

Vzorová implementace pokročilých kryptografických algoritmů

Bc. Jan Baslar

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Jan Baslar
Osobní číslo: A23330
Studijní program: N0613A140022 Informační technologie
Specializace: Kybernetická bezpečnost
Forma studia: Prezenční
Téma práce: Vzorová implementace pokročilých kryptografických algoritmů
Téma práce anglicky: Sample Implementation of Advanced Cryptographic Algorithms

Zásady pro vypracování

- Popište problematiku v kontextu diplomové práce.
- Navrhněte aplikaci včetně popisu požadavků.
- Zvolte vhodné technologie pro implementaci.
- Vyberte vhodné algoritmy moderní kryptografie pro vzorovou implementaci.
- Implementujte vybrané algoritmy pomocí zvolených technologií.
- Aplikaci vhodně otestujte a popište výsledky práce.

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

Seznam doporučené literatury:

1. SINGH, Simon; ECKHARTOVÁ, Dita a KOUBSKÝ, Petr. Kniha kódů a šifer. 3. vydání. Praha: Dokořán, 2017. ISBN 9788073632687.
2. KATZ, Jonathan a LINDELL, Yehuda. Introduction to modern cryptography. Third edition. Boca Raton: CRC Press, 2021. ISBN 978-0-8153-5436-9.
3. BURDA, Karel. Úvod do kryptografie. Brno: Akademické nakladatelství CERM, 2015. ISBN 9788072049257.
4. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 9797-1:2011, Information technology — Security techniques — Message Authentication Codes (MACs). Edition 2. 2011.
5. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 9564-1:2017, Financial services — Personal Identification Number (PIN) management and security. Edition 4. 2017.
6. GOOGLE LLC. Flutter documentation. Online. 2024. Dostupné z: <https://docs.flutter.dev/>. [cit. 2024-10-23].
7. GOOGLE LLC. Dart documentation. Online. 2024. Dostupné z: <https://dart.dev/guides>. [cit. 2024-10-23].

Vedoucí diplomové práce:

Ing. Petr Žáček, Ph.D.
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**

Termín odevzdání diplomové práce: **2. června 2025**

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



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

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou 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;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové 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 diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Diplomová práce se zaměřuje na projekt „Kryptokalkulačky“ - vzorové implementace pokročilých moderních kryptografických algoritmů - pomocí technologie Flutter, pro lokální firmu, jelikož externí řešení už není pro firemní potřeby dostatečné. Kromě klasických algoritmů jako DES a AES by se práce zaměřovala na kryptografii specifickou pro bankovní sektor, jako MAC algoritmy a další. Rozsah práce bude minimálně 4 až 5 algoritmů a celkově se bude jednat o jednotnou aplikaci.

Klíčová slova: Kryptografie, Multiplatformní aplikace, Dart, Flutter

ABSTRACT

The thesis focuses on the project "Cryptocalculator" - a sample implementation of advanced modern cryptographic algorithms - using Flutter technology for a local company, as external solutions are no longer sufficient for the company's needs. In addition to classical algorithms such as DES and AES, the work would focus on cryptography specific to the banking sector, such as MAC algorithms and others. The scope of the work will be at least 4 to 5 algorithms and the total will be a single application.

Keywords: Cryptography, Multiplatform apps, Dart, Flutter

Děkuji Ing. Petru Žáčkovi, Ph.D. za vedení této diplomové práce a za veškerou zpětnou vazbu, kterou mi průběžně s ochotou poskytoval. Dále svým kolegům z firmy Monet+, jmenovitě teamleaderům Tomáši Jačkovi, kterým se mnou projekt průběžně konzultoval a poskytl nejednu věcnou připomínku a Pavlu Hoke, za předání cenných rad a zkušeností. V neposlední řadě své rodině, která mě za dobu mého studia podporovala jak psychicky, tak finančně a svým přátelům, kteří se mnou měli schovívavost v těchto nelehkých časech.

OBSAH

ÚVOD.....	13
I TEORETICKÁ ČÁST	14
1 DES (DATA ENCRYPTION STANDARD)	15
1.1 PRINCIP FUNGOVÁNÍ DES	16
1.1.1 Feistelova síť	16
1.1.2 Funkce f_i	17
1.2 ÚTOKY NA DES	18
1.3 3DES (TRIPLE DES)	19
1.3.1 Možnosti použití klíče	19
1.4 PROVOZNÍ REŽIMY BLOKOVÝCH ŠIFER	20
1.4.1 ECB (Electronic Codebook)	20
1.4.2 CBC (Cipher-Block Chaining)	22
1.4.3 CTR (Counter Mode)	23
1.5 PADDING ZPRÁVY	24
1.5.1 Padding nulovými bajty	24
1.5.2 ANSI X9.23	24
1.5.3 ISO 10126-2	25
1.5.4 ISO 7816-4	25
1.5.5 PKCS#5 a PKCS#7	25
2 AES (ADVANCED ENCRYPTION STANDARD).....	26
2.1 PRINCIP FUNGOVÁNÍ AES	26
2.1.1 Dílčí operace	26
2.2 BEZPEČNOST AES	28
2.2.1 Útok postranními kanály	29
2.3 KCV (KEY CHECKSUM VALUE)	29
3 MAC (MESSAGE AUTHENTICATION CODE)	31
3.1 PRINCIP FUNGOVÁNÍ MAC	31
3.1.1 Vytváření MAC	31
3.1.2 Ověřování MAC	32
3.2 HMAC (HASH-BASED MAC)	33
3.3 CMAC (CIPHER-BASED MAC)	34
3.4 RETAIL MAC	35
4 PIN BLOCK.....	37
4.1 PIN BLOCK FORMÁTY	37
4.1.1 ISO-0 (Format 0)	37
4.1.2 ISO-1 (Format 1)	38
4.1.3 ISO-2 (Format 2)	38

4.1.4	ISO-3 (Format 3)	38
4.1.5	ISO-4 (Format 4)	38
II	PRAKTICKÁ ČÁST	40
5	NÁVRH APLIKACE	41
5.1	ANALÝZA POŽADAVKŮ	41
5.1.1	Funkcionální požadavky	41
5.1.2	Nefunkcionální požadavky	42
5.2	NÁVRH UŽIVATELSKÉHO ROZHRAŇÍ	43
5.2.1	Koncept uživatelského rozhraní	43
5.2.2	Wireframy	43
6	IMPLEMENTACE.....	47
6.1	STRUKTURA PROJEKTU	47
6.2	IMPLEMENTACE ÚLOŽIŠTĚ	48
6.2.1	StorageService	49
6.2.2	SecureKeyProvider	52
6.3	KEY MANAGER	54
6.3.1	Sekce Key Manager	55
6.3.2	Export a import klíčů	58
6.3.3	Tlačítko Key Manager	61
6.4	UI A NASTAVENÍ APLIKACE	63
6.4.1	Vlastní UI	63
6.4.2	Nastavení aplikace	64
6.5	SEKCE DES/3DES	66
6.6	SEKCE AES	69
6.7	SEKCE MAC	70
6.7.1	Podsekce HMAC	71
6.7.2	Podsekce CMAC	72
6.7.3	Podsekce ISO-9797-1	74
6.8	SEKCE PIN BLOCK	75
6.9	SEKCE KCV	77
7	TESTOVÁNÍ.....	79
7.1	TC01 – PLATFORMA WINDOWS	80
7.2	TC02 – PLATFORMA ANDROID	82
7.3	TC03 – PLATFORMA WEBOVÝ PROHLÍZEČ	84
	ZÁVĚR.....	86
	SEZNAM POUŽITÉ LITERATURY	87
	SEZNAM OBRÁZKŮ	91
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	93

SEZNAM PŘÍLOH.....	95
---------------------------	-----------

ÚVOD

Už od dob starověku provázela lidstvo touha nejen spolu komunikovat, ale i komunikovat způsobem, aby komunikace zůstala utajená před ostatními. A tak začaly postupně vznikat první šifry, od Caesarovy, přes Vigenèrovu, se lidstvo postupně učilo kryptografií zdokonalovat a s tím vznikaly i daleko důmyslnější způsoby, jak dané šifry pomocí kryptoanalýzy prolamovat. Kryptografie začala s postupem času nabývat na významu. Už nerozhodovala jenom o výsledcích bitev, ale i celých válek. S rozvojem počítačů se pak šifrování stalo výsadou i v komunikaci firem a společností a ne jen armády a vlád jak doposud. Zlom pak představovala asymetrická kryptografie. Díky ní v dnešní době používá kryptografii opravdu každý, kdo vlastní počítač nebo mobilní telefon a díky ní může naše komunikace zůstat utajená.

Ve firmě Monet+ se už nějakou dobu zabývám vývojem systémů pro autorizaci a zpracování bankovních transakcí. I zde hraje kryptografie nemalou roli. Při platbě kartou, ať už v supermarketu, nebo online, bychom přece nechtěli, aby veškeré údaje jako číslo karty, či CVC posílal terminál, nebo platební brána, nešifrovaně, takže kdokoliv, kdo by byl schopný komunikaci odposlechnout, by měl naše platební údaje k dispozici. S vývojem těchto systémů se pojí potřeba nástrojů, které nám takový vývoj usnadní. Mezi tyto nástroje patří i tzv. kryptografické kalkulačky, programy, pomocí kterých lze provést šifrovací a dešifrovací operace na různých datech. Díky tomu lze například ověřit, zda se náš systém chová správně a zda šifruje tak, jak by měl. Takové nástroje existují v omezené míře online a i jako placený software na PC. S rostoucí velikostí firmy však vzniká potřeba po vlastním interním nástroji, který bude vyladěn přesně pro potřeby firmy, vývojářů a testerů.

V rámci této diplomové práce jsou nejdříve popsány jednotlivé kryptografické algoritmy, od symetrických blokových šifer DES a AES, přes autentizační algoritmus MAC, až po pro bankovní sektor typický PIN Block. Praktická část se zaměřuje na návrh multiplatformní aplikace, která dané algoritmy implementuje, na implementaci těchto algoritmů s použitím vhodných technologií a na řádné otestování výsledné aplikace. Cílem této práce je pak projekt, který obsahuje základní set kryptografických modulů a je zároveň připravený na budoucí rozšíření o novou funkcionalitu.

I TEORETICKÁ ČÁST

1 DES (DATA ENCRYPTION STANDARD)

V první kapitole se zaměříme na blokovou šifru zvanou DES, neboli Data Encryption Standard. Roku 1934 přichází do Spojených států německý emigrant Horst Feistel. Ten ve výzkumné laboratoři IBM Thomase J. Watsona vyvíjí komerční šifrovací systém zvaný Lucifer. Ten pracuje s bloky o 128 bitech. Takový blok rozdělí na dvě části, levou a pravou. Ty potom tzv. „mandluje“ a sčítá mezi sebou, tento proces se opakuje celkem 16 krát. Roku 1973 začíná americký úřad National Bureau for Standards poptávat šifrovací algoritmus, který by se stal standardem pro obchodní a firemní komunikaci. Lucifer je zatím široce využíván řadou organizací a je považován za jeden z nejsilnějších komerčních šifrovacích produktů. Dalo se tedy předpokládat, že se stane šifrovacím standardem. To ovšem nevyhovovalo National Security Agency (NSA). Šifra Lucifer jim přišla až příliš silná a nelíbila se jim myšlenka, že by civilní organizace mohly používat šifrovací standard, který by NSA nedokázala prolomit. Šifra Lucifer byla nakonec roku 1976 schválena jako standard pod novým názvem Data Encryption Standard (DES), nicméně s jedním výrazným omezením, a to, že byla délka klíče zkrácena na 56 bitů. Díky tomu byla NSA schopna za pomoci přístupu k nejvýkonnější výpočetní technice tuto šifru prolomit, ale civilní organizace mohly tuto šifru používat, aniž by konkurence dokázala rozšifrovat jejich komunikaci [1].



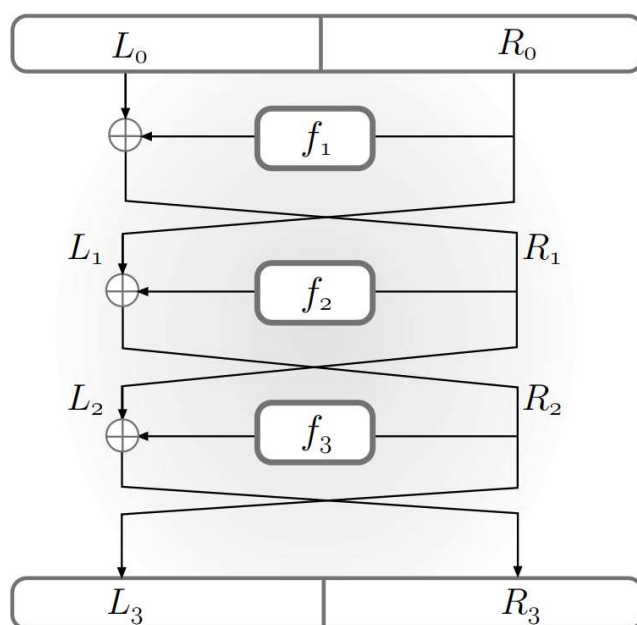
Obrázek 1 Pečeť National Security Agency [2]

1.1 Princip fungování DES

Nyní se podrobněji popíšeme fungování DES. Jak už víme, samotný standard DES vychází z šifry Lucifer. Ta pro svoje fungování používá tzv. Feistelovu síť. Ta zůstala i v základu samotné šifry DES [3].

1.1.1 Feistelova síť

Feistelova síť pracuje v několika kolech (rounds). Každé takové kolo obsahuje funkci, často S-box, či permutaci, která na rozdíl od funkcí v substitučně-permutačních sítích, nemusí být invertibilní. Díky tomu působí Feistelova síť jako síť s více „neuspořádaným“ či náhodným chováním v porovnání se substitučně-permutačními sítěmi [3].



Obrázek 2 Schéma Feistelovy sítě [3]

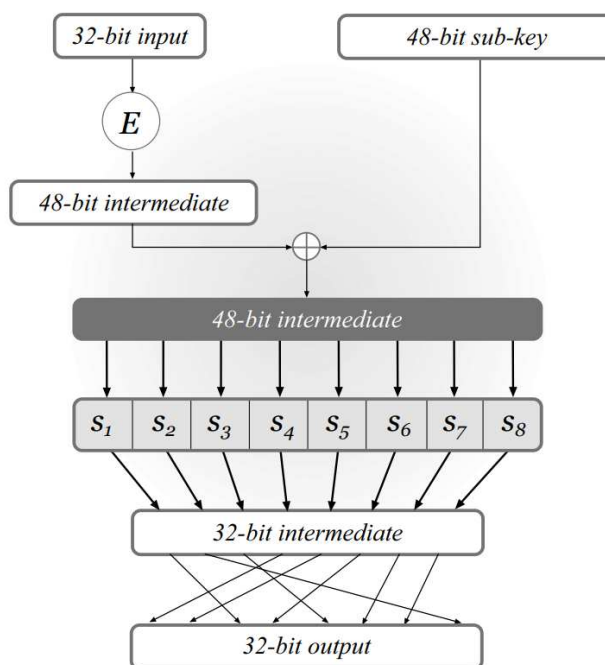
Síť se skládá z i kol a hlavní klíč je rozdělen na jednotlivé podklíče k_i . Každé kolo f_i pak používá svůj vlastní podklíč k_i . Pro výpočet jednoho kola platí, že vstupní data, ať už počáteční, nebo jako výstupy z předchozího kola jsou rozděleny na dvě části, levou L_{i-1} a pravou R_{i-1} . Pro výstupní data L_i a R_i pak platí následující vzorec [3]:

$$L_i = R_{i-1} \quad (1)$$

$$R_i = L_{i-1} \oplus f_i(R_{i-1}) \quad (2)$$

1.1.2 Funkce f_i

V každém jednom kole ze šestnácti prochází data tzv. „mandlovací“ funkcí f_i . Tato funkce v DES algoritmu pracuje se 48bitovým podklíčem a 32bitovou půlkou vstupních dat R_{i-1} [3].



Obrázek 3 Schéma funkce f_i [3]

Funkce f_i pak pracuje ve čtyřech různých krocích:

1. **Expanze E** – Abychom mohli provést operaci XOR mezi vstupními daty a podklíčem, je třeba vstupní data rozšířit ze 32 bitů na 48, aby měla stejnou délku. Právě k tomu slouží expanzní funkce E , která rozšiřuje vstup duplikací některých bitů na výstupu podle předem dané tabulky.
2. **XOR dat s podklíčem** – Nyní, když už mají data požadovanou délku 48 bitů, lze na nich provést operaci XOR (Exclusive Or) s podklíčem k_i .
3. **Substituce** – Výstup z předchozího kroku je rozdělen na 8 bloků po 6 bitech. Každý z těchto bloků je pak zpracováván v jiném S-boxu. Výstupem z S-boxu jsou 4 bity, čímž je délka dat zredukována opět na 32 bitů. Jelikož každému výstupu z S-boxu odpovídají přesně 4 různé vstupy, tak se tato operace nedá invertovat a tak je zajištěna nelinearita celého systému.
4. **Permutace** – Nakonec je na datech provedena permutace. Ta byla navržena tak, aby každý výstup z S-boxu ovlivnil vstupy do 6 různých S-boxů v následujícím kole (Díky tomu, že je provedena expanze na datech před vstupem do S-boxů).

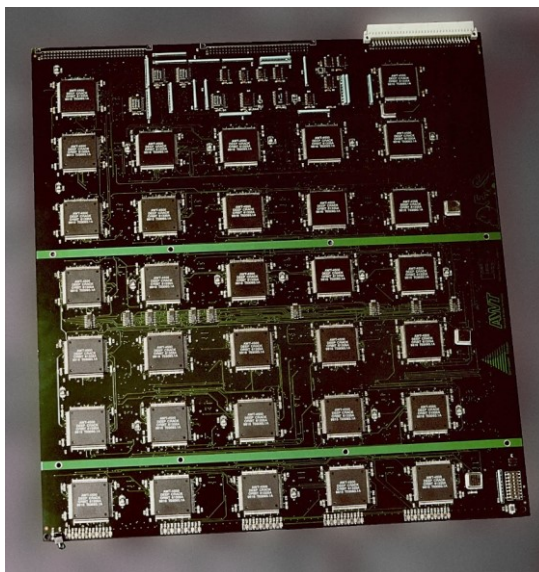
Díky této struktuře dochází při šifrování k tzv. lavinovému efektu. To znamená, že i pouhá malá změna v šifrovaných datech na vstupu povede k obří změně na výstupních datech. Pokud bychom měli například dva vstupy, které se od sebe liší rozdílem pouze v jednom bitu, tak po prvním kole DES je rozdíl v pravé polovině výstupu R_1 . Po druhém kole se budou data lišit minimálně ve třech různých bitech, po sedmi kolech už bude odlišná celá pravá polovina R_7 a po osmi kolech změna v jednom bitu na vstupu ovlivní bity v levé i pravé polovině L_8 a R_8 . Takové chování posiluje odolnost DES vůči útokům [3].

1.2 Útoky na DES

Ačkoliv je DES v jádru dobře navrženou a silnou šifrou, tak její slabina tkví právě v příliš krátkém 56bitovém klíči. Rozhodnutí použít pouze takto krátký klíč budilo kritiku už koncem 70. let minulého století. Nicméně v té době nikdo kromě NSA nedisponoval takovým výpočetním výkonem, aby mohl v reálném čase projít veškerý klíčový prostor o velikosti 2^{56} [1][3].

V roce 1997 vznikl projekt zvaný DESCHALL (DES Challenge). Ten si dal za úkol DES prolomit a to metodou hrubé síly, tj. vyzkoušet postupně všech 72 057 594 037 927 936 možností, dokud se nepodaří najít ten správný klíč. Toho se projekt snažil dosáhnout díky síti dobrovolníků koordinované přes Internet. Dobrovolníci byli takto schopni projít kolem 600 biliónů klíčů za den, ale i to představovalo méně jak 1 % klíčového prostoru. Nakonec se jim 18. června podařilo klíč se štěstím najít a to za zhruba pět měsíců od spuštění a to po projití přibližně jedné čtvrtiny všech klíčů. Autor projektu Matt Curtin poté uvedl, že vzhledem k rostoucí efektivitě testování a množství dobrovolníků, by projití zbylých tří čtvrtin klíčového prostoru zabralo pouhých 36 dní [4].

Později, téhož roku, byla DES prolomena znovu, a to za pouhých 41 dní, tentokrát v rámci projektu *distributed.net*. Roku 1998 přišel další průlom. Společnost Electronic Frontier Foundation navrhla a postavila zařízení zvané „Deep Crack“. Toto zařízení, které stálo kolem 250 000 \$, bylo schopno prolomit DES za pouhých 56 hodin. V roce 1999 dokázal „Deep Crack“ společným úsilím s *distributed.net* najít klíč pouze za 22 hodin [3].



Obrázek 4 Deska osazená čipy Deep Crack [5]

1.3 3DES (Triple DES)

S úspěšnými pokusy o prolomení DES začalo být jasné, že standard bude muset být nahrazen novým a silnějším. Zde vznikly dva různé přístupy. Jeden navrhoval změnit vnitřní strukturu DES a tím vytvořit novou a silnější šifru. Druhý považoval DES za téměř „perfektní“ šifru, jejíž jediná slabina spočívala pouze v příliš krátkém klíči, a dal si za cíl vytvořit novou šifru, která bude vnitřně používat původní a nezměněnou DES [3].

První pokusy, nazvané 2DES, navrhovaly šifru, která by jednoduše zašifrovala data 2x s použitím klasické DES. Odolnost této šifry by se však zvýšila pouze minimálně oproti DES [3]. Později představil Walter Tuchman schéma operací, které by se mělo použít, a to $E_1D_2E_1$ pro šifrování a $D_1E_2D_1$ pro dešifrování, kde E znamená DES v módu šifrování a D DES v módu dešifrování. Dolní indexy pak určují klíč, který se zrovna používá. Oproti DES používá 3DES v původním návrhu 112bitový klíč místo 56bitů. Klíč je však rozdělen na části a pracuje se s ním jako s dvěma DES klíči K_1 a K_2 . To přináší výhodu, že pokud budou K_1 a K_2 shodné, tak se 3DES šifruje data stejně jako DES, díky tomu, že se operace E_1 a D_2 navzájem „vyruší“, což činí 3DES zpětně kompatibilní s DES [3][6].

1.3.1 Možnosti použití klíče

3DES můžeme rozdělit podle toho, jaký typ klíče používáme. V původním návrhu šifry byl klíč sice 112bitový, tuhle variantu však implementovalo pouze několik bankovních standardů [6]. Operace provádíme nicméně tři - $E_1D_2E_3$ a pro každou z nich můžeme použít

různý, 56bitový klíč K_1 , K_2 a K_3 , celkem tedy 168 bitů. Podle toho, jaké klíče použijeme lze 3DES rozdělit do 3 kategorií [7]:

- **Možnost 1**, někdy také známá jako 3TDEA, používá 3 nezávislé 56bitové klíče K_1 , K_2 a K_3 . Díky tomu je 3TDEA nejsilnější ze všech možností.
- **Možnost 2**, někdy rovněž nazývaná 2TDEA používá různé klíče K_1 , K_2 a K_3 , ale klíč K_3 je stejný jako K_1 . 2TDEA nabízí kompromis mezi možnostmi 1 a 3, kdy nabízí vyšší bezpečnost než možnost 3, ale zároveň kratší klíč než možnost 1.
- **Možnost 3**, pro kterou platí, že klíče K_1 , K_2 i K_3 jsou shodné. Tato možnost je sice zpětně kompatibilní s DES, nicméně nenabízí o nic vyšší bezpečnost, než původní DES.

Ačkoliv se délka klíčů uvádí jako 56, 112 a 168 bitů, 3DES ve skutečnosti pracuje s klíči o délce 64, 128 a 192 bitů. To je způsobeno tím, že 8 posledních bitů každého klíče slouží jako tzv. paritní bity, které slouží k detekci chyby v předchozích bitech klíče, aniž by nějak rozšiřovaly klíčový prostor [3][7].

1.4 Provozní režimy blokových šifer

Blokové šifry šifrují data po blocích. Délka klíče je pak buď shodná s délkou bloku (DES), nebo v některých případech i delší (AES). Šifrovaná data však málokdy zabírají pouze jeden blok, ve většině případů jsou podstatně delší. Klíč tedy při šifrování používáme opakovaně, což ale vede k oslabení síly šifry. Jako řešení tohoto problému se nabízí distribuovat klíč pro každý blok dat zvlášť, tj. klíč by měl stejnou délku jako šifrovaná data. V praxi by to však znamenalo nutnost distribuovat o délce až miliardy bitů, což by bylo přinejmenším velmi nepraktické. Z tohoto důvodu vznikly různé tzv. provozní režimy blokových šifer. Ty zlepšují bezpečnost dané šifry bez nutnosti zvyšování délky klíče. V této podkapitole se seznámíme se třemi nejpoužívanějšími [8].

