_path = self.ui.private_key_file_DSA.text()
            public_key_path = self.ui.public_key_file_DSA.text()
            private_key, public_key = load_key_pair_from_files(private_key_path, public_key_path)
        except:
            error_messages.append("Invalid public or private key.")
```

Obrázek 61. Ukázka implementace funkce pro DSA část první – klíče

V druhé části v obrázku číslo 62 se funkce stará o vytváření podpisu, či ověřování podpisu ve formě souborů. První bere cestu k souboru, kterou validuje. V případě vytváření podpisu se zeptá uživatele kam chce podpis uložit. Poté s touto cestou pracuje a dle sign_DSA_file podpis vytvoří a informuje o tom uživatele. V případě ověření bere i cestu k podpisu a pomocí verify_DSA_file určí, zda je ověření validní či nikoliv. Opět o tom uživatele informuje, v případě opačném přidává chybovou hlášku.

```
# Perform signing or verification based on user selection
if self.ui.input_DSAcombo.currentIndex() == 1:
    # Sign operation
    if value == 0:
        input_path = self.ui.input_file_DSA.text().encode('utf-8')
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            signature_path, _ = QFileDialog.getSaveFileName(self, "Save Signature", "", "Binary Files (*.bin)")
            try:
                sign_DSA_file(input_path, private_key, signature_path)
                set_Text(self.ui.sign_file_DSA, lambda: signature_path)
                self.ui.output_text_DSA.clear()
                self.ui.sign_verify_DSA.setText("Signature was created in input path.")
            except:
                error_messages.append("Failed to create signature.")
    # Verify operation
    else:
        input_path = self.ui.input_file_DSA.text().encode('utf-8')
        if not validate_file_path(input_path):
            error_messages.append("Invalid input file path.")
        else:
            signature_path = self.ui.sign_file_DSA.text()
            try:
                verified = verify_DSA_file(input_path, signature_path, public_key)
                if verified:
                    self.ui.sign_verify_DSA.setText("Signature verified.")
                else:
                    self.ui.sign_verify_DSA.setText("Signature is not verified.")
            except:
                error_messages.append("Failed to verify signature.")
```

Obrázek 62. Ukázka implementace funkce pro DSA část druhá – vstupy ve formě souboru

V třetí části v obrázku číslo 63 se funkce stará o vytváření podpisu, či ověřování podpisu ve formě textu. První načte jednotlivé vstupy a validuje ho. V případě vytváření podpisu tak učiní pomocí sign_dsa a výsledek vloží a informuje o tom uživatele. V případě ověření načte samotný podpis a a provede jej díky verify_dsa, poté o ověření informuje uživatele. V případě jakékoliv chyby během procesu přidává chybovou hlášku a nakonec je vypíše.

```
else:
    # Sign operation
    if value == 0:
        input_text = self.ui.input_text_DSA.toPlainText().encode('utf-8')
        if not validate_input_data_AES(input_text):
            error_messages.append("Invalid input text.")
        else:
            try:
                signature = sign_dsa(input_text, private_key)
                encoded_signature = base64.b64encode(signature)
                self.ui.output_text_DSA.setText(encoded_signature.decode('utf-8'))
                self.ui.sign_file_DSA.clear()
                self.ui.sign_verify_DSA.setText("Signature was created in text.")
            except:
                error_messages.append("Failed to create signature.")
    # Verify operation
    else:
        input_text = self.ui.input_text_DSA.toPlainText().encode('utf-8')
        if not validate_input_data_AES(input_text):
            error_messages.append("Invalid input text.")
        else:
            signature_text = self.ui.output_text_DSA.toPlainText()
            try:
                decoded_signature = base64.b64decode(signature_text)
                verified = verify_dsa(input_text, decoded_signature, public_key)
                if verified:
                    self.ui.sign_verify_DSA.setText("Signature verified.")
                else:
                    self.ui.sign_verify_DSA.setText("Signature is not verified.")
            except:
                error_messages.append("Failed to verify signature.")

# Display all error messages, if any
if error_messages:
    error_message = " ".join(error_messages)
    self.ui.sign_verify_DSA.setText(error_message)
```

Obrázek 63. Ukázka implementace funkce pro DSA část třetí – vstupy ve formě textu

## 7.6 Hashovacích algoritmů k uživatelskému rozhraní

Jedná se o jednu funkci, kterou pak jednotlivě volají vlastní hashovací algoritmy ukázka v obrázku číslo 64. Na vstupu bere samotnou hashovací funkci, která je použita. Bere typ vstupu a výstupu a dle toho pokračuje v procesu vytváření hashe. V první části upravuje vstup dle typu, to znamená validuje buď cestu k souboru nebo samotný text. Oba pak načítá do proměnné s kterou dále pracuje a vytváří z ní hash. V případě chybové hlášky je check roven hodnotě jedna a funkce končí chybovou hláškou. Dále dle typu výstupu výsledek buď vloží do textového pole nebo dá uživateli možnost vybrat cestu, kde výstup uloží jako soubor. Nakonec uživatele o úspěšném vytvoření hashe informuje, nebo ho informuje chybovou hláškou.

```
def get_hash(self, hash_func):
    try:
        check = 0
        input_type = self.ui.input_hashcombo.currentText()
        output_type = self.ui.output_hashcombo.currentText()
        if input_type == "Input text type":
            input_data = self.ui.input_text_hash.toPlainText()
            if not validate_input_data_AES(input_data):
                self.ui.info_label_hash.setText("Invalid input text.")
                check = 1
            else:
                hashed_text = hash_func(input_data.encode('utf-8'))
        elif input_type == "Input file type":
            file_path = self.ui.input_file_hash.text()
            if not validate_file_path(file_path):
                self.ui.info_label_hash.setText("Invalid input path.")
                check = 1
            else:
                with open(file_path, 'rb') as file:
                    data = file.read()
                hashed_text = hash_func(data)
        if check == 0:
            if output_type == "Output text type":
                self.ui.output_text_hash.setText(hashed_text)
                self.ui.info_label_hash.setText("Hash was successfully created.")
            elif output_type == "Output file type":
                output_file_path, _ = QFileDialog.getSaveFileName(self, "Save File", "", "Binary Files (*.bin)")
                if output_file_path:
                    with open(output_file_path, 'w') as file:
                        file.write(hashed_text)
                self.ui.info_label_hash.setText("Hash was successfully created.")
    except:
        self.ui.info_label_hash.setText("Invalid input")
```

Obrázek 64. Ukázka implementace funkce pro Hashovací algoritmy

## 7.7 Další funkce k uživatelskému rozhraní

Patří sem další funkce, které jsou využity v uživatelském rozhraní a nepracují s algoritmy, nebo ještě nebyly zmíněny.

### 7.7.1 Kopírování v AES

Jedná se o jednoduché funkce, jejichž účelem je vzít jednotlivé vstupy z jedné stránky, zkopírovat je a přenést do druhé. Na obrázku 65 je funkce copy_to_decrypt_AES, která bere zašifrovaný text, klíč, inicializační vektor a tag z šifrování a přenáší je do části pro dešifrování. Taktéž je přepínána hodnota na dešifrování a nastavuje se zobrazená stránka na dešifrování. Druhá funkce pod ní na obrázku 65 je copy_to_encrypt, která pracuje stejně, ale s dešifrovaným textem, klíčem a inicializačním vektorem ze stránky pro dešifrování.

