VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

IMPLEMENTACE KRYPTOGRAFICKÝCH PRIMITIV

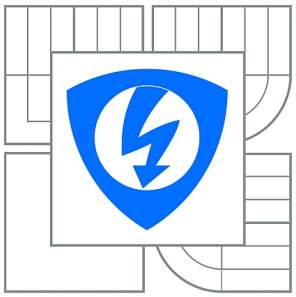
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ELIŠKA JÉGROVÁ
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

IMPLEMENTACE KRYPTOGRAFICKÝCH PRIMITIV

IMPLEMENTATION OF CRYPTOGRAPHIC PRIMITIVES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

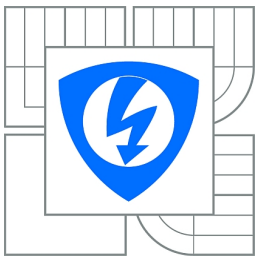
AUTOR PRÁCE — ELIŠKA JÉGROVÁ

AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR LEŽÁK

SUPERVISOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Bakalářská práce

bakalářský studijní obor
Teleinformatika

Studentka: Eliška Jégrová ***ID:*** 158153
Ročník: 3 ***Akademický rok:*** 2014/2015

NÁZEV TÉMATU:

Implementace kryptografických primitiv

POKYNY PRO VYPRACOVÁNÍ:

V jazyce Java implementuje blokové šifry Blowfish a Triple DES a hešovací funkce SHA-3 a Tiger. Začleňte je do knihovny Aalg. Dále porovnejte efektivitu jejich implementace s existujícími implementacemi a navrhněte metody, jak jejich efektivitu co nejvýše zvýšit.

DOPORUČENÁ LITERATURA:

[1] SMART, Nigel. Cryptography: An Introduction. [online]. 3rd Edition. [cit. 2012-11-27]. Dostupné z: http://www.cs.bris.ac.uk/~nigel/Crypto_Book/

[2] LEŽÁK, Petr. The abstract algebra library. [online]. [cit. 2014-04-14]. Dostupné z: http://sourceforge.net/projects/aalg/

Termín zadání: 9.2.2015 ***Termín odevzdání:*** 2.6.2015

Vedoucí práce: Ing. Petr Ležák
Konzultanti bakalářské práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

ABSTRAKT

Tato bakalářská práce je zaměřena na implementaci kryptografických metod. Část se věnuje blokovým šifrám, kde popisuje algoritmy Blowfish a 3DES. Dále se zabývá hashovacími funkcemi, z nich podrobně rozebírá algoritmy SHA-3 a Tiger. Implementace šifer i hashovacích funkcí jsou dále porovnány s jinými realizacemi.

KLÍČOVÁ SLOVA

Kryptografie, implementace, 3DES, DES, Blowfish, SHA-3, Tiger, Java

ABSTRACT

This bachelor thesis is focused on cryptographic methods. Part of it deals with block ciphers, where are described algorithms of Blowfish and 3DES. It also deals with hash functions of which are analysed algorithms of SHA-3 and Tiger in detail. Implementation of ciphers and hash functions are further compared with other implementations.

KEYWORDS

Cryptography, implementation, 3DES, DES, Blowfish, SHA-3, Tiger, Java

JÉGROVÁ, Eliška *Implementace kryptografických primitiv*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2015. 61 s. Vedoucí práce byl Ing. Petr Ležák

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Implementace kryptografických primitiv“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu bakalářské práce panu Ing. Petru Ležákovi za odborné vedení, konzultace, trpělivost a cenné rady při zpracování této práce.

Brno ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

V nynější době probíhá velké množství komunikace elektronicky. Důležité je, aby se data nedostala k útočníkovi, nebo aby je nemohl zneužít ve svůj prospěch.

Věda zabývající se bezpečným přenosem těchto dat se nazývá kryptologie. Ta se dále dělí na dva podobory. Prvním je kryptografie, mezi její hlavní cíle patří zajištění autentičnosti. Neboť je nutné mít důkaz o tom, že osoby jsou skutečně těmi, za které se vydávají a nebylo nijak manipulováno s jejich daty. Dalším cílem je zajištění utajení dat, které obstarává přístup ke chráněným datům pouze pro oprávněné uživatele. Druhým podoborem je kryptoanalýza, která zkoumá bezpečnost šifer a zabývá se jejich prolamováním.

Podle [1] existují dva druhy kryptografických systémů: symetrické, nebo asymetrické.

Symetrické šifrování užívá pro šifrování i dešifrování stejný klíč, dělí se na dva druhy. Proudové šifry, které pracují s bity, popř.bajty, které šifrují samostatně. Některými z představitelů proudových šifer jsou RC4, A5, Fish. Dalším druhem jsou blokové šifry, věnuje se jim velká část mé práce. Pracují s celistvými bloky textu.

DES [2] je bezpochyby nejvýznamnější moderní symetrickou šifrou, byl vyvinut na začátku 70.let a je odvozen z šifry Lucifer, což byla interní šifra IBM. DES byl přijat jako standart a začal se hojně mezinárodně využívat. Nejprve byl schválen na pětileté období, následně byla tato doba postupně prodloužena až na 20 let. Jeho slabinou je krátký klíč (56 bitů), a proto se přechází na algoritmus 3DES, který využívá stejného algoritmu, ale vzhledem k jeho opakování může být délka klíče trojnásobná. Takto vytvořený algoritmus je ale velmi pomalý a je přijat nový standard, nazvaný AES (Rijndael).

Blowfish [3] byl publikován v roce 1993 Brucem Schneierem a byl zamýšlen jako alternativa ke stárnoucí šifře DES, ale novým standardem se nestal. Stejně jako algoritmus DES využívá Feistelova schématu. Uživatelé algoritmu Blowfish si musí dát pozor na určité třídy klíčů, které jsou náchylné proti útoku.

Dalšími blokovými šiframi jsou například: AES, Twofish, IDEA, Serpent.

Asymetrické šifrování je vhodné, pokud probíhá komunikace mezi více lidmi (institucemi), je popsáno v [4]. Základní rozdíl je v použití odlišných klíčů pro šifrování a dešifrování. První klíč je veřejně známý a použije se pro zašifrování zprávy, soukromým klíčem se data dešifrují. Princip asymetrie je založený na jednocestných funkcích. Například násobení: je velmi jednoduché získat součin dvou čísel, ale rozklad součinu na činitele je mnohem složitější (využito u algoritmu RSA). Používá se také jako elektronický podpis pro zabezpečení autentičnosti zprávy. Nejrozšířenější algoritmy asymetrického šifrování jsou RSA, a ElGamal.

Další částí kryptografie, které se v práci věnuji podrobněji, jsou algoritmy hashovacích funkcí, které jsou v kryptografii hojně využívány. Můžeme zde uvést několik důležitých využití uvedených v [5].

U digitálních podpisů jsou využívány ke generování hodnoty hash. Jedná se o přidání nadbytečných dat za zprávu tak, aby zpráva obsahovala rozpoznatelné informace pro příjemce.

V asymetrické kryptografii jsou hashovací funkce široce používány pro realizaci mechanismu k ověření správnosti kryptogramu. Takový mechanismus je nezbytný pro dosažení prokazatelného zabezpečení proti aktivním útočníkům.

V širokém rozsahu kryptografických aplikací, kde se vyžaduje pseudo-náhodnost, jsou hashovací funkce používány jako pseudonáhodné funkce. Tyto aplikace zahrnují: dohodu o klíčích (jako vstup pro hash se užije náhodné číslo a získá se společná hodnota klíče), ověřovací protokoly (dva účastníci protokolu si zašlou po dokončení protokolu hodnotu hash).

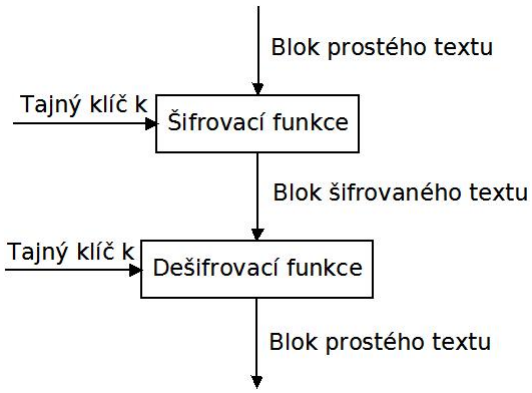
Keccak [6] je funkce, která byla publikována v roce 2012 a později se stala standardem SHA-3. Po útocích na SHA-1 a SHA-2 bylo potřeba přejít na odlišný princip algoritmu, kterým je právě SHA-3. Je možné použít čtyři různé typy algoritmu, které se liší ve výstupní délce hashe.

Další hashovací funkcí je Tiger, ze zdroje [7]: Ross Anderson a Eli Biham ji navrhli v roce 1995, využívají Merkle-Damgårdovy konstrukce. Není známý žádný efektivní útok na plný, 24 rundový tiger.

Bakalářská práce se zabývá převážně kryptografií, stěžejní částí jsou celkově čtyři algoritmy. V první kapitole, o blokových šifrách, budu podrobně rozebírat algoritmy šifer Blowfish a Triple DES. Ve druhé kapitole pojednávám o hashovacích funkcích, konkrétně o algoritmech funkcí SHA-3 a Tiger. Následující kapitola se zabývá samotnou implementací uvedených algoritmů a v poslední kapitole jsou k dispozici výsledky měření rychlostí daných implementací.

1 BLOKOVÉ ŠIFRY

Základní schéma je zobrazeno na obr. 1.1. Blokové šifry pracují s bloky prostého textu na vstupu, výstupem je šifrovaný text a po dešifrování je získán text shodný se vstupním textem.



Obr. 1.1: Princip blokových šifer

Velikost bloku na vstupu je vždy shodná s velikostí bloku na výstupu, v modernějších šifrách se užívá bloků dat o velikosti alespoň 128 bitů.

Následující rovnice z [8] popisují základní algoritmus blokových šifer:

$$c = E_k(m), \tag{1.1}$$

$$m = D_k(c), \tag{1.2}$$

kde

- m - prostý text
- c - kryptogram
- E - šifrovací funkce
- D - dešifrovací funkce
- k - tajný klíč

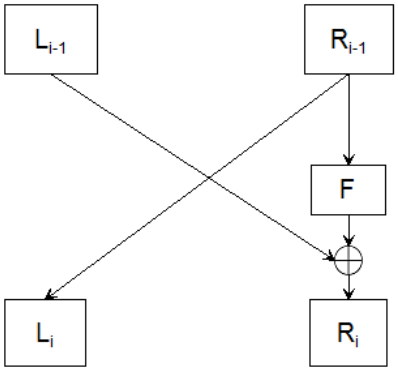
Z rovnic (1.1) a (1.2) je zřejmé, že se při šifrování i dešifrování použije stejný tajný klíč a šifry jsou reverzibilní – existuje dešifrovací funkce, pomocí které se získá původní text.

Feistelovo schéma

Jedná se o strukturu, která je užita v mnoha blokových šifrách. Podle článku [9] je pojmenována po fyziku a kryptografovi Horstu Feistelovi, který použil tuto konstrukci poprvé v šifře Lucifer.

Feistelovo schéma má výhodu, že šifrovací a dešifrovací operace jsou podobné, v některých případech dokonce totožné, což velmi zjednodušuje implementaci [10].

Na obr. 1.2 je zobrazena jedna runda, algoritmus je popsaný v [8]. Nejprve se blok textu rozdělí na dvě poloviny, značené L_0 a R_0. Data L_{i+1} jsou shodná s daty R_i a R_{i+1} jsou získány pomocí operace xor z dat L_i a výstupu funkce f. Iterací může být libovolné množství, např. u šifry DES je užito 16 iterací. Výsledným kryptogramem je výstup poslední iterace v opačném pořadí: $c = R_n \,||\, L_n$.



Obr. 1.2: Feistelovo schéma

Další výhodou [10] Feistelova schématu je fakt, že je invertovatelný bez ohledu na funkci f. Vstupními daty funkce f je blok R_i a klíč K_i. V každé iteraci se používá různého klíče k zašifrování. Pokud máme užity při šifrování klíče $k_1, k_2, \ldots, k_n$, pak při dešifrování otočíme posloupnost klíčů a použijeme $k_n, k_{n-1}, \ldots, k_1$.