1.4.1 ECB (Electronic Codebook)

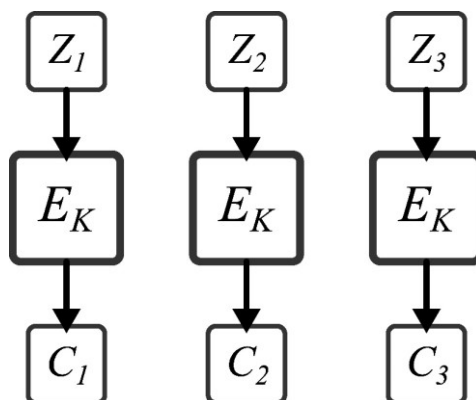
První režim, který lze použít, se nazývá ECB, neboli Electronic Codebook. Blok dat pak lze zašifrovat dle následujícího vzorce [8]:

$$C_i = E(Z_i, K), i = 1, 2, \dots, M \quad (3)$$

Kde Z_i značí určitý blok zprávy, C_i určitý blok kryptogramu a M celkový počet bloků. K je pak klíč a E šifrovací funkce. Pro dešifrování lze použít tento vzorec [8]:

$$Z_i = D(C_i, K), i = 1, 2, \dots, M \quad (4)$$

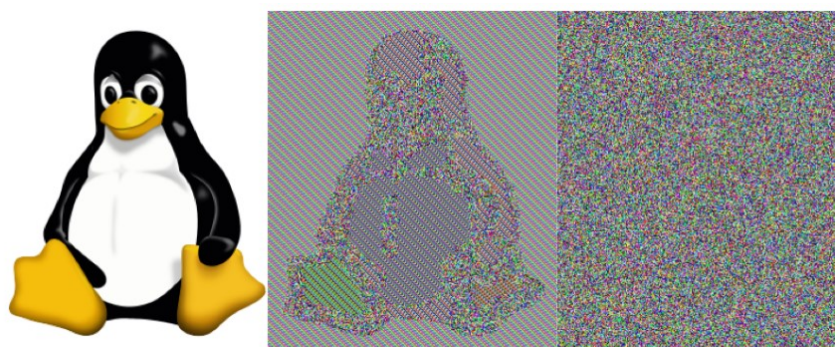
Kde D značí dešifrovací funkci. ECB tedy funguje velmi prostě, že postupně bere jednotlivé bloky a šifruje je nezávisle na ostatních. To sice přináší výhodu v podobě jednoduchosti implementace i použití, nicméně přináší i jeden zásadní problém. A to, že pokud jsou některé bloky Z_i shodné, budou shodné i bloky C_i kryptogramu [8].



Obrázek 5 Schéma provozního režimu ECB

(zdroj: vlastní)

Problém, který při použití režimu ECB nastává, je krásně demonstrován na následujícím obrázku. Na něm Larry Ewing zašifroval obrázek Tuxe, maskota Linuxu, a to nejdříve s použitím ECB režimu a poté pokročilejšího a bezpečnějšího CTR režimu [9].



Obrázek 6 Demonstrace nedostatku provozního režimu ECB [9]

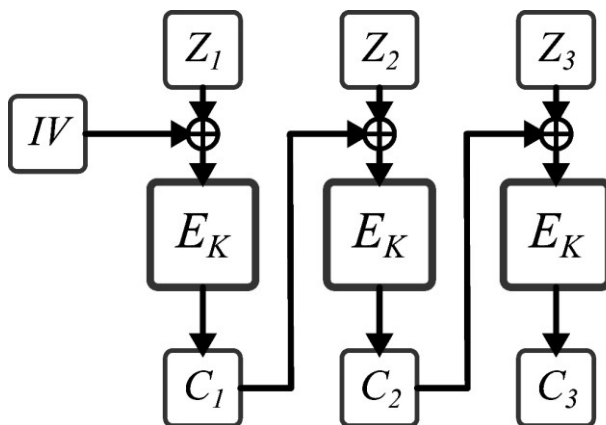
Jak je vidět, tak u dat s vysokou mírou redundance se snadno stane, že jsou částečně čitelné i po zašifrování, což výrazně ulehčuje kryptoanalytikům útok na takto zašifrovanou zprávu. Z tohoto důvodu by se provozní režim ECB neměl používat. Jedinou výjimkou, kde by jeho použití nepředstavovalo bezpečnostní hrozbu, je šifrování velmi krátkých zpráv s neredundantními daty, jako například jiné šifrovací klíče, které byly testovány na náhodnost. [8].

1.4.2 CBC (Cipher-Block Chaining)

Další provozní režim, který lze při šifrování použít, se nazývá CBC, neboli Cipher-Block Chaining. Ten se hodí k použití v komunikačních kanálech, kde je nízká pravděpodobnost chybně přenesených bitů, nebo v případech, kdy data šifrujeme, ale nepřenášíme. Vzorcem tento režim lze vyjádřit následovně [8]:

$$C_i = E(Z_i \oplus C_{i-1}, K), i = 1, 2, \dots, M, C_0 = IV \quad (5)$$

Hlavní rozdíl oproti ECB spočívá v tom, že před zašifrováním je blok Z_i xorován s předchozím blokem kryptogramu C_{i-1} . U bloku Z_1 však žádný předchozí blok kryptogramu neexistuje, jelikož se ještě nezačalo šifrovat. Z tohoto důvodu je stanoven tzv. inicializační vektor, často značen pouze jako IV . Ten má velikost stejnou jako blok a slouží k operaci XOR mezi sebou samým a prvním blokem Z_1 před zašifrováním. Díky tomu, pokud šifrujeme stejná data stejným klíčem, ale pozměníme IV , tak se budou i výsledná data od sebe lišit. IV není tajnou hodnotou jako klíč a může se příjemci posílat spolu se zašifrovanými daty. Pokud nechceme IV při šifrování použít, tak se jeho hodnota může nastavit na samé nulové bity. Tak bude výsledek XOR operace mezi IV a Z_1 samotný blok Z_1 beze změny [8].



Obrázek 7 Schéma provozního režimu CBC

(zdroj: vlastní)

Dešifrování v CBC režimu pak lze vyjádřit takto [8]:

$$Z_i = D(C_i, K) \oplus C_{i-1}, i = 1, 2, \dots, M, C_0 = IV \quad (6)$$

Nejprve se dešifruje blok C_1 , který se pak xoruje s blokem kryptogramu C_{i-1} . Tak se získají postupně všechny bloky zprávy Z .

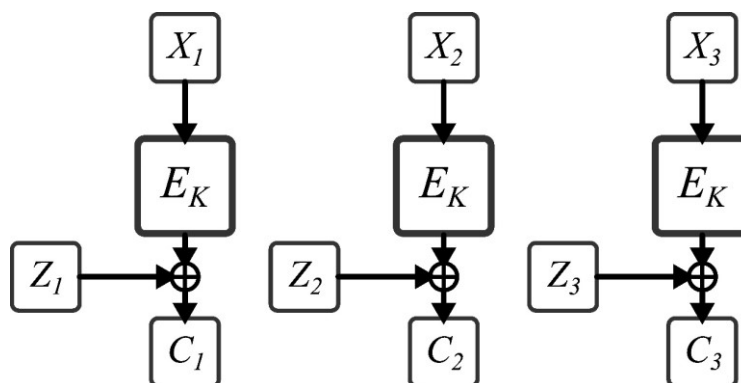
Oproti ECB provoznímu režimu se nám u CBC nemůže stát, že bychom dva stejné bloky Z_i zašifrovali stejně, jelikož provádíme XOR s předchozím blokem kryptogramu. Celý režim rovněž posiluje i inicializační vektor IV . Nevýhodou režimu CBC však může být, že předchozí bloky kryptogramu ovlivňují šifrování bloků dalších. To může představovat problém v přenosových kanálech, ve kterých dochází často k chybám. Pouhá změna v jednom bitu na při přenosu, pak může při dešifrování vytvořit tzv. kumulovanou chybu a vytvořit několikanásobně víc chyb v dešifrované zprávě [8].

1.4.3 CTR (Counter Mode)

Problém šíření kumulované chyby způsobené rušením v přenosovém kanálu dokáže řešit poslední režim provozu zvaný CTR, neboli Counter Mode. CTR nefunguje tak zcela jako bloková šifra, ale jako proudová, která používá blokovou šifru pouze jako deterministický generátor náhodných bitů. Šifrování lze vyjádřit takto [8]:

$$C_i = Z_i \oplus E(X_i, K), i = 1, 2, \dots, M \quad (7)$$

Kde X_i představuje hodnotu tzv. čítače. Čítač je n bitové číslo, které vzniká cyklickou inkrementací předchozí hodnoty o hodnotu jedna. Počáteční hodnota čítače X_i nemusí být utajována a může být opět nastavena s použitím IV , nicméně je potřeba zajistit, aby se hodnota X_i nikdy neopakovala a všechny použité hodnoty byly unikátní [3][8].



Obrázek 8 Schéma provozního režimu CTR

(zdroj: vlastní)

Dešifrování lze pak provést následovně [8]:

$$Z_i = C_i \oplus E(X_i, K), i = 1, 2, \dots, M \quad (8)$$

Nelze si nevšimnout, že dešifrování probíhá stejným způsobem jako šifrování. I to nám může prozradit, že režim CTR je ve skutečnosti proudovou šifrou. Díky tomu se chyba, která

může vzniknout při přenosu, nekumuluje, takže počet chybných bitů po dešifrování bude odpovídat počtu chybných bitů vzniklých při přenosu.

Režim CTR nám přináší oproti předchozím ještě jednu výhodu. Délka dat nemusí být celistvý násobek délky bloku dané šifry. To je způsobeno tím, že data šifrujeme operací XOR mezi daty a blokem pseudonáhodných bitů vytvořených blokovou šifrou. Nevýhodou tohoto režimu je jako u každé proudové šifry možnost útoku na autentičnost zprávy, proto je dobré jej kombinovat s nějakou autentizační technikou [8].

1.5 Padding zprávy

Jak už jsme zmínili v předchozí podkapitole, tak u velké části režimů provozu blokových šifer je nutné, aby délka dat šifrované zprávy odpovídala násobku délky bloku dané blokové šifry. Jen velmi zřídka se nám poštěstí, že zpráva splňuje tuto podmínku, aniž bychom ji museli před šifrováním upravit. K tomuto účelu se často používají techniky, anglicky zvané jako „padding“, neboli v překladu zarovnání [8]. V této podkapitole si několik těchto technik představíme.

1.5.1 Padding nulovými bajty

Jedná se o nejjednodušší způsob, jak zarovnat poslední blok zprávy. Máme-li kupříkladu zprávu, hexadecimálně zapsanou jako AA BB CC DD EE FF AA BB CC DD EE FF, o délce 12 bajtů a potřebujeme ji zarovnat na délku bloku 8 bajtů, tj. vytvořit z ní zprávu o celkové délce 16 bajtů, tak k ní přidáme 4 nulové bajty [10]:

AA BB CC DD EE FF AA BB CC DD EE FF **00 00 00 00**

Tato metoda se nepoužívá u běžného šifrování, protože má jednu zásadní nevýhodu a to pokud samotná data zprávy končila nulovými bajty, tak nejsme schopni tyto data odlišit od nulových zarovnávacích bajtů. Přesto se s touto technikou můžeme setkat v některých hashovacích a MAC funkcích, tedy tam, kde nepotřebujeme daná data dešifrovat [10].

1.5.2 ANSI X9.23

Tento padding je specifikován standardem ANSI X9.23. Zpráva je opět zarovnána nulovými bajty, nicméně poslední bajt nám tentokrát udává počet použitých nulových bajtů k zarovnání [10]:

AA BB CC DD EE FF AA BB CC DD EE FF **00 00 00 04**

Díky tomu ví příjemce zprávy přesné množství bajtů, které bylo použito k zarovnání a po dešifrování zprávy je může od původní zprávy oddělit [10].

1.5.3 ISO 10126-2

Podobně funguje i padding u standardu ISO 10126-2. S tím rozdílem, že nulové bajty jsou nahrazeny náhodnými daty. Poslední bajt opět udává počet bajtů použitých k zarovnání [10]:

AA BB CC DD EE FF AA BB CC DD EE FF **3C 42 7F 04**

Tento způsob zarovnání nám přináší výhodu, že dvě stejné zprávy budou mít s velkou pravděpodobností různé zarovnání, takže i po zašifrování stejným způsobem a klíčem se budou kryptogramy těchto zpráv výrazně lišit.

1.5.4 ISO 7816-4

Další padding, co se dá použít, je definován ve standardu ISO 7816-4. Zde je k nezarovnaným datům nejdříve přidán bajt s hexadecimální hodnotou **80** a zbytek bloku je doplněn nulovými bajty [10]:

AA BB CC DD EE FF AA BB CC DD EE FF **80 00 00 00**

1.5.5 PKCS#5 a PKCS#7

Poslední padding, který si představíme, je definováno standardem PKCS#5 a PKCS#7. Zatímco PKCS#5 definuje zarovnání pouze pro bloky o velikosti 64 bitů, tak PKCS#7 se dá použít na jakýkoliv blok až do velikosti 255 bitů. Blok je opět doplněn bajty do úplné velikosti, tentokrát ale každý bajt udává počet bajtů použitých k zarovnání [10]:

AA BB CC DD EE FF AA BB CC DD EE FF **04 04 04 04**

2 AES (ADVANCED ENCRYPTION STANDARD)

Píše se rok 1997, kdy organizace NIST (National Institute of Standards) oznámila, že hledá nový šifrovací standard, který by měl nahradit doposud používaný DES. Hledaný algoritmus by měla být symetrická bloková šifra, která bude mít velikost jednoho bloku 128 bitů a bude podporovat klíče o velikostech 128, 192 a 256 bitů [11].

V srpnu roku 1998 uskuteční NIST první konferenci (AES1), kde představí celkem 15 kandidátů na nový standard a požádá veřejnost o případné připomínky k vybraným kandidátům. Na druhé konferenci (AES2), pořádané v březnu roku 1999, oznámí NIST pět finálních algoritmů – MARS, RC6, Rijndael, Serpent a Twofish [11].

NIST se při výběru vítězné šifry soustředil na tři hlavní kritéria – bezpečnost, náklady a charakteristiky daného algoritmu a jeho implementaci. Po obsáhlém testování a porovnávání všech kandidátů NIST nakonec vybral jako vítěznou šifru algoritmus Rijndael, pojmenovaný podle svých belgických autorů Vincenta Rijmena a Joana Daemena. Oproti ostatním kandidátům vynikal hlavně ve výkonu, jednoduché implementaci a nízkým nárokům na hardware [11].

2.1 Princip fungování AES

Podobně jako DES, tak i AES pracuje v několika kolech. Místo Feistelovy sítě je zde tentokrát použita substitučně-permutační síť. Počet kol se také liší a to v závislosti na délce klíče. Pro 128bitový klíč se provádí 10 kol, pro 192bitový klíč 12 kol a pro 256bitový klíč 14 kol. V každém kole se provádí několik dílčích operací [8][12].

2.1.1 Dílčí operace

Některé dílčí operace se provádí v AES pouze jedenkrát, některé, konkrétně čtyři, se pak pravidelně opakují v jednotlivých kolech. Celkem můžeme definovat šest různých dílčích operací [12]:

1. **Rozdělení dat do bloků a jejich zarovnání** – Podobně jako DES šifruje bloky o velikosti 64 bitů, tak AES používá 128bitové bloky. Data je potřeba rozdělit do bloků a poslední blok zarovnat [12]. AES používá nejčastěji PKCS#7 padding [13].
2. **Expanze klíče** – Protože pro každé kolo potřebujeme vlastní podklíč, tak je původní klíč tzv. expandován a tím se vytvoří dílčí podklíče [8][12].

3. **Přičtení iteračního klíče** – K šifrovaným datům se přičte podklíč daného kola, čímž se vytvoří blok nový [8][12].
4. **Substituce bajtů** – Každý bajt je nahrazen kódem podle předem dané tabulky zvané Rijndael S-box. Bajt 42 je kupříkladu nahrazen bajtem 2C [12].

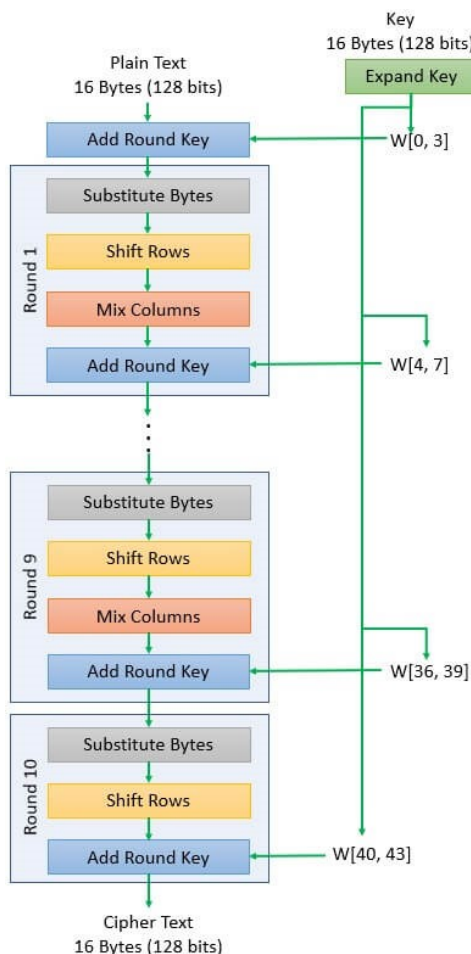
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Obrázek 9 Rijndael S-box [14]

5. **Rotace řádků** – V tomto kroku AES posune řádky bloku, které byly získány během předchozí operace. Na prvním řádku se neprovádí žádná rotace, druhý se posune doleva o 1 bajt, třetí o 2 bajty a čtvrtý o 3 bajty vlevo [12].
6. **Substituce sloupců** – Tato operace provádí substituci sloupců stavové matice. Tu získáme vynásobením tzv. mixovací matice M a stavové matice U . Sloupec S výchozí stavové matice U je násoben mixovací maticí M a vznikne tak sloupec S' transformované matice U [8].

Tyto operace se při šifrování v různém pořadí střídají a opakují v 10, 12 nebo 14 kolech podle délky klíče. Před začátkem šifrování se musí provést **Rozdělení dat do bloků a jejich zarovnání** a také **Expanze klíče**. Tyto operace se provedou pouze jedenkrát na začátku. Následuje takzvaně iniciační šifra. V ní se provede jediná operace a to **Přičtení iteračního klíče**. Pak následují už samotná kola. Ve všech kolech je pořadí operací následující: **Substituce bajtů** → **Rotace řádků** → **Substituce sloupců** → **Přičtení iteračního klíče** [8]. Jedinou výjimku představuje kolo poslední. Tam je vynechána operace **Substituce sloupců**.

a to z důvodu, že spotřebovává zbytečně mnoho výpočetního výkonu, aniž by výrazně modifikovala data. Jinými slovy je v posledním kole prakticky redundantní [12].



Obrázek 10 Schéma AES šifrování [15]

Při dešifrování funguje AES jako tzv. inverzní šifrování. Dílčí operace jsou v každém kole prováděny v opačném pořadí a i iterační klíče jsou použity v inverzním pořadí [12].

Podobně jako DES, tak i AES může pracovat v různých provozních režimech jako například ECB, CBC či CTR, s tím rozdílem, že tentokrát pracuje se 128bitovými bloky místo 64bitových [8].

2.2 Bezpečnost AES

Díky tomu, že šifra AES používá delší 128, 192 a 256bitové klíče, tak je v současné době považována za neprolomitelnou použitím metody hrubé síly. Kupříkladu AES-256, u které je klíčový prostor 2^{256} . V praxi to znamená, že i moderním počítačům by zabralo vyzkoušet všechny klíče déle, než je stáří vesmíru [16].

Nicméně útok hrubou silou není jediný způsob, jak na šifru útočit. Bezpečnost samotné AES, ostatně i jako jakékoliv jiné šifry, závisí na faktorech jako její implementace, správa klíčů, jejich generování a distribuce. Může existovat sebebezpečnější a neprolomitelný šifrovací algoritmus, ale jeho bezpečnost je rázem prolomena, pokud si nedokážeme uhlídat únik tajných dat jako například klíčů [16]. Existuje však ještě jiný způsob, jak na AES zaútočit.

2.2.1 Útok postranními kanály

Tento typ útoku neútočí na algoritmus samotný, ale místo toho se zaměřuje na slabiny v implementaci či fyzikální efekty spojené se šifrováním, která můžou útočníkovi prozradit cenná vodítka k prolomení šifry [17].

Jedna z metod, kterou lze při útoku postranními kanály použít, je tzv. diferenciální výkonová analýza (DPA). Každé elektromagnetické zařízení totiž spotřebovává různé množství energie v závislosti na právě prováděných operacích. Pomocí měření a následné analýzy těchto výkyvů může útočník získat tajné klíče ze zařízení jako jsou mobilní telefony či čipové karty, aniž by potřeboval jakoukoliv znalost vnitřní konstrukce těchto zařízení [17].

Proti DPA sice existují protiopatření, jako například přidání šumu, vyrovnaní spotřeby energie, nebo maskování tajných dat, nicméně úplná eliminace úniku dat postranními kanály zůstává pro kryptografy výzvou [17].

2.3 KCV (Key Checksum Value)

Jak už bylo zmíněno, k udržení šifrovaných dat v bezpečí před útočníky patří i proces distribuce a sdílení klíčů. Různé šifrovací systémy používají při komunikaci více klíčů, nebo klíče často mění, aby zvýšily bezpečnost. To může vést k tomu, že můžou dva různé spolu komunikující systémy nedopatřením používat různé symetrické šifrovací klíče. Aby tomuto problému předešly, tak nemohou sdílet kryptografické klíče otevřeným komunikačním kanálem, protože to by mohlo vést k jejich kompromitaci. Co ale spolu sdílet mohou, je tzv. Key Checksum value, neboli zkráceně KCV.

Dle specifikace GlobalPlatform, lze KCV pro DES vypočítat tak, že zašifrujeme 8 bajtů s hodnotou 00. Jako KCV pak považujeme první 3 bajty výsledku. Pro výpočet hodnoty KCV u algoritmu AES šifrujeme 16 bajtů, tentokrát s hodnotou 01. Za KCV opět považujeme první 3 bajty kryptogramu [18]. Pokud bychom měli kupříkladu klíč

s hexadecimální hodnotou F7A8B946259D686B4C1A38F23B927F2C, tak jeho DES KCV bude 17D0FA a jeho AES KCV bude BA7218.

Tohle ovšem není jediný způsob, jak KCV vypočítat. V oblasti platebních karet se například používá způsob výpočtu KCV definovaný normou ANSI X9.24-1-2017. Ta specifikuje 2 různé metody. První je opět zašifrování bloku nul použitím prvních 3 bajtů z výsledku. Druhá používá algoritmus CMAC k vypočtení MAC z bloku nul. Výsledek je ořezán na 10 bitů. Šifra 3DES může použít k výpočtu KCV obě metody, zatímco AES pouze tu druhou [19].

3 MAC (MESSAGE AUTHENTICATION CODE)

Message Authentication Code, neboli zkráceně MAC lze definovat jako řetězec bitů, který se posílá spolu se zprávou. MAC musí splňovat dvě základní vlastnosti. Musí být **kompletní** a **nepadělatelný**. Kompletní znamená, že samotný MAC nám stačí k ověření zprávy. Nepadělatelný zase to, že pokud je zachycen neoprávněnou třetí stranou, tak ta jej nesmí být schopna padělat pomocí dat zachycených u další zprávy [20].

MAC slouží k dvěma základním účelům – Ověření **legitimity odesílatele** a ověření **integrity dat** zprávy. Díky tomu pokud obdržíme zprávu s validním MAC, tak si můžeme být jisti, že s daty nebylo po cestě manipulováno, a že zpráva pochází od legitimního odesílatele [20].

MAC naopak danou zprávu nešifruje ani nijak jinak nezabezpečuje. Při počítání MAC se sice používá symetrický kryptografický klíč, ten ale slouží pouze k výpočtu MAC a jeho následného ověření [20].

Podle popisu, nám MAC může svým účelem a podobou připomínat digitální podpis, který taky slouží k ověření legitimacy odesílatele a integrity posílaných dat. Na rozdíl od digitálního podpisu, který používá asymetrický klíčový pár, používá MAC symetrický kryptografický klíč. Strany, které si mezi sebou posílají zprávy s MAC, tedy používají stejný klíč. Samotnou problematikou distribuce tohoto tajného klíče se algoritmy pro vytváření a validaci MAC nezabývají [20].

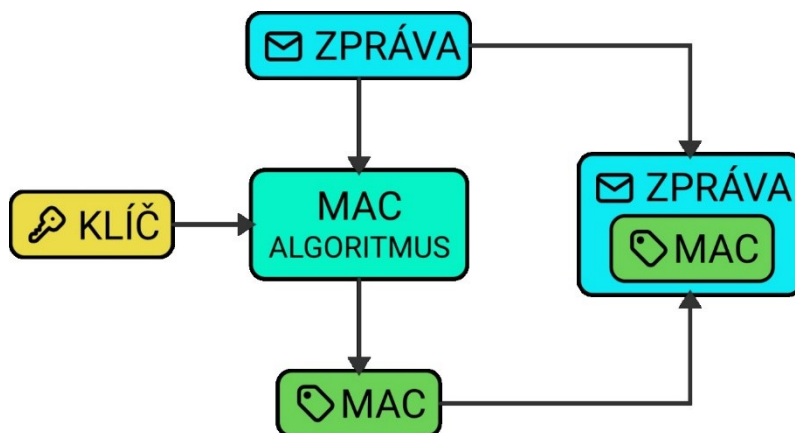
3.1 Princip fungování MAC

Proces vytváření a ověřování MAC probíhá mezi odesílatelem zprávy a jejím příjemcem. Před samostatným procesem se musí splnit dva předpoklady. Mezi odesílatelem a příjemcem musí být stanoveno, jaký algoritmus se bude pro vytváření MAC používat. Obě strany musí rovněž mít stejný tajný klíč, který bývá zpravidla poskytnut jednou ze stran a sdílen zabezpečeným komunikačním kanálem [21].

3.1.1 Vytváření MAC

Aby odesílatel mohl poslat zprávu s MAC, tak nad ní musí provést daný MAC algoritmus. Ten se provádí buď nad celou zprávou, nebo často i na pouhém hashi (otisku) této zprávy. Vstup do tohoto algoritmu je originální zpráva a tajný symetrický klíč. Výstupem z tohoto

algoritmu je pak samotný MAC. Ten je přidán ke zprávě, zpravidla na její úplný konec a odeslán spolu se zprávou [21].