```
#function for copy output text to decryption
def copy_to_decrypt_AES(self):
    ciphertext = self.ui.output_text_AES.toPlainText()
    key = self.ui.key_text_AES.toPlainText()
    iv = self.ui.iv_AES_encrypt.text()
    tag = self.ui.tag_AES_encrypt.text()

    self.ui.input_text_decrypt_AES.setText(ciphertext)
    self.ui.key_text_AES_2.setText(key)
    self.ui.iv_AES_decrypt.setText(iv)
    self.ui.tag_AES_decrypt.setText(tag)

    self.ui.stackedWidget.setCurrentWidget(self.ui.AES_Decrypt_page)
    self.ui.radioButton_4.setChecked(True)

def copy_to_encrypt_AES(self):
    output = self.ui.output_text_AES_2.toPlainText()
    key = self.ui.key_text_AES_2.toPlainText()
    iv = self.ui.iv_AES_decrypt.text()

    self.ui.input_text_encrypt_AES.setText(output)
    self.ui.key_text_AES.setText(key)
    self.ui.iv_AES_encrypt.setText(iv)

    self.ui.stackedWidget.setCurrentWidget(self.ui.AES_Encrypt_page)
    self.ui.radioButton.setChecked(True)
```

Obrázek 65. Ukázka funkcí na kopírování textu do jiné stránky

### 7.7.2 Načtení souboru skrze GUI

Na obrázku číslo 66 je zobrazena funkce load_file_into_textedit, ta pracuje s cestou souboru kterou uživatel zadá skrze dialogové okno a vrací ji do specifického texteditu. Využito při tlačítku Choose file.

```
#file operating function
def load_file_into_textedit(self, text_edit):
    # Prompt the user to select a file
    file_path, _ = QFileDialog.getOpenFileName(self, "Open File", "", "All Files (*)")

    if file_path:
        text_edit.setText(file_path)
    else:
        text_edit.setText(None)
```

Obrázek 66. Ukázka implementace načítání souboru do textového pole

### 7.7.3 Přepínání stránek pomocí radioboxu

Spadá sem funkce pro přepínání stránky z šifrování na dešifrování a opačně u AES jak je prezentováno na obrázku číslo 67. Dle typu souboru vybraného v šifrování nastaví v dešifrování na stejné hodnoty, přepne stránku na požadovanou a zároveň vizuálně změní radiobox na požadovanou hodnotu.

```
#routing functions for radioboxes
def changeToEncrypt(self):
    index_input = self.ui.input_AEScombo.currentIndex()
    index_key = self.ui.key_AEScombo.currentIndex()
    self.handle_AES_input_combo_changeEnc(index_input)
    self.handle_AES_key_combo_changeEnc(index_key)
    self.ui.stackedWidget.setCurrentWidget(self.ui.AES_Encrypt_page)
    self.ui.radioButton.setChecked(True)

def changeToDecrypt(self):
    index_input = self.ui.input_AEScombo_2.currentIndex()
    index_key = self.ui.key_AEScombo_2.currentIndex()
    self.handle_AES_input_combo_changeDec(index_input)
    self.handle_AES_key_combo_changeDec(index_key)
    self.ui.stackedWidget.setCurrentWidget(self.ui.AES_Decrypt_page)
    self.ui.radioButton_4.setChecked(True)
```

Obrázek 67. Ukázka funkcí pro změnu stránky u AES

### 7.7.4 Funkce pro přepínání výběrových polí

Dle obrázku 68 se jedná o handle_input_change, která bere postupně textové pole, tlačítko a hodnotu indexu která určuje zda je výběrové pole ve formě textu nebo souboru. Dle toho index nabývá hodnot 0 nebo 1. Dle toho nabývá hodnota input_enabled buď True nebo False. To vstupuje do funkce set_text_widget_state, dle toho odemkne nebo zablokuje dané textové pole, či tlačítko a v druhém odstraní text.

```
def handle_input_change(self, index, input_text_widget, input_file_widget, input_filebtn_widget):
    input_enabled = index == 0
    self.set_text_widget_state(input_text_widget, input_file_widget, input_filebtn_widget, input_enabled)

def set_text_widget_state(self, text_widget, file_widget, btn_widget, enabled):
    text_widget.setEnabled(enabled)
    file_widget.setEnabled(not enabled)
    btn_widget.setEnabled(not enabled)
    if enabled:
        file_widget.clear()
    else:
        text_widget.clear()
```

Obrázek 68. Ukázka funkcí pro změnu pomocí výběrového pole

Funkce handle_input_change je pak volána jak na obrázku číslo 69 dle specifických názvů textových polí a tlačítek. Index je pak tedy hodnota, která určuje typ vstupu.

```
def handle_RSA_input_combo_change(self, index):
    self.handle_input_change(
        index,
        self.ui.input_text_RSA,
        self.ui.input_file_RSA,
        self.ui.input_file_RSAbtn
    )
```

Obrázek 69. Příklad použití funkce handle_input_change

### 7.7.5 Přepínání menu

Funkce toggleMenu na obrázku číslo 70 buď menu schová nebo ukáže, následně přidá výplň o 20 pixelů aby zakryl mezeru, nebo ji vrátí jak byla.

```
def toggleMenu(self):
    if self.ui.left_side_menu.isVisible():
        self.ui.left_side_menu.hide()
        layout = self.ui.center_main_pages.layout()
        if layout:
            layout.setContentsMargins(120, 9, 9, 9)
    else:
        self.ui.left_side_menu.show()
        layout = self.ui.center_main_pages.layout()
        if layout:
            layout.setContentsMargins(9, 9, 9, 9)
```

Obrázek 70. Funkce pro přepínání menu

### 7.7.6 Volání jednotlivých funkcí do GUI

Všechny funkce, které jsou zmíněné v této kapitole je třeba nějakým způsobem volat, to samé co týká použití těchto funkcí a jejich napojení na specifické tlačítka.

Na obrázku číslo 71 můžeme vidět příklady pro přepínání jednotlivých stránek. Je zde volána funkce setCurrentWidget a u AES ještě nastavování hodnot u daných radioboxů.

```
self.ui.AES.clicked.connect(lambda: self.ui.stackedWidget.
                            setCurrentWidget(self.ui.AES_Encrypt_page))
self.ui.AES.clicked.connect(lambda: self.ui.radioButton.setChecked(True))
self.ui.DES3.clicked.connect(lambda: self.ui.stackedWidget.
                            setCurrentWidget(self.ui.DES3_page))

self.ui.DES.clicked.connect(lambda: self.ui.stackedWidget.
                            setCurrentWidget(self.ui.DES_page))

self.ui.Blowfish.clicked.connect(lambda: self.ui.stackedWidget.
                            setCurrentWidget(self.ui.BF_page))
```

Obrázek 71. Ukázka přepínání stránek v GUI

Dále na obrázku číslo 72 je zobrazeno jakým způsobem je manipulováno s jednotlivými částmi pro generování klíčů a šifrování, či dešifrování.