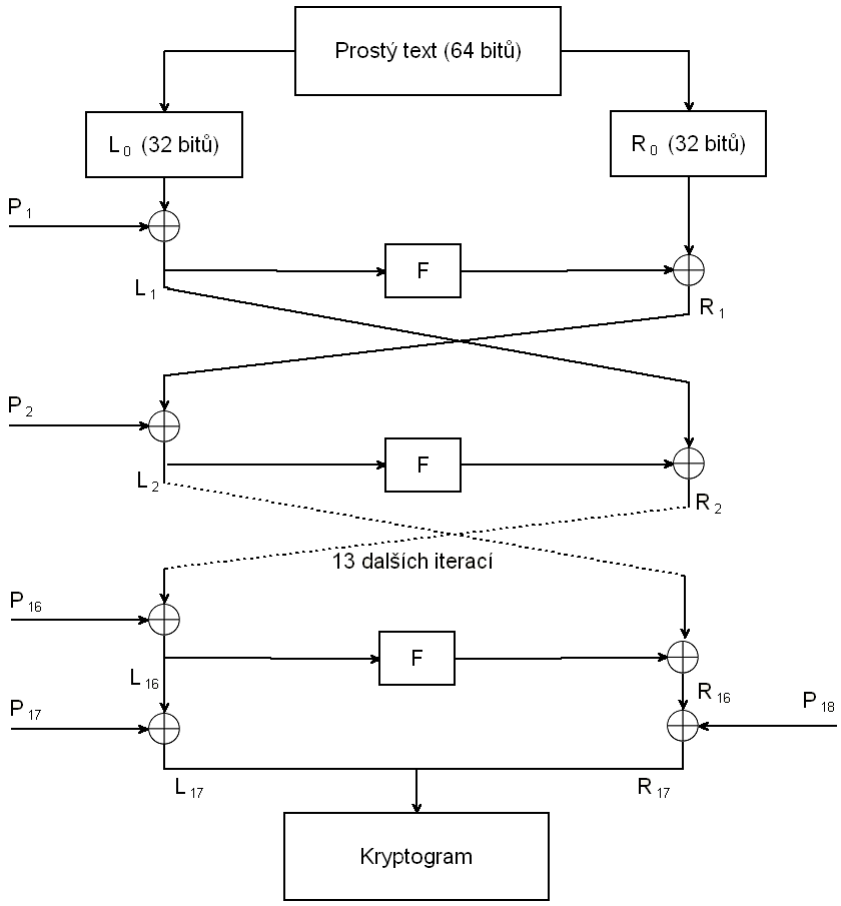
1.1 Blowfish

Blowfish vznikl jako alternativa k šifře DES. Pracuje s 64 bitovým blokem vstupních dat, velikost klíče může být libovolná v rozmezí 32 – 448 bitů.

Algoritmus, popsaný v [11], se skládá ze dvou hlavních částí, rozvoje klíče a šifrování dat. Rozvoj klíče spočívá v generování 18 vstupních hodnot velikosti 32 bitů pole P a 4 S-boxů, každý z nich obsahuje 256 hodnot o velikosti 32 bitů. Šifrování dat využívá 16 rund Feistelova schématu.

1.1.1 Šifrování dat

Blokový diagram je zobrazen na obr.1.3. Podle [12] se nejprve blok 64 bitů rozdělí na dvě části stejné velikosti L_0 a R_0. S 32 bity L_0 a s hodnotou pole $P[1]$ se provede operace xor, čímž získáme blok L_1. Blok R_1 se získá pomocí operace xor funkce $\mathrm{F}(L_1)$



Obr. 1.3: Algoritmus Blowfish

a R_0. Následující rovnice (1.3) a (1.4) platí pro $i \in \langle 2,\, 16\rangle$. Nakonec se provede xor bloků L_{16} a $P[17]$, tím vznikne blok L_{17} a pomocí xoru bloků R_{16} a $P[18]$ vznikne blok R_{17}. Následně získáme kryptogram spojením bloků L_{17} a R_{17} dohromady.

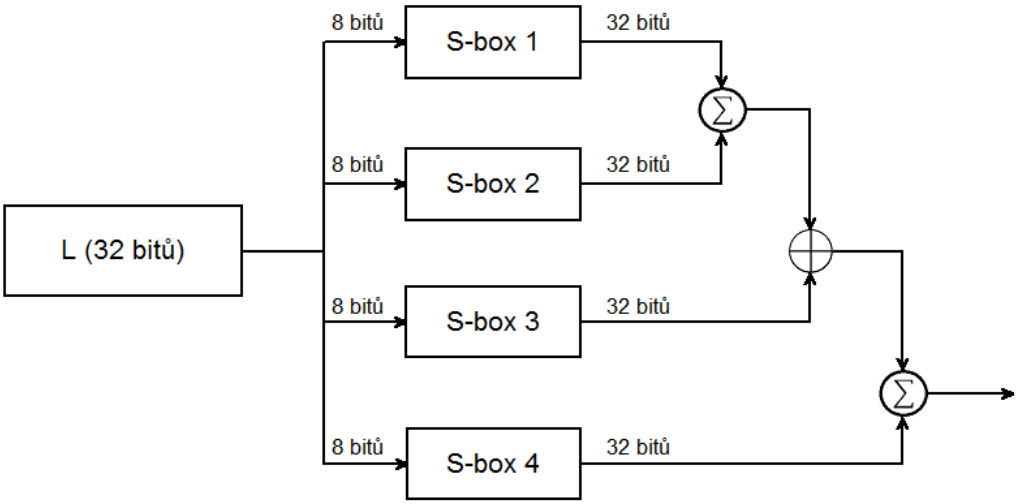
$$L_i = R_{i-1} \oplus P[i] \tag{1.3}$$

$$R_i = L_{i-1} \oplus \mathrm{F}\,(L_i) \tag{1.4}$$

Funkce F, obr.1.4, používá 2 jednoduché operace – operaci součet modulo 2 (xor), operaci součet modulo 2^{32}. Dle zdroje [11] je aplikace následovná:

1. Rozdělme L (32 bitů) na 4 části po 8 bitech a označme je a, b, c, d.
2. Pak $\mathrm{F}(L) = ((S_1[a] + S_2[b]) \oplus S_3[c]) + S_4[d]$,
   kde $+$ je součet modulo 2^{32}, $\oplus$ značí xor a zápis $S_1[a]$ znamená hodnotu prvního S-boxu na pozici a.

Dešifrování probíhá stejným způsobem, jediný rozdíl je v hodnotách pole P[i], které jsou použity v obráceném pořadí.



Obr. 1.4: Funkce F šifry Blowfish

1.1.2 Rozvoj klíče

Algoritmem popsaným v [13] se generuje pole P a 4 pole S:

1. Nejprve se inicializuje pole P, a pak všechny pole S pomocí daného řetězce. Tento řetězec sestává z desetinného rozvoje π vyjádřeného v šestnáctkové soustavě. P[1]=0x243f6a88, P[2]=0x85a308d3, P[3]=0x13198a2e, atd.
2. Provede se xor P[1] s prvními 32 bity klíče, xor P[2] s dalšími 32 bity klíče, atd. Jakmile se použijí všechny bity klíče, pokračuje se opět od prvních bitů klíče, dokud nebudou změněny všechny hodnoty pole P.
3. Zašifrujte řetězec, který obsahuje pouze nuly pomocí algoritmu Blowfish s hodnotami vygenerovanými v krocích 1 a 2.
4. Nahraďte P[1] a P[2] výstupními daty z kroku 3.
5. Zašifrujte výstupní data kroku č.3 pomocí algoritmu Blowfish s modifikovanými hodnotami P[1] a P[2].
6. Nahraďte P[3] a P[4] výstupními daty z kroku 5.
7. Pokračujte dále v uvedeném postupu, dokud nebudou nahrazeny všechny pole P a následně všech 4 S-boxů.

1.2 Triple DES

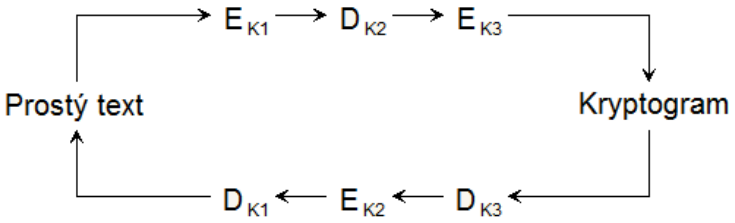
Triple Data Encryption Standard (3DES) vznikl jako aplikace algoritmu šifry Data Encryption Standard (DES) třikrát za sebou. DES má délku klíče 56 bitů, což je z hlediska zabezpečení nedostačující.

Algoritmus 3DES, popsán v [14], je specifikován pro 3 různé varianty klíčů (K_1, K_2, K_3).

1. K_1, K_2, K_3 jsou navzájem nezávislé klíče – nejčastější varianta, klíč má celkovou délku 168 bitů.
2. K_1 a K_2 jsou nezávislé klíče, $K_3 = K_1$, velikost klíče je rovna 112 bitům.
3. $K_1 = K_2 = K_3$.

Nechť $E_K(I)$ a $D_K(I)$ reprezentují šifrování a dešifrování vstupu I pomocí algoritmu DES při užití klíče K. Každá operace šifrování a dešifrování 3DES je složena z operací šifrování i dešifrování DES viz obr.1.5. Operace jsou použity následovně:

- Šifrování 3DES: transformace 64 bitového bloku I na 64 bitový blok O.
  $O = E_{K3}(D_{K2}(E_{K1}(I)))$
- Dešifrování 3DES: transformace 64 bitového bloku I na 64 bitový blok O.
  $O = D_{K1}(E_{K2}(D_{K3}(I)))$



Obr. 1.5: Algoritmus 3DES

1.2.1 Algoritmus DES

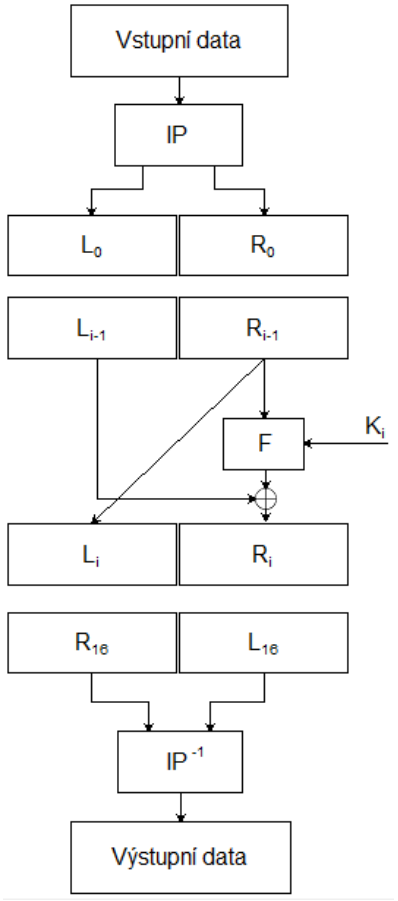
Šifrování

DES pracuje s bloky dat o velikosti 64 bitů. V souhrnu se provedou následující operace, jak je popsáno v [8]:

- Proběhne prvotní permutace IP vstupního bloku dat.
- Data se rozdělí na dvě části, levou L a pravou R, každá obsahuje 32 bitů.
- Provede se 16 rund.
- Obě části se spojí zpět dohromady na blok 64 bitů.
- Uskuteční se závěrečná permutace IP^{-1} dat.

Prvotní permutace

Permutace je definovaná tabulkou 1.1 ze zdroje [14]. Vstupní bit na 58. pozici bude po permutaci na první pozici, 50. bit na druhé pozici, atp.