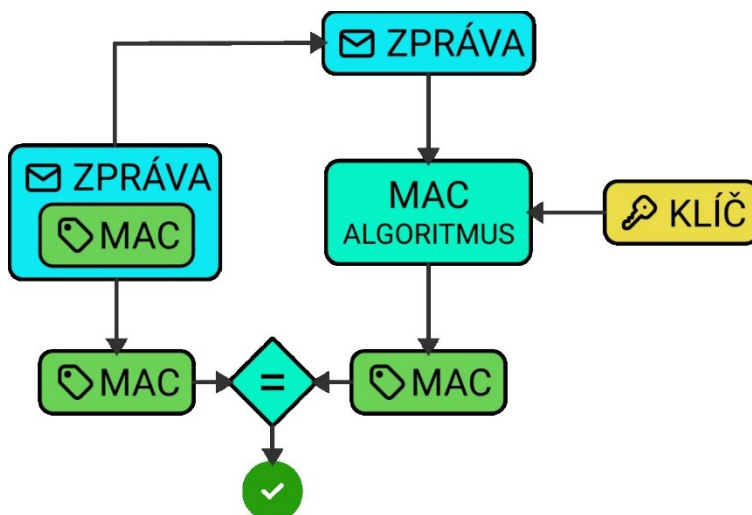


Obrázek 11 Schéma vytváření MAC

(zdroj: vlastní)

3.1.2 Ověřování MAC

Příjemce obdržel zprávu s MAC. Ten oddělí MAC od zprávy a na zprávě bez MAC provede stejný MAC algoritmus jako odesílatel s použitím stejného tajného klíče. Výsledný MAC pak porovná s tím, který obdržel spolu se zprávou. Pokud se rovnají, tak to znamená, že integrita dat zprávy nebyla během přenosu porušena a že zpráva pochází od legitimního odesílatele [21].



Obrázek 12 Schéma ověření MAC

(zdroj: vlastní)

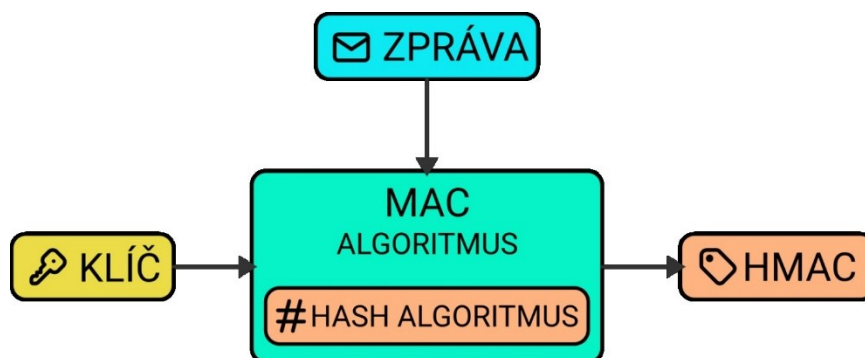
Pokud se MAC navzájem nerovnejí, tj. MAC, který nám přišel se zprávou, se liší od toho, který jsme získali výpočtem, tak to může znamenat následující [20][21]:

- Integrita dat zprávy byla porušena. To mohlo být způsobeno ať už záměrně neoprávněnou třetí stranou, nebo omylem chybovostí přenosového kanálu. V obou případech jsme však obdrželi jiná data, než jaká byla odeslána.
- Odesílatel není legitimní, ale někdo se za něj snaží vydávat a posílá nám buď zprávu s falešným MAC, nebo vůbec žádným (v tom případě se často část zprávy chybně vyhodnotí jako MAC).
- Odesílatel, nebo příjemce použil špatný tajný klíč. K tomu může nedopatřením dojít například při špatném nastavení systému, nebo v rámci nějaké plánované výměny klíče.

Ve všech případech by však příjemce neměl takovou zprávu akceptovat a měl by ji zahodit a v případě potřeby si od odesílatele vyžádat zprávu novou.

3.2 HMAC (Hash-based MAC)

Jedna z možností, jak MAC vytvořit, je použít nějakou z kryptografických hashovacích funkcí jako například MD5, SHA-1 či SHA-256. HMAC také do jisté míry připomíná hash. Na rozdíl však od HMAC, při jehož výpočtu používáme symetrický klíč, nelze pomocí hash ověřit legitimitu odesílatele, ale pouze integritu dat [20]. Během výpočtu HMAC se tak kombinují data zprávy s tajným klíčem a s použitím hashovací funkce se vytvoří HMAC dané zprávy [22][23].



Obrázek 13 Schéma vytváření HMAC

(zdroj: vlastní)

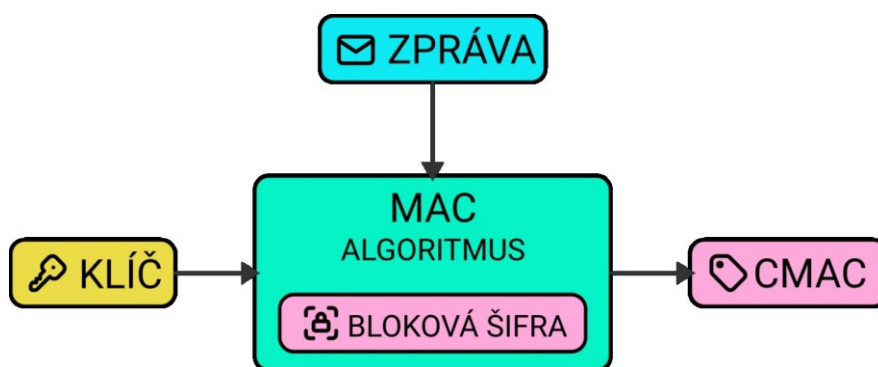
HMAC má v praxi široké využití. To hlavně díky tomu, že jeho výpočet bývá rychlý a je nenáročný na hardware. Oproti digitálním podpisům bývají HMAC menší, což se hodí

v případech, kdy chceme přenášet co nejmenší množství dat. HMAC je rovněž jednoduchý na implementaci a je podporován v celé řadě programovacích jazyků. Díky tomu se používá například v IoT (Internet of Things). HMAC je také široce používán a podporován v celé řadě protokolů, jako například HTTPS (Hypertext Transfer Protocol Secure), či FTPS (File Transfer Protocol Secure) [21][22][23].

HMAC ale může mít i řadu nevýhod. Pokud MAC algoritmus používá zastaralý hashovací algoritmus, například MD5, tak nebude HMAC odolný proti útoku hrubou silou. Důležité je rovněž zvolit dostatečně dlouhý tajný klíč, krátké klíče mohou rovněž vést ke snadno prolomitelným HMAC. Nakonec je nutné brát v potaz, že ve výjimečných případech může nastat tzv. kolize, což znamená, že dvě různé zprávy vytvoří stejný HMAC. Je to sice vysoce nepravděpodobné, ale dojít k tomu může [22].

3.3 CMAC (Cipher-based MAC)

Podobně, jako u HMAC můžeme použít hashovací funkci jako základ MAC algoritmu, tak u CMAC použijeme nějakou z blokových šifer, jako například 3DES nebo AES. Vstupem do CMAC algoritmu je opět symetrický klíč, který musí tentokrát splňovat délku, kterou vyžaduje daná bloková šifra. Dalším vstupem je opět zpráva. Ta musí mít rovněž takovou délku, aby se dala rozdělit na bloky o délce, kterou vyžaduje daná bloková šifra. Pokud se délka zprávy liší, tak lze opět použít padding, nebo lze CMAC počítat z hashe dané zprávy [20][24].



Obrázek 14 Schéma vytváření CMAC

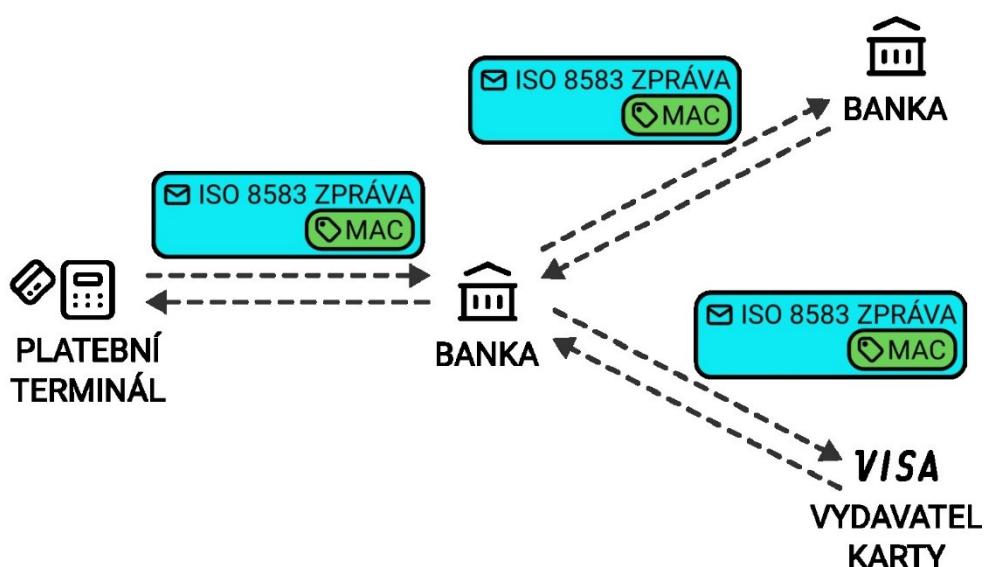
(zdroj: vlastní)

Podobně jako HMAC, tak i CMAC má v praxi řadu využití. Co se týče výkonu, tak rovněž dosahuje dobrých výsledků. Jeho síla a odolnost proti útokům závisí na použité blokové šifře, tzn. CMAC algoritmus je pouze tak silný, jako jeho bloková šifra. Obdobně

jako HMAC se CMAC používá hlavně v oblastech jako IoT a v protokolu TLS (Transport Layer Security) [20][24].

3.4 Retail MAC

Poslední skupinou MAC, kterou se budeme zajímat, je tzv. Retail MAC. Tentokrát se nejedná o novou implementaci algoritmu, ale skupinu algoritmů jako HMAC či CMAC, které se využívají v oblasti bankovníctví a v platebních systémech. Ať už pro komunikaci platebního terminálu se systémem banky, nebo komunikaci mezi jedním finančním systémem s druhým [25].



Obrázek 15 Komunikace s použitím Retail MAC

(zdroj: vlastní)

Finanční systémy spolu komunikují za použití různých protokolů. Jedním z nejpoužívanějších je pak protokol definovaný standardem ISO 8583. Ten lze využít téměř všude v oblasti bankovních a finančních systémů. Terminál může při platbě kartou tímto protokolem komunikovat se serverem autorizačního systému. Autorizační systém je pak schopen stejným způsobem komunikovat se serverem banky nebo se systémem vydavatele karty [26].

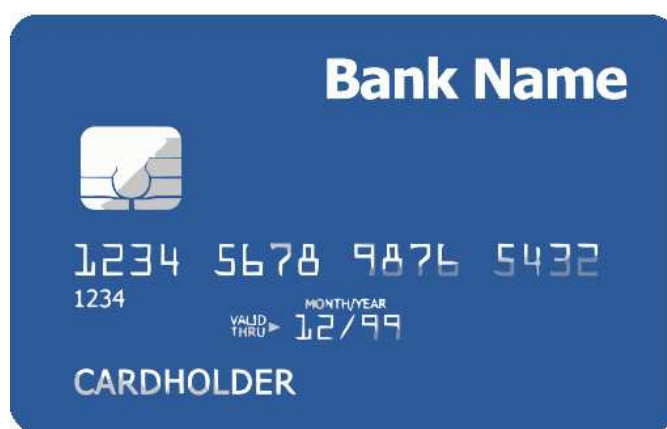
Každá zpráva ISO 8583 má pevně danou strukturu. Hlavička zprávy obsahuje tzv. MTI neboli Message Type Indicator. Ten udává informace, jako o jakou verzi protokolu se jedná, jaký je účel zprávy, zda se jedná o požadavek, či odpověď a kdo začal komunikaci. Dále zpráva obsahuje primární a případně i sekundární bitmapu. Ty udávají, co všechno je ve zprávě obsaženo za data. Nakonec je zde datová část zprávy, tzn. 64 až 128 samostatných

datových polí o různých délkách a s různými datovými typy. Každé z těchto polí nese nějakou informaci o finanční transakci, například pole 2 nese číslo platební karty (PAN), pole 4 částku a pole 49 kód měny transakce. Část těchto polí je pevně definována standardem ISO 8583, část z nich je ovšem ponechána pro pozdější definice a často využívána ke specifickým účelům daného platebního systému [26].

Retail MAC se posílá vždy na posledním místě zprávy. Pokud finanční transakce obsahuje pouze primární bitmapu (64 polí), tak se MAC nachází v poli 64. Pokud obsahuje i sekundární bitmapu (128 polí), tak se posílá v poli 128. MAC by nikdy neměl být v poli 64 i 128 zároveň. Velikost MAC je zpravidla 8 bajtů. Systém, který zprávu obdrží, z ní MAC oddělí a na zbytku zprávy vypočítá MAC vlastní. Výsledný MAC porovná s tím, co přišel se zprávou, pokud se nerovnájí, tak danou transakci zamítne [26].

4 PIN BLOCK

Posledním algoritmem, kterým se bude tato práce zabývat, je tzv. PIN Block. Na každé platební kartě bývá přítomen tzv. PAN neboli Primary Account Number. V češtině nazýván jako číslo platební karty, což je zpravidla číslo skládající se z 8 až 19 číslic, unikátně určující danou platební kartu. Zároveň při platbě kreditní či debetní kartou je od určité částky potřeba zadat tzv. PIN neboli Personal Identification Number. Tím se potvrzuje, že jsme opravdu držitelem karty a ne někdo, kdo kartu nevlastní a chce ji pouze zneužít k získání finančních prostředků na našem účtu [27][28].



Obrázek 16 Kreditní karta s PAN [29]

PIN se ale mezi terminálem a autorizačním systémem nesmí přenášet v čisté, pro člověka snadno čitelné podobě. K tomuto účelu se používají právě PIN Blocky. Ten lze vytvořit kombinací upraveného PINu s jinými daty, často PANu [27][28].

4.1 PIN block formáty

Pro vytváření PIN Blocků existuje mnoho různých standardů. Tato kapitola nám některé z nich přiblíží. Pro všechny uvedené formáty se bude používat smyšlený PAN 1234 5678 9876 5432 a smyšlený PIN 1234.

4.1.1 ISO-0 (Format 0)

Jeden z nejčastěji používaných PIN Block formátů, vycházející ze standardu ISO 9564, se nazývá ISO-0, nebo také Format 0. Nejdříve se upraví PIN na velikost 16 znaků. První znak nám udává číslo formátu (0), druhý délku PINu, následuje samotný PIN a zbytek je zarovnán znakem "F" [30].

0 4 1 2 3 4 F F F F F F F F F F

Dále se upraví PAN. Z něho se použije 12 znaků nejvíc vpravo (bez kontrolní číslice PANu). Ty se zarovnají zprava na 16 znaků [30].

0 0 0 0 1 2 3 4 5 6 7 8 9 8 7 6

Mezi takto upraveným PINem a PANem se provede operace XOR, a tak se získá požadovaný PIN Block [30].

0 4 1 2 7 1 9 8 7 6 7 8 9 A B C

Formáty ECI-1, ANSI X9.8 a VISA-1 fungují stejným způsobem [30].

4.1.2 ISO-1 (Format 1)

Formát 1, který je ekvivalentní formátu ECI-4, nepoužívá pro tvorbu PIN Blocku PAN, ale pouze náhodná data k zarovnání. První znak opět udává číslo formátu (1) a druhý délku PINu, následuje PIN zarovnaný náhodnými daty, často sekvenčním číslem [30].

1 4 1 2 3 4 C 7 F 9 4 2 C A 6 1

4.1.3 ISO-2 (Format 2)

Tento formát by se měl používat pouze u off-line systémů, jako například u čipových karet. První znak je opět číslo formátu (2), druhý délka PINu. Další znaky jsou číslice PINu zarovnané zprava znakem "F" [30].

2 4 1 2 3 4 F F F F F F F F F F

4.1.4 ISO-3 (Format 3)

Formát 3 funguje podobným způsobem jako formát 0. Hlavní rozdíl je, že k zarovnání PINu jsou použity náhodné hexadecimální číslice A až F. Nad PINem a PANem je opět provedena operace XOR [30].

3 4 1 2 7 1 5 9 1 C B 9 4 F 4 5

4.1.5 ISO-4 (Format 4)

Na závěr je zde formát 4. Ten je oproti ostatním formátům poněkud složitější. Nejdříve je předpřipraven PIN. Ten opět začíná číslem formátu (4), délkou PINu, následovaný PINem a zarovnáním, tentokrát znakem "A" na délku 16. Následuje dalších 16 náhodných znaků 0 až F, takže celková délka je 32 znaků [30].

4 4 1 2 3 4 A A A A A A A A A A E E 9 4 D 4 E F 1 A 1 9 D C D E

V dalším kroku je třeba si připravit PAN. První znak je délka PANu minus 12. Pokud je PAN kratší, tak je znak 0 a pan je zleva zarovnán nulami. Následuje celý PAN a zbytek bloku je zarovnán nulami opět na 32 znaků [30].

4 1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Předpřipravený PIN je zašifrován pomocí AES-128 v ECB módu. Na výsledném zašifrovaném bloku je provedena operace XOR s předpřipraveným PANem. Výsledek je znova zašifrován pomocí AES [30].

II PRAKTICKÁ ČÁST

5 NÁVRH APLIKACE

Tato část práce se zaměřuje na definování funkcionálních a nefunkcionálních požadavků aplikace, definování jejích klíčových vlastností a uživatelského rozhraní.

5.1 Analýza požadavků

Společně s konzultantem z firmy Monet+ jsme provedli analýzu požadavků a definovali následující funkcionální a nefunkcionální požadavky. Požadavky jsme definovali pro první verzi aplikace, u které se počítá s tím, že se v budoucnu bude aktualizovat a rozšiřovat o novou funkcionalitu.

5.1.1 Funkcionální požadavky

Do kategorie funkcionálních požadavků spadá veškerá funkcionalita, kterou bude aplikace schopna vykonávat:

1. **DES/3DES** – Aplikace bude schopna šifrovat a dešifrovat hexadecimální data symetrickou blokovou šifrou DES a 3DES a to v několika různých módech a s použitím různých paddingů.
2. **AES** – Aplikace bude schopna šifrovat a dešifrovat hexadecimální data symetrickou blokovou šifrou AES a to v několika různých módech a s použitím různých paddingů.
3. **MAC** – Aplikace bude schopna vytvářet MAC z hexadecimálních dat a to jak HMAC, CMAC, tak i MAC specifikovaný standardem ISO9797-1.
4. **PIN Block** – Aplikace bude schopna vytvářet z PINu a PANu PIN Block a rovněž PIN z PIN Blocku opět získávat. Bude podporovat několik různých formátů.
5. **KCV** – Aplikace bude schopna počítat KCV kryptografických klíčů, rovněž bude podporováno několik formátů KCV.
6. **Key Manager** – Jedna z dalších důležitých vlastností bude tzv. Key Manager, neboli správce klíčů. Ten bude umožňovat uživatelům si ukládat vlastní klíče k znovupoužití, editovat je, mazat, nebo si generovat klíče náhodné. Součástí bude i import a export těchto klíčů za účelem sdílení mezi uživateli či jejich zálohování.

7. **Konzole** – Aplikace bude obsahovat vlastní konzoli, kam budou vypisovány výsledky kryptografických operací. Obsah konzole bude moci být kopírován a mazán.
8. **Validace vstupů** – Aplikace bude schopna validovat veškeré vstupy od uživatele a v případě zadání nevalidních vstupů uživateli jasně sdělit, proč je vstup neplatný.
9. **Uživatelské rozhraní** – Aplikace bude mít moderní a přehledné uživatelské rozhraní včetně tmavého a světlého motivu. Uživatel si bude moci přizpůsobit font konzole a jeho velikost.
10. **Ukládání stavu** – Aplikace bude schopna držet v paměti jednotlivá nastavení uživatele i jím poslední zadané vstupy a to i po vypnutí a opětovném zapnutí aplikace.

5.1.2 Nefunkcionální požadavky

Do kategorie nefunkcionálních požadavků spadají požadavky, které specifikují vlastnosti aplikace, její rozsah a omezení:

1. **Operační systém** – Aplikace bude schopna fungovat primárně na desktopových operačních systémech, konkrétně Windows, Linux a macOS. Vývojáři, testéři a DevOps používají ve firmě Monet+ všechny tři zmíněné operační systémy, proto je třeba podporovat všechny. Další podporovaná platforma bude web. Díky tomu bude aplikace snadno dostupná komukoliv s připojením na Internet bez nutnosti její instalace. Poslední podporovanou platformou bude OS Android. Díky tomu bude aplikace použitelná i na tabletech a mobilních telefonech.
2. **Bezpečnost** – Aplikace bude ukládat citlivé údaje, jako například kryptografické klíče, na disku dané platformy pouze šifrované. Klíč, který byl použit pro jejich šifrování, pak bude uložen na bezpečném úložišti specifickém pro danou platformu.
3. **Použitelnost** – Aplikaci bude schopen používat zaměstnanec v IT sektoru s průměrnou znalostí angličtiny. K používání aplikace nebude třeba pokročilá znalost kryptografie. Aplikace by měla být schopna v případě potřeby uživateli jasně a srozumitelně vysvětlit její použití.
4. **Údržba a škálovatelnost** – Aplikace by měla být dobře dokumentována a navržena tak, aby ji bylo možno v budoucnu rozšířit o nové funkce a vylepšení.

5.2 Návrh uživatelského rozhraní

V této podkapitole se nachází návrh uživatelského rozhraní, rozebrání jeho klíčových vlastností a navržená požadovaná podoba aplikace pomocí wireframů.

5.2.1 Koncept uživatelského rozhraní

Jak už bylo zmíněno v analýze požadavků, tak aplikace bude koncipována z různých sekcí jako například AES, DES/3DES, MAC, atd. Jakákoliv akce provedená v sekcích (šifrování, dešifrování) se projeví ve formě logu na konzoli aplikace. Logovat operace na konzoli dává uživatelům výhodu, že pokud chtějí například sdílet výsledek nějaké kryptografické operace, tak text z konzole jednoduše zkopírují a pošlou jako textovou zprávu. Velikost konzole vůči zbytku aplikace bude možné rovněž měnit, nebo ji schovat úplně. V tom případě se v aplikaci objeví plovoucí tlačítko, jenž vrátí konzoli zpátky do aplikace.

Přepínat mezi jednotlivými sekcemi bude možné pomocí levého postranního navigačního menu. Zde budou jednotlivé sekce rozděleny do skupin. AES a DES/3DES bude například spadat do skupiny Generic, zatímco KCV a Key Manager budou v sekci Keys. Speciální sekce bude sekce Home, která se zobrazí po každém novém otevření aplikace a sekce Settings, ve které si bude moct uživatel nastavit, jak se bude aplikace chovat a jak bude vypadat.

5.2.2 Wireframy

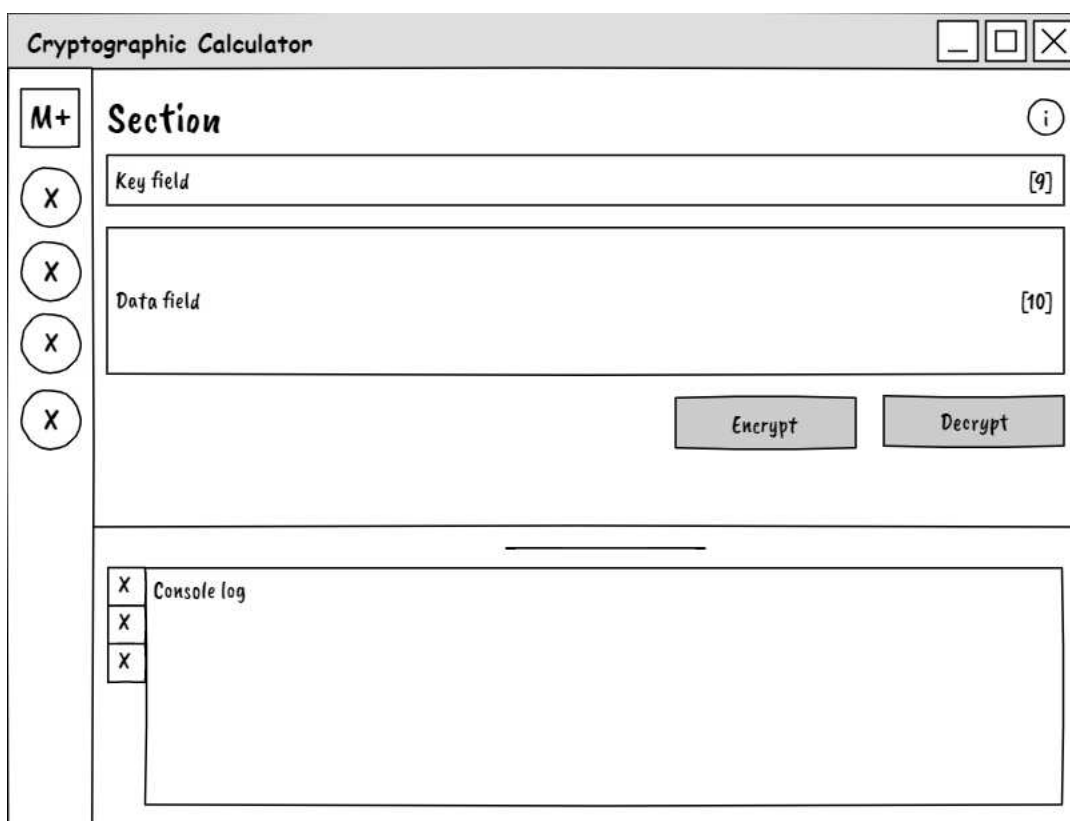
Tato část obsahuje navrhnuté wireframy, které nám dají před zahájením vývoje lepší představu, jak bude výsledná aplikace vypadat.

Jako první je zde wireframe obecné sekce aplikace. Celá aplikace bude situována jako one page, což znamená, že konzole aplikace i navigační menu zůstanou přítomné v okně aplikace po celou dobu používání aplikace. Přepínáním sekcí v navigačním menu se bude pouze měnit obsah sekce nad konzolí. Pro zvětšení prostoru obsahu dané sekce se bude levé postranní menu skládat pouze z ikon daných sekcí či jejich skupin. Po kliknutí na ikonu bude buď uživatel rovnou přesměrován do dané sekce, anebo pokud je ve skupině více jak jedna sekce, se mu v aplikaci objeví drop-down menu, které mu umožní si danou sekci vybrat ze seznamu.