```
#buttons for Blowfish value = 0 for encryption; value = 1 for decryption
self.ui.encrypt_BFbtn.clicked.connect(lambda: self.process_BF(0))
self.ui.decrypt_BFbtn.clicked.connect(lambda: self.process_BF(1))
self.ui.generate_key_BFbtn.clicked.connect(self.generate_keyBF)
self.ui.generate_key_file_BFbtn.clicked.connect(self.generate_key_file_BF)
```

Obrázek 72. Ukázka použití process_BF, generate_keyBF a generate_key_file_BF

Podobně je použito volání například i pro načítání souboru do textového pole. V neposlední řadě na obrázku 73 je ukázána i manipulace s funkcemi na výběrová pole, zde je třeba zjistit jejich aktuální hodnoty a při spuštění programu jednotlivé prvky dle toho upravit.

```
#comboboxes
input_AES = self.ui.input_AEScombo.currentIndex()
key_AES = self.ui.key_AEScombo.currentIndex()
self.ui.input_AEScombo.currentIndexChanged.connect(self.handle_AES_input_combo_changeEnc)
self.ui.key_AEScombo.currentIndexChanged.connect(self.handle_AES_key_combo_changeEnc)
self.handle_AES_input_combo_changeEnc(input_AES)
self.handle_AES_key_combo_changeEnc(key_AES)
```

Obrázek 73. Použití funkcí pro výběrová pole

Celá třída je pak načítána z jiného souboru, který byl vygenerován po vytvoření samotného GUI popsanou v kapitole 6. To patří pod QMainWindow a funkcí __init__ ta je pak volána pro vytvoření okna v třídě main.

# 8 TESTOVÁNÍ APLIKACE

V této části práce jsou popsány jednotlivé testy provedené na aplikaci za cílem ověření správné funkcionality. Všechny texty jsou upravené pomocí base64 a .encode na utf-8 s výjimkou vstupu pro otevřený text.

## 8.1 Šifrování, dešifrování textu a generování textového klíče s IV u AES

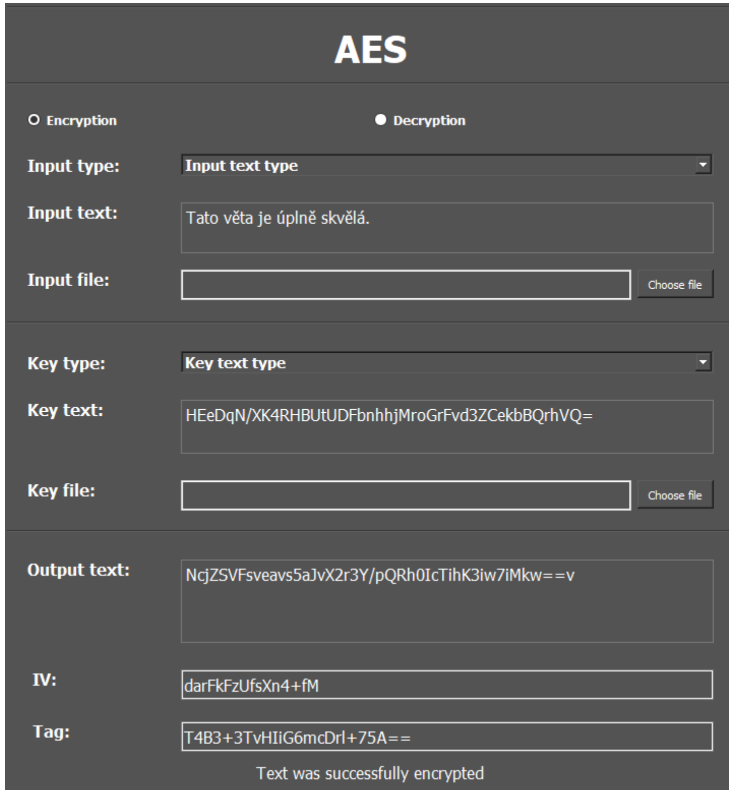
Test ověřuje správnost funkcionality šifrování a dešifrování textu pomocí AES v režimu GCM. Také generování klíče o velikosti 32 bytů, který je po úpravě o velikosti 44 bytů v samotném GUI. Generování také vytváří IV, ten je o velikosti 12 bytů, v textu 16 bytů.

**Popis testu:**

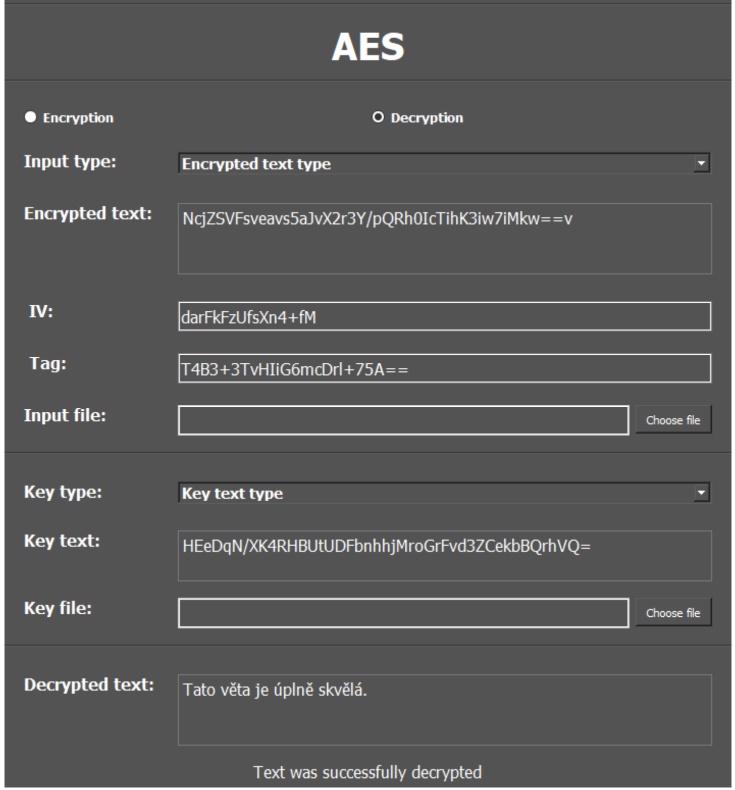
1. Vstupní text: „Tato věta je úplně skvělá.“
2. Vygenerování klíče a IV
3. Zašifrování textu a vytvoření Tag
4. Zkopírovat zašifrovaný text, klíč, IV a Tag na stránku pro dešifrování
5. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen a správně dešifrován a odpovídá původnímu vstupnímu textu.




Obrázek 74. Šifrování textu AES pro ukázku k bodu číslo 3




Obrázek 75. Dešifrování textu AES pro ukázku k bodu číslo 5

## 8.2 Šifrování, dešifrování souboru a generování klíče v souboru s IV u AES

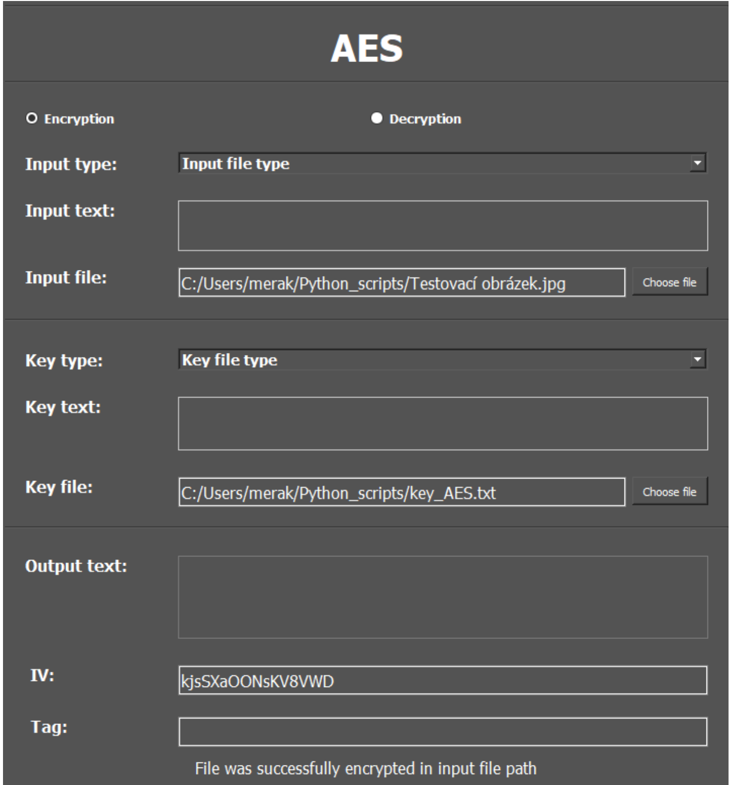
Test ověřuje správnost funkcionality šifrování a dešifrování souboru pomocí AES v režimu GCM. Také generování klíče jako soubor. Generování také vytváří IV, který je zapsán v textové podobě a poté uvnitř zašifrovaného souboru.

**Popis testu:**

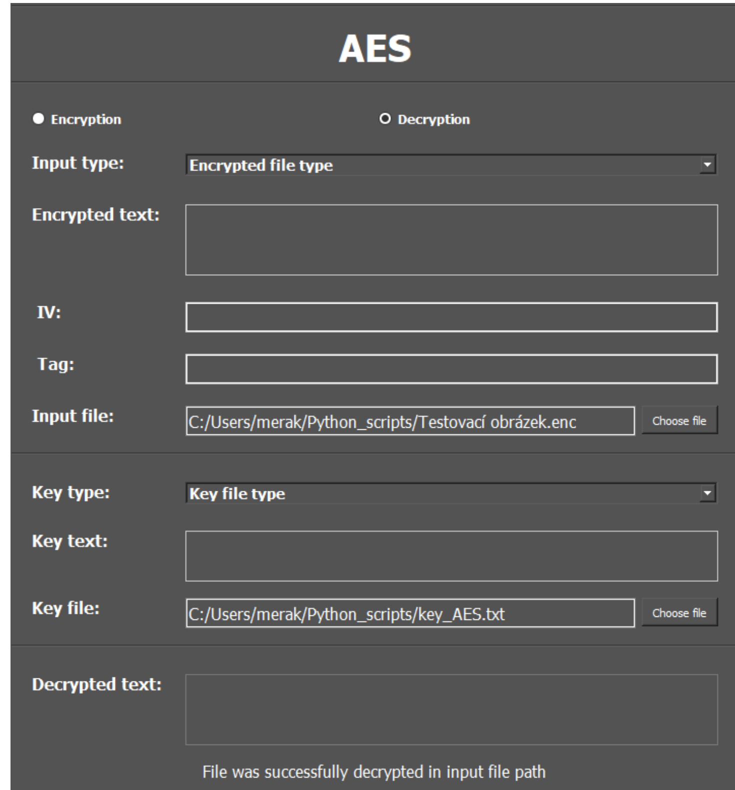
1. Vstupní soubor je obrázek ve formátu .jpg pod jménem „Testovací obrázek"
2. Vygenerování klíče jako soubor a IV
3. Zašifrování souboru a vytvoření Tag uvnitř souboru
4. Zkopírovat požadované cesty k souborů, na vstupu do stránky na dešifrování
5. Dešifrování souboru s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

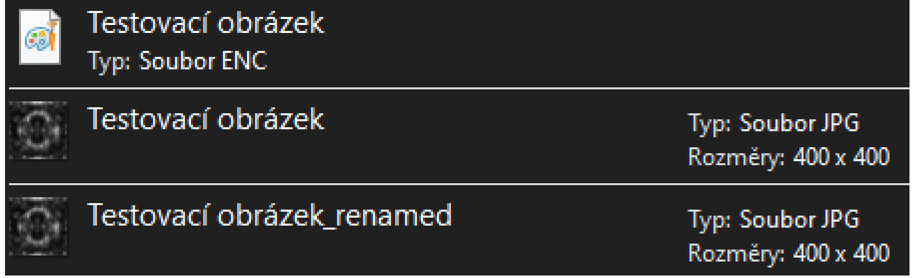
Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen ve formátu .enc, správně dešifrován a odpovídá původnímu vstupnímu souboru. Klíč byl vytvořen v textovém souboru ve formátu .txt.




Obrázek 76. Šifrování souboru AES pro ukázku k bodu číslo 3




Obrázek 77. Dešifrování souboru AES pro ukázku k bodu číslo 5




Obrázek 78. Ukázka originálního přejmenovaného, zašifrovaného a dešifrovaného souboru AES

## 8.3 Šifrování, dešifrování textu a generování textového klíče s IV u DES

Test ověřuje správnost funkcionality šifrování a dešifrování textu pomocí DES v režimu CBC. Také generování klíče o velikosti 8 bytů, který je po úpravě o velikosti 12 bytů v samotném GUI. Generování také vytváří IV, ten je o velikosti 8 bytů, v textu 12 bytů.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá."
2. Vygenerování klíče a IV
3. Zašifrování textu
4. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen a správně dešifrován a odpovídá původnímu vstupnímu textu.

## 8.4 Šifrování, dešifrování souboru a generování klíče v souboru s IV u DES

Test ověřuje správnost funkcionality šifrování a dešifrování souboru pomocí DES v režimu CBC. Také generování klíče jako soubor. Generování také vytváří IV, který je zapsán v textové podobě a poté uvnitř zašifrovaného souboru.

**Popis testu:**

1. Vstupní soubor je obrázek ve formátu .jpg pod jménem „Testovací obrázek"
2. Vygenerování klíče jako soubor a IV
3. Zašifrování souboru
4. Dešifrování souboru s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen ve formátu .enc, správně dešifrován a odpovídá původnímu vstupnímu souboru. Klíč byl vytvořen v textovém souboru ve formátu .txt.