Obr. 1.6: Algoritmus DES

Iterační cyklus

Cyklus se skládá ze 16 iterací, které jsou popsány v rovnicích (1.5) a (1.6) [14]. Vstupními daty jsou data po permutaci, rozdělena na levou L_0 a pravou R_0 část.

$$L_i = R_{i-1}, \tag{1.5}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i), \tag{1.6}$$

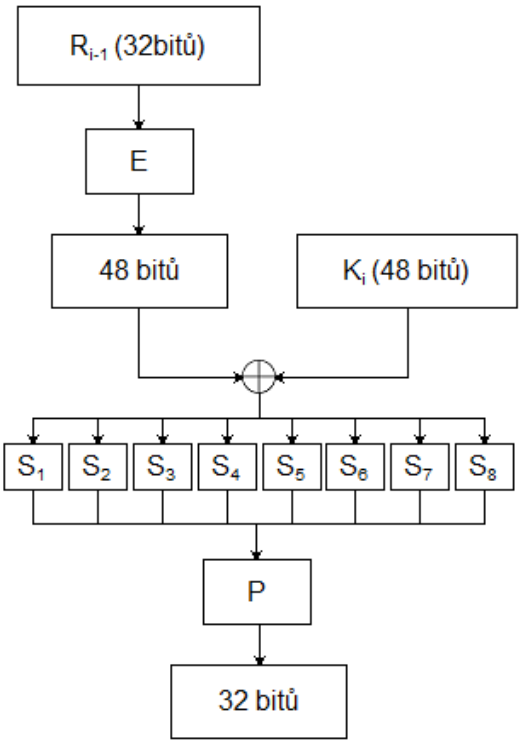
kde i je hodnota od 1 do 16, K_i - blok 48 bitů klíče, F - šifrovací funkce a $\oplus$ - operace xor, neboli bitový součet modulo 2.

Šifrovací funkce F, popsána ve [14], pracuje se dvěma vstupními bloky, prvním je blok o velikosti 32 bitů funkce R_{i-1}, druhým je blok K_i se 48 bity. Výstupem této funkce je 32-bitový blok. Algoritmus funkce F je zobrazen na obr.1.7. Funkce E rozšíří blok R na 48 bitů dle tab.1.2, ve které jsou označeny nové pozice bitů bloku R_{i-1}. Např. na první pozici bude nyní bit z původní 32. pozice a zároveň bude i na pozici č. 47.

Tab. 1.1: IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

S nově vytvořeným blokem a blokem K_i se následně provede operace bitového součtu modulo 2. Dále se data rozdělí na 8 částí po 6 bitech.



Obr. 1.7: Funkce F

Tab. 1.2: Selekční tabulka E

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Z každé částí se pomocí funkcí S_1,S_2,...,S_8, které jsou zobrazeny v [14] a v tab.A.1, získají 4 bity. Blok 6 bitů označíme B. V každém bloku B definujeme čísla k a l následujícím způsobem: První a poslední bit B je binární číslo k, zbylé 4 bity pak binární číslo l. Tato čísla reprezentují k-tý řádek a l-tý sloupec definiční tabulky funkce S_i, kdy řádky i sloupce jsou číslovány od nuly. Daná hodnota této funkce se převede zpět do binární soustavy, čímž získáme 4 bitový výstup funkce S_i. Například pro S_1(B), kdy B = 101110, je hodnota řádku 10 a sloupce 0111. Po převedení do dekadické soustavy jde o 2. řádek a 7. sloupec, tomu odpovídá pro funkci S_1 hodnota 11 a v binární soustavě hodnota 1011.

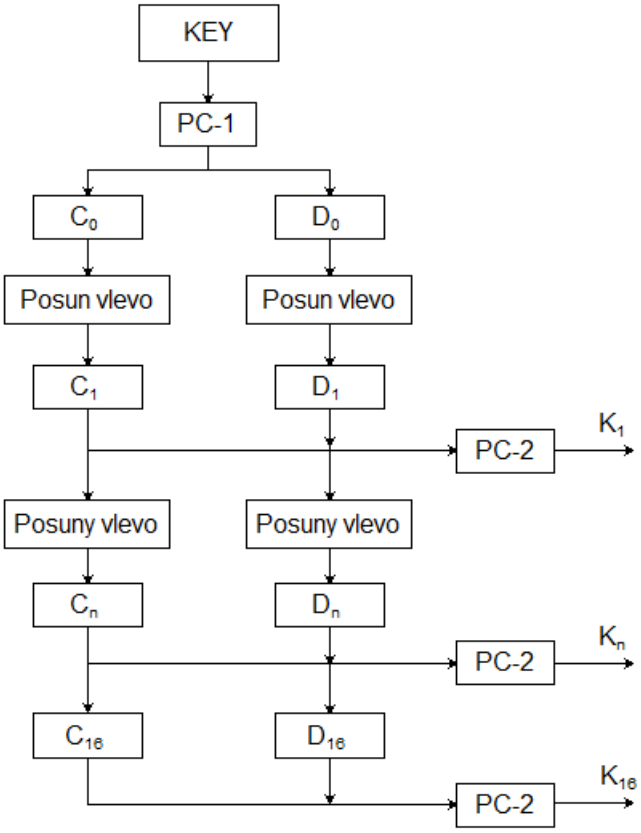
Tab. 1.3: P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Z výstupu funkcí S_1,S_2,...,S_8 dostaneme blok dat o velikosti 32 bitů. Poté provedeme permutaci P, zobrazenou v tab.1.3 [14] a tím získáme výstupní data funkce F.

Při každé iteraci je použit odlišný blok K, který je vybrán z 64 bitů klíče (KEY) pomocí funkce KS. Funkce je zobrazena na obr. 1.8

Při výpočtu KS se nejprve provede permutace nazvaná PC-1, definovaná v tab.1.4, stanovená ve [14]. Data jsou rozdělena na dvě části. První část nazveme C a druhou část D. Bity klíče KEY jsou číslovány 1 - 64, bity C_0 jsou bity $57, 49, 41, ..., 36$ a D_0 obsahuje bity $63, 55, ..., 4$ klíče KEY.



Obr. 1.8: Funkce KS

Tab. 1.4: PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Bloky C_i a D_i se získávají pomocí tzv. posunutí vlevo – definované v [14]. Posunutím vlevo je myšlena rotace bitů o jednu pozici doleva. Pokud máme blok dat, který obsahuje 5 bitů: a, b, c, d, e, budou po jedné operaci přesunuty na jiné pozice: b, c, d, e, a.

Například C_3, D_3 jsou získány z C_2, D_2 pomocí dvou posunů vlevo a C_{16}, D_{16} je získán z C_{15}, D_{15} jedním posunutím vlevo.

Tab. 1.5: Přehled posunutí vlevo při konkrétní iteraci

Iterace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Počet posunutí vlevo	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Blok K_i obdržíme pomocí permutace PC-2 bloku C_iD_i, která je definovaná v tab.1.6 a v [14].

Tab. 1.6: PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Závěrečná permutace

Výstupní bloky R_{16} a L_{16} sjednotíme na 64 bitový blok výstupních dat v pořadí R_{16}, L_{16}. Závěrečná permutace IP^{-1}je uvedena v tab.1.7.

Tab. 1.7: IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Dešifrování

Při dešifrování se užívá stejného algoritmu jako při šifrování, pouze se změní pořadí jednotlivých operací. Jelikož je prvotní permutace IP funkcí inverzní k závěrečné permutaci IP^{-1}, užijeme nejprve závěrečnou permutaci.

Následně proběhne iterační cyklus, který bude probíhat opačným směrem než původně. Popsán je rovnicemi (1.7) a (1.8) z [8]. Nyní již nebudeme generovat bloky K_i, ale musíme použít ty, které se získali při šifrování, a to ve správném pořadí. Vstupními daty je blok $R_{16}L_{16}$, pomocí bloku K_{16} dostaneme blok $R_{15}L_{15}$, atd. Výstupními daty bude blok L_0R_0, se kterými na závěr provedeme prvotní permutaci IP.

$$R_{i-1} = L_i, \tag{1.7}$$

$$L_{i-1} = R_i \oplus F(L_i, K_i) \tag{1.8}$$

2 HASHOVACÍ FUNKCE

Vstupní data kryptografických hashovacích funkcí h funkce mají libovolnou délku a z nich se vytvoří tzv. hash o pevně dané délce, jeho velikost značíme |h|, typicky se uživá 160 bitů. Je nutné, aby každá hashovací funkce splňovala dle [8] následující vlastnosti:

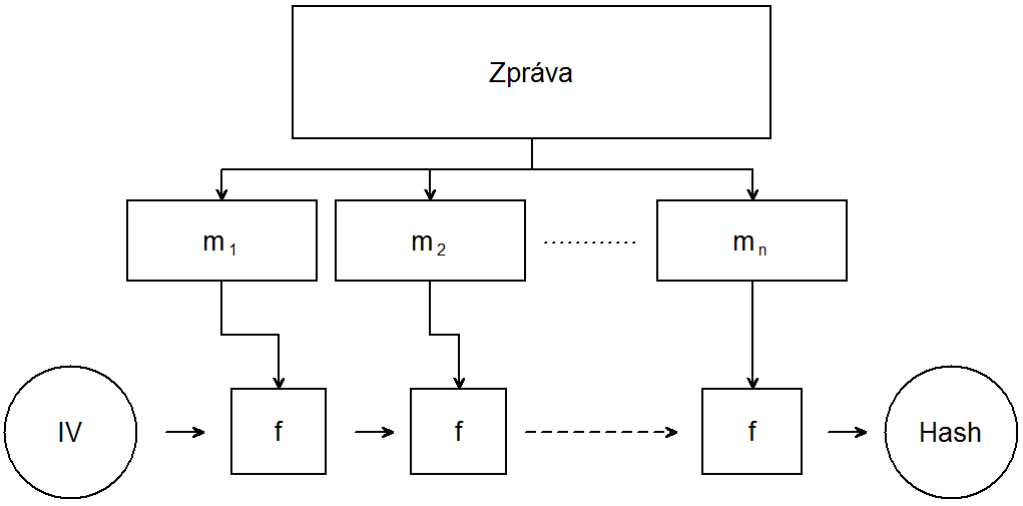
- Z daného hashe H nelze zjistit vstupní data x.
- Nelze najít vstupy x, y, $x \neq y$ takové, aby měli stejný výstup: $\mathrm{h}(x) = \mathrm{h}(y)$.
- Ze znalosti $\mathrm{H} = \mathrm{h}(x)$ nelze najít y takové, že $\mathrm{H} = \mathrm{h}(y)$

Merkle-Damgårdova konstrukce

Předpokládejme funkci f, která je kompresí z s bitů na n bitů. Tuto funkci použijeme při konstrukci hashovací funkce h, která má libovolnou délku vstupních dat a výstupem je hash délky n bitů. Ze zdroje [8] je algoritmus následovný :

1. Nechť $l = s - n$
2. Vstupní data m se doplní pomocí 0 tak, aby byla násobkem délky l
3. Data m se rozdělí na t bloků o délce l
4. Nechť řetězec H má danou délku n, pro $i \in \langle 1, t \rangle$: $H = \mathrm{f}(H \,||\, m_i)$
5. Vrať H

Po každé iteraci se změní hodnota H, která je použita v dalším kroku iterace. Na konci cyklu je H výsledným hashem této konstrukce.



Obr. 2.1: Merkle-Damgårdova konstrukce

Merkle-Damgårdova konstrukce je snadno napadnutelná tzv. útokem prodloužením zprávy viz [15]. Jedná se o typ útoku, který zneužívá určité typy hashů jako autentizační kódy zprávy (MAC), což umožňuje zahrnout další informace k hashi.

Tento útok může být proveden na hashi s konstrukcí H(klíč || zpráva), kde je zpráva a délka klíče známa. Zranitelné hashovací funkce pracují se vstupní zprávou a použijí ji ke transformaci vnitřního stavu H. Po zpracování všech vstupů, je generován hash, který může být použit pro zpracování nových dat. Tímto způsobem se může rozšířit zpráva a vypočítat hash, který je platný pro podpis nové zprávy.

Algoritmy, jako MD5 a SHA-1, které jsou založeny na Merkle–Damgårdově konstrukci jsou náchylné k tomuto druhu útoku. Existence tohoto útoku byla důvodem pro modifikaci MAC na autentizační kódy zprávy založené na hashi (HMAC). HMAC používá odlišnou konstrukci – H(klíč || H(klíč || zpráva)), která již není náchylná tomuto útoku.