Samotná konzole bude mít rovněž na levé straně sadu tlačítek, kterými bude možno obsah konzole zkopírovat do schránky, vymazat, nebo se v konzoli přesunout na začátek či

konec logu. Konzolí pak bude uživatel moct hýbat nahoru či dolů a upravovat tak její výšku vůči zbytku aplikace nebo ji úplně schovat.

Samotná sekce pak často bude obsahovat běžné UI elementy jako drop-down menu, textová pole a tlačítka. Textová pole zde oproti běžným textovým polím obsahují ještě indikátor délky vstupu na konci daného pole. To poskytne uživateli lepší představu o délce vstupu a povaze zadávaných dat. Celý wireframe základního rozložení je zobrazen na následujícím obrázku.

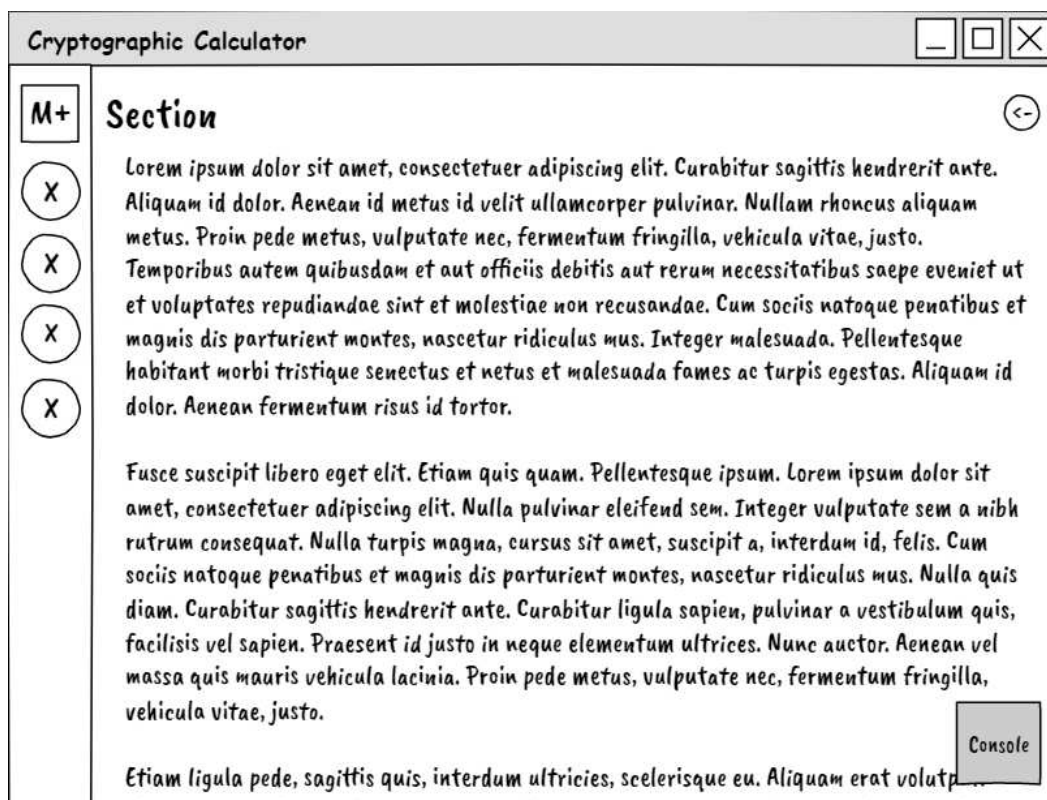


Obrázek 17 Wireframe jedné sekce

(zdroj: vlastní)

Každá sekce bude mít ještě navíc malé tlačítko v pravém horním rohu. Po kliknutí na něj se uživateli otevře stránka s podrobnějšími informacemi o dané sekci. Ta může obsahovat například základní informace o dané šifře, popis jak danou sekci používat, popis jak fungují například různé způsoby provozních režimů, atd. Vrátit se z téhle části bude moct uživatel opět kliknutím na tlačítko v pravém horním rohu, kde bude šipka zpět.

Pokud uživatel stáhne konzoli až naspod aplikace, tak konzole zmizí a v pravém dolním rohu se objeví plovoucí tlačítko na opětovné vrácení konzole zpět do aplikace. Informační část sekce a tlačítko jsou zobrazeny na následujícím wireframu.



Obrázek 18 Wireframe sekce s podrobnějšími informacemi

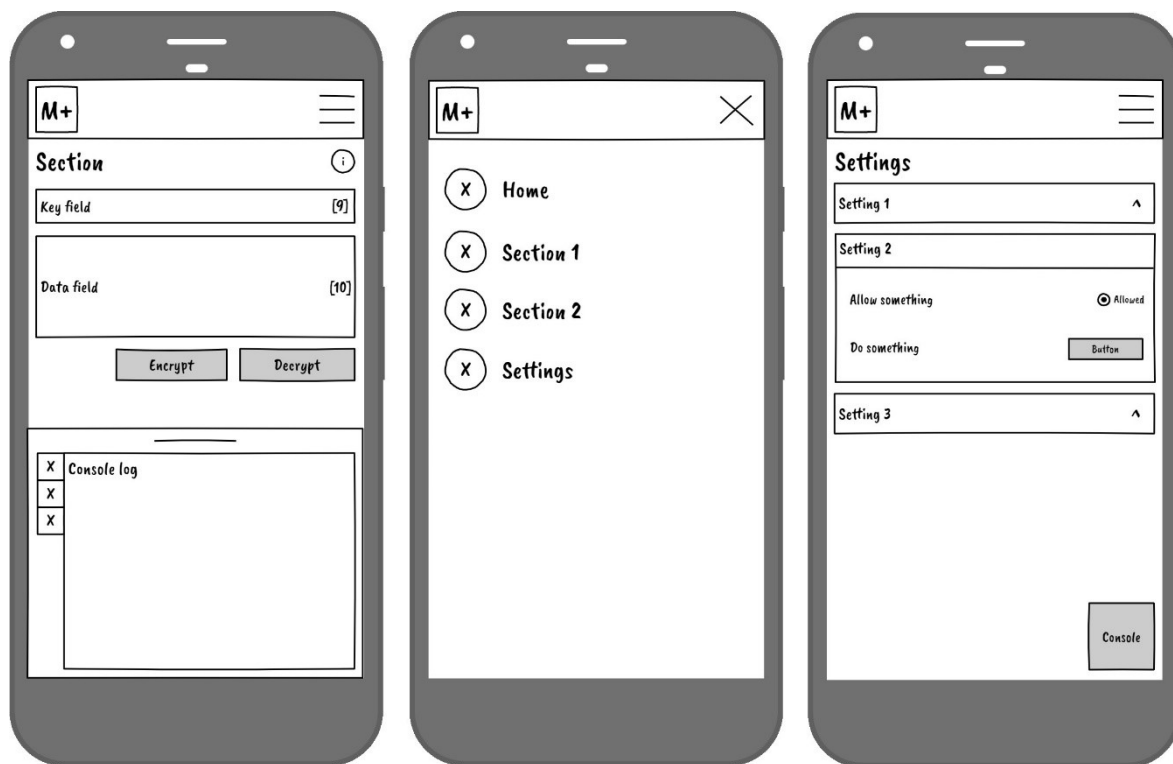
(zdroj: vlastní)

Poslední věc, na kterou se musí u návrhu uživatelského rozhraní myslet, je to, že vyvíjená aplikace je multiplatformní, tudíž je nutné počítat i s tím, že ji uživatelé budou používat i na podstatně menších mobilních displejích a to ať už v Android verzi aplikace, nebo ve webovém prohlížeči.

V tomhle případě je rozložení sekce a konzole stejné. Sekce bude zabírat horní část obrazovky, zatímco konzole bude zabírat část spodní. Konzoli bude opět možné schovat a po stisknutí tlačítka znova zobrazit. Tlačítko pro zobrazení navigačního menu se přesune do pravého horního rohu a po stisknutí navigačního tlačítka se zobrazí navigační dialog přes celou obrazovku.

Pokud se uživatel přesune do sekce s nastavením aplikace, ať už na mobilní, nebo desktopové verzi aplikace, tak zde nalezne jednotlivá nastavení seskupená opět do různých sekcí, které spolu souvisí, například vzhled aplikace, chování konzole nebo podrobnější informace o verzi aplikace a platformě, na které aplikace zrovna běží.

Mobilní rozložení sekce, navigačního menu a sekce s nastavením aplikace jsou zobrazeny na následujících wireframech.



Obrázek 19 Wireframy mobilního UI

(zdroj: vlastní)

6 IMPLEMENTACE

V této kapitole je postupně popsán průběh implementace a vývoj samotné aplikace. První a velmi důležitý krok byla volba frameworku. Jako framework pro vývoj jsem zvolil Flutter [31]. Ten nabízí možnost vyvíjet vysoce výkonné aplikace pro Android, iOS, Linux, macOS, web i Windows, přičemž pro všechny platformy je použit stejný kód psaný v programovacím jazyce Dart [32]. Tento přístup zrychluje a zefektivňuje vývoj na více platformách zároveň. Zároveň obsahuje velké množství moderních UI elementů, takže při jejich správném použití nebude aplikace vypadat jako 20 let starý projekt.

Na druhou stranu ale přichází s určitými omezeními jako například horší přístup k API daných platform, který se dá částečně řešit s použitím balíčků, či větší velikost výsledných aplikací v porovnání s nativním vývojem pro danou platformu.

6.1 Struktura projektu

Neméně důležitou součástí při vývoji aplikace, která má být do budoucna rozšiřitelná a dobře udržitelná, je i vhodná struktura projektu. Každý Flutter projekt obsahuje adresáře **android**, **ios**, **linux**, **macos**, **web** a **windows**, které jsou nutné k sestavení a spuštění projektu na daných platformách. Kromě nich je zde ještě adresář **lib**, který po vytvoření nového projektu obsahuje pouze jeden soubor **main.dart**. Právě v adresáři **lib** tvoříme multiplatformní kód psaný v jazyce Dart. Struktura tohoto adresáře není nijak pevně daná a bývá čistě na vývojářích, jakou strukturu pro svůj projekt zvolí.

V adresáři **lib** jsem se tak rozhodl vytvořit dva podadresáře – **core** a **modules**. V adresáři **core** jsou obsaženy všechny komponenty aplikace, které se v kódu používají opakovaně, jedná se tedy o pomyslné jádro projektu. Tyto komponenty jsou rovněž rozděleny do adresářů podle účelu, ke kterým se využívají. Adresář **extension** obsahuje tzv. rozšíření už existujících tříd. V jazyce Dart lze jakoukoliv už existující třídu (například **String**) rozšířit o nové vlastní metody. To pomáhá zkrátit a zefektivnit kód bez nutnosti vytváření vlastních nových tříd. Další adresář **model** obsahuje třídy se zapouzdřenými daty používané v kódu opakovaně a tzv. **enum** třídy. V adresáři **resources** lze najít základní nastavení aplikace při prvním spuštění. Adresář **service** v sobě obsahuje globálně používané služby, jako například službu na obsluhu úložiště či stahování souborů. Adresář **theme** je pak dedikován na části kódu, které se týkají vzhledu a motivu aplikace, ať už jde o barvu motivu, tmavý a světlý režim, ikony aplikace či použité fonty. Adresář **utils** obsahuje všechny třídy se znovupoužitelnými, často statickými metodami. Poslední adresář **widgets**

obsahuje všechny možné UI elementy opakovaně používané v aplikaci. Každý adresář má ještě vlastní strukturu podadresářů, které ještě víc rozdělují části projektu na logické celky v rámci daného adresáře.

Adresář **modules** pak obsahuje jednotlivé části aplikace, které se v projektu vyskytují pouze jednou. Například podadresář **generic** obsahuje adresáře **aes** a **des**. Každý z nich je ještě rozdělen na podadresáře **model**, **view** a **controller** po vzoru návrhového vzoru MVC. Strukturu projektu lze vidět na následujícím obrázku.



Obrázek 20 Struktura projektu

(zdroj: vlastní)

6.2 Implementace úložiště

Další, neméně důležitou částí, byla implementace úložiště aplikace. To slouží ke dvěma základním účelům – aby si aplikace pamatovala stavy jednotlivých sekcí během přepínání mezi sekcemi a aby byla aplikace rovněž schopna načíst svůj stav po vypnutí a opětovném zapnutí, aby si pamatovala nastavení a byla schopna ukládat šifrovaně citlivé údaje jako kryptografické klíče.

Původně jsem chtěl na implementaci úložiště použít balíček **shared_preferences** [33], který nabízí jednoduchou formu ukládání primitivních datových typů do interního úložiště zařízení. Zde jsem ale při implementaci narazil na několik problémů, kdy balíček nefungoval tak, jak by měl. Proto jsem tento balíček zaměnil za **hive** [34], jednoduchou NoSQL, databázi, která vyniká výkonem a dá se použít na všech desktopových a mobilních platformách i na webu. Hive navíc umožňuje dělit data do separátních boxů a ty i případně šifrovat pomocí AES-256.

6.2.1 StorageService

Třidu, která bude hive databázi obsluhovat jsem pojmenoval **StorageService**. Abychom zabránili tomu, že bude s hive boxy manipulovat více částí aplikace zároveň, tak jsem StorageService navrhl jako singleton třídu, tzn. vždy bude moci existovat pouze jediná instance této třídy v celé aplikaci. Kód vypadá následovně:

```
class StorageService {
    StorageService._();
    static final StorageService _instance = StorageService._();

    /// StorageService is used for storing app data into local storage.
    factory StorageService() {
        return _instance;
    }

    /// Box dedicated to storing data that are not key or settings parameter.
    late Box<String> _dataBox;

    /// Box dedicated to storing keys used in calculator.
    late Box<String> _keyBox;

    /// Box dedicated to storing all data related to settings.
    late Box<String> _settingsBox;

    /// Box dedicated to store keys managed by Key Manager.
    late Box<String> _keyManager;
```

Boxy, pro ukládání dat, jsem vytvořil čtyři, **_keyBox**, kam se budou šifrovaně ukládat všechny uživatelem naposledy použité klíče, **_settingsBox**, kam se budou ukládat všechna nastavení aplikace, **_keyManager**, kam se budou šifrovaně ukládat všechny klíče spravované Key Managerem a nakonec **_dataBox**, kam se budou ukládat všechna ostatní data. Při startu aplikace je potřeba určit adresář, do kterého bude moct hive zapisovat. Aplikace si tak při startu vytvoří ve složce s dokumenty dané platformy (pokud se nejedná o web) adresář **CryptoCalc**, kam bude zapisovat:

```
if (!RunningOn.web()) {
    final Directory docDir = await getApplicationDocumentsDirectory();
    Hive.init('${docDir.path}/CryptoCalc');
}
await StorageService().init();
```

Následně se zavolá metoda **init**, která otevře jednotlivé boxy, aby se do nich mohlo zapisovat. Pokud se adresář nebo některý z boxů v adresáři nenachází, tak jsou aplikací

vytvořeny. K otevření boxů `_keyBox` a `_keyManager` jsou potřeba AES klíče. Ty ale nelze skladovat nešifrovaně na disku, protože to by znamenalo bezpečnostní riziko a hrozila by jejich kompromitace. Tyto klíče ukládám pomocí třídy **SecureKeyProvider**, jejíž účel a fungování si popíšeme později. Celý proces inicializace tedy vypadá následovně:

```
Future<void> init() async {
  _dataBox = await Hive.openBox('data');
  _settingsBox = await Hive.openBox('settings');

  KeyInitResult initResultDb = await SecureKeyProvider.initDbKey();
  HiveAesCipher aesDb = HiveAesCipher(initResultDb.key);
  _keyBox = await Hive.openBox('keys', encryptionCipher: aesDb);

  // Clears old data if in key box that were encrypted by old key
  if (initResultDb.isNewKey) await _keyBox.clear();

  KeyInitResult initResultManager = await SecureKeyProvider.initManagerKey();
  HiveAesCipher aesManager = HiveAesCipher(initResultManager.key);
  _keyManager =
    await Hive.openBox('keyManager', encryptionCipher: aesManager);

  // Clears old data if in keyManager box that were encrypted by old key
  if (initResultManager.isNewKey) await _keyManager.clear();
}
```

Úložiště je tedy připravené. Teď zbývají metody, pomocí kterých budeme moct data do úložiště zapisovat. Hive umožňuje použít jako primární klíč k hodnotě jakýkoliv řetězec znaků. Pokud bychom ale v každé části aplikace ukládali hodnoty takovýmto způsobem, mohl by lehce vzniknout zmatek a hrozilo by například, že by jedna část aplikace nechtěně přepisovala data jiné části aplikace. Z tohoto důvodu jsem vytvořil speciální třídu zvanou **StorageKey**. Ta v sobě obsahuje atributy `_section`, `_component` a `_subtype`. Jedná se o enum třídy, které nám pomůžou zasadit ukládanou hodnotu do její sekce, přiřadit ji ke své komponentě a popřípadě jí přiřadit i podtyp, pokud používáme více stejných komponent v rámci jedné sekce.

Pokud máme například sekci `Section.aes` a komponentu `Component.padding`, tak její klíč vytvoříme následujícím způsobem:

```
StorageKey(Section.aes, Component.padding);
```

`StorageKey` pak využívají metody při ukládání do hive databáze. Z objektu vytvoří jeho `String` reprezentaci – pro klíč z příkladu by to bylo například `“aes_padding”`. Podle `StorageKey` se rovněž `StorageService` rozhodne, do jakého hive boxu data uloží. Pokud

StorageKey obsahuje sekci *Section.settings*, tak jsou data uložena do boxu `_settingsBox`, pokud se jedná o komponentu *Component.key*, tak se hodnota uloží šifrovaně do boxu `_keyBox`. Zbytek dat se ukládá do boxu `_dataBox`. Ukládání klíčů do lokálního úložiště lze ještě vypnout v nastavení aplikace. V tom případě se klíče nebudou do úložiště ukládat vůbec.

Uložit String hodnoty do úložiště lze pomocí metody **store**, získat je zase lze pomocí metody **get**. Ta má navíc ještě vstupní argument *defaultValue*, který nahradí výstupní hodnotu, pokud v úložišti není nebo při čtení dojde k chybě.

```
Box<String> _selectBox(StorageKey key) {
    if (key.isSettingsKey()) {
        return _settingsBox;
    } else if (key.isKeyKey()) {
        return _keyBox;
    } else {
        return _dataBox;
    }
}

Future<void> store(StorageKey key, String value) async {
    if (key.isKeyKey()) {
        if (getBool(StorageKey.canStoreKeys(), true)) {
            _keyBox.put(key.create(), value);
        }
    } else {
        _selectBox(key).put(key.create(), value);
    }
}

String get(StorageKey key, String defaultValue) {
    try {
        return _selectBox(key).get(key.create()) ?? defaultValue;
    } on Exception {
        return defaultValue;
    }
}
```

K ukládání jiných datových typů pak slouží metody **storeInt** a **storeBool**. Jediné metody, které nepoužívají k ukládání jako klíč StorageKey, jsou metody dedikované k obsluze dat Key Manageru. Ten ukládá data šifrovaně do svého speciálního boxu zvaného `_keyManager`. Každý klíč ukládá jako JSON objektu ManagedKey a jako primární klíč používá String název klíče. StorageService implementuje metody jako **storeManagedKey**,

`getAllManagedKeys`, `deleteManagedKey` a `managedKeyExists`, které slouží ke správě uživatelem vytvořených klíčů.

```
Future<bool> storeManagedKey(String name, ManagedKey key) async {
  try {
    await _keyManager.put(name, jsonEncode(key.toJson()));
    return true;
  } on Exception {
    return false;
  }
}

List<ManagedKey> getAllManagedKeys() {
  try {
    return _keyManager.values
      .map(((value) => ManagedKey.fromJson(jsonDecode(value))))
      .toList();
  } on Exception {
    return List.empty();
  }
}

Future<bool> deleteManagedKey(String name) async {
  try {
    await _keyManager.delete(name);
    return true;
  } on Exception {
    return false;
  }
}

bool managedKeyExist(String name) => _keyManager.containsKey(name);
```

Klíče se v boxu `_keyManager` ukládají šifrovaně. Ke správě klíčů k němu a `_keyBox` nám slouží třída, kterou jsem nazval **SecureKeyProvider**. Ta je popsána v následující podkapitole.

6.2.2 SecureKeyProvider

Balíček `hive` nám umožňuje jednotlivé boxy šifrovat pomocí AES-256, tj. velmi silné a standardizované šifry. Data na disku tak jsou uložena zašifrovaná a v případě například odcizení zařízení nebude mít zloděj přístup k tajným datům bez klíče. Otázkou ovšem zůstává, kam uložit klíče použité k zašifrování `hive` boxů. První, co nás napadne, je uložit je rovněž na disk zařízení, nicméně nechat na disku klíč v plaintextu by bylo podobně nebezpečné, jako boxy nešifrovat vůbec. Další, co by nás mohlo napadnout, by bylo klíče

napevno zadat do zdrojového kódu aplikace. Zde nastávají další problémy. Klíče by se daly z kódu získat pomocí jeho reverzní analýzy. Navíc by každý uživatel používal k šifrování stejný klíč. Další možnost by bylo ukládat klíče na vzdáleném serveru, tak by ale aplikace nebyla schopna plně fungovat bez připojení k Internetu a zároveň by to znamenalo nutnost implementovat serverovou část aplikace.

K vyřešení tohoto problému jsem se rozhodl využít balíček zvaný **flutter_secure_storage** [35]. Ten umožňuje ukládat citlivá či tajná data do zabezpečených úložišť daných platform. U Androidu využívá systém zvaný **Keystore**, u macOS a iOS pak zase **Keychain**, Linux používá knihovnu **libsecret**, Windows interface zvaný **Data Protection API** a prohlížeče **Web Crypto API**. SecureKeyProvider tento balíček používá při ukládání klíčů následujícím způsobem:

```
class SecureKeyProvider {  
  /// Secure key is stored in storage under this value  
  static const String _dbKey = 'HiveDBKey';  
  
  /// Key used for storing keys under in the Key Manager  
  static const String _keyManagerKey = 'KeyManagerKey';  
  
  /// Inits key for [Hive] box called keys  
  static Future<KeyInitResult> initDbKey() {  
    return _initKey(_dbKey);  
  }  
  
  /// Inits key for [Hive] box called keyManager  
  static Future<KeyInitResult> initManagerKey() {  
    return _initKey(_keyManagerKey);  
  }  
  
  /// Changes key for [Hive] box called keys  
  static Future<List<int>> changeDbKey() {  
    return _changeKey(_dbKey);  
  }  
  
  /// Changes key for [Hive] box called keyManager  
  static Future<List<int>> changeManagerKey() {  
    return _changeKey(_keyManagerKey);  
  }  
  
  /// Inits secure key during app startup.  
  /// Gets it from secure storage or generate new if not exist  
  static Future<KeyInitResult> _initKey(String keyName) async {  
    const FlutterSecureStorage secureStorage = FlutterSecureStorage();  
    if (await secureStorage.containsKey(key: keyName)) {
```

```

    String? storedKey = await secureStorage.read(key: keyName);
    return KeyInitResult(false, base64Decode(storedKey!));
  } else {
    List<int> secureKey = Hive.generateSecureKey();
    await secureStorage.write(key: keyName, value: base64Encode(secureKey));
    return KeyInitResult(true, secureKey);
  }
}

/// Generates new secure key and rewrite old one in secure storage
static Future<List<int>> _changeKey(String keyName) async {
  const FlutterSecureStorage secureStorage = FlutterSecureStorage();
  List<int> newSecureKey = Hive.generateSecureKey();
  await secureStorage.write(key: keyName, value: base64Encode(newSecureKey));
  return newSecureKey;
}
}

```

Hive boxy `_keyManager` a `_keyBox` používají každý svůj separátní klíč. O něj si při inicializaci `StorageService` zažádají provoláním statických metod **initManagerKey** a **initDbKey**. Klíč v zabezpečeném úložišti buď už existuje, nebo je vytvořen nový. Pokud je vytvořen klíč nový, tak je obsah daného boxu smazán, protože i kdyby v něm byla zašifrovaná data, tak by nešla novým klíčem rozšifrovat.