## 8.5 Šifrování, dešifrování textu a generování textového klíče s IV u 3DES

Test ověřuje správnost funkcionality šifrování a dešifrování textu pomocí 3DES v režimu CBC. Také generování klíče o velikosti 24 bytů, který je po úpravě o velikosti 44 bytů v samotném GUI. Generování také vytváří IV, ten je o velikosti 8 bytů, v textu 12 bytů.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá."
2. Vygenerování klíče a IV
3. Zašifrování textu
4. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen a správně dešifrován a odpovídá původnímu vstupnímu textu.

## 8.6 Šifrování, dešifrování souboru a generování klíče v souboru s IV u 3DES

Test ověřuje správnost funkcionality šifrování a dešifrování souboru pomocí 3DES v režimu CBC. Také generování klíče jako soubor. Generování také vytváří IV, který je zapsán v textové podobě a poté uvnitř zašifrovaného souboru.

**Popis testu:**

1. Vstupní soubor je soubor word ve formátu .docx pod jménem „Testovací word soubor"
2. Vygenerování klíče jako soubor a IV
3. Zašifrování souboru
4. Dešifrování souboru s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen ve formátu .enc, správně dešifrován a odpovídá původnímu vstupnímu souboru. Klíč byl vytvořen v textovém souboru ve formátu .txt.

## 8.7 Šifrování, dešifrování textu a generování textového klíče s IV u Blowfish

Test ověřuje správnost funkcionality šifrování a dešifrování textu pomocí Blowfish v režimu CBC. Také generování klíče o velikosti 56 bytů, který je po úpravě o velikosti 88 bytů v samotném GUI. Generování také vytváří IV, ten je o velikosti 8 bytů, v textu 12 bytů.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá."
2. Vygenerování klíče a IV
3. Zašifrování textu
4. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen a správně dešifrován a odpovídá původnímu vstupnímu textu.

## 8.8 Šifrování, dešifrování souboru a generování klíče v souboru s IV u Blowfish

Test ověřuje správnost funkcionality šifrování a dešifrování souboru pomocí Blowfish v režimu CBC. Také generování klíče jako soubor. Generování také vytváří IV, který je zapsán v textové podobě a poté uvnitř zašifrovaného souboru.

**Popis testu:**

1. Vstupní soubor je soubor word ve formátu .docx pod jménem „Testovací word soubor"
2. Vygenerování klíče jako soubor a IV
3. Zašifrování souboru
4. Dešifrování souboru s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen ve formátu .enc, správně dešifrován a odpovídá původnímu vstupnímu souboru. Klíč byl vytvořen v textovém souboru ve formátu .txt.

## 8.9 Šifrování, dešifrování textu a generování textových klíču u RSA

Test ověřuje správnost funkcionality šifrování a dešifrování textu. Také generování klíče v textové formě v PEM formátu.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá.“
2. Vygenerování soukromého a veřejného klíče
3. Zašifrování textu
4. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen, správně dešifrován a odpovídá původnímu vstupnímu textu.

## 8.10 Šifrování, dešifrování souboru a generování klíčů v souboru u RSA

Test ověřuje správnost funkcionality šifrování a dešifrování souborů. Také generování klíče v souboru v PEM formátu.

**Popis testu:**

1. Vstupní soubor je klíč AES v .txt pod jménem „key_AES“
2. Vygenerování soukromého a veřejného klíče
3. Zašifrování souboru
4. Dešifrovat soubor s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen, správně dešifrován a odpovídá původnímu vstupnímu souboru.

## 8.11 Šifrování, dešifrování textu a generování textového klíče a nonce u ChaCha20

Test ověřuje správnost funkcionality šifrování a dešifrování textu. Také generování klíče o velikosti 32 bytů v textové formě o 44 bytech a nonce o velikosti 16 bytů v textové formě o 24 bytech.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá.“
2. Vygenerování klíče a nonce v textové formě
3. Zašifrování textu
4. Dešifrovat text s očekávaným výsledkem vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný text byl vytvořen, správně dešifrován a odpovídá původnímu vstupnímu textu.

## 8.12 Šifrování, dešifrování souboru a generování klíče a nonce v souboru u ChaCha20

Test ověřuje správnost funkcionality šifrování a dešifrování souboru. Také generování klíče a nonce jako soubory v .txt formátu.

**Popis testu:**

1. Vstupní soubor je obrázek v .jpg pod jménem „Testovací obrázek“
2. Vygenerování klíče a nonce ve formě souborů .txt
3. Zašifrování souboru
4. Dešifrovat soubor s očekávaným výsledkem vstupního souboru z bodu 1

**Závěr:**

Test byl úspěšně dokončen, zašifrovaný soubor byl vytvořen, správně dešifrován a odpovídá původnímu vstupnímu souboru.

## 8.13 Vytváření podpisu, ověřování a generování klíčů v textové formě u DSA

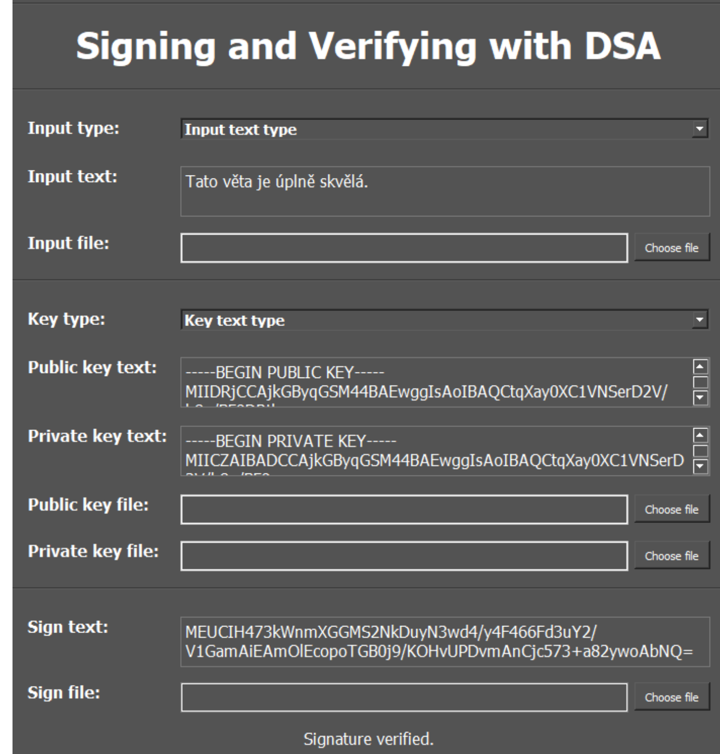