2.1 Secure Hash Algorithm-3

V rovnicích (2.1) – (2.4) z [16] jsou definovány 4 hashovací funkce SHA-3 pomocí funkce nazvané Keccak[c], její algoritmus bude podrobně rozebrán dále. Ke zprávě M se připojí dva bity 01 a kapacita c je dvojnásobkem požadované délky, tj. $c = 2d$.

$$\text{SHA3-224(M)} = \text{KECCAK[448]}(M \,||\, 01,\, 224), \tag{2.1}$$
$$\text{SHA3-256(M)} = \text{KECCAK[512]}(M \,||\, 01,\, 256), \tag{2.2}$$
$$\text{SHA3-384(M)} = \text{KECCAK[768]}(M \,||\, 01,\, 384), \tag{2.3}$$
$$\text{SHA3-512(M)} = \text{KECCAK[1024]}(M \,||\, 01,\, 512) \tag{2.4}$$

2.1.1 Permutace Keccak-p

Předpis permutace je značen Keccak-$p[b, n_r]$, je popsán z [16], kde b je počet bitů a n_r počet iterací této funkce. Soubor vstupních hodnot bitů b se nazývá stavové pole, přičemž $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. Cyklus permutací je definován jako funkce Rnd a skládá se z pěti transformací.

Stavové pole

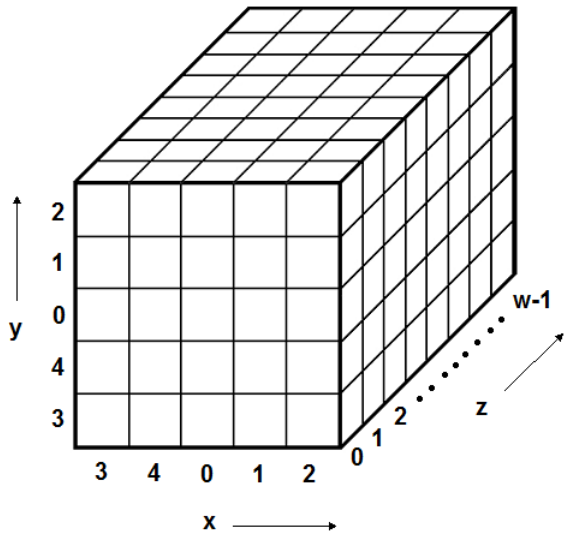
Pro stavové pole definujeme šířku $w = \frac{b}{25}$ a binární logaritmus hloubky pole $l = log_2(\frac{b}{25})$, tyto hodnoty jsou pro každé b uvedeny v tab.2.1 [16].

Původní řetězec S je tvořen z b bitů, které jsou indexovány od 0 do b-1 následovně: $S = S[0] \,||\, S[1] \,||...||\, S[b-2] \,||\, S[b-1]$. Jednotlivé bity z řetězce se přeskupí

Tab. 2.1: Hodnoty parametrů stavového pole

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

do trojrozměrného prostoru, konkrétně do tvaru pravidelného čtyřbokého hranolu, jehož délka i výška má velikost 5 bitů a šířka je rovna velikosti w. Pro stavové pole A, jak je znázorněné v [16], je jeden bit jednoznačně určen pomocí souřadnic A[x,y,z], kde $x \in \langle 0,\ 5)$, $y \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, viz obr.2.2. Jednotlivé souřadnice na osách x a y nejsou seřazeny chronologicky, ale dle uvedeného pořadí: $3, 4, 0, 1, 2$. Souřadnice osy z zůstávají seřazeny vzestupně: 0 – w-1.



Obr. 2.2: Stavové pole

Konverze stavového pole na řetězec

Odpovídající řetězec S může být vytvořen, pomocí algoritmu z [16], z pole A následovně: nejprve definujeme řetězec Řada(i, j) takový, že $i, j \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$.

$$\text{Řada}(i,j) = A[i,j,0] \,||\, A[i,j,1] \,||\, A[i,j,2] \,||...||\, A[i,j,w-2] \,||\, A[i,j,w-1] \quad (2.5)$$

Například pro b=1600 a w=64:

Řada$(0,0) = A[0,0,0] \,||\, A[0,0,1] \,||\, A[0,0,2] \,||...||\, A[0,0,62] \,||\, A[0,0,63]$

Řada$(1,0) = A[1,0,0] \,||\, A[1,0,1] \,||\, A[1,0,2] \,||...||\, A[1,0,62] \,||\, A[1,0,63]$

Řada$(2,0) = A[2,0,0] \,||\, A[2,0,1] \,||\, A[2,0,2] \,||...||\, A[2,0,62] \,||\, A[2,0,63]$

$Plocha(j)$ určuje rovinu pole, která je rovnoběžná s osou x a z a $j \in \langle 0,\, 5)$.

$$Plocha(j) = \check{\mathrm{R}}ada(0,j) \,||\, \check{\mathrm{R}}ada(1,j) \,||\, \check{\mathrm{R}}ada(2,j) \,||\, \check{\mathrm{R}}ada(3,j) \,||\, \check{\mathrm{R}}ada(4,j), \quad (2.6)$$

pak řetězec S má podobu:

$$S = Plocha(0) \,||\, Plocha(1) \,||\, Plocha(2) \,||\, Plocha(3) \,||\, Plocha(4). \quad (2.7)$$

Příklad pro b=1600 a w=64:

$$\begin{aligned} S = {} & A[0,0,0] \,||\, A[0,0,1] \,||\, A[0,0,2] \,||...|| \, A[0,0,62] \,||\, A[0,0,63] \\ & ||\, A[1,0,0] \,||\, A[1,0,1] \,||\, A[1,0,2] \,||...|| \, A[1,0,62] \,||\, A[1,0,63] \\ & ||\, A[2,0,0] \,||\, A[2,0,1] \,||\, A[2,0,2] \,||...|| \, A[2,0,62] \,||\, A[2,0,63] \\ & \qquad\qquad\qquad\qquad \vdots \\ & ||\, A[3,4,0] \,||\, A[3,4,1] \,||\, A[3,4,2] \,||...|| \, A[3,4,62] \,||\, A[3,4,63] \\ & ||\, A[4,4,0] \,||\, A[4,4,1] \,||\, A[4,4,2] \,||...|| \, A[4,4,62] \,||\, A[4,4,63] \end{aligned}$$

Konverze řetězce na stavové pole

Pro každý bit řetězce S platí při převodu na bit pole A vztah uvedený v rovnici(2.8), obsažený v [16].

$$\mathrm{A}\,[x,y,z] \;\; = \;\; \mathrm{S}\,[w(5y+x)+z] \quad (2.8)$$

Například pro b=1600 a w=64:

$$\begin{array}{llll} \mathrm{A}\,[0,0,0] = \mathrm{S}[0] & \mathrm{A}\,[1,0,0] = \mathrm{S}[64] & \mathrm{A}\,[4,0,0] = \mathrm{S}[256] & \mathrm{A}\,[0,1,0] = \mathrm{S}[320] \\ \mathrm{A}\,[0,0,1] = \mathrm{S}[1] & \mathrm{A}\,[1,0,1] = \mathrm{S}[65] & \mathrm{A}\,[4,0,1] = \mathrm{S}[257] & \mathrm{A}\,[0,1,1] = \mathrm{S}[321] \\ \vdots & \vdots & \vdots & \vdots \\ \mathrm{A}\,[0,0,63] = \mathrm{S}[63] & \mathrm{A}\,[1,0,63] = \mathrm{S}[127] & \mathrm{A}\,[4,0,63] = \mathrm{S}[319] & \mathrm{A}\,[0,1,63] = \mathrm{S}[383] \\ & & & \\ \mathrm{A}\,[0,2,0] = \mathrm{S}[640] & \mathrm{A}\,[1,2,0] = \mathrm{S}[704] & \mathrm{A}\,[4,2,0] = \mathrm{S}[896] & \mathrm{A}\,[0,3,0] = \mathrm{S}[960] \\ \vdots & \vdots & \vdots & \vdots \\ \mathrm{A}\,[0,2,63] = \mathrm{S}[703] & \mathrm{A}\,[1,2,63] = \mathrm{S}[767] & \mathrm{A}\,[4,2,63] = \mathrm{S}[959] & \mathrm{A}\,[0,3,63] = \mathrm{S}[1023] \end{array}$$

atd.

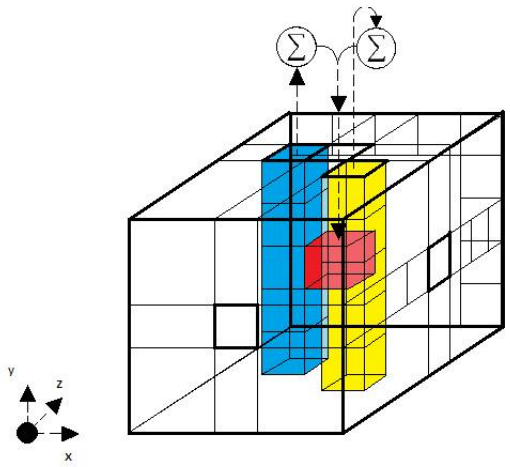
Transformace

Funkce Keccak-$p[b, n_r]$ zahrnuje pět transformací, které se značí θ, ρ, π, χ a ι a jsou definovány v [16]. Vstupními daty těchto transformací je stavové pole A a výstupními daty je pole A′. Je znám parametr b, a proto se ze zápisu vynechává velikost pole A. Transformace ι má i druhý vstup a tím je hodnota dané iterace i_r.

Algoritmus $\theta(A)$

1. Pro všechny dvojice (x, z) takové, že $x \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $C\,[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$.
2. Pro všechny dvojice (x, z) takové, že $x \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $D\,[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$.
3. Pro všechny trojice (x, y, z) takové, že $x \in \langle 0,\ 5)$, $y \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $A'[x, y, z] = A[x, y, z] \oplus D\,[x, z]$.

Funkce θ spočívá v použití operace xor na součet bitů ve dvou sloupcích pole A a ten pak uloží na pozici bitu v poli A′.
Pro bit $A[x_0,y_0,z_0]$: první sloupec má souřadnici x rovnu $(x_0-1) \bmod 5$ a souřadnicí z je z_0. Souřadnice x druhého sloupce je $(x_0 + 1) \bmod 5$ a z je rovna $(z_0 - 1) \bmod w$.



Obr. 2.3: Algoritmus θ

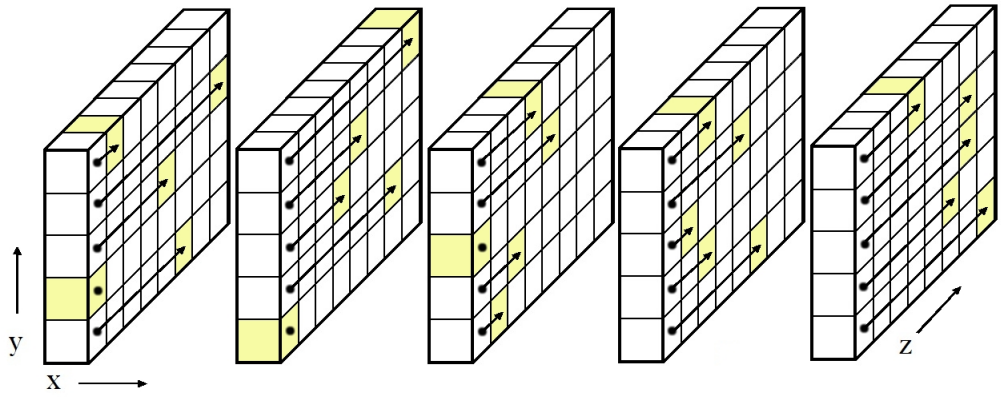
Algoritmus $\rho(A)$

1. Pro $\forall z : z \in \langle 0,\ w)$, nechť $A'[0, 0, z] = A[0, 0, z]$.
2. Nechť $(x, y) = (1, 0)$.
3. Pro $t \in \langle 0,\ 23\rangle$:
   (a) Pro $\forall z : z \in \langle 0,\ w)$, nechť $A'[x, y, z] = A[x, y, z - \frac{(t+1)(t+2)}{2} \bmod w]$.
   (b) Nechť $(x, y) = (y, (2x + 3y) \bmod 5)$.
4. Vrať A′.

Algoritmus způsobí rotaci bitů podél řady z o délku nazvanou offset, která závisí na souřadnicích x a y. Pro každý bit je souřadnice z dána součtem původní souřadnice z a offsetu modulo velikostí řady w. Hodnota offsetu pro každou řadu je uvedena v tab. 2.2. Případ w=8 je zobrazen na obr. 2.4, počátek každé šipky značí bit, který se přesunuje na jinou pozici. Tato pozice je označena žlutým zabarvením. Ostatní bity řady jsou posunuty o stejný offset.