Pokud uživatel chce například smazat všechny klíče v `_keyBoxu` (v nastavení aplikace), tak při smazání boxu dojde rovněž k výměně klíče v zabezpečeném úložišti. K tomu se používá metoda nazvaná **changeDbKey**. Ve třídě `StorageService` vypadá její použití následovně:

```

Future<void> deleteKeys() async {
  await _keyBox.clear();
  await _keyBox.close();

  List<int> secureKey = await SecureKeyProvider.changeDbKey();
  HiveAesCipher aes = HiveAesCipher(secureKey);
  _keyBox = await Hive.openBox('keys', encryptionCipher: aes);
}

```

6.3 Key Manager

Jednou z klíčových vlastností kryptokalkulačky je tzv. Key Manager, komponenta, která umožňuje uživateli vytvářet, upravovat a mazat klíče, které používá opakovaně napříč celou aplikací. Díky tomu nebude muset při každém novém spuštění aplikace zdlouhavě opisovat

nebo kopírovat klíče. Jako vlastnosti, kterými by měl Key Manager disponovat, jsem definoval následující:

- Ukládání, upravování a mazání klíčů ve vlastní sekci aplikace
- Generování klíčů různých délek a to i v jiných částech aplikace
- Výběr klíče v jakékoliv jiné části aplikace
- Filtrování klíčů podle názvu, hodnoty či KCV
- Automatický výpočet KCV při vytváření nového klíče
- Možnost uložit právě používaný klíč v jiné části aplikace do Key Manageru
- Export a import klíčů z a do aplikace a to v šifrované i nešifrované podobě
- Šifrování uložených klíčů na disku zařízení

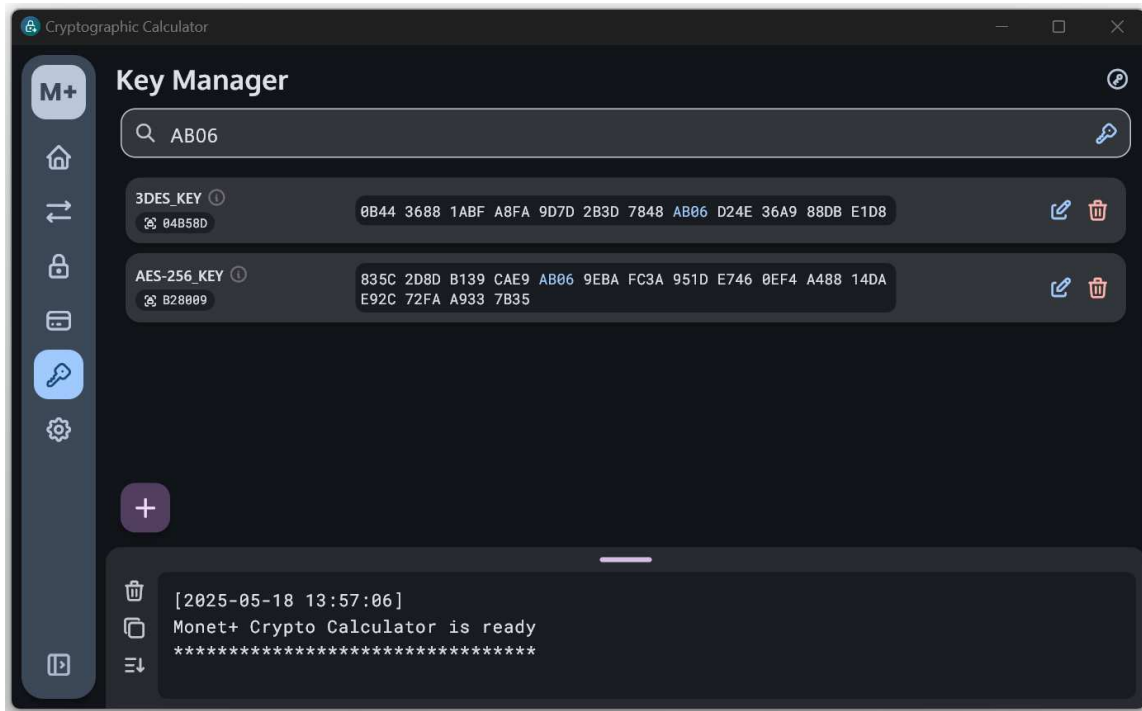
Jakým způsobem je řešeno šifrované ukládání na disk, bylo popsáno v předchozí podkapitole. Nyní se zaměříme na implementaci dalších bodů.

6.3.1 Sekce Key Manager

Key Manager využívá k ukládání klíčů separátní a šifrovaný hive box `_keyManager`. Ten je spravován pomocí `StorageService`. Tu ovšem v sekci Key Manager nepoužíváme napřímo, ale používáme třídu **KeyManager**, která obsahuje různé metody pro ukládání, mazání a filtrování klíčů. Pokud chceme například vyhledat klíče podle jejich *názvu*, nebo jejich *KCV*, či jejich *hodnoty*, můžeme použít metodu **get**.

```
List<ManagedKey> get(ManagedKeyFilter filter) {
    return _storage
        .getAllManagedKeys()
        .where((key) => switch (filter.type) {
            ManagedKeyFilterType.name =>
                key.name.toUpperCase().contains(filter.searched.toUpperCase()),
            ManagedKeyFilterType.value => key.data
                .toUpperCase()
                .contains(filter.searched.withoutWhitespaces().toUpperCase()),
            ManagedKeyFilterType.kcv => (key.kcv ?? '')
                .toUpperCase()
                .contains(filter.searched.toUpperCase()),
        })
        .toList();
}
```

Ta vezme všechny klíče z lokálního úložiště a vyfiltruje je pomocí předem specifikovaných parametrů v objektu **ManagedKeyFilter**. Pokud chceme například najít klíč, který má část své hodnoty **AB06**, tak nám tato metoda vrátí pouze ty klíče, které tuto podmínku splňují. V UI se navíc ještě barevně zvýrazní hledaná část klíče.



Obrázek 21 Filtrování klíčů v sekci Key Manager

(zdroj: vlastní)

Vlevo dole se nachází plovoucí tlačítko se symbolem „+“. To slouží k přidání nového klíče do Key Managera. Po kliknutí se uživateli otevře dialogové okno s formulářem pro vytvoření nového klíče. Zde vyplní název klíče (který musí být unikátní), jeho hodnotu (kterou si může nechat i vygenerovat), KCV daného klíče a nepovinný popis klíče v případě, že si chce k danému klíči něco poznamenat. Poté klikne na tlačítko „Create“ a nový klíč se zapíše do úložiště.

Podobným způsobem lze klíče i editovat. Vpravo u každého klíče jsou dvě ikonky. Ikonka tužky otevře uživateli ten samý dialog, tentokrát ale sloužící k editaci. Uživatel může editovat vše kromě názvu klíče. Ikonka odpadkového koše pak slouží ke smazání daného klíče z Key Managera. Po kliknutí je uživatel upozorněn dialogem, že se daný klíč trvale smaže z úložiště a jeho smazání musí ještě potvrdit. Tím se lze vyvarovat situaci, že uživatel klikne na ikonku koše omylem a klíč nedopatřením natrvalo smaže.

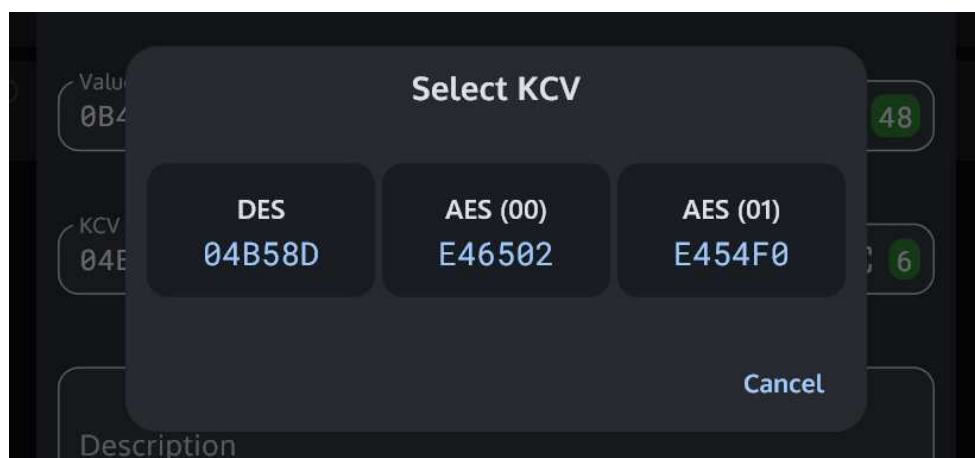
Dialog pro editaci klíče vypadá následovně:



Obrázek 22 Dialog pro editaci klíče

(zdroj: vlastní)

I když kalkulačka obsahuje separátní sekci pro výpočet KCV, tak počítat jeho hodnotu pro každý klíč zvlášť před jeho uložením by bylo otravné a nepraktické. Proto při vytváření klíče a jeho jakékoliv úpravě aplikace nabízí možnost automatického přepočtu. Problém ale nastává v tom, že existuje víc jak jeden způsob, jak vypočítat KCV (minimálně 1 pro DES klíče a 2 pro AES klíče). V případě, že pro daný klíč existuje více způsobů, jak KCV vypočítat, tak aplikace uživateli v dialogovém okně jednoduše nabídne, kterou hodnotu chce použít.

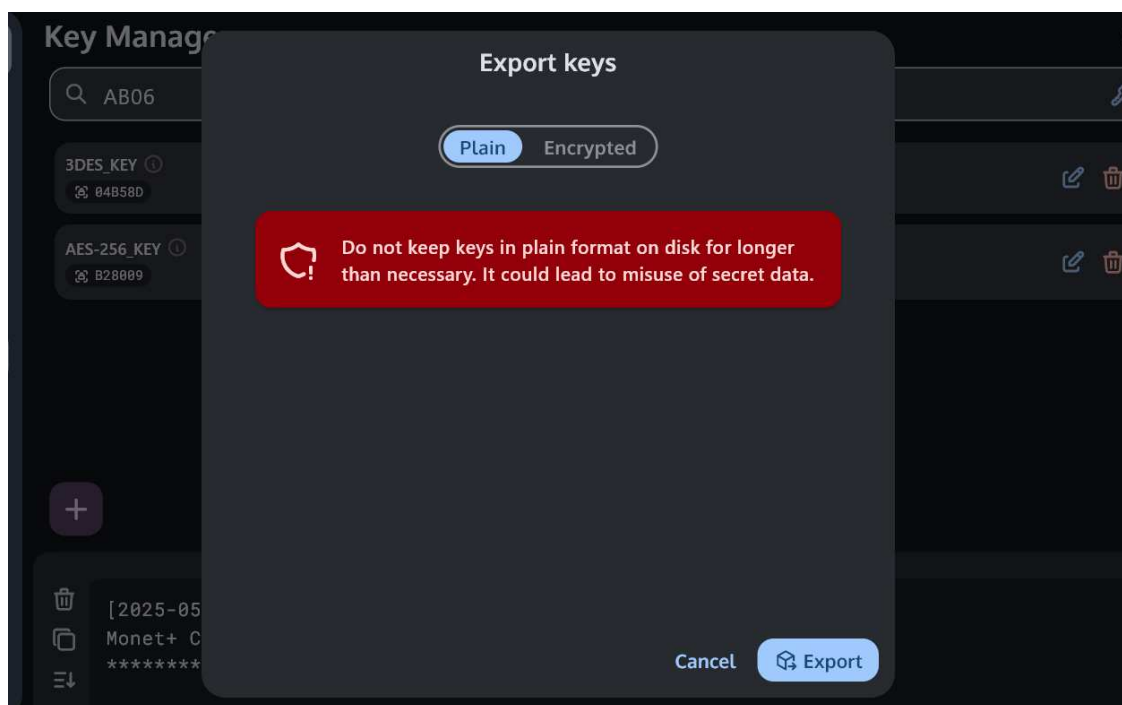


Obrázek 23 Výběr KCV při ukládání klíče

(zdroj: vlastní)

6.3.2 Export a import klíčů

V sekci Key Manager si lze v pravém horním rohu všimnout malé ikonky klíče v kruhu. Tuto ikonku jsem zvolil jako reprezentaci Key Managera napříč celou aplikací. V této sekci slouží ale ke dvěma účelům, exportu a importu klíčů z a do aplikace. Po zvolení možnosti exportu klíčů se objeví následující dialog:



Obrázek 24 Export nešifrovaných klíčů

(zdroj: vlastní)

Exportovat klíče lze ve dvou variantách – nešifrovaně (plain) či zašifrovaně (encrypted). Jak nás dialog upozorňuje, tak klíče vyexportované v nešifrované podobě bychom neměli nechávat na disku po dobu delší než je nutné. Tento export slouží například pouze k tomu, pokud chce uživatel klíče s někým sdílet, nebo je migrovat z jedné platformy na druhou (například z desktopu na Android). Po migraci klíčů by měl uživatel soubor s nešifrovanými daty okamžitě smazat.

Po kliknutí na tlačítko export se všechny uložené klíče exportují do jednoho souboru s příponou **.cckeys** a uloží se ve složce se staženými soubory dané platformy s názvem souboru nastaveným podle data a času exportu ve formátu **yyyy-MM-dd_HH-mm-ss** (například **2025-05-05_16-39-02.cckeys**). O export klíčů se stará třída **KeyImportExportController**. K exportu nezašifrovaných klíčů je použita následující metoda:

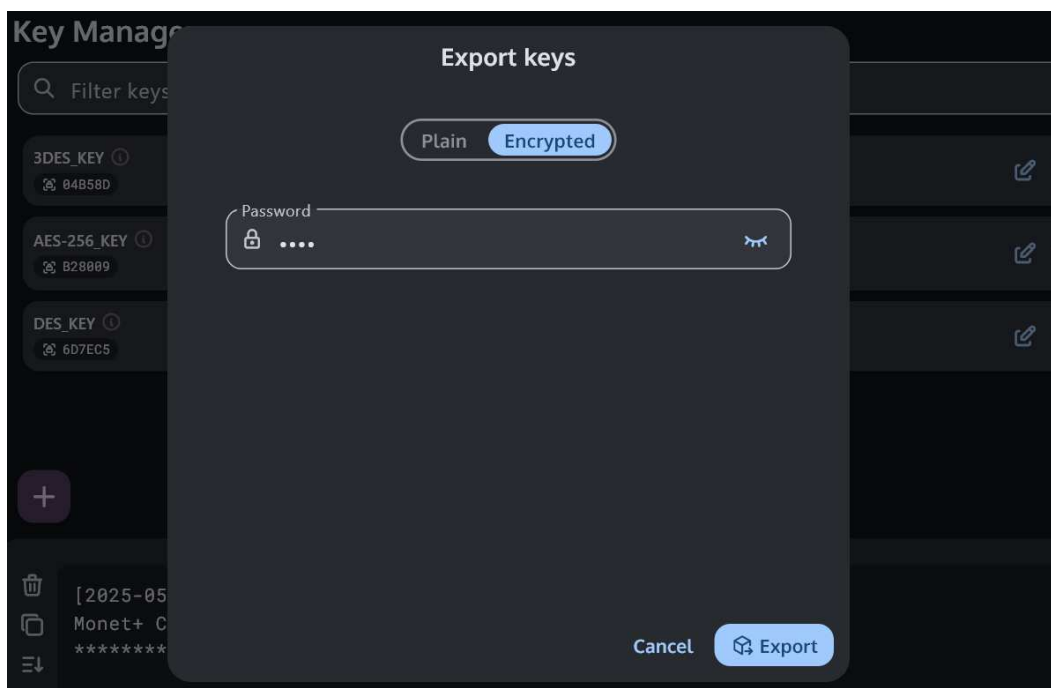
```
Future<OperationResult> exportPlain(List<ManagedKey> keys) async {  
  try {  
    final String encoded = _encodeKeys(keys);  
    final String fileName = '${DateTime.now().asFileName()}.${plainSuffix}';  
    return await FileHandler.download(fileName, encoded);  
  } on Exception catch (e) {  
    return OperationResult.fail(e.message);  
  }  
}
```

Pro uložení klíčů je použita třída **FileHandler**, která zajišťuje ukládání souborů tak, aby fungovalo na všech platformách. Pokud se vyexportovaný soubor otevře v textovém editoru, tak je jasné, že se jedná o JSON obsahující jednotlivé atributy klíčů:

```
[{"name":"3DES_KEY","data":"0B4436881ABFA8FA9D7D2B3D7848AB06D24E36A988DBE1D8","kcv":"04B58D","description":null}, {"name":"AES-256_KEY","data":"835C2D8DB139CAE9AB069EBAFC3A951DE7460EF4A48814DAE92C72FAA9337B35","kcv":"B28009","description":null}, {"name":"DES_KEY","data":"FC9471B6221A AFC9","kcv":"6D7EC5","description":null}]
```

Díky tomu je obsah souboru CCKEYS jednoduše čitelný i pro člověka. Zároveň, pokud se v budoucnu přidá klíči nějaký nepovinný atribut, tak budou i klíče exportované z předchozí verze aplikace zpětně kompatibilní s novou verzí.

Pokud se klíče exportují šifrovaně, tak se použije následující dialog:



Obrázek 25 Export šifrovaných klíčů

(zdroj: vlastní)

Zde je potřeba ještě navíc zadat heslo, podle kterého se obsah exportovaného souboru zašifruje. Ve třídě KeyImportExportController toto provádí následující metoda:

```
Future<OperationResult> exportEncrypted(
    List<ManagedKey> keys,
    String password,
) async {
    try {
        final Uint8List key = FileEncryption.generateKey(password);
        final String encrypted = FileEncryption.encrypt(_encodeKeys(keys), key);
        String fileName = '${DateTime.now().asFileName()}.${encryptedSuffix}';
        return await FileHandler.download(fileName, encrypted);
    } on Exception catch (e) {
        return OperationResult.fail(e.message);
    }
}
```

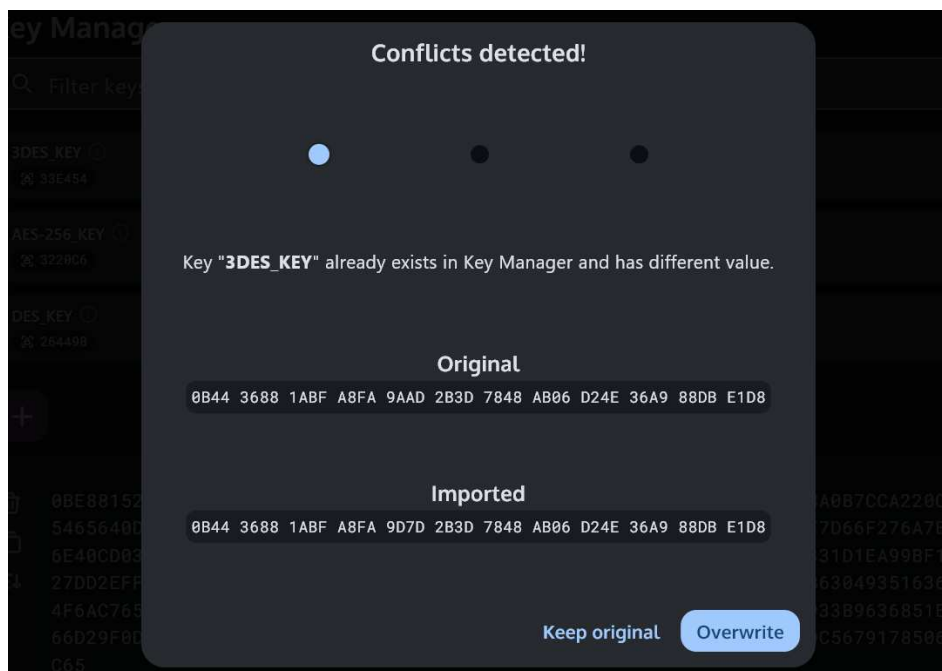
K zašifrování se využívá modul, který jsem nazval **FileEncryption**. Ten nejdříve z uživatelem zadaného hesla vytvoří AES-256 klíč a to tak, že vytvoří SHA-256 otisk hesla, jehož bajty použije jako kryptografický klíč. JSON s exportovanými klíči poté zašifruje AES-256 v CBC módu a soubor uloží s příponou **.ccekeys**. Obsah souboru je zašifrovaný a kódovaný v Base64. Pokud se tento soubor otevře v textovém editoru, tak obsahuje následující:

```
1H11roWsWZw0hvUF9mb4sZzKfYs2ebBBnMxwq8XBsJwTWP73JSc+ugqw4fmtImOMQKvbx6cJTK9Fzk+qQYLqdqh
AuJJZu+WvFmYK0xMjasWye/bwxN+4aaxYL84Nsu7/LK63UxswiIIqrq7B/DMdWpI/5X0i4VM44E/1mbayC+iBUl
ikUTuURx5W0Vqz4ALrC/YBfPrVR1waW8t02dJYoLfMoiDA9WG8lUZWQNN+laF2A4pr5eF3rb1FBMlvM2nMNM+v0f
QTDcyp99ZvJ2p72MvJe25AzQNVsworEvXZeX7xu92MXyA/nSPYS5Un/kQP+zPXQzHR6pm/GxpK5yJ90u/071LhfK
30To1EUvJKPnznG35R/CY8D6JZMKRzGIYwSTUWNonHM2RPasdlNMxby8+KAinobr0phcH4xb/0q9fI4ct+i8vy7i
kzuWNoUeoZuDQWbSnw0ki150yEvKBoB859S08CDS59qMuArSgbXXvKRpsMVnkXhQYunqYkx1
```

O klíčích teď nelze zjistit nic, jelikož jsou zašifrované a navíc kódované v Base64. Díky tomu může uživatel tento soubor bez starostí zálohovat na disku, protože bez tajného hesla nepůjde jeho obsah dešifrovat (to i v případě, že uživatel heslo zapomene).

Import funguje podobně jako export, ale v opačném pořadí. Nejdříve uživatel zvolí soubor s klíči (CCKEYS nebo CCEKEYS), které chce do kalkulačky importovat. Podle přípony souboru aplikace pozná, zda jsou klíče šifrované či nikoliv. Pokud ano, tak vyzve uživatele k zadání tajného hesla. Obsah souboru pak eventuálně dešifruje a následuje samotný proces importu. Může ovšem nastat situace, že uživatel už klíč s daným jménem v Key Manageru uložený má. V ten moment provede aplikace tzv. detekci konfliktů. Porovná importované klíče s těmi, které už v Key Manageru jsou a vyhodnotí případné

konflikty. Pokud už importovaný klíč v aplikaci je, ale má stejnou hodnotu, nebo v aplikaci není vůbec, tak je automaticky importován. Pokud ovšem už v Key Manageru existuje a zároveň má rozdílnou hodnotu, tak je detekován konflikt a uživateli se zobrazí dialogové okno, kde může rozhodnout, zda se hodnota klíče přepíše hodnotou importovaného klíče, nebo se ponechá původní klíč.



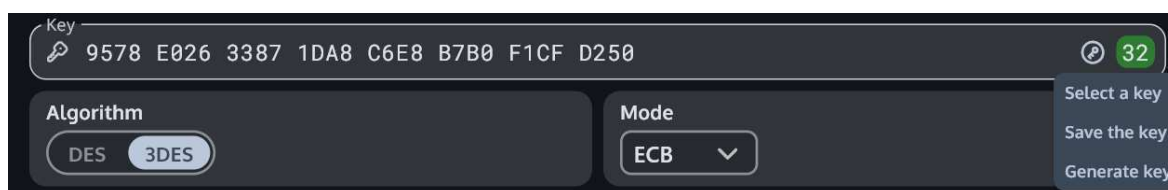
Obrázek 26 Konflikt při importu klíčů

(zdroj: vlastní)

Díky tomu se uživatel nemusí obávat, že by při importu nedopatřením přepsal klíče, které už v kryptokalkulačce jsou.

6.3.3 Tlačítko Key Manager

V ostatních částech aplikace se tlačítko Key Managera vyskytuje kdekoli, kde lze využít uložený klíč, typicky v textovém poli na zadání kryptografického klíče. Key Manager je zde reprezentován ikonkou klíče v kruhu.



Obrázek 27 Tlačítko Key Manager v textovém poli

(zdroj: vlastní)

Po kliknutí na ikonku jsou uživateli nabídnuty tři možnosti. Buď kryptografický klíč vybrat z uložených klíčů, nebo klíč, který se právě nachází v textovém poli uložit do Key Managera, či vygenerovat nový klíč.

Po kliknutí na „**Select a key**“ se uživateli otevře dialogové okno se seznamem všech uložených klíčů. Ty lze opět filtrovat podle názvu, KCV, či hodnoty. Po kliknutí na klíč je klíč automaticky doplněn do textového pole.

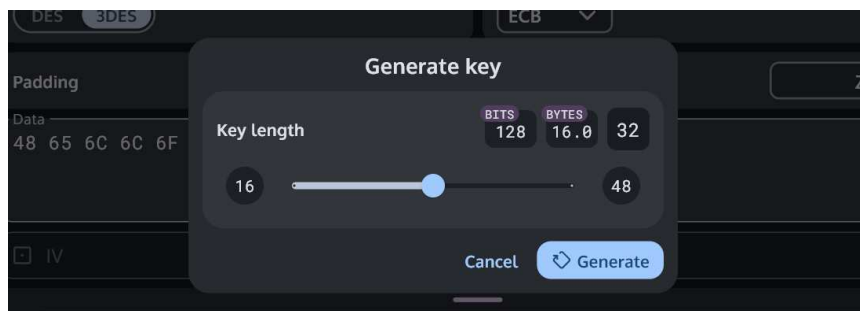


Obrázek 28 Dialog pro výběr klíče

(zdroj: vlastní)

Po kliknutí na „**Save the key**“ se uživateli objeví dialogové okno s formulářem pro vytvoření nového klíče jako v sekci Key Manager. Hodnota klíče se vyplní z textového pole. Uživatel doplní název klíče a další případné atributy a může klíč uložit.

Po kliknutí na „**Generate key**“ se zobrazí následující dialog:



Obrázek 29 Dialog pro generování klíče

(zdroj: vlastní)

Zde uživatel může zvolit délku klíče a ten následně vygenerovat pseudonáhodným generátorem. Délka klíče je zde primárně uvedena v nibblech, což se rovná délce hexadecimálních znaků, které je třeba použít na zapsání bajtové hodnoty klíče. Délka se rovněž zobrazuje i v bitech a bajtech. Po zvolení délky a kliknutí na tlačítko Generate se vygenerovaný klíč automaticky doplní do textového pole.

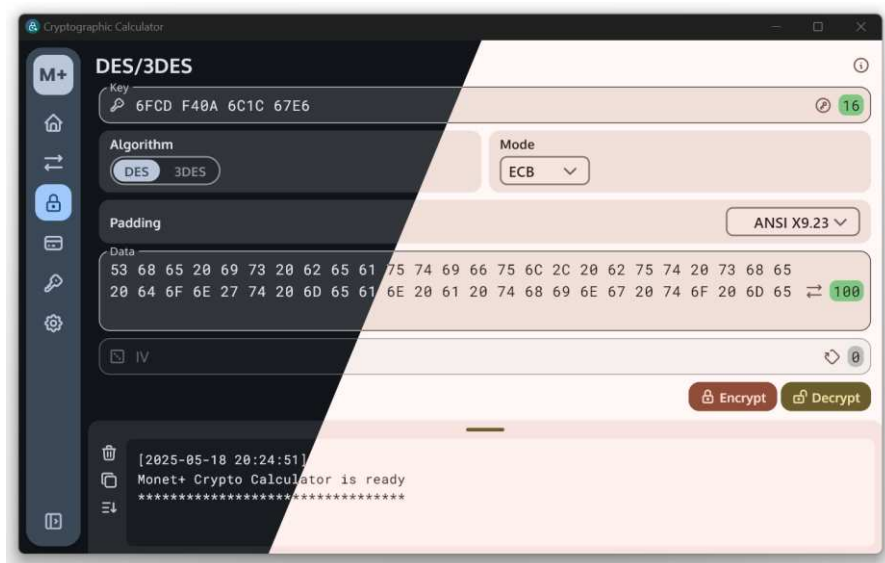
Díky tomuto tlačítku bude Key Manager uživateli dostupný v kterékoliv části aplikace, kde se zrovna manipuluje s kryptografickými klíči.