Test ověřuje správnost funkcionality vytváření podpisu a jeho ověřování. Také generování soukromého a veřejného klíče v textové formě v PEM formátu.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá."
2. Vygenerování soukromého a veřejného klíče
3. Vytvoření podpisu
4. Ověření podpisu dle vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, podpis byl vytvořen a správně ověřen.




Obrázek 79. Vytvořený a ověřený podpis
v textové formě u DSA

## 8.14 Vytváření podpisu, ověřování a generování klíčů ve formě souboru u DSA

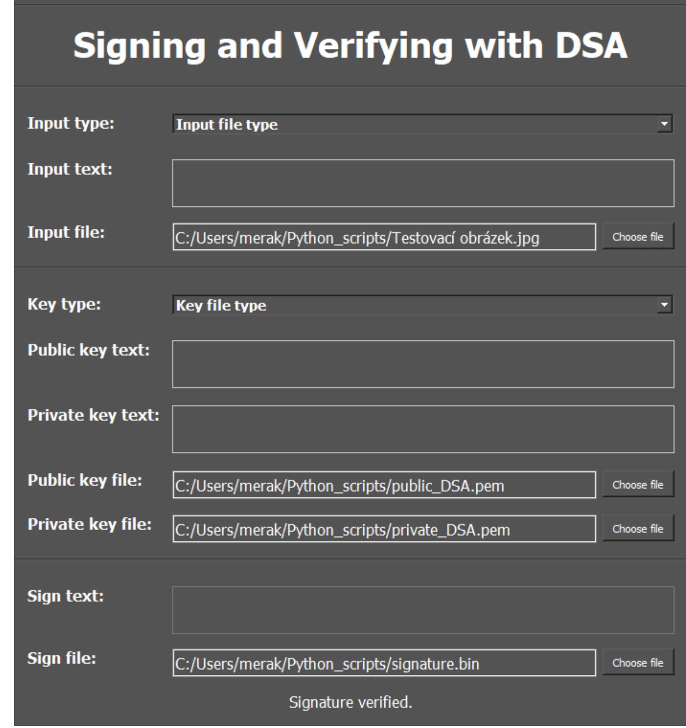
Test ověřuje správnost funkcionality vytváření podpisu a jeho ověřování. Také generování soukromého a veřejného klíče ve formě souboru v PEM formátu.

**Popis testu:**

1. Vstupní soubor soubor je obrázek v .jpg pod jménem „Testovací obrázek“
2. Vygenerování soukromého a veřejného klíče
3. Vytvoření podpisu
4. Ověření podpisu dle vstupního textu z bodu 1

**Závěr:**

Test byl úspěšně dokončen, podpis byl vytvořen a správně ověřen.




Obrázek 80. Vytvořený a ověřený podpis ve formě souboru u DSA

## 8.15 Vytvoření hashe jako text ze souboru u SHA-1, SHA-256, SHA-3 a MD5

Test ověřuje správnost funkcionality vytváření hashe ze souboru do textové formy u SHA-1, SHA-256, SHA-3 a MD5

**Popis testu:**

1. Vstupní soubor soubor je obrázek v .jpg pod jménem „Testovací obrázek“
2. Vytvoření hashe dle SHA-1
3. Vytvoření hashe dle SHA-256
4. Vytvoření hashe dle SHA-3
5. Vytvoření hashe dle MD5

**Výsledky hashovacích algoritmů:**

SHA-1: „08756cd2650203c449ab597dafecf1c8ca925b48“

SHA-256: „ada335c43d0e5fe840b0582398e07b0650327eb5b20138d135bceaa1df8f8cdc“

SHA-3: „8f449ee9d3ae8f93c3605337a33fb413b5ba523d413dcd61e9b2b3996d8280ea“

MD5: „8f449ee9d3ae8f93c3605337a33fb413b5ba523d413dcd61e9b2b3996d8280ea“

**Závěr:**

Test byl úspěšně dokončen, z obrázku byly vytvořeny hashe dle jednotlivých hashovacích algoritmů.

## 8.16 Vytvoření hashe jako soubor z textu u SHA-1, SHA-256, SHA-3 a MD5

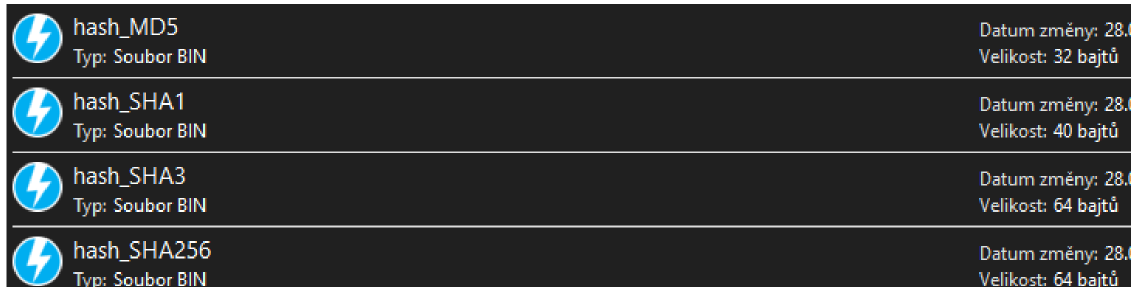
Test ověřuje správnost funkcionality vytvoření hash souboru z textu u SHA-1, SHA-256, SHA-3 a MD5.

**Popis testu:**

1. Vstupní text: „Tato věta je úplně skvělá."
2. Vytvoření hash souboru dle SHA-1
3. Vytvoření hash souboru dle SHA-256
4. Vytvoření hashe souboru dle SHA-3
5. Vytvoření hashe souboru dle MD5

**Závěr:**

Test byl úspěšně dokončen, ze vstupního textu byly vytvořeny hashe dle jednotlivých hashovacích algoritmů.




Obrázek 81. Vytvořené hash soubory z textu

# ZÁVĚR

Cílem práce bylo seznámit čtenáře převážně s problematikou kryptografie a seznámit je s jednotlivými algoritmy. Následně vytvořit aplikaci kryptografické kalkulačky kterou by mohl uživatel jak zkušenější, tak nováček využívat pohodlně bez potřeby psaní různých příkazů za využití vlastního uživatelského rozhraní.

V teoretické části byly představeny jak historické tak moderní šifry, včetně těch implementovaných v praktické části. Vysvětlen byl převážně jejich princip a jak fungují s obecným náhledem.

V praktické je pak rozebrána samotná aplikace do detailu, včetně jednotlivých funkcí, vytváření uživatelského rozhraní i jednotlivé manuální testy. Ty mohou taktéž sloužit pro prezentování funkcionality aplikace kryptografické kalkulačky a jak s ní může uživatel pracovat. Program má samozřejmě ještě co zlepšovat, je zde možné vytvářet různé velikosti klíčů, což v aplikaci zavedené není. Následně bylo možné lépe provést vizuální prvky jako animace menu a jak se chová. Dalším velkým nedostatkem je vstup klíčů, jelikož když jsou generovány, tak jednotlivý klíč má určitý počet bytů a skrze base64 a utf-8 nafoukne na víc. Tudíž neodpovídá původní velikosti pro uživatele, což může být poněkud matoucí. Taktéž je možné že pro lepší bezpečnost a lepší logiku za programem bylo lepší používat .bin formát klíčů u souborů, avšak volba tak byla spíše učiněna pro kontrolu klíče a podobné. Další velkou věcí je ohledně RSA která přijímá jen omezenou velikost souborů skrze klíče, což by určitě šlo zlepšit a tento nedostatek odstranit. Bohužel spoustu věcí které byly možné provést byly rozhodnuty jinak buď skrze nějaký specifický důvod, nadměrnému časovému vytížení její implementace nebo nalezení tohoto problému již v pokročilém stavu programu.

# SEZNAM POUŽITÉ LITERATURY

[1] KLIMA, Richard E.; KLIMA, Richard a SIGMON, Neil P. *Cryptology Classical and Modern*. 2nd Edition. New York: Chapman and Hall/CRC, 2018. ISBN 9781315170664.

[2] SINGH, Simon; KOUBSKÝ, Petr a ECKHARDTOVÁ, Dita. *Kniha kódů a šifer 2. vydání Tajná komunikace od starého Egypta po kvantovou kryptografii*. 2. vydání. Argo, 2009. ISBN 978-80-7363-268-7.

[3] RICHARDS, Kathleen. *Cryptography*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/cryptography. [cit. 2024-04-30].

[4] ROSENCRANCE, Linda. *Cryptoanalysis*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/cryptanalysis. [cit. 2024-02-19].