Tab. 2.2: Hodnoty offsetu

	x=3	x=4	x=0	x=1	x=2
y=2	153	231	3	10	171
y=1	55	276	36	300	6
y=0	28	91	0	1	190
y=4	120	78	210	66	253
y=3	21	136	105	45	15

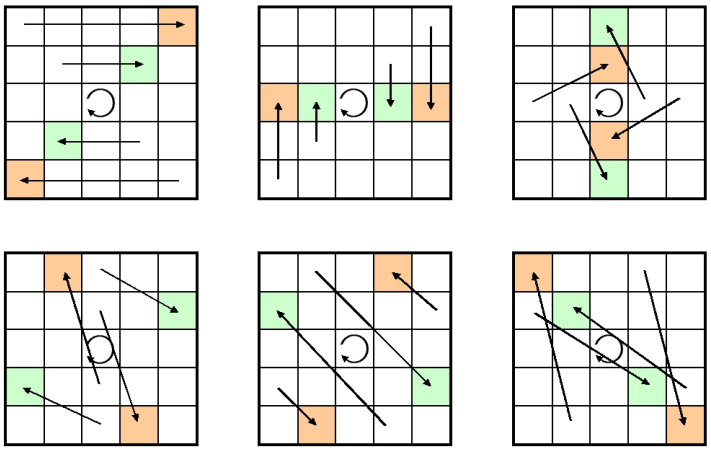


Obr. 2.4: Algoritmus ρ pro w=8

Algoritmus $\pi(A)$

1. Pro všechny trojice (x, y, z) takové, že $x \in \langle 0,\ 5)$, $y \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $A'[x, y, z] = A[(x + 3y) \bmod 5, x, z]$.
2. Vrať A′.

Efektem π je přeskupení bitů v rámci každé roviny rovnoběžné s osami x a y. Bit A[0,0,z] je středem, okolo kterého se otáčí všechny ostatní bity v rovině, viz obr. 2.5.



Obr. 2.5: Algoritmus π

Algoritmus $\chi(A)$

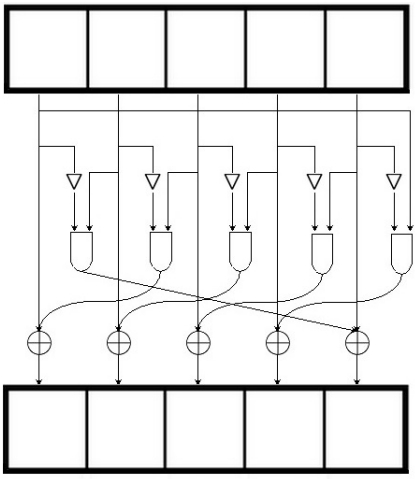
1. Pro všechny trojice (x, y, z) takové, že $x \in \langle 0,\ 5)$, $y \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1) \bmod 5, y, z] \oplus 1) \cdot A[(x + 2) \bmod 5, y, z])$.
2. Vrať A′.

Algoritmus χ je zobrazen na obr. 2.6. Provede se operace xor s dvěma dalšími bity pole A řady x, z nichž jeden bit má invertovanou hodnotu.

Algoritmus $\iota(A, i_r)$

1. Pro všechny trojice (x, y, z) takové, že $x \in \langle 0,\ 5)$, $y \in \langle 0,\ 5)$ a $z \in \langle 0,\ w)$, nechť $A'[x, y, z] = A[x, y, z]$.
2. Nechť $RC = 0^w$.
3. Pro $j \in \langle 0,\ l\rangle$, nechť $RC[2^j - 1] = rc(j + 7i_r)$.
4. Pro $\forall\ z$: $z \in \langle 0,\ w)$, nechť $A'[0, 0, z] = A[0, 0, z] \oplus RC[z]$.
5. Vrať A′.

Zápis 0^w značí w po sobě jdoucích bitů s hodnotou 0 a l binární logaritmus hloubky pole.



Obr. 2.6: Algoritmus χ jedné řady

Funkce $rc(t)$ má vstupní hodnotu t a výstupní hodnotou funkce je bit $rc(t)$,

1. Pokud $t \bmod 255 = 0$, vrať 1.
2. Nechť R = 10000000.
3. Pro $i \in \langle 1, t \bmod 255 \rangle$, nechť:
   (a) R = 0 || R
   (b) R[0] = R[0] + R[8]
   (c) R[4] = R[4] + R[8]
   (d) R[5] = R[5] + R[8]
   (e) R[6] = R[6] + R[8]
   (f) R = Trunc_8[R].
4. Vrať R[0].

V algoritmu je užita funkce Trunc_s(X): pro řetězec X o délce s jsou výstupním řetězcem bity od X[0] do X[s-1]. Například $\text{Trunc}_4(100110) = 1001$.

Efektem ι je dle [16] modifikace některých bitů v Řad$\check{\text{e}}(0, 0)$, kdy záleží na iteraci i_r. Ostatní řady nejsou nijak ovlivněny.

Keccak-$p[b, n_r]$

Je dáno stavové pole A a hodnota dané iterace i_r. V rovnici (2.9) ze [16] definujeme posloupnost uvedených transformací. Permutace Keccak-$p[b, n_r]$ se skládá z n_r iterací funkce Rnd.

$$\text{Rnd}(A, i_r) \quad = \quad \iota(\chi(\pi(\rho(\theta(A)))), i_r). \tag{2.9}$$

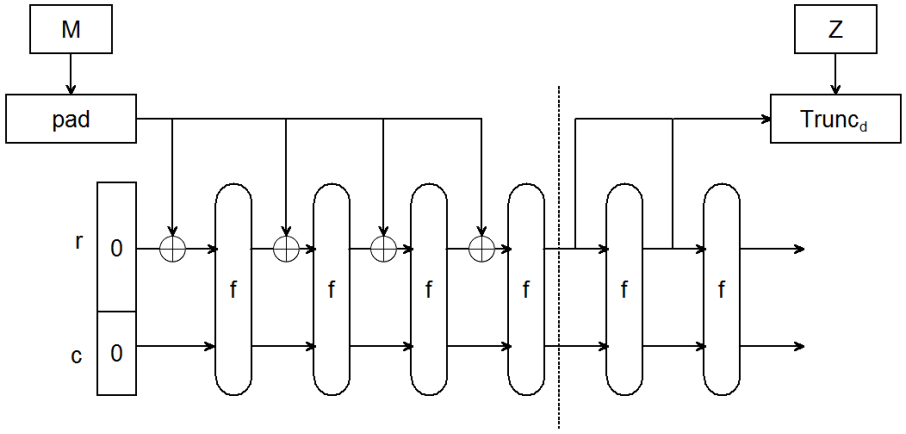
1. Konvertujte řetězec S na stavové pole A.
2. Pro $i_r \in \langle 2l + 12 - n_r, 2l + 12 - 1\rangle$, nechť $A = Rnd(A, i_r)$.
3. Konvertujte stavové pole A na řetězec S′ o délce b.
4. Vrať S′.

Srovnání s Keccak-f

Keccak-f je speciálním případem permutací Keccak-p, popisuje ho rovnice (2.10) z [16], $n_r = 12+12l$. Například Keccak-$p[1600, 24]$ je shodná s funkcí Keccak-$f[1600]$,

$$\text{Keccak-}f[b] \quad = \quad \text{Keccak-}p[b, 12 + 12l]. \tag{2.10}$$

2.1.2 Konstrukce Sponge



Obr. 2.7: Konstrukce sponge

Konstrukce sponge je popsána rovnicí (2.11) z [16] a zobrazena na obr. 2.7. Využívá tří komponent:

- funkce f, která může mít libovolnou délku dat b
- parametr r, nazývající se míra, je vždy menší než b. Kapacita c je hodnota $b - r$, takže platí $r + c = b$.
- funkce pad je funkcí, která doplňuje bity do velikosti určitého řetězce. Podle [16] zajišťuje možnost rozdělení zprávy do řetězců velikosti r. Pokud máme kladnou hodnotu x a nezápornou hodnotu m, pak výstup $\text{pad}(x, m)$ je řetězec takový, že $m + len(\text{pad}(x, m))$ je velikostí násobkem x.

$$Z \quad = \quad \text{SPONGE}[f, \text{pad}, r](M, d), \tag{2.11}$$

kde M je vstupní zpráva a d je požadovaná délka výstupních dat v bitech.

Algoritmus sponge

1. Nechť $P = M \,||\, \text{pad}(r, len(M))$
2. Nechť $n = \frac{len(P)}{r}$
3. Nechť $c = b - r$
4. Nechť $P_0, \ldots, P_{n-1}$ je posloupnost řetězců délky r takových, že $P = P_0 \,||\, \ldots \,||\, P_{n-1}$
5. Nechť $S = 0^b$
6. Pro $i \in \langle 0,\, n-1 \rangle$ nechť $S = f(S \oplus (P_i || 0^c))$
7. Nechť Z je prázdný řetězec
8. Nechť $Z = Z \,||\, \text{Trunc}_r(S)$
9. Pokud $d \leq |Z|$, pak vrať $\text{Trunc}_d(Z)$, jinak se pokračuje na bod č.10
10. Nechť $S = f(S)$ a zpět na bod č.8

Vstupními daty algoritmu jsou řetězec M a nezáporná hodnota d, výstupními daty je řetězec Z, jehož délka $len(Z) = d$.

2.1.3 Keccak[c]

Keccak[c] je definován z [16] pomocí funkce Keccak-$p[b, 2l + 12]$, pad10*1 a volbou parametrů míry r a kapacity c tak, aby $r + c$ odpovídalo hodnotám b z tab. 2.1, tj.$25, 50, 100, 200, 400, 800, 1600$.

V případě $b = 1600$ se jedná o funkci Keccak[c], viz rovnice (2.12).

$$\text{Keccak}[c] \;=\; \text{SPONGE}[\text{Keccak-}p[1600, 24], \text{pad10*1}, 1600 - c] \tag{2.12}$$

Při dané zprávě M a výstupní délce d:

$$\text{Keccak}[c](M, d) = \text{SPONGE}[\text{Keccak-}p[1600, 24], \text{pad10*1}, 1600 - c](M, d) \tag{2.13}$$

$$\tag{2.14}$$

Algoritmus pad10*1

Algoritmus je převzat z [16]. Zápis pad10*1 označuje, že bit 0 je buď úplně vynechán, nebo je opakován, aby bylo dosaženo požadované délky výstupního řetězce, která se rovná $m + len(Z)$.

1. Nechť $j = (-m - 2) \bmod x$
2. Vrať $1 \,||\, 0^j \,||\, 1$

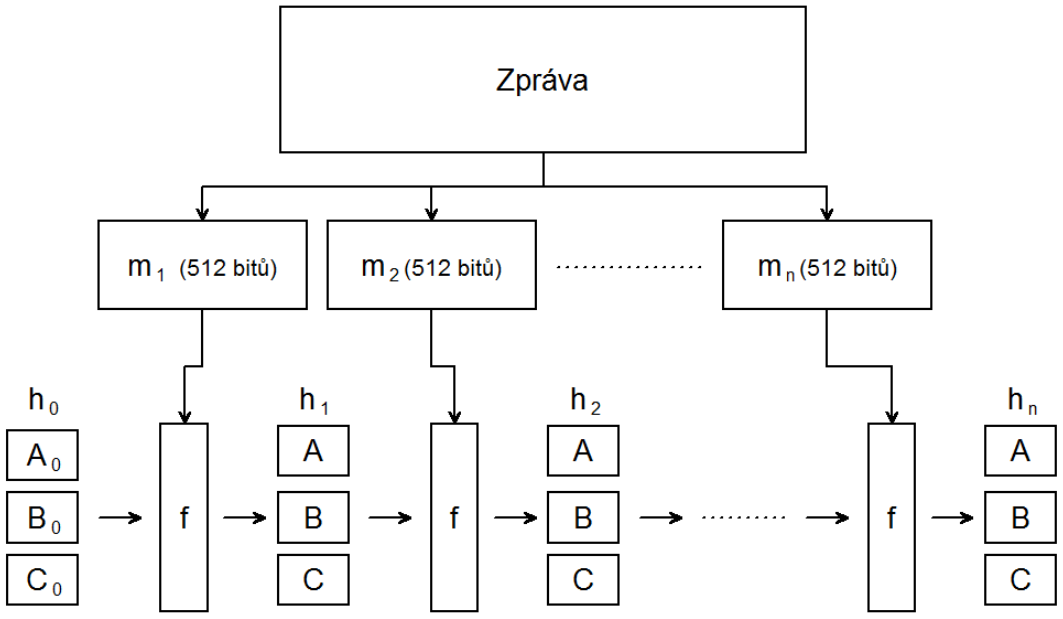
2.2 Tiger

Hashovací funkce Tiger je založena na Merkle-Damgårdově konstrukci, popsán v [17]. Vstupní zpráva je rozdělena na bloky m_n o velikosti 512 bitů. Bity, které zbydou jsou doplněny bitem 1 a tolika bity s hodnotou 0, aby vznikl celý blok 512 bitů. Výsledný hash nejčastěji obsahuje 192 bitů, pak se funkce značí Tiger/192. Existují i varianty Tiger/160 a Tiger/128.