6.4 UI a nastavení aplikace

Než přijde popis implementací jednotlivých sekcí, tak mi dovolu krátkou zmínku o UI vytvořeném pro aplikaci, a co vše lze v aplikaci nastavit při jejím používání.

6.4.1 Vlastní UI

U aplikace jsem chtěl docílit toho, aby měla vlastní, moderní, vizuálně čisté, intuitivní a přehledné UI. Flutter v základu nabízí vytvářet UI ve dvou stylech – Material, napodobující UI Androidu a Cupertino, vycházející z UI iOS a macOS. Jako základ jsem zvolil právě Material widgety, každý jednotlivý prvek jsem však upravil a přizpůsobil tak, aby tvořil jednotnou vizuální identitu aplikace. Kryptokalkulačka rovněž podporuje tmavý i světlý motiv, který lze přepínat v nastavení. Celé UI si lze taktéž sladit do jedné šesti barev.

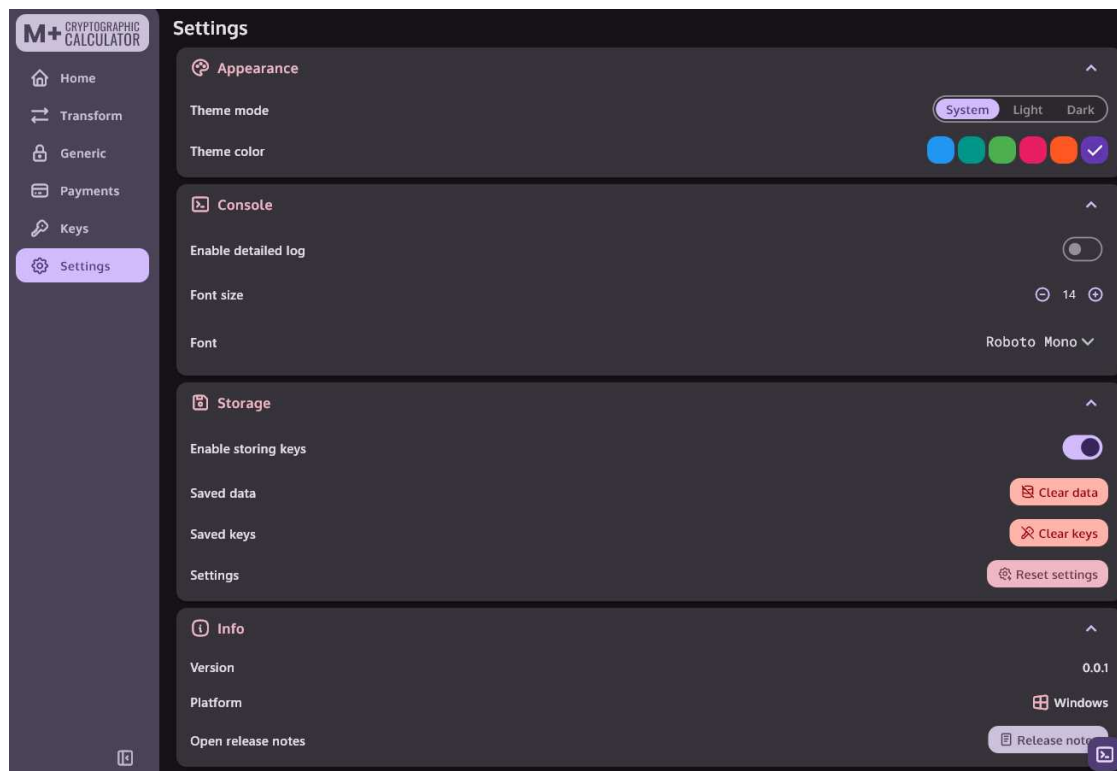


Obrázek 30 Tmavě modrý a světle oranžový motiv

(zdroj: vlastní)

6.4.2 Nastavení aplikace

Aby se uživatel s aplikací co nejlépe pracovalo, tak má aplikace sekci nastavení, kde si může pracovní prostor přizpůsobit dle svých potřeb.



Obrázek 31 Sekce Settings

(zdroj: vlastní)

Kromě již zmíněného motivu UI si dle vlastních potřeb přizpůsobit i konzoli aplikace. V konzoli lze měnit velikost fontu. Dále si lze vybrat jeden z pěti různých neproporciálních fontů, který bude konzole používat. Nakonec lze pro konzoly povolit tzv. detailní log. Pokud nás zajímá pouze výsledek kryptografických operací, tak jej necháme vypnutý. Pokud nás ovšem zajímá k dané kryptografické operaci více podrobností, třeba jak vypadají data po aplikaci paddingu, tak jej zapneme. To lze demonstrovat například u sekce DES/3DES. Pokud je detailní log vypnutý, tak se na konzoli objeví následující:

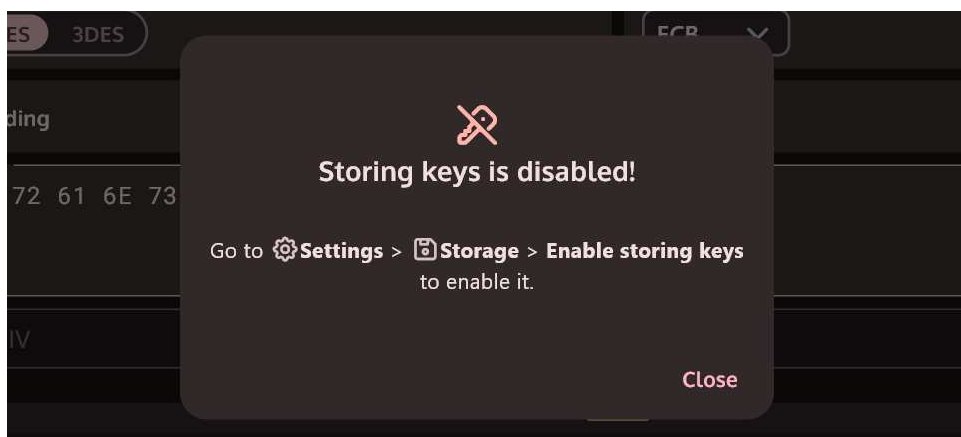
```
[2025-05-19 11:29:38]
DES/3DES ECB Encryption
*****
Key:      A9D8A6FB1B7DF25FEDDE76C1A997A876
Padding:  ANSI X9.23
Data:     5472616E7361746C616E74696369736D
-----
8DB8BF58EB5F9653ED3E5E83F3BCF4E91EE058DFC5773192
```

V logu konzole nyní vidíme pouze vstupní data, použitý klíč a padding. Pokud je ovšem detailní log v nastavení povolený, vypíše se následující:

```
[2025-05-19 11:38:18]
DES/3DES ECB Encryption
*****
Key:      A9D8A6FB1B7DF25FEDDE76C1A997A876
Padding: ANSI X9.23
Data:     5472616E7361746C616E74696369736D
-----
Used key:      A9D8A6FB1B7DF25FEDDE76C1A997A876A9D8A6FB1B7DF25F
Keying option: 2TDEA (Option 2)
Padded data:   5472616E7361746C616E74696369736D0000000000000008
-----
8DB8BF58EB5F9653ED3E5E83F3BCF4E91EE058DFC5773192
```

Nyní vidíme kromě vstupních parametrů i další podrobnosti, například jaký klíč byl použit pro 3DES (jeho celá délka), jaká možnost použití klíče byla zvolena (2TDEA) a jak vypadají data po aplikaci paddingu.

Další nastavení se týká úložiště, zde lze smazat uložená data v aplikaci (kromě klíčů), neboli hive box `_dataBox`. Všechny klíče mimo ty uložené v Key Manageru, neboli hive box `_keyBox`. Nebo obnovit nastavení na původní (`_settingsBox`). Před smazáním dat či klíčů je ještě uživatel upozorněn dialogem, který musí potvrdit. Pokud uživatel nechce, aby se z bezpečnostních důvodů zapisovaly klíče do lokálního úložiště, tak to lze v aplikaci zakázat vypnutím „**Enable storing keys**“. V ten moment jsou všechny zatím uložené klíče smazány a to včetně těch v Key Manageru. Funkce Key Managera jsou rovněž omezeny pouze na generování klíčů. Pokud se pak pokusíme v kterékoliv části aplikace například uložit klíč, tak se objeví následující banner:



Obrázek 32 Banner Storing keys is disabled!

(zdroj: vlastní)

Poslední část nastavení je pouze informační. Je zde uvedena verze aplikace a platforma, na které aplikace zrovna běží (pokud by uživatel náhodou zapomněl, jaký má OS). Nakonec je zde tlačítko, které nás přesměruje do sekce **Release Notes**, kde jsou vypsány podrobnější body ke každé nové verzi aplikace.

6.5 Sekce DES/3DES

Nyní se přesuneme k popisu jednotlivých sekcí aplikace. Jako první je zde implementace DES a 3DES. Sekci DES/3DES jsem umístil do skupiny zvané *Generic*, tedy skupiny, která obsahuje obecné kryptografické algoritmy. Sekce vypadá následovně:

```
[2025-05-19 16:29:11]
DES/3DES CTR Encryption
*****
Key: 64AC36C35A58D2B3
Padding: ISO 10126
Data: 4920647265616D742077652073706F6B6520616761696E
IV: BD1FCC3757A2170F
-----
2BA5AD09F75CF2DA27A9BF8F6C86A8591C012E8339BBC702
```

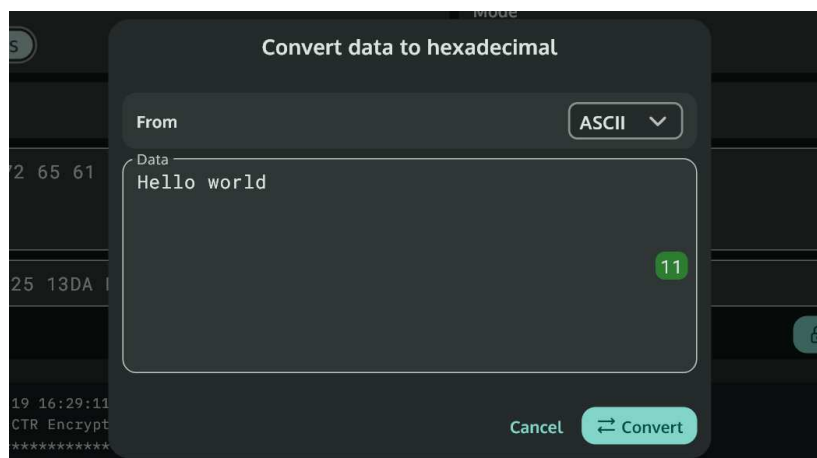
Obrázek 33 Sekce DES/3DES

(zdroj: vlastní)

Sekce obsahuje textová pole pro zadání klíče, dat a IV. Dále lze zvolit, zda chceme použít DES či 3DES, jaký mód blokové šifry bude použit a jaký padding se použije. Na konci textových polí si můžeme všimnout čísla, které ukazuje počet znaků zadaných v poli. V případě hexadecimálních dat jsou ignorovány mezery. Pokud uživatel zadal vstup, který má správnou délku (např. u DES klíče, jenž má délku 16 nibblů), tak číslo zezelená. Pokud ne, tak je číslo oranžové a zároveň se deaktivují tlačítka Encrypt a Decrypt. Po najetí kurzorem na tlačítko je uživateli sděleno, proč nemůže provést operaci (např. klíč nemá správnou délku).

Dále si můžeme všimnout v poli pro data malého tlačítka se dvěma šipkami. To po kliknutí otevře dialogové okno, do kterého můžeme zadat data v jiném kódování než jako

hexadecimální reprezentace bajtů. Po kliknutí na Convert jsou tyto data převedena ze svého kódování na hexadecimální data.



Obrázek 34 Dialog pro konverzi dat

(zdroj: vlastní)

Díky tomu může uživatel převést například ASCII text bez nutnosti si text převádět z jiného kódování předem, například v sekci **Data Convertor** této aplikace.

V pravém horním rohu sekce je malá ikona i v kroužku. Ta po kliknutí otevře informační část dané sekce.

M+
Home
Back
Lock
List
Key
Settings
Info

DES/3DES

Born in Germany, **Horst Feistel** immigrated to the United States in 1934. Decades later, while working at IBM's Thomas J. Watson Research Center, Feistel and his team developed an early block cipher called **Lucifer**. In 1973, the US National Bureau of Standards (NBS, now NIST) began soliciting proposals for a federal encryption standard to secure business and government communications. IBM submitted a variant of Lucifer for consideration.

The National Security Agency (NSA) reviewed the proposal and influenced some aspects of the cipher's design, most notably reducing the key length from the original 128 bits to **56 bits**. This reduction made the cipher more vulnerable to brute force attacks as computing power increased, but was considered sufficient for commercial use at the time. The modified cipher was adopted as the **Data Encryption Standard (DES)** in 1976.

How DES works

The Data Encryption Standard is a **symmetric block cipher** that encrypts data in **64-bit blocks** using a key that is technically **64 bits** long, but only 56 bits are used for encryption, while the remaining 8 bits are used for parity checking and do not contribute to security. It works by repeatedly applying a complex sequence of substitutions and permutations over **16 rounds** each round mixing the data with a subkey derived from the main key. The process involves splitting the data block, transforming it through S-boxes for non-linearity, and swapping the halves in a structure called a **Feistel network**. Due to its relatively short key length, it is now considered insecure against modern brute force attacks.

Triple DES

Triple DES (3DES, TDES or DES-EDE) is a **symmetric block cipher** that enhances the security of DES by applying the DES algorithm **three times** in sequence to each **64-bit blocks** of data, typically using either two or three DES keys. For encryption we can use the following formula:

$$C = E_{K3}(D_{K2}(E_{K1}(P)))$$

Where **E** is the DES encryption function, **D** is the DES decryption function, and **K1**, **K2** and **K3** are the keys used. **P** stands for plain data and **C** for encrypted data. The decryption reverses the encryption steps:

$$P = D_{K1}(E_{K2}(D_{K3}(C)))$$

Keying options

3DES standards define three keying options:

←

Obrázek 35 Popis sekce DES/3DES

(zdroj: vlastní)

Zde je konkrétní šifra více popsána, včetně stručné historie vzniku, a také toho, jak funguje, všech přítomných módů provozu blokových šifer či paddingů. Díky tomu se může i uživatel, který je v kryptografii nováčkem, dozvědět něco málo o dané šifře, či zjistit, co jednotlivá nastavení znamenají.

Samotné šifrování má na starosti třída zvaná **DesController**. Ta dědí abstraktní třídu **CryptoController**. Ten slouží k provádění kryptografických operací jako šifrování a dešifrování. Na konci každé operace updatuje konzoli a přidá do ní nový log. Rovněž odchyťává případné výjimky při provádění operací. Pokud se například pokusíme šifrovat bez paddingu (padding *None*) a zároveň data nemají délku odpovídající násobku bloků, tak se na konzoli vypíše následující:

```
[2025-05-19 17:12:35]
DES/3DES ECB Encryption
*****
Key:      8FE171D4F6307A52
Padding: None
Data:     4920647265616D7420776520737066F6B6520616761696E
-----
Error: Data must be 64-bit blocks. Perhaps use some padding.
```

CryptoController rovněž odchyťává i výjimky, které mohou vzniknout v balíčcích nebo ve frameworku samotném. Ty značí konzole jako Unexpected error.

K provádění kryptografických operací ve většině částí aplikace jsem použil balíček zvaný **pointycastle** [36]. Jedná se o balíček od tvůrců populární kryptografické Java knihovny Bouncy Castle. Ten, nejen že obsahuje široké množství různých šifer a algoritmů, ale rovněž funguje na všech Flutterem podporovaných platformách. Balíček se rovněž používá podobným způsobem jako jeho Java varianta, takže vývojář, který má zkušenosti s Bouncy Castle v Javě, nebude mít problém implementovat stejné algoritmy i v Dartu.

V případě 3DES je implementace pomocí pointycastle opravdu jednoduchá:

```
String _desProcess(DesMode mode, String key, String data, String iv, bool encrypt) {
  var params = mode == DesMode.ecb
    ? KeyParameter(key.unhex())
    : ParametersWithIV(KeyParameter(key.unhex()), iv.unhex());
  var cipher = BlockCipher('DESede/${mode.label}')..init(encrypt, params);
  final Uint8List input = data.unhex();
  final Uint8List output = Uint8List(input.length);
  for (var offset = 0; offset < input.length; offset += mode.blockSize) {
    cipher.processBlock(input, offset, output, offset);
  }
  return output.hex();
}
```

Používám BlockCipher zvanou DESede (neboli DES encrypt, decrypt, encrypt), což je pouze jiný název pro triple DES. Mode nastavím podle toho, který uživatel zvolil. Padding aplikuji na data ještě před začátkem šifrování. Balíček pointycastle ovšem neobsahuje klasický DES algoritmus, jelikož se už nepovažuje v dnešní době za bezpečný. V tomhle případě si upravím před šifrováním klíč tak, aby platilo, že $K1 = K2 = K3$. V tenhle moment sice šifruji pomocí triple DES, nicméně ve zpětně kompatibilním režimu k DES. Operace decrypt vyruší jednu operaci encrypt.

6.6 Sekce AES

Nyní se přesuneme k o něco modernější šifře a to konkrétně k AES. I ta má ve skupině *Generic* dedikovanou vlastní sekci.

The screenshot shows the AES encryption interface. The Key field contains the hexadecimal string: 9096 243F 12BD 54CE 4F39 10E0 53B9 5D58 1C6C DF13 8133 3BE8. The Mode is set to CBC and the Padding is set to PKCS#7. The Data field contains the hexadecimal string: 48 65 72 65 20 74 6F 20 66 6F 72 65 76 65 72. The IV field contains the hexadecimal string: 4DE6 5CD3 A724 ADC6 A706 56DE 8451 C8B5. The Encrypt and Decrypt buttons are visible. The output section shows the following text:

```
[2025-05-19 20:50:50]
AES-192 CBC Encryption
*****
Key: 9096243F12BD54CE4F3910E053B95D581C6CDF1381333BE8
Padding: PKCS#7
Data: 4865726520746F20666F7265766572
IV: 4DE65CD3A724ADC6A70656DE8451C8B5
-----
CA1A539AD64525B4407833884BB2530E
```

Obrázek 36 Sekce AES

(zdroj: vlastní)

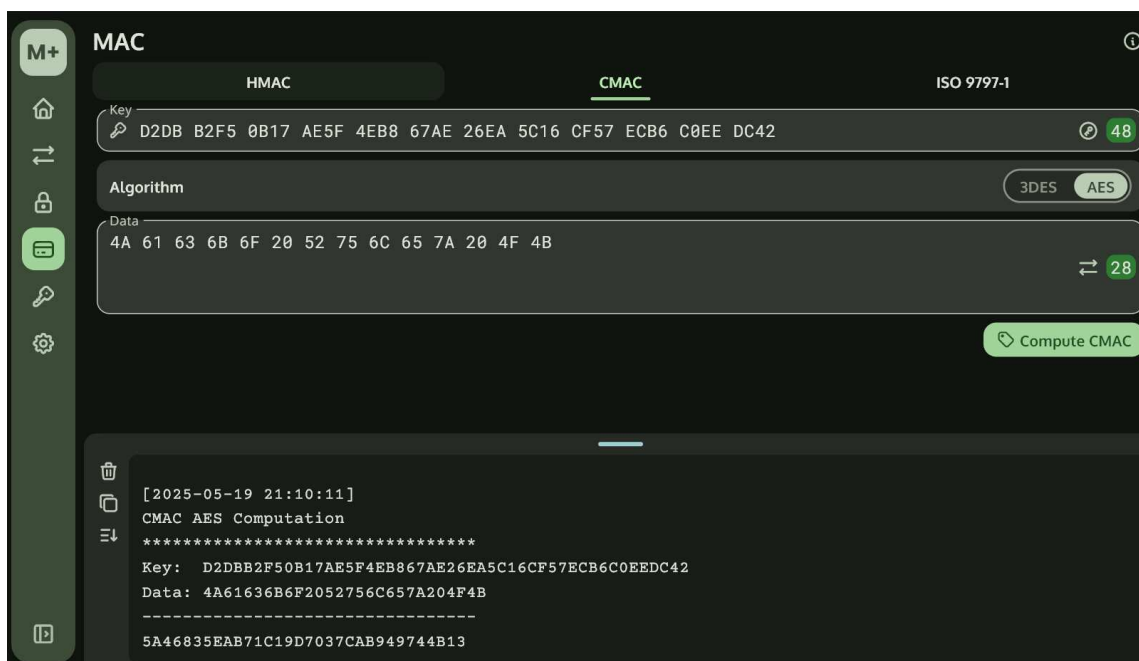
I zde je přítomna informační část popisující AES, tlačítko Key Manager v textovém poli klíče, tlačítko na konverzi dat či tlačítko na vygenerování náhodného IV. Lze si vybrat z několika módů blokové šifry i z několika různých paddingů. Není zde sice na výběr, zda chceme použít AES-128, AES-192 a nebo AES-256, nicméně to si aplikace automaticky zjistí podle délky klíče a zalogue tuto informaci na konzoli.

Implementace logiky po stisku tlačítka Encrypt či Decrypt je opět řešena ve třídě zvané **AesController**, která dědí abstraktní **CryptoController**. Implementace pomocí balíčku pointycastle je podobně jednoduchá jako v případě 3DES:

```
String _aesProcess(AesMode mode, String key, String data, String iv, bool encrypt) {
    var params = mode == AesMode.ecb
        ? KeyParameter(key.unhex())
        : ParametersWithIV(KeyParameter(key.unhex()), iv.unhex());
    var cipher = BlockCipher('AES/${mode.label}')..init(encrypt, params);
    final Uint8List input = data.unhex();
    final Uint8List output = Uint8List(input.length);
    for (var offset = 0; offset < input.length; offset += mode.blockSize) {
        cipher.processBlock(input, offset, output, offset);
    }
    return output.hex();
}
```

6.7 Sekce MAC

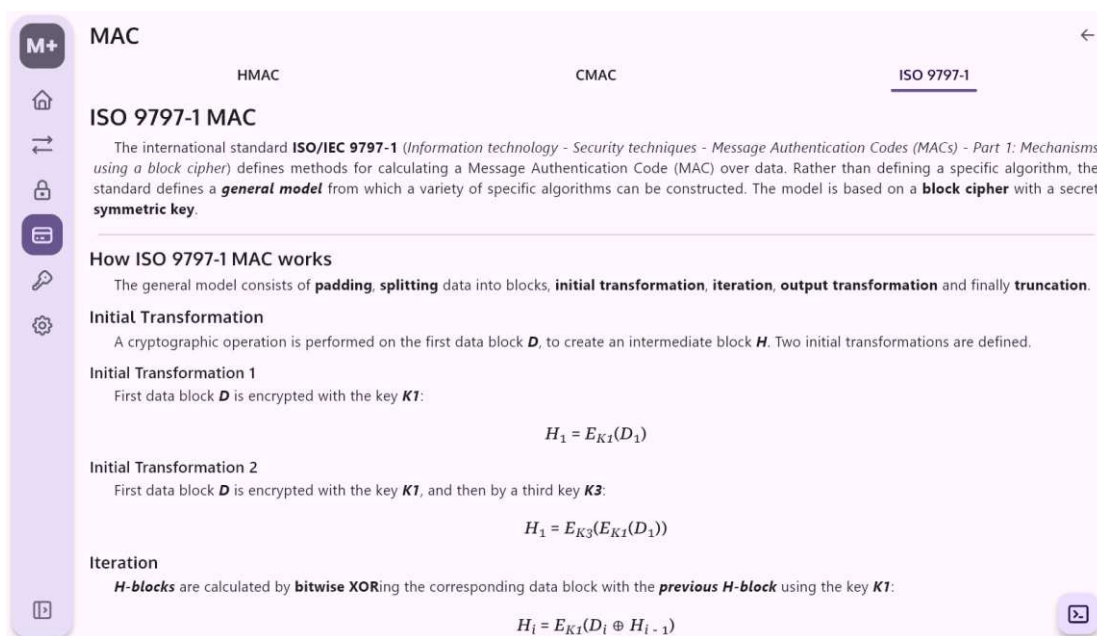
Ted' se z oblasti obecné kryptografie přesuneme do oblasti bankovníctví, neboli k částem aplikace, které jsem zařadil do skupiny nazvané *Payments*. MAC se sice používá i mimo oblast bankovníctví, nicméně v kontextu firmy Monet+ se používá především tam. Sekce MAC vypadá následovně:



Obrázek 37 Sekce MAC

(zdroj: vlastní)

Sekce je tentokrát rozdělena do tří podsekci – HMAC, CMAC a ISO 9797-1. Zde jsou implementovány rozdílné druhy MAC algoritmů. Kliknutím na taby v horní části aplikace lze mezi nimi přepínat. Všechny podsekce obsahují textové pole pro zadání klíče a dat. V sekci HMAC si lze zvolit který HASH algoritmus chce uživatel k výpočtu využít. V sekci CMAC pak zase která bloková šifra se použije. V ISO 9797-1 MAC implementaci pak uživatel nastavuje, který ze čtyř podporovaných algoritmů se použije, a který padding bude aplikován. Každá podsekce má rovněž svoji informační část, mezi kterými lze opět přepínat za pomoci tabů.



Obrázek 38 Popis sekce MAC

(zdroj: vlastní)

6.7.1 Podsekce HMAC

Za pomoci balíčku pointycastle, který HMAC plně podporuje, byla jeho implementace opravdu nenáročná.

```
String _hmacProcess(String key, String data, HashAlgorithm algorithm) {
    final hmac = Mac('${algorithm.label}/HMAC')
        ..init(KeyParameter(key.unhex()));
    final Uint8List output = hmac.process(Uint8List.fromList(data.unhex()));
    return output.hex();
}
```

Aplikace podporuje široké množství různých hash algoritmů, jako například algoritmy z rodin BLAKE, MD (Message Digest), RIPEMD, SHA (Secure Hash Algorithm) či hash funkce SM3, Tiger a Whirlpool.