[5] SHELDON, Robert. *Encryption*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/encryption. [cit. 2024-02-19].

[6] LOSHIN, Peter a ROSENCRANCE, Linda. *Cipher*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/cipher. [cit. 2024-02-19].

[7] KATZ, Jonathan a LINDELL, Yehuda. *Introduction to Modern Cryptography*. Boca Raton: CRC Press, 2007. ISBN 9780429143809.

[8] VERMA, Anjlee a KAUR, Navjot. A Comparative Study of Classical Substitution Ciphers. *International Journal of Engineering Research & Technology (IJERT)*. 2014, s. 360-364. ISSN 2278-0181.

[9] NECKÁŘ, Jan. *Afinní šifra*. Online. C2016-2024. Dostupné z: https://www.algoritmy.net/article/49/Afinni-sifra. [cit. 2024-02-20].

[10] JANČÍK, David. *Ukázka Vigenerova šifra*. Online. C2024. Dostupné z: https://www.itnetwork.cz/algoritmy/kryptografie/algoritmus-ukazka-sifrace-textu-pomoci-vigenerovi-sifry. [cit. 2024-02-21].

[11] KOWALCZYK, Chris. *Polygraphic Substitution Ciphers*. Online. 2020. Dostupné z: https://www.crypto-it.net/eng/simple/polygraphic-substitution-ciphers.html. [cit. 2024-02-21].

[12] KOWALCZYK, Chris. *Playfair Cipher*. Online. 2020. Dostupné z: https://www.crypto-it.net/eng/simple/playfair-cipher.html. [cit. 2024-02-22].

[13] SIMMONS, Gustavus J. *Cryptology: Cipher systems*. Online. C2024. Dostupné z: https://www.britannica.com/topic/cryptology/Cryptography#ref233468. [cit. 2024-02-25].

[14] KOWALCZYK, Chris. *Columnar Transposition*. Online. 2020. Dostupné z: https://www.crypto-it.net/eng/simple/columnar-transposition.html. [cit. 2024-02-27].

[15] *ADFGX – ADFGVX*. Online. 2005. Dostupné z: http://soutez2005.crypto-world.info/images/ADFGX.pdf. [cit. 2024-02-27].

[16] LYONS, James. *ADFGX Cipher*. Online. C2009-2012. Dostupné z: http://practicalcryptography.com/ciphers/adfgx-cipher/. [cit. 2024-02-28].

[17] COBB, Michael. *Symmetric vs. Asymmetric encryption: What's the difference?* Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/answer/What-are-the-differences-between-symmetric-and-asymmetric-encryption-algorithms. [cit. 2024-03-05].

[18] AWATI, Rahul; BERNSTEIN, Corinne a COBB, Michael. *Advanced Encryption Standard*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard. [cit. 2024-03-07].

[19] AWATI, Rahul. *Blowfish*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/Blowfish. [cit. 2024-03-08].

[20] National Institute of Standards and Technology (NIST). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC Online. 2007. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf. [cit. 2024-04-04]

[21] National Institute of Standards and Technology (NIST). Recommendation for Block Cipher Modes of Operation: Methods and Techniques Online. 2001. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf. [cit. 2024-04-04]

[22] National Institute of Standards and Technology (NIST). Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography [online]. 2019 [cit. 2024-04-04]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Br2.pdf

[23] CRAWFORD, Douglas. *What is ChaCha20?* Online. 2023. Dostupné z: https://protonvpn.com/blog/chacha20/. [cit. 2024-04-04].

[24] National Institute of Standards and Technology (NIST). Hash Functions [online]. 2023. Dostupné z: https://csrc.nist.gov/projects/hash-functions. [cit. 2024-04-04]

[25] SPYDER DOC CONTRIBUTORS. *Where to Spyder's Documentation*. Online. C2009-2024. Dostupné z: https://docs.spyder-ide.org/current/index.html. [cit. 2024-04-08].

[26] KARTHIKEYAN, Nagaraj. *Understanding ChaCha20 Encryption: A Secure and Fast Algorithm for Data Protection | 2023*. Online. 2023. Dostupné z: https://cyberw1ng.medium.com/understanding-chacha20-encryption-a-secure-and-fast-algorithm-for-data-protection-2023-a80c208c1401. [cit. 2024-04-22].

[27] COBB, Michael. *RSA algorithm (Rivest-Shamir-Adleman)*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/RSA. [cit. 2024-04-23].

[28] *Digital Signature Algorithm (DSA)*. Online. 2024. Dostupné z: https://www.geeksforgeeks.org/digital-signature-algorithm-dsa/. [cit. 2024-04-23].

[29] *Digital Signature Algorithm (DSA)*. Online. C2000-2024. Dostupné z: https://crypto.ro/en/dictionary/digital-signature-algorithm-dsa/. [cit. 2024-04-23].

[30] SHACKLETT, Mary E. a LOSHIN, Peter. *MD5*. Online. C2000-2024. Dostupné z: https://www.techtarget.com/searchsecurity/definition/MD5. [cit. 2024-04-24].

[31] *Secure Hash Algorithms*. Online. 2024. Dostupné z: https://brilliant.org/wiki/secure-hashing-algorithms/. [cit. 2024-04-24].

[32] *SHA-256 algorithm*. Online. C2009-2024. Dostupné z: https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm. [cit. 2024-04-24].

[33] National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication 180-4 (FIPS PUB 180-4): Secure Hash Standard

(SHS) Online. 2015. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. [cit. 2024-04-24]