Stěžejní operací je využívání 4 S-boxů, každý má po 256 hodnotách. Další operace jsou uvedeny v tab.2.3 z [18].

Tab. 2.3: Značení operátorů

Značení	Význam
$A \oplus B$	operace xor
A'	bitový doplněk A
$A \ll n$	bitový posun A o n pozic doleva
$A \gg n$	bitový posun A o n pozic doprava
$A + B$	součet A a B modulo 2^{64}
$A - B$	rozdíl A a B modulo 2^{64}
$A \cdot B$	součin A a B modulo 2^{64}

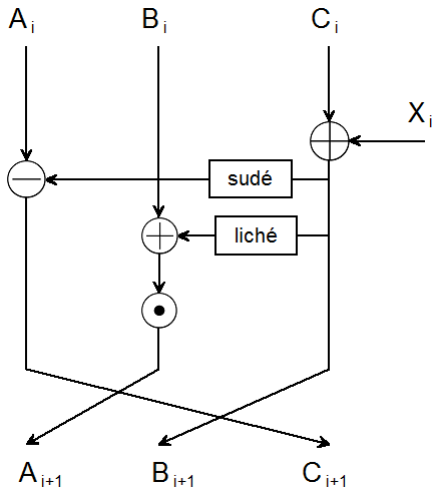


Obr. 2.8: Schéma algoritmu Tiger

2.2.1 Algoritmus

Celý algoritmus ze zdroje [18] se řídí podle schématu na obr.2.8. Hodnoty A_0, B_0, a C_0 jsou pevně stanoveny na 64 bitové slova. Tyto tři slova se souhrně značí h_0. A_0=0x0123456789abcdef, B_0=0xfedcba9876543210, C_0=0xf096a5b4c3b2e187

Každý blok m_n je rozdělen na osm 64 bitových slov $X_0, X_1, \ldots X_7$ a provede se pomocí h_i výpočet h_{i+1}. Jeden tento výpočet se skládá z 24 rund, ve kterých se mění hodnoty A, B, C. Na obr.2.9 je znázorněna jedna runda, popsána je v následujících rovnicích (2.15) – (2.18).



Obr. 2.9: Jedna runda funkce Tiger

$$C = C \oplus X_i \tag{2.15}$$
$$A = A - (S_1[c_0] \oplus S_2[c_2] \oplus S_3[c_4] \oplus S_4[c_6]) \tag{2.16}$$
$$B = B + (S_4[c_1] \oplus S_3[c_3] \oplus S_2[c_5] \oplus S_1[c_7]) \tag{2.17}$$
$$B = B \cdot mul, \tag{2.18}$$

kde $i \in \langle 0, 23 \rangle$. Hodnota $mul = 5$, při $i \in \langle 8, 15 \rangle$ bude $mul = 7$ a pokud je $i \geq 16$, pak je $mul = 9$.

$S_1, \ldots, S_4$ označují konkrétní S-boxy. Hodnota C se rozdělí na 8 bytů $c_7, \ldots, c_0$, kde c_7 je nejvýznamnější byte a c_0 nejméně významný. Pomocí hodnot z S-boxů se dále vypočítají A a B.

Po každé rundě se hodnoty zamění tak, že z A, B, C se stanou hodnoty B, C, A, atp. Po poslední rundě se opět inicializují hodnoty A, B, C a to kombinací prvotních hodnot A_0, B_0, C_0 a hodnot po posledním kole A_{24}, B_{24}, C_{24} viz rovnice (2.19) – (2.21), které se užijí pro další blok m_n.

$$A_{25} = A_0 \oplus A_{24} \tag{2.19}$$
$$B_{25} = B_0 - B_{24} \tag{2.20}$$
$$C_{25} = C_0 + C_{24} \tag{2.21}$$

Key Schedule

Pro získání $X_8, \ldots, X_{23}$ je definována funkce Key Schedule. Ta zajišťuje změnu malého počtu bitů ve zprávě a tím ovlivní velké množství bitů v dalších rundách. Aplikuje se následovně:

$(X_8, \ldots, X_{15}) = \text{KeySchedule}\,(X_0, \ldots, X_7)$
$(X_{16}, \ldots, X_{23}) = \text{KeySchedule}\,(X_8, \ldots, X_{15})$

Se vstupy $Y_0, \ldots, Y_7$ se provedou dle [18] úpravy:

$$\begin{aligned}
Y_0 &= Y_0 - (Y_7 \oplus \text{0xa5a5a5a5a5a5a5a5}) \\
Y_1 &= Y_1 \oplus Y_0 \\
Y_2 &= Y_2 + Y_1 \\
Y_3 &= Y_3 - (Y_2 \oplus ((Y_1') \ll 19)) \\
Y_4 &= Y_4 \oplus Y_3 \\
Y_5 &= Y_5 + Y_4 \\
Y_6 &= Y_6 - (Y_5 \oplus ((Y_4') \gg 23)) \\
Y_7 &= Y_7 \oplus Y_6
\end{aligned}$$

$$\begin{aligned}
Y_0 &= Y_0 + Y_7 \\
Y_1 &= Y_1 - (Y_0 \oplus ((Y_7') \ll 19)) \\
Y_2 &= Y_2 \oplus Y_1 \\
Y_3 &= Y_3 + Y_2 \\
Y_4 &= Y_4 - (Y_3 \oplus ((Y_2') \gg 23)) \\
Y_5 &= Y_5 \oplus Y_4 \\
Y_6 &= Y_6 + Y_5 \\
Y_7 &= Y_7 - (Y_6 \oplus \text{0x0123456789abcdef})
\end{aligned}$$

Finální hodnoty $Y_0, \ldots, Y_7$ jsou výstupem této funkce.

2.2.2 Generování S-Boxů

S-boxy, jak je uvedeno v [19] jsou generovány užitím kompresních funkcí, která produkují pseudonáhodná čísla, která jsou použita při přesunování sloupců různých bajtů. Algoritmus má za vstupní data řetězec náhodných hodnot a počet průchodů. Ty inicializují S-box tak, aby byl každý sloupec identickou permutací bajtů. Pak náhodně vymění páry vstupů každého sloupce bajtů. Každá výměna bajtů je pak závislá na předchozích výměnách ve všech sloupcích. S-boxy by měly mít následující vlastnosti:

1. Každý sloupec bajtů je permutací ze všech 256 možných hodnot bajtu.
2. Sloupce všech S-boxů by měly být co nejrozdílnější a měly by mít dlouhé cykly.
3. Všechny vstupy S-boxů by měly být odlišné a žádné dva vstupy by neměly mít shodné víc než tři bajty.
4. Žádné dva vstupy S-boxu ($S_i(t_1) \oplus S_i(t_2)$ a $S_j(t_3) \oplus S_j(t_4)$) by neměly mít více než čtyři shodné bajty.
5. Rychlost generování by neměla být příliš pomalá.
6. Řetězec náhodných hodnot je snadný na zapamatování.

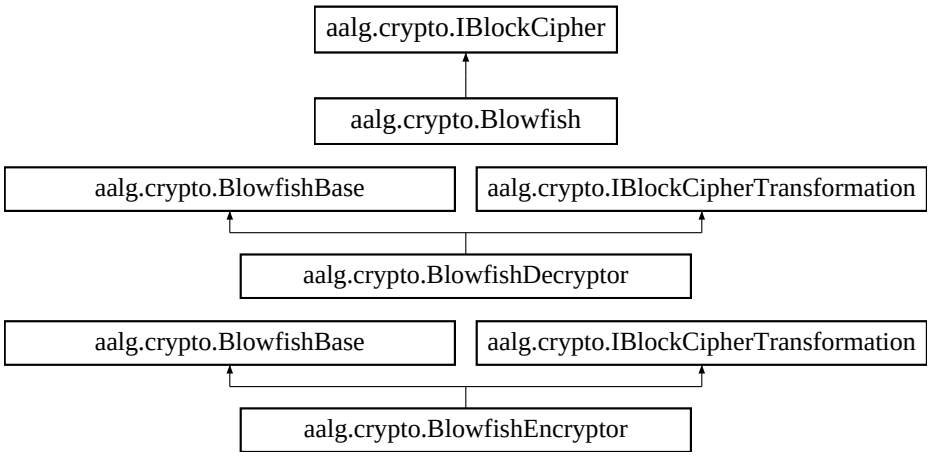
3 IMPLEMENTACE

Algoritmy jsou vytvořeny v knihovně Aalg, která je volně dostupná a je ji možno stáhnout z http://sourceforge.net/projects/aalg/. Tato knihovna obsahuje balíky aalg.algebra, aalg.crypto, aalg.mailer a aalgTest. V balíku aalg.crypto jsou realizované šifry a hashovací funkce. Jejich testování probíhá ve třídách balíku aalgTest, kde je pro každý šifrovací algoritmus vytvořena samostatná třídy a pro hashovací funkce je k dispozici třída HashTest.

Ve třídě HashTest jsou obsaženy testovací vektory hashovacích funkcí Tiger a Sha-3 a tři různé testy. Prvním je hashLengthTest, který porovnává požadovanou délku hashe se skutečným výstupem funkce. Druhý test, nazván correctnessTest, provádí kontrolu výsledku hashe se zadaným. Poslední test - partialUpdateTest, opět testuje výsledky hashů, ale tentokrát je vstupní zpráva předávána k hashování postupně.

3.1 Implementace Blowfish

Třída Blowfish implemetuje rozhraní IBlockCipher, které obsahuje metody pro získání délky bloku, velikosti klíče, šifrování a dešifrování. K šifrování slouží třída BlowfishEncryptor, pro dešifrování je zavedena třída BlowfishDecryptor. Obě tyto třídy využívají třídy BlowfishBase a implementují rozhraní IBlockCipherTransformation.



Obr. 3.1: diagramy tříd Blowfish

3.1.1 Metody třídy BlowfishBase

- protected void **encryptBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset, int value)
  Metoda se využívá k šifrování (hodnota *value* je rovna 0), popř. dešifrování bloků dat (*value* = 17). Pole sourceBlock obsahuje vstupní data, výstupní data jsou vložena do pole targetBlock.
- public long **feistel** (long origBlock, int value)
  Obsahuje Feistelovo schéma, opět se liší hodnotou value pro šifrování/dešifrování.
- public int **function_f** (int l) je metoda pro funkci F viz obr.1.4.
- public long **byteArray_to_long** (byte[] block, int offset, int length)
  Při volání této metody je vstupní blok bajtů délky length transformován na hodnotu typu long.
- public void **setKey** (ISymmetricKey key), metoda je užita pro rozvoj klíče.
- public void **calculateFields** (int[] keys)
  Metoda je volána při nastavování polí P a SBoxes.
- public byte[] **replaceEncryptor** (byte[] input, byte[] output, int[] array, int length)
  Je součástí metody calculateFields, slouží k postupnému přepsání hodnot v poli array délky length pomocí vstupního pole bajtů input.
- public int **byteKey_to_int** (ByteArraySymmetricKey key, int length)
  Klíč uložený v bajtovém poli je transformován na typ integer.

3.1.2 Metody třídy BlowfishEncryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)
  Slouží k volání metody encryptBlock, která zprostředkuje šifrování dat. Vstupními daty je sourceBlock a výstupními daty je targetBlock.

3.1.3 Metody třídy BlowfishDecryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)
  Slouží k volání metody encryptBlock, která zprostředkuje dešifrování dat. Vstupními daty je sourceBlock a výstupními daty je targetBlock.