6.7.2 Podsektce CMAC

Během implementace CMAC jsem ovšem narazil na problém. CMAC je počítán pomocí blokové šifry v CBC módu, kdy je úvodní inicializační vektor nastaven na nulové bajty. Balíček pointycastle sice opět CMAC podporuje, nicméně velikost IV nastavuje podle velikosti klíče. IV musí být stejně velký jako blok dat. To, že je klíč velký stejně jako blok dat platí pouze v případě AES-128. V případě AES-192 a AES-256 je klíč větší než blok, tudíž je i špatně nastavené IV. U 3DES je klíč dokonce 3x větší jak blok. Při snaze vypočítat CMAC tak knihovna ve všech případech kromě AES-128 vyhodí výjimku kvůli chybné velikosti IV. IV však nelze nastavit manuálně, jeho nastavení se provádí až v samotné knihovně. Z tohoto důvodu byl pro mě CMAC v knihovně pointycastle nepoužitelný.

Implementovat CMAC jsem proto musel sám. Implementoval jsem třídu zvanou **CmacEngine**. Ta využívá buď AESEngine nebo DESedeEngine z knihovny pointycastle.

```
class CmacEngine {
    /// Used block cipher
    late final BlockCipher cipher;

    /// Size of the block in bytes
    late final int blockSize;

    /// Reduction Polynomial for creating subkeys
    late final int rb;

    /// Custom engine for computing CMAC.
    CmacEngine(bool useAes) {
        cipher = useAes ? AESEngine() : DESedeEngine();
        blockSize = useAes ? 16 : 8;
        rb = useAes ? 0x87 : 0x1B;
    }

    /// Formats DES key so it can be used in 3DES engine.
    Uint8List _formatKey(Uint8List key) {
        return key.length == 8 ? key.bond(key).bond(key) : key;
    }

    /// Generate CMAC subkeys K1 and K2 used for XOR with last block
    List<Uint8List> _generateSubkeys(Uint8List key) {
        Uint8List zeroBlock = Uint8List(blockSize);
```

```

    UInt8List l = _encrypt(key, zeroBlock);

    UInt8List k1 = l.leftShift();
    if ((l[0] & 0x80) != 0) {
        k1[k1.length - 1] ^= rb;
    }

    UInt8List k2 = k1.leftShift();
    if ((k1[0] & 0x80) != 0) {
        k2[k2.length - 1] ^= rb;
    }

    return [k1, k2];
}

/// Encrypts block of [data] with [key] using current [cipher].
UInt8List _encrypt(UInt8List key, UInt8List data) {
    cipher.init(true, KeyParameter(key)); // Encrypt mode
    UInt8List output = UInt8List(blockSize);
    cipher.processBlock(data, 0, output, 0);
    return output;
}

/// Compute CMAC of [message] with given [key].<br>
/// Set [padApplied] to true, if [message] data were padded before.<br>
/// [padApplied] determines if subkey k1 or k2 will be used on last block.
UInt8List computeCMAC(UInt8List key, UInt8List message, bool padApplied) {
    key = _formatKey(key);
    List<UInt8List> subkeys = _generateSubkeys(key);
    UInt8List k1 = subkeys[0], k2 = subkeys[1];

    int n = (message.length / blockSize).ceil();

    // If data were padded, it uses subkey k2, else it uses k1
    var last = (message.sublist((n - 1) * blockSize)).xor(padApplied ? k2 : k1);

    UInt8List mac = UInt8List(blockSize);
    UInt8List temp = UInt8List(blockSize);

    // Processing full blocks
    for (int i = 0; i < n - 1; i++) {
        temp = temp.xor(message.sublist(i * blockSize, (i + 1) * blockSize));
        temp = _encrypt(key, temp);
    }

    // Processing last block
    temp = temp.xor(last);
    mac = _encrypt(key, temp);
}

```

```

    return mac;
}
}

```

Ten pak můžu v CmacControlleru použít místo CMAC implementace z balíčku.

```

String _cmacProcess(String key,
    String padded,
    CmacAlgorithm algorithm,
    bool padApplied,
) {
    CmacEngine cmacEngine = CmacEngine(algorithm == CmacAlgorithm.aes);
    return cmacEngine
        .computeCMAC(key.unhex(), padded.unhex(), padApplied)
        .hex();
}

```

6.7.3 Podsekcce ISO-9797-1

Posledním implementovaným algoritmem je ISO-9797-1 neboli MAC specifikovaný tímto standardem. Ten specifikuje obecný model, jak blokové šifry kombinovat, aby byl MAC vypočítán. Technicky může být použita jakákoliv bloková šifra pro výpočet, nicméně já svoji implementaci omezil pouze na DES, která se v oblasti bankovníctví používá nejčastěji. Balíček pointycastle tentokrát vlastní implementaci ISO-9797-1 nenabízí, nicméně opět s pomocí DESedeEngine si mohu vytvořit implementaci vlastní. Každý jednotlivý algoritmus se skládá z kombinace jedné ze dvou možných úvodních transformací, poté se zpracují jednotlivé bloky a provede se jedna ze tří možných výstupních transformací.

Například ISO 9797-1 Algoritmus 3, taktéž známý jako Retail MAC, je definován úvodní transformací 1 a výstupní transformací 3. Jeho implementace ve třídě, kterou jsem nazval **Iso9797MacEngine**, vypadá následovně:

```

/// Encrypts given [block] using given [key].
UInt8List _encryptBlock(UInt8List block, UInt8List key) {
    cipher.init(true, KeyParameter(key));
    return _processBlock(block);
}

/// Decrypts given [block] using given [key].
UInt8List _decryptBlock(UInt8List block, UInt8List key) {
    cipher.init(false, KeyParameter(key));
    return _processBlock(block);
}

/// Process given [block].
UInt8List _processBlock(UInt8List block) {

```

```
final output = Uint8List(cipher.blockSize);
cipher.processBlock(block, 0, output, 0);
return output;
}

/// D1 is encrypted with the key K
Uint8List _initTransform1(Uint8List block, Iso9797Keys keys) {
  return _encryptBlock(block, keys.k1);
}

/// Hq is decrypted with the key K' and the result encrypted with the key K
Uint8List _outTransform3(Uint8List block, Iso9797Keys keys) {
  return _encryptBlock(_decryptBlock(block, keys.k2), keys.k1);
}

/// This algorithm uses initial transformation 1 and output transformation 3.
Uint8List _macAlgorithm3(Uint8List message, Iso9797Keys keys) {
  var block = _initTransform1(message.sublist(0, cipher.blockSize), keys);

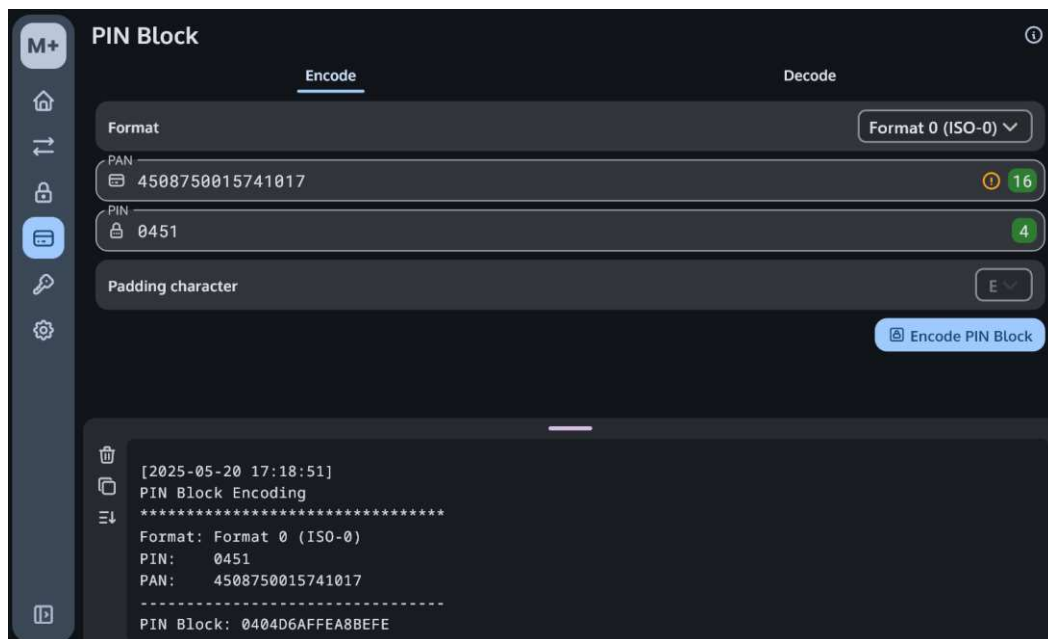
  for (int i = cipher.blockSize; i < message.length; i += cipher.blockSize) {
    block = _encryptBlock(
      block.xor(message.sublist(i, i + cipher.blockSize)),
      keys.k1,
    );
  }

  return _outTransform3(block, keys);
}
```

V současné době jsou implementovány Algoritmy 1 – 4 a Padding method 1 – 2 a pouze algoritmus DES, což je pro potřeby firmy zcela dostačující. Iso9797MacEngine je však koncipován tak, aby byl v případě potřeby rozšiřitelný například o podporu AES či dalších algoritmů.

6.8 Sekce PIN Block

Ve skupině Payments lze také nalézt sekci nazvanou PIN Block. Aplikace podporuje široké množství všemožných PIN Block formátů, ať už se jedná o ty definované standardem ISO 9564, ECI (Eurocheque International), IBM, VISA či AS2805. V informační části sekce je každý z těchto formátů popsán včetně postupu, jak jej vytvořit. Při zadání PANu do textového pole navíc aplikace ještě ověří, zda sedí takzvaný Luhnův algoritmus kontrolního součtu čísel. Pokud ne, tak se objeví na konci textového pole vykřičník v kroužku, jinak fajfka. I bez validního PANu však aplikace PIN Block vypočítá, tímto dává uživateli pouze jemně najevo, že PAN pravděpodobně není platný. Celá sekce vypadá následovně:



Obrázek 39 Sekce PIN Block

(zdroj: vlastní)

Implementace jednotlivých formátů PIN Blocku rovněž není složitá. Jedná se vesměs o jednoduché operace jako padding dat či vzájemný XOR dvou bloků dat. Například formát ISO-0 je implementován následujícím způsobem:

```
String _preparePin(int format, String pin, String padSequence) {
    String paddedPin = pin + padSequence;
    return '$format${pin.length.hex()}$paddedPin';
}

String _preparePan(String pan) {
    final String withoutLuhn = pan.substring(0, pan.length - 1);
    return '0000${withoutLuhn.substring(withoutLuhn.length - 12)}';
}

String _xor(String input1, String input2) {
    return input1.unhex().xor(input2.unhex()).hex();
}

PinBlockResult _encodeFormat0(String pin, String pan) {
    String preparedPin = _preparePin(0, pin, 'F' * (14 - pin.length));
    String preparedPan = _preparePan(pan);
    return PinBlockResult.fromPinBlock(_xor(preparedPin, preparedPan));
}

String _retrievePin(String rawPin) {
    int pinLength = int.parse(rawPin[1], radix: 16);
    return rawPin.substring(2, pinLength + 2);
}
```

```
String _decodeFormat0(String pinBlock, String pan) {  
    String preparedPan = _preparePan(pan);  
    String rawPin = _xor(pinBlock, preparedPan);  
    return _retrievePin(rawPin);  
}
```

Jednotlivé formáty se často navíc překrývají. Například ECI-1, VISA-1 či ANSI X9.8 je definován stejně jako ISO-0 formát.

6.9 Sekce KCV

Poslední implementovanou sekcí je sekce určená k výpočtu KCV z klíče ve skupině nazvané *Keys*. KCV si sice Key Manager dokáže u klíčů počítat sám, nicméně tahle sekce slouží uživateli, pouze pokud chce spočítat KCV nějakého klíče, aniž by ho chtěl uložit.

The screenshot displays the 'KCV' section of the 'M+ CRYPTOGRAPHIC CALCULATOR'. On the left is a sidebar with navigation options: Home, Transform, Generic, Payments, Keys (selected), and Settings. The main area has a header 'KCV' with an information icon. Below it is a 'Key' input field containing the hexadecimal string '0B44 3688 1ABF A8FA 9D7D 2B3D 7848 AB06 D24E 36A9 88DB E1D8' and a 'P' icon with the number '48'. A 'Compute all' button with a toggle switch is present. Below this, the 'Algorithm' is set to 'AES' (with 'DES' also available) and 'AES bytes value' is set to '00' (with '01' also available). A 'Compute KCV' button is at the bottom right. The output section shows a timestamp '[2025-05-20 17:40:34]', the text 'KCV Computation', and a separator line. Below the separator, the 'Key' is repeated, followed by the KCV results: 'DES: 04B58D', 'AES (00): E46502', and 'AES (01): E454F0'.

Obrázek 40 Sekce KCV

(zdroj: vlastní)

Sekce podporuje tři způsoby výpočtu KCV, jeden pomocí DES a dva pomocí AES. Uživatel může ovšem i zvolit možnost vypočítat všechny tři podporované způsoby zároveň, například v případě, že má klíč s uvedeným KCV, ale není si jistý, která metoda byla pro výpočet použita.

Pro počítání KCV jsem vytvořil třídu zvanou *KcvCalculator*. Ta počítá KCV šifrováním bloku nulových bajtů (či bajtů 01 v případě AES).

```
class KcvCalculator {
    KcvCalculator._();

    /// Formats DES key, DESede can't work with 8 bytes key natively
    static String _formatDesKey(String inKey) {
        String key = inKey.toUpperCase();
        if (key.length == 16) {
            return key + key + key;
        } else {
            return key;
        }
    }
}

/// Computes KCV for DES by encrypting 8 '00' bytes.
static String des(String key) {
    if (key.length == 16 || key.length == 32 || key.length == 48) {
        KeyParameter params = KeyParameter(_formatDesKey(key).unhex());
        BlockCipher cipher = BlockCipher('DESede/ECB')..init(true, params);
        return cipher.process(('00' * 8).unhex()).hex().substring(0, 6);
    } else {
        throw Exception('Invalid DES key length. Must be 8, 16 or 24 bytes!');
    }
}

/// Computes KCV for AES by encrypting given [bytes]. <br>
/// '00' bytes are usually used in retail, standard is however '01'.
static String aes(String key, [String bytes = '01']) {
    if (key.length == 32 || key.length == 48 || key.length == 64) {
        KeyParameter params = KeyParameter(key.unhex());
        BlockCipher cipher = BlockCipher('AES/ECB')..init(true, params);
        return cipher.process((bytes * 16).unhex()).hex().substring(0, 6);
    } else {
        throw Exception('Invalid AES key length. Must be 16, 24 or 32 bytes!');
    }
}
}
```

7 TESTOVÁNÍ

Tato kapitola popisuje poslední část práce a to testování. Průběžné testování jsem prováděl v průběhu celého vývoje. Výstupy kryptografických operací jsem často porovnával s výstupy z Python skriptů a knihoven, či jiných kryptografických nástrojů, abych měl nezávislý výsledek k porovnání.

Jelikož je aplikace multiplatformní, nabízí se provést testy na více platformách. Vybral jsem proto zástupce desktopové platformy – Windows, zástupce mobilní platformy – Android a zástupce webové platformy – prohlížeč Google Chrome. Definoval jsem následující uživatelský případ:

Preconditions: Nainstalovaná aplikace na dané platformě

Postconditions: Soubor s exportovanými klíči zůstane v zařízení i po vypnutí aplikace

Kroky:

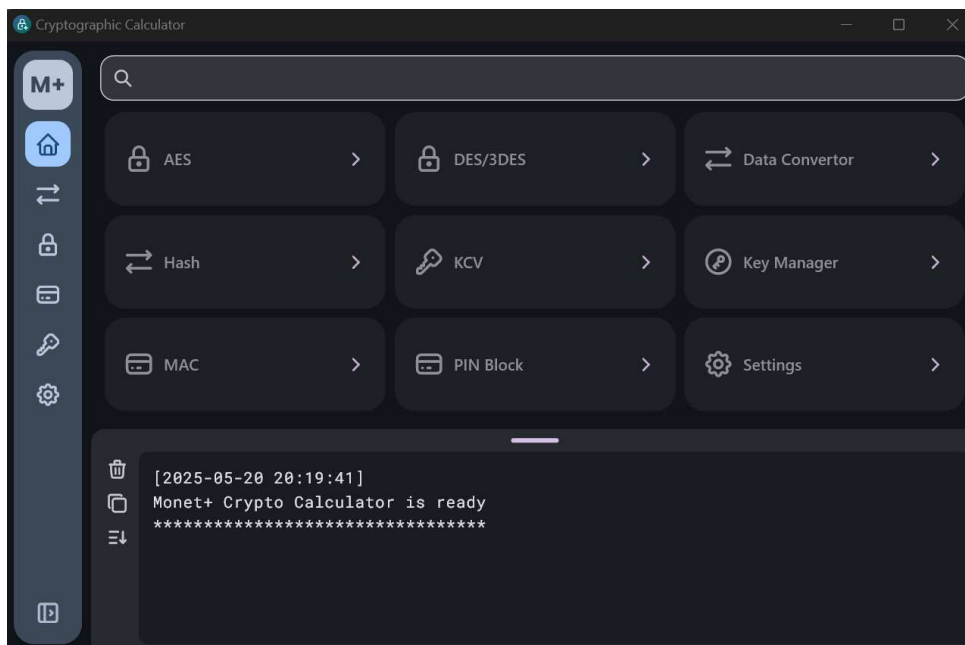
1. Uživatel spustí aplikaci *Cryptographic Calculator* na dané platformě
2. Uživatel přejde do sekce Key Manager, kde vytvoří nový klíč s názvem **MY_KEY** a hodnotou **000102030405060708090A0B0C0D0E0F**, KCV nechá dopočítat aplikaci (Zvolí metodu AES (00))
3. Uživatel klíč exportuje z Key Managera
4. Uživatel odstraní klíč **MY_KEY** z Key Managera
5. Uživatel klíč znovu naimportuje ze souboru CCKEYS
6. Uživatel přejde do sekce MAC a do podsekce ISO-9797-1. Vybere klíč s názvem **MY_KEY** z Key Managera, do textového pole zadá text „Hello Crypto“ kódovaný v ASCII a převede jej na hexadecimální reprezentaci. Jako algoritmus vybere **Algoritmus 3 (Retail)** a jako padding zvolí **ISO-9797-1 (1)** a klikne na tlačítko Compute ISO-9797-1 MAC
7. Zkontroluje, že se na konzoli objevil výsledek a vypne aplikaci

Očekávaný výsledek: Uživatel bez problému provede všechny uvedené kroky

Tímto uživatelským případem se otestuje několik funkcionalit aplikace, jako například Key Manager, počítání KCV či výpočet MAC. Měl by být proveditelný na všech testovaných platformách a na všech platformách by se aplikace měla chovat konzistentně a předvídatelně.

7.1 TC01 – Platforma Windows

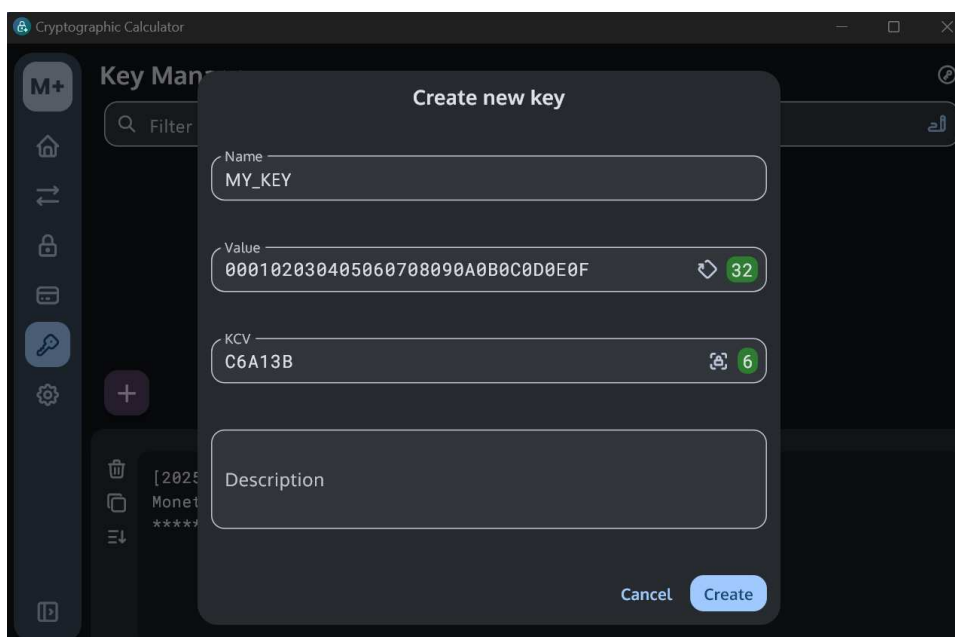
Na této platformě jsem prováděl většinu vývoje a průběžného testování. Operační systém je Windows 11 Pro v build verzi 22621.1555. Aplikace se zde spustí bez problému.



Obrázek 41 TC01 – Aplikace po spuštění

(zdroj: vlastní)

Přejdeme do sekce Key Manager, kde vytvoříme nový klíč.



Obrázek 42 TC01 – Vytvoření nového klíče

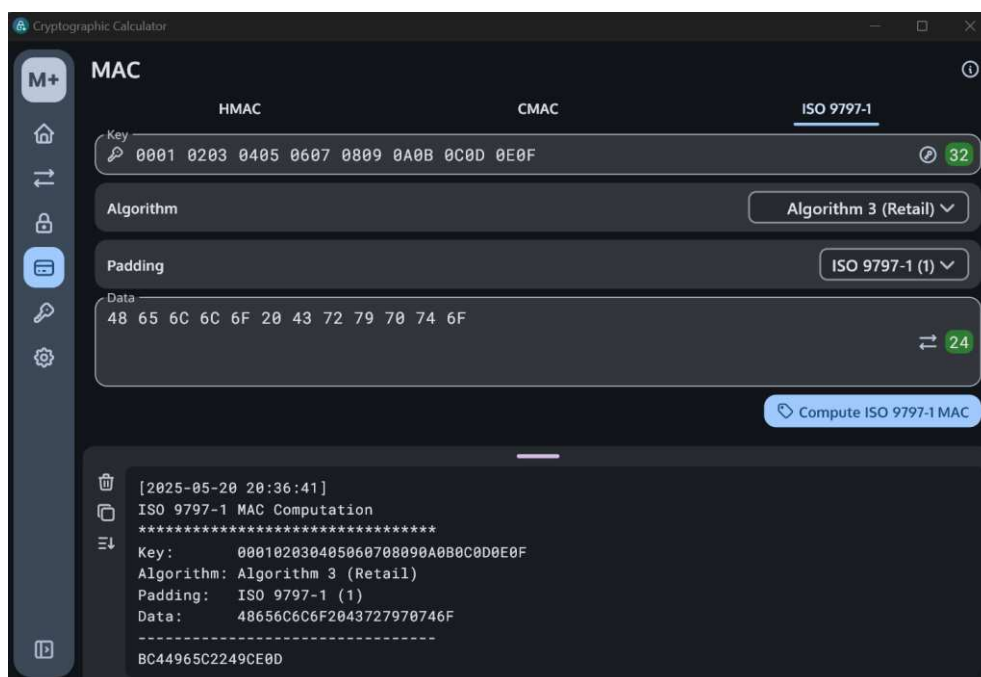
(zdroj: vlastní)

Klíč exportujeme z Key Managera. Uloží se do složky "C:\Users\jbaslar\Downloads".

Název	Datum změny	Typ	Velikost
Dnes			
2025-05-20_20-29-54.cckeys	20.05.2025 20:29	Soubor CCKEYS	1 kB

Obrázek 43 TC01 – Exportovaný klíč

Klíč smažeme a importujeme zpět do Key Managera. Nyní přejdeme do sekce MAC a do podsekce ISO-9797-1, doplníme a nastavíme vše potřebné a spočítáme MAC.



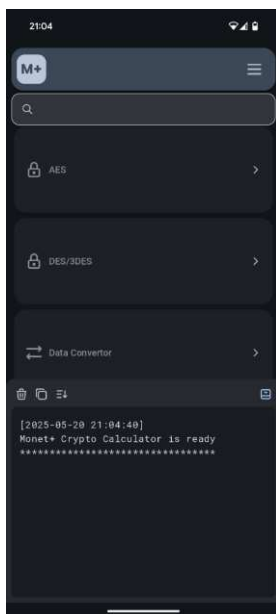
Obrázek 44 TC01 – Výpočet ISO 9797-1 MAC

(zdroj vlastní)

Aplikaci zavřeme křížkem v pravém horním rohu. Aplikace se zavře. Soubor s klíčem zůstane exportovaný v adresáři Downloads.

7.2 TC02 – Platforma Android

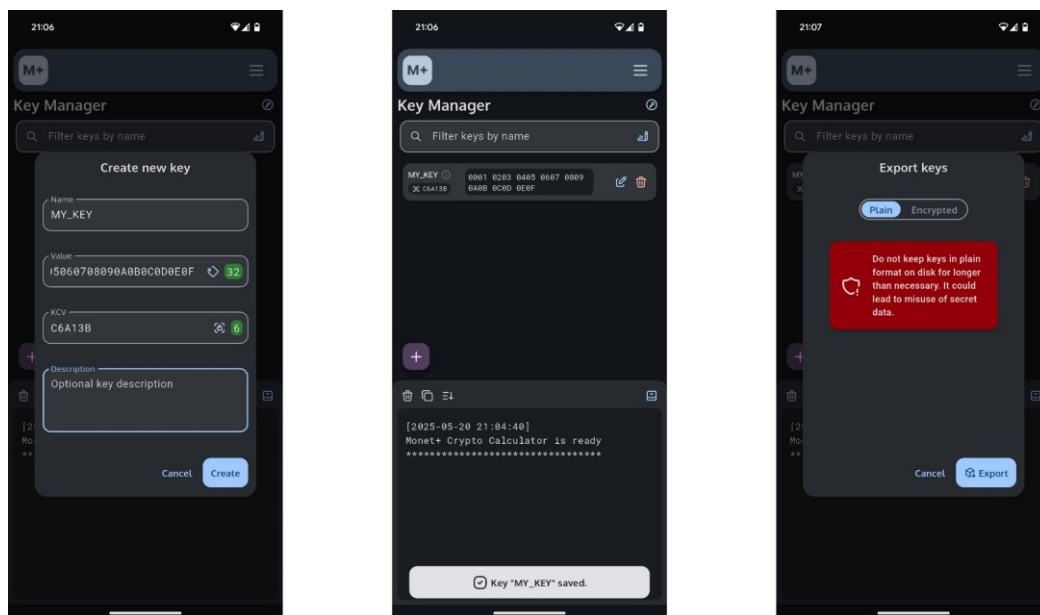
Následuje testování na platformě Android. Testuji na operačním systému Android 15 ve verzi jádra 6.1.99. Aplikace po spuštění vypadá následovně.