[34] National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication 202 (FIPS PUB 202): SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions Online. 2015. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. [cit. 2024-04-24]

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

| | |
|---|---|
| AES | Advanced Encryption Standard |
| CBC | Cipher Block Chaining |
| DES | Data Encryption Standard |
| DSA | Digital Signature Algorithm |
| GCM | Galois/Counter Mode |
| GUI | Graphical User Interface |
| HTTPS | Hypertext Transfer Protocol Secure |
| IBM | International Business Machines Corporation |
| IDE | Integrated Development Environment |
| IDEA | International Data Encryption Algorithm |
| IV | Initialization Vector |
| MGF | Mask Generation Function |
| MIT | Massachusetts Institute of Technology |
| NIST | National Institute of Standards and Technology |
| NSA | National Security Agency |
| OAEP | Optimal Asymmetric Encryption Padding |
| PEM | Privacy Enhanced Mail |
| RSA | Rivest-Shamir-Adleman |
| S/MIME | Secure/Multipurpose Internet Mail Extensions |
| SSL | Secure Sockets Layer |
| SSH | Secure Shell |
| TLS | Transport Layer Security |

# SEZNAM OBRÁZKŮ

Obrázek 1. Ukázka Caesarovy šifry [2]....15
Obrázek 2. Ukázka Vigenerova čtverce [2]....18
Obrázek 3. Ukázka principu Skytale [2]....22
Obrázek 4. Ukázka implementace funkcí encrypt_aes a decrypt_aes....36
Obrázek 5. Ukázka implementace funkce encrypt_file_aes....37
Obrázek 6. Ukázka implementace funkce decrypt_file_aes....38
Obrázek 7. Ukázka implementace encrypt_des a decrypt_des....39
Obrázek 8. Ukázka implementace funkce encrypt_file_des....40
Obrázek 9. Ukázka implementace funkce decrypt_file_des....40
Obrázek 10. Ukázka implementace encrypt_3des a decrypt_3des....41
Obrázek 11. Ukázka implementace funkcí encrypt_file_3des a decrypt_file_3des...42
Obrázek 12. Ukázka implementace funkcí encrypt_BF a decrypt_BF....43
Obrázek 13. Ukázka implementace funkcí encrypt_file_bf a decrypt_file_bf....44
Obrázek 14. Ukázka implementace funkcí encrypt_RSA a decrypt_RSA....45
Obrázek 15. Ukázka implementace funkcí encrypt_file_RSA a decrypt_file_RSA..46
Obrázek 16. Ukázka implementace funkcí encrypt_chacha20 a decrypt_chacha20..47
Obrázek 17. Ukázka implementace funkce encrypt_file_chacha20....48
Obrázek 18. Ukázka implementace funkce decrypt_file_chacha20....48
Obrázek 19. Ukázka implementace funkcí sign_dsa a verify_dsa....49
Obrázek 20. Ukázka implementace funkcí sign_DSA_file a verify_DSA_file....50
Obrázek 21. Ukázka implementace funkce calculate_md5....51
Obrázek 22. Ukázka implementace funkce calculate_sha3_256....51
Obrázek 23. Ukázka implementace funkcí calculate_sha256 a calculate_sha1....52
Obrázek 24. Ukázka implementace generování klíče a IV pro AES v GUI....53
Obrázek 25. Ukázka implementace generování klíče pro AES....53
Obrázek 26. Ukázka implementace generování klíče v souboru pro AES v GUI....54
Obrázek 27. Ukázka implementace funkce pro generování klíče v souboru....54
Obrázek 28. Ukázka implementace funkce pro generování klíče a IV pro DES v GUI ....55
Obrázek 29. Ukázka implementace funkce pro generování klíče v souboru pro DES v GUI....55
Obrázek 30. Ukázka implementace generování klíčů pro RSA v GUI....56

Obrázek 31. Ukázka implementace generování klíčů pro RSA ................................56
Obrázek 32. Ukázka implementace generování klíčů v souboru pro RSA v GUI .....57
Obrázek 33. Ukázka implementace generování klíčů v souboru pro RSA ...............57
Obrázek 34. Ukázka implementace generování klíčů pro DSA v GUI .....................57
Obrázek 35. Ukázka a implementace generování klíčů u DSA................................58
Obrázek 36. Ukázka a implementace generování klíčů v souboru pro RSA v GUI...58
Obrázek 37. Ukázka implementace generování klíčů v souboru pro RSA ...............59
Obrázek 38. Struktura prvků v GUI.........................................................................61
Obrázek 39. Ukázka stránky AES pro šifrování......................................................63
Obrázek 40. Ukázka stránky AES pro dešifrování...................................................64
Obrázek 41. Ukázka stránky DES ...........................................................................65
Obrázek 42. Ukázka stránky RSA ...........................................................................66
Obrázek 43. Ukázka stránky ChaCha20 ..................................................................67
Obrázek 44. Ukázka stránky DSA............................................................................68
Obrázek 45. Ukázka stránky Hash functions............................................................69
Obrázek 46. Ukázka implementace funkce pro šifrování AES část první – klíč........70
Obrázek 47. Ukázka implementace funkce pro šifrování AES část druhá – IV.........71