3.1.4 Testovací třída BlowfishTest

Třída obsahuje následující testy:

- public void **Test_function_f** ()
  Pomocí zadané vstupní a výstupní hodnoty se testuje správná funčknost funkce F.
- public void **Test_encryption** ()
  Metoda slouží k ověření šifrování, jsou zde inicializovány proměnné klíč, vstup a očekávaný výstup. Pomocí objektu encryptor třídy BlowfishEncryptor je nejprve volána metoda setKey a následně transformBlock, její výstup je porovnán s hodnotou očekávaného výstupu.
- public void **Test_decryption** ()
  Zde probíhá kontrola dešifrování, metoda obsahuje klíč, vstup, očekávaný výstup a objekt decryptor třídy BlowfishDecryptor. Opět jsou volány metody setKey a transformBlock, hodnota výstupu je porovnána s očekávaným výstupem.

Vstupní a výstupní hodnoty byly získány pomocí programu SCV Cryptomanager [16] a jsou uvedeny v příloze B.3.

3.2 Implementace TDES

Základem je algoritmus DES, jehož metody jsou volány prostřednictvím Triple DES.

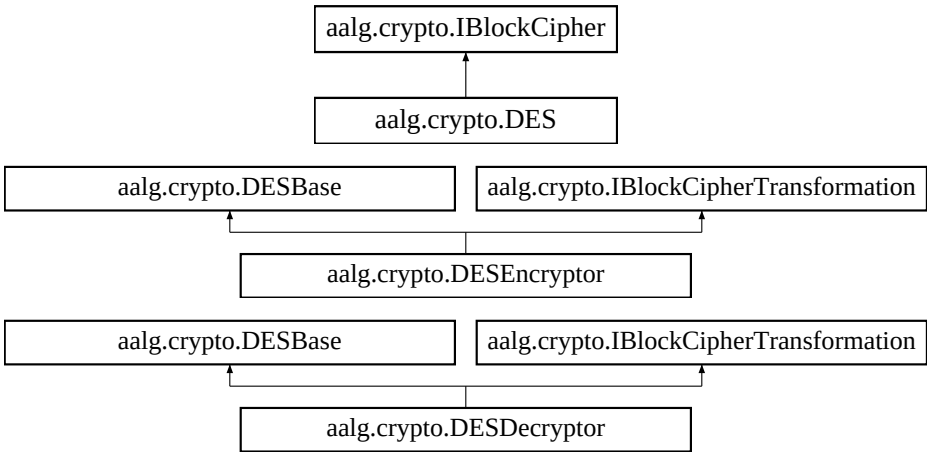
3.2.1 Algoritmus DES

Hlavní třída DES implementuje rozhraní IBlockCipher, šifrování probíhá pomocí třídy DESEncryptor a k dešifrování slouží třída DESDecryptor. Tyto třídy jsou rozšířeny o třídu DESBase a implementují rozhrani IBlockCipherTransformation viz obr.3.2.

Metody třídy DESBase

- protected long **feistel** (long origBlock)
  Obsahuje Feistelovo schéma, vstupní data origBlock jsou šifrována, vznikne kryptogram.
- protected long **feistel_deciphering** (long cipher)
  Metoda představuje Feistelovo schéma, vstupní data cipher jsou dešifrována na prostý text.

- public long **round** (final long r, final long k)
  Tato metoda je funkcí F v obr.1.6, hodnoty S-boxů jsou uloženy v proměnné ROUND_TABLE.
- private static long[][] **initRoundTable()**
  Slouží k inicializaci proměnné $ROUND\_TABLE$, do které jsou uloženy hodnoty z daných S-boxů a zároveň jsou zde provedeny permutace P a E.
- public long[] **calculateRoundKeys** (final long key)
  Metoda přijme klíč, pomocí kterého získá klíče K_1 - K_{16}, které jsou uloženy do pole roundKeys.
- public long **byteArray_to_long** (byte[] block, int offset, int length)
  Bajtové pole je transformováno na typ long.
- public long **byteKey_to_long** (ByteArraySymmetricKey key, int length)
  Klíč uložený v bajtovém poli je transformován na typ long.



Obr. 3.2: diagramy tříd DES

Metody třídy DESEncryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)
  Přijímá vstupní data sourceBlock, která nejprve transformuje na hodnotu typu long. Následně se volá metoda feistel a do pole targetBlock se vloží výstupní hodnoty.
- public void **setKey** (ISymmetricKey key)
  Metoda zajišťuje výpočet klíčů voláním calculateRoundKeys, předává klíč jako hodnotu long.

Metody třídy DESDecryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)
  Data sourceBlock jsou pomocí metody byteArray_to_long vložena do proměnné typu long a dále jsou dešifrována metodou feistel_deciphering.
- public void **setKey** (ISymmetricKey key)
  Metoda zajišťuje výpočet klíčů voláním calculateRoundKeys, předává klíč jako hodnotu long.

Testovací třída DesTest

- public void **Test_round**
  V testu probíhá volání metody round objektu encryptor, výsledek metody je porovnán s očekávaným výsledkem.
- public void **Test_encryption**
  Obsahuje objekt encryptor třídy DESEncryptor, vstupní data, klíč a očekávaná výstupní data. Nejprve se nastaví klíče metodou setKey, poté se provede zašifrování vstupních dat a ověří se výsledek s očekávanými daty.
- public void **Test_decryption**
  Metoda kontroluje správné dešifrování dat, objekt decryptor volá metody setKey a transformBlock se zadanými testovacími hodnotami. Nakonec se testuje shoda výstupu s očekávanými daty.

Testovací hodnoty pro všechny testy jsou uvedeny v příloze B.1. Vektory se nacházejí na webové stránce [21]. Pouze hodnoty pro funkci F byly získány přepočtem vstupních dat podle teorie.

3.2.2 Algoritmus TDES

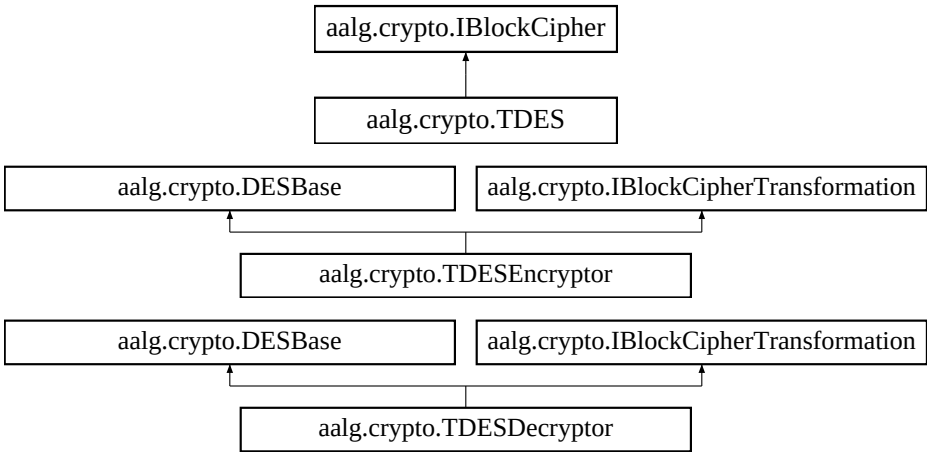
Algoritmus Triple DES sestává ze třídy TDES, která implementuje rozhraní IBlockCipher, tříd TDESEncryptor a TDESDecryptor. Obě třídy implementují IBlockCipherTransformation a rozšiřuje je třída DESBase, viz obr.3.3

Metody třídy TDESEncryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)

V metodě je definován objekt enc třídy DESEncryptor a objekt dec třídy DESDecryptor. Transformace bloku odpovídá schématu dle obr.1.5. Zašifrování probíhá voláním metody setKey s klíčem key1, případně key3 a dále se volá metoda transformBlock. Při dešifrování je postup stejný, jen se aplikuje s objektem dec a klíčem key2.

- public void **setKey** (ISymmetricKey key)
  Klíč přejatý parametrem je postupně rozdělen na tři různé klíče o stejné délce.



Obr. 3.3: diagramy tříd TDES

Metody třídy TDESDecryptor

- public IBlockCipher **getCipher** (), metoda vrací proměnnou cipher.
- public void **transformBlock** (byte[] sourceBlock, int sourceOffset, byte[] targetBlock, int targetOffset)
  V metodě je definován objekt enc třídy DESEncryptor a objekt dec třídy DESDecryptor. Transformace bloku odpovídá schématu dle obr.1.5. Dešifrování probíhá voláním metody setKey s klíčem key1, případně key3 a dále se volá metoda transformBlock. Při šifrování je postup stejný, jen se aplikuje s objektem enc a klíčem key2.
- public void **setKey** (ISymmetricKey key)
  Klíč přejatý parametrem je rozdělen na tři různé klíče o stejné velikosti.

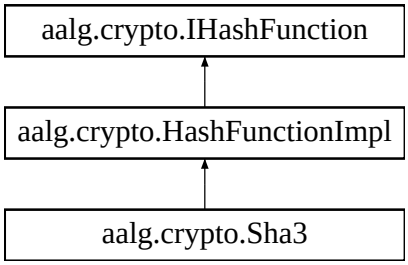
Testovací třída TDESTest

Ověřování šifrování a dešifrování algoritmu Triple DES probíhá téměř stejně jako testování ve třídě DesTest. Třída obsahuje pouze testy Test_encryption() a Test_decryption(), které jsou rozdílné oproti metodám DesTestu pouze jinou délkou klíče. Ke zjištění

testovacích hodnot byl použit program SCV Cryptomanager ze [20] a vektory jsou uvedeny v příloze B.2.

3.3 Implementace Secure Hash Algorithm-3

Algoritmus SHA-3 je vytvořený ve třídě Sha3. Ta je rozšířená o třídu HashFunctionImpl, která implementuje rozhraní IHashFunction. Ověření funkčnosti je realizované ve třídě HashTest, popsané v úvodu kapitoly. Testovací vektory jsou uvedeny v příloze B.4 a byly získány ze zdroje [23].



Obr. 3.4: diagram SHA-3

3.3.1 Metody třídy Sha3

- public **Sha3** (int HashLength)
  Konstruktor přebírá parametrem hodnotu HashLength, která odpovídá délce hashe v bitech.
- public void **theta** (), metoda odpovídá transformaci θ.
- public void **ro** (), provede se transformace ρ.
- public void **pi** (), v metodě se data upraví pomocí transformace π.
- public void **chi** (), odpovídá transformaci χ.
- public void **iota** (int roundIndex)
  Je přejímán parametr roundIndex, který je roven hodnota konkrétní iterace i_r.
  V metodě se provede transformace ι.
- public void **rc** () Metoda předpočítá veškeré výsledky hodnot funkce $rc(t)$ a uloží je do pole *RCvalue*.
- public byte[ ] **convert_array_to_string** ()
  Pomocí metody je konvertováno stavové pole na pole bajtů.
- public byte[ ][ ][ ] **convert_string_to_array** (byte[ ] message)
  V metodě probíhá konverze pole bajtů na stavové pole.

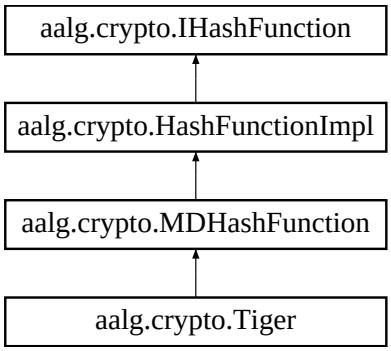
- public byte[] **KeccakP** (byte[] message, int rounds)
  Přijímá vstupní zprávu message, se kterou probíhají transformace dat s počtem rund uvedených jako rounds.
- public byte[] **padding** ()
  Metoda vytváří pole bajtů, které má v nultém bajtu hodnotu 6 a v posledním bajtu hodnotu 0x80.
- public void **UpdateFieldS** () Provede se operace xor bajtů zprávy a pole S, do kterého je pak uložen výsledek volání metody KeccakP.
- public void **initialize** ()
  V metodě se inicializují členské proměnné a volá se metoda rc().
- public void **update** (byte[] data, int offset, int size)
  Metoda prostřednictvím UpdateFieldS () zajišťuje postupné hashování bloků vstupních dat o velikosti size.
- public byte[] **finish** ()
  Data se doplní o padding, provede se UpdateFieldS (). Následně se provádí s polem bajtů Z úpravy, dokud pole nesplní zadané podmínky. Pak jsou vráceny hodnoty hashe v podobě bajtového pole.
- public int **getOutputByteCount** ()
  Vrací počet bajtů hashe.
- public int **getBlockSize** ()
  Metoda vrací velikost bloku.
- public IHashFunction **copy** ()
  Vrátí hodnoty získáné konstruktorem.