Obrázek 45 TC02 – Aplikace po spuštění

(zdroj: vlastní)

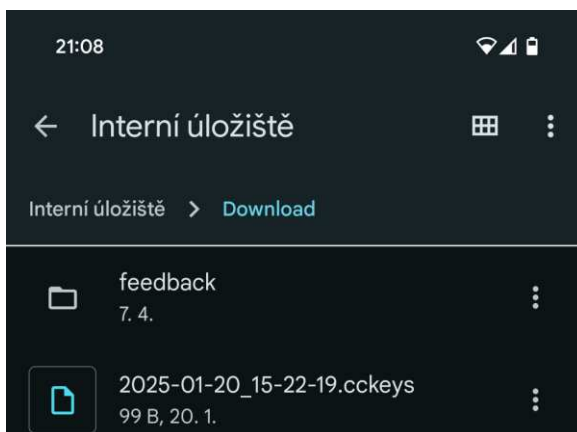
Přejdeme do sekce Key Manager a vytvoříme nový klíč.



Obrázek 46 TC02 – Vytvoření nového klíče

(zdroj: vlastní)

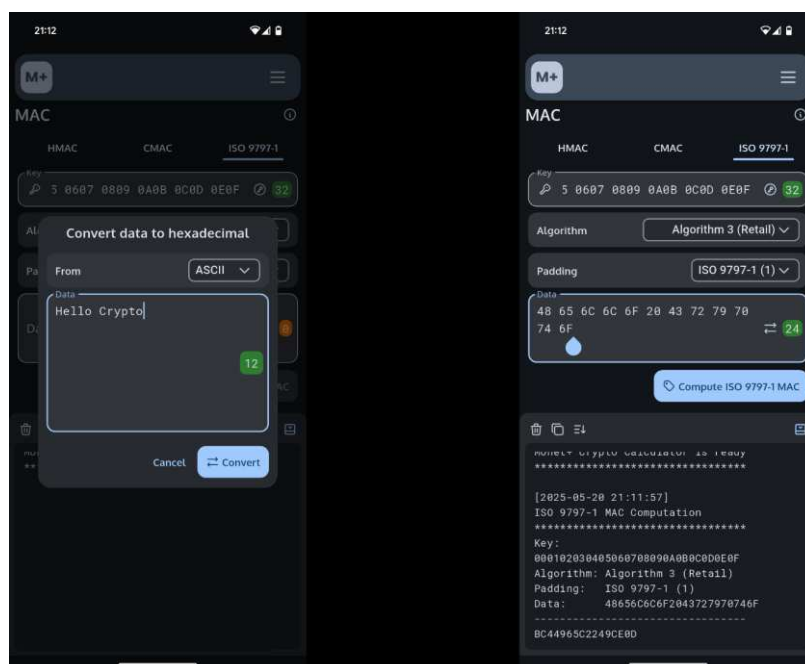
Ten následně exportujeme, uloží se do zařízení do složky Download:



Obrázek 47 TC02 – Exportovaný klíč

(zdroj: vlastní)

Klíč smažeme, znovu importujeme a přejdeme do sekce MAC a do podsekce s názvem ISO-9797-1, vyplníme vstupní hodnoty a vypočítáme MAC.



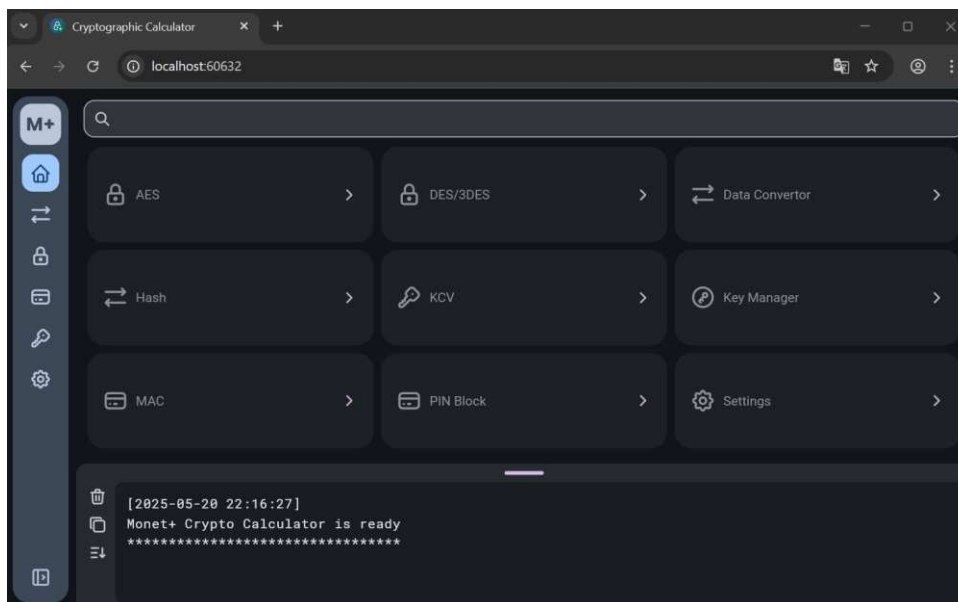
Obrázek 48 TC02 - Výpočet ISO 9797-1 MAC

(zdroj: vlastní)

Všechny kroky se podařily provést stejně jako v desktopové verzi aplikace. Vypočítaný MAC vyšel totožný s tím v desktopové aplikaci.

7.3 TC03 – Platforma webový prohlížeč

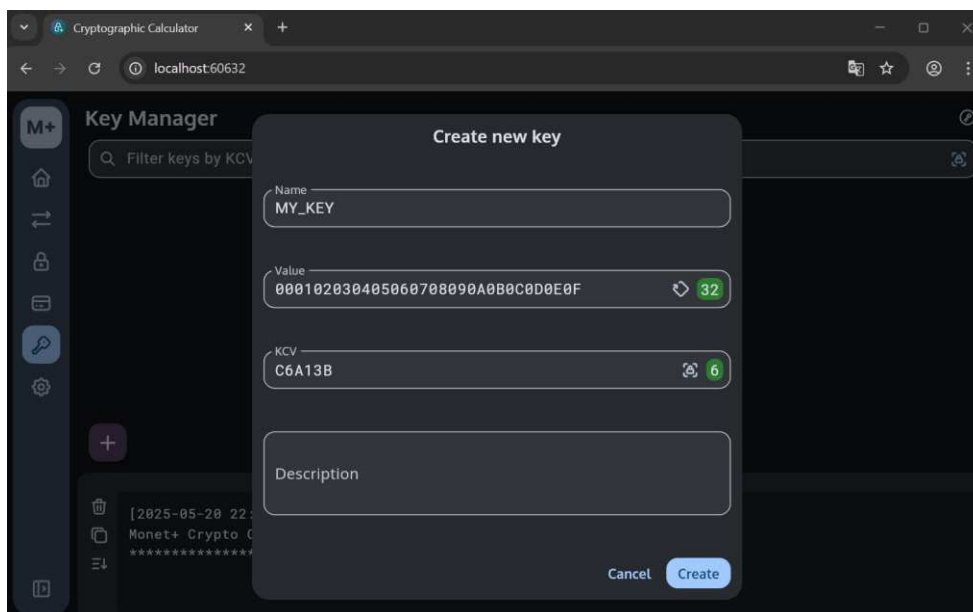
Nakonec si stejný scénář otestujeme i ve webové verzi aplikace. Aplikaci spouštím v prohlížeči Google Chrome ve verzi 136.0.7103.114.



Obrázek 49 TC03 – Aplikace po spuštění

(zdroj: vlastní)

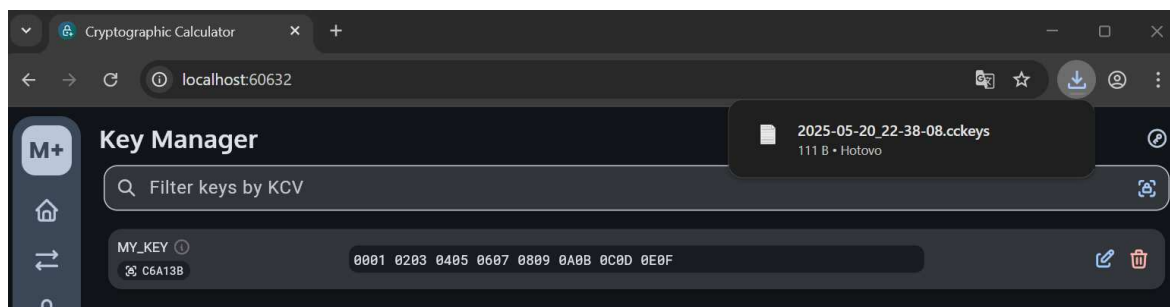
Aplikace se vzhledem hodně podobá desktopové verzi. Přejdu do sekce Key Manager, kde mě čeká vytvoření nového klíče.



Obrázek 50 TC03 – Vytvoření nového klíče

(zdroj: vlastní)

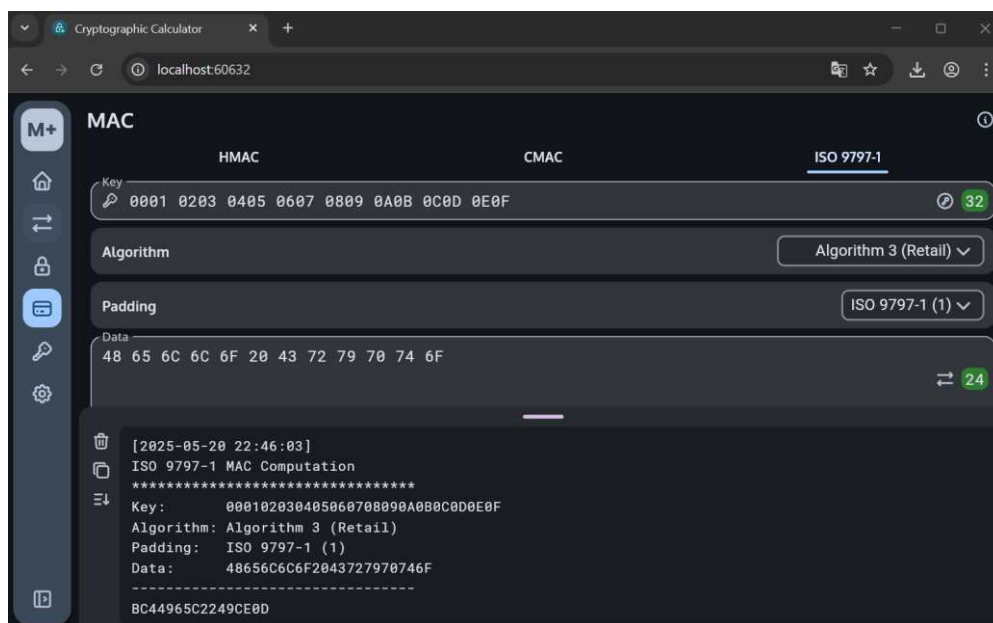
Ten následně exportuji do zařízení – prohlížeč jej stáhne jako nový soubor.



Obrázek 51 TC03 – Exportovaný klíč

(zdroj: vlastní)

Klíč je následně smazán a znovu importován do Key Managera. V sekci MAC, v podsekci ISO-9797-1 jej pak použijeme a spolu s dalšími vstupy vypočítáme MAC.



Obrázek 52 TC03 – Výpočet ISO 9797-1 MAC

(zdroj: vlastní)

I ve webové verzi aplikace se podařilo splnit všechny kroky z uživatelského případu.

ZÁVĚR

Cílem této diplomové práce bylo nastudovat a popsat vybrané kryptografické algoritmy a ty následně implementovat do podoby multiplatformní aplikace.

V teoretické části této práce byly popsány vybrané kryptografické algoritmy, jmenovitě pak DES, AES, KCV, MAC a PIN Block. Rozebrán byl princip fungování těchto algoritmů a jejich účel a význam pro kryptografii. U vybraných algoritmů byly zmíněny i důležité standardy či formáty.

Cílem praktické části pak bylo vytvořit multiplatformní aplikaci, která tyto algoritmy implementuje. Aplikace byla navržena tak, aby byla moderní, přehledná a uživatelsky přívětivá. Jejím účelem není uživateli jen poskytnout nástroj pro práci s kryptografickými algoritmy, ale taky mu samotnou kryptografii přiblížit a poskytnout přehled a informace ke každému algoritmu. Aplikace rovněž obsahuje část zvanou Key Manager, která uživateli umožňuje bezpečně ukládat, editovat, spravovat a znova používat kryptografické klíče v rámci celé aplikace. Aplikace je vytvořena ve frameworku Flutter a je multiplatformní, takže je schopna fungovat jak na desktopových platformách, tak i v mobilních zařízeních s operačním systémem Android a dokonce i jako webová aplikace v prohlížeči.

Výstupem z praktické části je pak Flutter projekt, který zvládá zmíněné kryptografické operace, je zdokumentovaný a připravený na budoucí rozšíření o novou funkcionalitu. Aplikace může být nasazena ve verzi 1.0.0 jako interní vývojářský nástroj ve firmě Monet+.

SEZNAM POUŽITÉ LITERATURY

- [1] SINGH, Simon; ECKHARDTOVÁ, Dita a KOUBSKÝ, Petr. Kniha kódů a šifer. 3. vydání. Praha: Dokořán, 2017. ISNB 9788073632687.
- [2] Seal of the U.S. National Security Agency. Online. In: Wikipedia Commons. 2017. Dostupné z: https://commons.wikimedia.org/wiki/File:Seal_of_the_U.S._National_Security_Agency.svg. [cit. 2025-02-22].
- [3] KATZ, Jonathan a LINDELL, Yehuda. Introduction to modern cryptography. Second edition. Boca Raton: CRC Press, 2015. ISBN 978-1-4665-7027-6.
- [4] FROMKIN, A. Michael. A Dispatch from the Crypto Wars (Review of Matt Curtin, Brute Force: Cracking The Data Encryption Standard (2005)). Online. A JOURNAL OF LAW AND POLICY. 2006, roč. 2, č. 2, s. 346 - 363. Dostupné z: [https://www.semanticscholar.org/paper/A-Dispatch-from-the-Crypto-Wars-\(Review-of-Matt-The-Froomkin/368ae645bd90c6bcffbc700255ea2b919dc59ed7](https://www.semanticscholar.org/paper/A-Dispatch-from-the-Crypto-Wars-(Review-of-Matt-The-Froomkin/368ae645bd90c6bcffbc700255ea2b919dc59ed7). [cit. 2025-02-23].
- [5] DES Cracker circuit board fitted with Deep Crack chips. Online. In: Wikipedia Commons. 2004. Dostupné z: <https://commons.wikimedia.org/wiki/File:Board300.jpg>. [cit. 2025-02-23].
- [6] SIMMONS, Gustavus J. Data Encryption Standard. Online. ENCYCLOPÆDIA BRITANNICA, INC. Britannica. 2025. Dostupné z: <https://www.britannica.com/topic/Data-Encryption-Standard>. [cit. 2025-03-01].
- [7] NIST. NIST SP 800-57, Recommendation for Key Management: Part 1 – General. Revision 4. 2016.
- [8] BURDA, Karel. Úvod do kryptografie. Brno: Akademické nakladatelství CERM, 2015. ISBN 9788072049257.
- [9] EWING, Larry. Tux ecb. Online. In: Wikipedia Commons. 2006. Dostupné z: https://commons.wikimedia.org/wiki/File:Tux_ecb.jpg. [cit. 2025-03-01].
- [10] KOWALCZYK, Chris. Padding Mechanisms. Online. Crypto-IT. 2024. Dostupné z: <http://www.crypto-it.net/eng/theory/padding.html>. [cit. 2025-03-02].
- [11] NECHVATA, James; BARKER, Elaine; BASSHAM, Lawrence; BURR, William; DWORKIN, Morris et al. Report on the Development of the Advanced Encryption

- Standard (AES). Online. Journal of Research of the National Institute of Standards and Technology. 2001, roč. 106, č. 3, s. 511 - 576. Dostupné z: <https://doi.org/10.6028/jres.106.023>. [cit. 2025-03-07].
- [12] RIMKIENĖ, Rūta. What is AES encryption and how does it work? Online. Cybernews. 2022. Dostupné z: <https://cybernews.com/resources/what-is-aes-encryption/>. [cit. 2025-03-08].
- [13] Padding schemes for block ciphers. Online. CryptoSys. 2024. Dostupné z: https://www.cryptosys.net/pki/manpki/pki_paddingschemes.html. [cit. 2025-03-08].
- [14] ÁLVAREZ-CUBERO, José Antonio. Vector Boolean Functions: applications in symmetric cryptography. Thesis for PhD. Madrid: Universidad Politécnica de Madrid, 2015.
- [15] Advanced Encryption Standard (AES) Encryption. Online. In: Binary Terms. 2025. Dostupné z: <https://binaryterms.com/advanced-encryption-standard-aes.html>. [cit. 2025-03-09].
- [16] SHAH, Javed. What Is AES Encryption? The Complete Guide. Online. 1KOSMOS. 2023. Dostupné z: <https://www.1kosmos.com/authentication/aes-encryption/>. [cit. 2025-03-16].
- [17] KOCHER, Paul; JAFFE, Joshua; JUN, Benjamin a ROHATGI, Pankaj. Introduction to differential power analysis. Online. Journal of Cryptographic Engineering. 2011, roč. 1, č. 1, s. 5-27. Dostupné z: <https://doi.org/10.1007/s13389-011-0006-y>. [cit. 2025-03-16].
- [18] GLOBALPLATFORM. GPC_SPE_034, GlobalPlatform Technology Card Specification. Version 2.3.1. 2018.
- [19] AMERICAN NATIONAL STANDARD FOR FINANCIAL SERVICES. ANSI X9.24-1-2017, Retail Financial Services Symmetric Key Management Part 1: Using Symmetric Techniques. 2017.
- [20] What Is a Message Authentication Code (MAC)? Online. The SSL Store. 2023. Dostupné z: <https://www.thesslstore.com/blog/what-is-a-message-authentication-code-mac/>. [cit. 2025-03-22].

- [21] SHELDON, Robert. Message authentication code (MAC). Online. Tech Target. 2023. Dostupné z: <https://www.techtarget.com/searchsecurity/definition/message-authentication-code-MAC>. [cit. 2025-03-22].
- [22] HMAC Algorithm in Computer Network. Online. GeeksForGeeks. 2024. Dostupné z: <https://www.geeksforgeeks.org/hmac-algorithm-in-computer-network/>. [cit. 2025-03-22].
- [23] What is Hash-Based Message Authentication Codes (HMAC)? Online. Kinde. 2024. Dostupné z: <https://kinde.com/guides/authentication/passwords/hash-based-message-authentication-codes-hmac/>. [cit. 2025-03-23].
- [24] DWORKIN, Morris. NIST SP 800-38B, Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication. 2005.
- [25] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 9797-1:2011, Information technology — Security techniques — Message Authentication Codes (MACs). Edition 2. 2011.
- [26] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 8583:2023, Financial-transaction-card-originated messages — Interchange message specifications. Edition 3. 2023
- [27] MASTERCARD. PIN Management. Online. MasterCard Developers. 2025. Dostupné z: <https://developer.mastercard.com/mastercard-processing-core/documentation/guides/pin-management/>. [cit. 2025-03-24].
- [28] KUMAR, Amit. Understanding : PIN BLOCK. Online. LinkedIn. 2023. Dostupné z: <https://www.linkedin.com/pulse/understanding-pin-block-amit-kumar>. [cit. 2025-03-24].
- [29] A credit card (front). Online. In: Wikipedia Commons. 2013. Dostupné z: <https://commons.wikimedia.org/wiki/File:Credit-card-front.svg>. [cit. 2025-03-24].
- [30] EFTLAB. Complete list of PIN-blocks. Online. EFTlab. 2022. Dostupné z: <https://www.eftlab.com/knowledge-base/complete-list-of-pin-blocks>. [cit. 2025-03-25].
- [31] Flutter. Online. Google, 2025. Dostupné z: <https://flutter.dev/>. [cit. 2025-04-20].

- [32] GOOGLE LLC. Dart documentation. Online. 2025. Dostupné z: <https://dart.dev/guides>. [cit. 2025-04-20].
- [33] Shared_preferences. Online. GOOGLE LLC. Pub.dev. 2025. Dostupné z: https://pub.dev/packages/shared_preferences. [cit. 2025-04-27].
- [34] Hive. Online. GOOGLE LLC. Pub.dev. 2025. Dostupné z: <https://pub.dev/packages/hive>. [cit. 2025-04-27].
- [35] Flutter_secure_storage. Online. GOOGLE LLC. Pub.dev. 2025. Dostupné z: https://pub.dev/packages/flutter_secure_storage. [cit. 2025-05-09].
- [36] Pointycastle. Online. GOOGLE LLC. Pub.dev. 2025. Dostupné z: <https://pub.dev/packages/pointycastle>. [cit. 2025-05-19].

SEZNAM OBRÁZKŮ

Obrázek 1 Pečeť National Security Agency [2]	15
Obrázek 2 Schéma Feistelovy sítě [3]	16
Obrázek 3 Schéma funkce f_i [3]	17
Obrázek 4 Deska osazená čipy Deep Crack [5]	19
Obrázek 5 Schéma provozního režimu ECB	21
Obrázek 6 Demonstrace nedostatku provozního režimu ECB [9]	21
Obrázek 7 Schéma provozního režimu CBC	22
Obrázek 8 Schéma provozního režimu CTR	23
Obrázek 9 Rijndael S-box [14]	27
Obrázek 10 Schéma AES šifrování [15]	28
Obrázek 11 Schéma vytváření MAC	32
Obrázek 12 Schéma ověření MAC	32
Obrázek 13 Schéma vytváření HMAC	33
Obrázek 14 Schéma vytváření CMAC	34
Obrázek 15 Komunikace s použitím Retail MAC	35
Obrázek 16 Kreditní karta s PAN [29]	37
Obrázek 17 Wireframe jedné sekce	44
Obrázek 18 Wireframe sekce s podrobnějšími informacemi	45
Obrázek 19 Wireframy mobilního UI	46
Obrázek 20 Struktura projektu	48
Obrázek 21 Filtrování klíčů v sekci Key Manager	56
Obrázek 22 Dialog pro editaci klíče	57
Obrázek 23 Výběr KCV při ukládání klíče	57
Obrázek 24 Export nešifrovaných klíčů	58
Obrázek 25 Export šifrovaných klíčů	59
Obrázek 26 Konflikt při importu klíčů	61
Obrázek 27 Tlačítko Key Manager v textovém poli	61
Obrázek 28 Dialog pro výběr klíče	62
Obrázek 29 Dialog pro generování klíče	62
Obrázek 30 Tmavě modrý a světle oranžový motiv	63
Obrázek 31 Sekce Settings	64
Obrázek 32 Banner Storing keys is disabled!	65
Obrázek 33 Sekce DES/3DES	66
Obrázek 34 Dialog pro konverzi dat	67

Obrázek 35 Popis sekce DES/3DES	67
Obrázek 36 Sekce AES	69
Obrázek 37 Sekce MAC	70
Obrázek 38 Popis sekce MAC	71
Obrázek 39 Sekce PIN Block.....	76
Obrázek 40 Sekce KCV	77
Obrázek 41 TC01 – Aplikace po spuštění	80
Obrázek 42 TC01 – Vytvoření nového klíče	80
Obrázek 43 TC01 – Exportovaný klíč	81
Obrázek 44 TC01 – Výpočet ISO 9797-1 MAC.....	81
Obrázek 45 TC02 – Aplikace po spuštění	82
Obrázek 46 TC02 – Vytvoření nového klíče	82
Obrázek 47 TC02 – Exportovaný klíč	83
Obrázek 48 TC02 - Výpočet ISO 9797-1 MAC	83
Obrázek 49 TC03 – Aplikace po spuštění	84
Obrázek 50 TC03 – Vytvoření nového klíče	84
Obrázek 51 TC03 – Exportovaný klíč	85
Obrázek 52 TC03 – Výpočet ISO 9797-1 MAC.....	85

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

3DES	Triple DES
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
CBC	Cipher-Block Chaining
CMAC	Cipher-based MAC
CTR	Counter Mode
CVC	Card Verification Code
DES	Data Encryption Standard
DPA	Differential Power Analysis
ECB	Electronic Codebook
FTPS	File Transfer Protocol Secure
HMAC	Hash-based MAC
HTTPS	Hypertext Transfer Protocol Secure
IBM	International Business Machines Corporation
IoT	Internet of Things
ISO	International Organization for Standardization
IV	Initial Vector
JSON	JavaScript Object Notation
KCV	Key Checksum Value
MAC	Message Authentication Code
NIST	National Institute of Standards
NSA	National Security Agency
OS	Operating System
PAN	Primary Account Number
PC	Personal Computer

PIN	Personal Identification Number
PKCS	Public Key Cryptographic Standards
TLS	Transport Layer Security
UI	User Interface
XOR	Exclusive Or

SEZNAM PŘÍLOH

Příloha P I: cryptocalc

PŘÍLOHA P I: CRYPTOCALC

Příloha obsahuje adresář cryptocalc, ve kterém se nachází Flutter projekt obsahující zdrojové kódy aplikace.