Obrázek 48. Ukázka implementace funkce pro šifrování AES část třetí – šifrování souboru ...........................................................................................71
Obrázek 49. Ukázka implementace funkce pro šifrování AES část čtvrtá – šifrování textu ..............................................................................................72
Obrázek 50. Ukázka implementace funkce pro dešifrování AES část první – dešifrování souboru ..............................................................................72
Obrázek 51. Ukázka implementace funkce pro dešifrování AES část druhá – dešifrování textu .....................................................................................73
Obrázek 52. Ukázka implementace funkce pro DES část první – klíč a IV..............74
Obrázek 53. Ukázka implementace funkce pro DES část druhá – vstup ve formě souboru ...........................................................................................75
Obrázek 54. Ukázka implementace funkce pro DES část třetí – vstup ve formě textu ......................................................................................................75
Obrázek 55. Ukázka implementace funkce pro ChaCha20 část první – klíč a nonce 76
Obrázek 56. Ukázka implementace funkce pro ChaCha20 část druhá – vstup ve formě souboru ...........................................................................................77

Obrázek 57. Ukázka implementace funkce pro ChaCha20 část třetí – vstup ve formě textu ........ 77
Obrázek 58. Ukázka implementace funkce pro RSA část první – klíče ........ 78
Obrázek 59. Ukázka implementace funkce pro RSA část druhá – vstupy ve formě souboru ........ 78
Obrázek 60. Ukázka implementace funkce pro RSA část třetí – vstupy ve formě textu ........ 79
Obrázek 61. Ukázka implementace funkce pro DSA část první – klíče ........ 79
Obrázek 62. Ukázka implementace funkce pro DSA část druhá – vstupy ve formě souboru ........ 80
Obrázek 63. Ukázka implementace funkce pro DSA část třetí – vstupy ve formě textu ........ 81
Obrázek 64. Ukázka implementace funkce pro Hashovací algoritmy ........ 82
Obrázek 65. Ukázka funkcí na kopírování textu do jiné stránky ........ 83
Obrázek 66. Ukázka implementace načítání souboru do textového pole ........ 83
Obrázek 67. Ukázka funkcí pro změnu stránky u AES ........ 84
Obrázek 68. Ukázka funkcí pro změnu pomocí výběrového pole ........ 84
Obrázek 69. Příklad použití funkce handle_input_change ........ 84
Obrázek 70. Funkce pro přepínání menu ........ 85
Obrázek 71. Ukázka přepínání stránek v GUI ........ 85
Obrázek 72. Ukázka použití process_BF, generate_keyBF a generate_key_file_BF 85
Obrázek 73. Použití funkcí pro výběrová pole ........ 86
Obrázek 74. Šifrování textu AES pro ukázku k bodu číslo 3 ........ 88
Obrázek 75. Dešifrování textu AES pro ukázku k bodu číslo 5 ........ 88
Obrázek 76. Šifrování souboru AES pro ukázku k bodu číslo 3 ........ 89
Obrázek 77. Dešifrování souboru AES pro ukázku k bodu číslo 5 ........ 90
Obrázek 78. Ukázka originálního přejmenovaného, zašifrovaného a dešifrovaného souboru AES ........ 90
Obrázek 79. Vytvořený a ověřený podpis v textové formě u DSA ........ 97
Obrázek 80. Vytvořený a ověřený podpis ve formě souboru u DSA ........ 98
Obrázek 81. Vytvořené hash soubory z textu ........ 99

# SEZNAM TABULEK

Tabulka 1. Playfair pravidlo řádkové ........................................................................19
Tabulka 2. Playfair pravidlo sloupcové ......................................................................20
Tabulka 3. Playfair pravidlo křížové ..........................................................................20
Tabulka 4. Ukázka Rail-fence .....................................................................................22
Tabulka 5. Ukázka otevřeného textu u Transpozice s klíčem ...................................23
Tabulka 6. Ukázka zašifrovaného textu u Transpozice s klíčem. ...............................23
Tabulka 7. Ukázka Zubatky.........................................................................................24
Tabulka 8. Ukázka tabulky ADFGVX šifry ..............................................................25
Tabulka 9. ADFGVX s klíčem HYBRIDNI před transpozicí ...................................25
Tabulka 10. ADFGVX s klíčem HYBRIDNI po transpozici ....................................26

# SEZNAM PŘÍLOH

PŘÍLOHA P I: OBSAH CD

# PŘÍLOHA P I: OBSAH CD

Struktura obsahu CD:

- Text bakalářské práce v elektronické formě
- Adresář Zdrojové kódy – obsahuje všechny zdrojové kódy, soubor s grafickým rozhraním a dodatečný obrázek pro grafické rozhraní
- Adresář CryptographicCalculator – obsahuje spustitelnou verzi programu ve formátu .exe a potřebným obrázkem ke grafickému rozhraní