3.4 Implementace Tiger

Na obr. 3.5 je vidět struktura algoritmu Tiger. Třída Tiger je potomkem třídy MDHashFunction, ta je rozšířena o HashFunctionImpl, která implementuje rozhraní IHashFunction. K testování se využívá třídy HashTest, která byla popsána v úvodu kapitoly. Testovací vektory jsou uvedeny v příloze B.5 a byly převzaty z [22].

3.4.1 Metody třídy Tiger

- protected void **updateHashByBlock** ()
  Zajišťuje hashování jednoho bloku dat.
- protected void **parseBlock** (byte[] input, int offset)
  Metoda převede hodnoty z bajtového pole do pole hodnot typu long.



Obr. 3.5: diagram pro Tiger

- protected byte[ ] **hashToBytes** (long[ ] hash2)
  Vstupní proměnnou je hash, výstupními daty je bajtové pole s hodnotami hashe.
- public void **save_abc** ()
  Metoda zachová hodnoty původního hashe.
- public void **round** (int offset, long x)
  Pracuje stejně jako runda z obr.2.9. Podle vstupní hodnoty offset, která nabývá hodnot $0 - 2$, jsou rozlišovány indexy a, b a c. Proměnná long x značí vstupní blok dat.
- public void **key_schedule** ()
  Metoda představuje Key Schedule.
- public void **feed_forward** ()
  V této metodě se upraví hodnoty hashů pomocí hodnot uložených v save_abc ().
- public long **multiply5** (long b)
  Hodnota b je násobena pěti pomocí bitového posunu.
- public long **multiply7** (long b)
  Hodnota b je násobena sedmi pomocí bitového posunu.
- public long **multiply9** (long b)
  Hodnota b je násobena devíti pomocí bitového posunu.
- public void **initialize** ()
  Proběhne inicializace hashe a volá se metoda initialize () předka.
- public byte[ ] **finish** ()
  V metodě finish se doplní data o padding, provede se updateHashByBlock () a následně jsou vráceny hodnoty hashe v podobě bajtového pole.
- public int **getOutputByteCount** ()
  Vrací počet bajtů hashe.
- public int **getBlockSize** ()

Metoda vrací velikost bloku.

- public IHashFunction **copy** ()
  Vrátí hodnoty získáné konstruktorem.

3.4.2 Metody třídy MDHashFunction

- public void **initialize** ()
  Inicializuje členské proměnné.
- public void **update** (final byte[] data, final int offset, final int size)
  Metoda zajišťuje prostřednictvím updateHashByBlock () postupné hashování bloků vstupních dat o velikosti size.
- public byte[] **finish** ()
  Provádí část paddingu a vrací hodnoty hashe v poli bajtů.
- protected abstract byte[] **hashToBytes** (long[] hash2)
- protected abstract void **updateHashByBlock** ()
- protected abstract void **parseBlock** (final byte[] input, final int offset)

4 POROVNÁNÍ RYCHLOSTÍ

V balíku aalgTest se nachází dvě testovací třídy ke měření, nazvané CrossBlockCipherTest a CrossHashTest. Měření bylo prováděno na procesoru Intel Core i5−3210M, 2.50 GHz.

První ze tříd testuje blokové šifry celkem ve čtyřech různých testech. Každá šifra se porovnává s výsledky javovské implementace, tím se potvrdí její správnost. Dále se měří čas, který uplyne za dobu testu a pomocí něj je získána rychlost transformace dat. Oba testy probíhají zvlášť pro šifrování a dešifrování.

Tab. 4.1: Rychlosti šifrování dat algoritmů

Blowfish		DES		TDES	
Aalg	Java	Aalg	Java	Aalg	Java
38 868	73 014	9 492	35 837	1 099	12 480
36 168	65 651	9 481	37 924	1 109	12 019
38 675	73 014	9 333	37 202	1 097	12 187
38 675	74 404	9 527	37 741	1 110	12 641
38 485	74 404	9 446	35 511	1 109	12 480
33 530	73 702	9 504	38 485	1 103	11 695
25 201	74 404	9 367	37 026	1 104	11 695
38 296	73 702	9 158	37 924	1 103	12 361
38 675	76 593	9 753	37 380	1 136	12 913
38 868	76 593	9 597	36 851	1 141	12 641
38 541	73 548	9 466	37 188	1 111	12 311

V tabulce 4.1 a 4.2 jsou uvedeny rychlosti jednotlivých algoritmů v jednotkách kB/s. Na posledním řádku tabulky jsou uvedeny průměrné hodnoty ze všech měření. Při šifrování rychlost algoritmu Blowfish knihovny Aalg dosahuje 52,4% vůči rychlosti knihovny javy, DES je na 24,45% a algoritmus Triple DES odpovídá přibližně 9% rychlosti javy. Při dešifrování jsou výsledky obdobné.

Ve druhé třídě CrossHashTest se měří rychlosti hashovacích funkcí. Java bohužel neobsahuje vlastní implementace požadovaných hashovacích funkcí, takže se odtud získají pouze hodnoty rychlostí funkcí z knihovny Aalg. Proto jsem ze zdroje [24] získala knihovnu sphlib 3.0, která má taktéž implementovánu hashovací funkci Tiger. Výsledky se nachází v tabulce 4.3. Hodnota průměrné rychlosti Tigeru je rovna přibližně 15,3% rychlosti knihovny sphlib. Rychlost hashovací funkce SHA-3 je velmi pomalá, pohybuje se okolo 455 kB/s.

Tab. 4.2: Rychlosti dešifrování dat algoritmů

Blowfish		DES		TDES	
Aalg	Java	Aalg	Java	Aalg	Java
28 617	60 096	9 333	34 568	1 109	12 460
31 375	61 035	9 633	37 380	1 115	12 641
30 517	62 003	9 356	34 416	1 107	12 187
30 517	57 444	9 084	35 837	1 113	12 560
32 552	61 035	9 094	34 877	1 105	12 460
34 265	60 096	9 322	35 511	1 110	12 400
21 945	58 302	9 356	36 851	1 115	12 361
31 758	60 096	9 469	36 678	1 113	12 037
31 502	63 004	9 668	37 026	1 140	12 828
31 001	63 004	9 256	33 103	1 135	12 460
30 405	60 612	9 357	35 625	1 116	12 439

Tab. 4.3: Rychlosti hashovacích funkcí

Tiger		SHA-3
Aalg [kB/s]	sphlib [MB/s]	Aalg [kB/s]
9 765	51,23	478
10 850	67,24	445
9 765	91,09	481
9 765	39,10	447
9 765	43,78	450
10 850	85,22	417
9 765	75,89	445
9 765	91,40	454
10 850	62,46	450
9 765	38,36	481
10 091	64,58	455

5 ZÁVĚR

Cílem bakalářské práce bylo podrobně prostudovat požadované algoritmy. V úvodu jsem uvedla obecné informace o kryptologii, jejím rozdělení a základní pojmy, se kterými se v kryptologii setkáváme.

První kapitola se zabývá principem blokových šifer a dále jsou popsány algoritmy dvou z nich – Blowfish a Triple DES. Obě šifry využívají Feistelovo schéma.

Další část práce se věnuje hashovacím funkcím. Z nich byly vybrány a důkladně rozebrány funkce Tiger a SHA-3.

Ve třetí kapitole se popisuje základní struktura knihovny Aalg a následně je zde uvedena struktura algoritmů, se kterými jste byli seznámeni v předchozích kapitolách. U každého z nich jsou vypsány a okomentovány metody, které využívají třídy daného algoritmu a to i třídy testovací.

Poslední kapitola porovnává efektivitu jednotlivých implementací. V případě blokových šifer jsou algoritmy testovány vůči jejich implementacím knihovny Java. Dobře dopadl Blowfish, který má průměrnou rychlost okolo 37,64 MB/s při šifrování 37,64 MB/s při dešifrování, což jsou přibližně poloviční hodnoty oproti Javě. Hodnoty rychlostí Triple DES dosahují 1 MB/s a odpovídají přibližně dvanáctině požadované rychlosti.

U hashovacích funkcí nebyla možnost srovnání s javovou implementací. Tiger byl tedy srovnán s knihovnou sphlib, kdy dopadl poněkud hůře, s rychlostí cca 9,85 MB/s, vůči druhé implementaci. Pro Secure Hash Algorithm 3 se mi bohužel nepovedlo získat knihovnu, se kterou bych mohla hodnoty porovnat, ale již ze samotného výsledku měření vlastní implementace je patrné, že daná realizace algoritmu není efektivní.

LITERATURA

[1] AMIROVÁ, Kamilla. *Úvod do kryptografie* [online]. 2007 [cit. 2014-12-13]. Dostupné z: http://sifrovani.fd.cvut.cz/index.html

[2] HAVLÍČEK, Michal. *DES* [online]. 28. března 2011 [cit. 2014-12-13]. Dostupné z: https://kmlinux.fjfi.cvut.cz/ balkolub/Vyuka/leto2011/Havlicek.pdf

[3] SCHNEIER, Bruce. *Schneier on Security* [online]. 2014 [cit. 2014-12-13]. Dostupné z: https://www.schneier.com/blowfish.html

[4] PŘISPĚVATELÉ WIKIPEDIE. Asymetrická kryptografie. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 25. 11. 2014 [cit. 2014-12-13]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Asymetrick%C3%A1_kryptografie&oldid=12029358

[5] MAO, Wenbo. *Modern Cryptography: Theory and Practice*. 5th ed. Upper Saddle River: Prentice Hall, 2004, 707 s. ISBN 01-306-6943-1.

[6] PŘISPĚVATELÉ WIKIPEDIE. SHA-3. In: *Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2001-, 7. 08. 2013 [cit. 2014-12-14]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=SHA-3&oldid=10619579

[7] WIKIPEDIA CONTRIBUTORS. Tiger (cryptography). In: *Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2001, 24 November 2014 [cit. 2014-12-14]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Tiger_(cryptography)&oldid=635277170

[8] SMART, Nigel. *Cryptography: An Introduction* [online]. 3rd Edition. [cit. 2015-06-01]. Dostupné z: http:/www.cs.bris.ac.uk/~nigel/Crypto_Book/

[9] PŘISPĚVATELÉ WIKIPEDIE. Feistelova šifra. In: *Wikipedia: the free encyclopedia* [online]. Wikimedia Foundation, 2001, 16. 09. 2013 [cit. 2014-12-14]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Feistelova_%C5%A1ifra&oldid=10721943

[10] PŘIKRYL, Jan. *Blokové šifry* [online]. 2011, aktualizováno 15.12.2013 [cit. 2014-12-10]. Dostupné z: http://euler.fd.cvut.cz/predmety/y2kk/kzk-krypto-blok.pdf

[11] *World Journal of Science and Technology: Image encryption and decryption using blowfish algorithm* [online]. 2012 [cit. 2014-12-14]. ISSN 2231 – 2587. Dostupné z: https://www.academia.edu/5132258/Image_encryption_and_decryption_using_blowfish_algorithm

[12] GATLIFF, Bill. Encrypting data with the Blowfish algorithm. In: *Embedded: cracking the code to systems development* [online]. July 15, 2003 [cit. 2014-12-14]. Dostupné z: http://www.embedded.com/design/configurable-systems/4024599/Encrypting-data-with-the-Blowfish-algorithm

[13] SCHNEIER, Bruce. The Blowfish Encryption Algorithm – One Year Later. In: *Schneier on Security* [online]. September 1995 [cit. 2014-12-14]. Dostupné z: https://www.schneier.com/paper-blowfish-oneyear.html

[14] FIPS PUB 46-3. *Data Encryption Standard (DES).* [online]. National Institute of Standards and Technology, 25. 10. 1999 [cit. 2014-11-27]. Dostupné z: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[15] WIKIPEDIA CONTRIBUTORS. Length extension attack. *Wikipedia: The Free Encyclopedia* [online]. 2014, 9 September 2014 [cit. 2014-12-12]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Length_extension_attack&oldid=624837648

[16] FIPS PUB 202. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* [online]. National Institute of Standards and Technology, 28. 05. 2014 [cit. 2014-11-27]. Dostupné z: http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf

[17] ANDERSON, Ross a Eli BIHAM. *Tiger: A Fast New Hash Function* [online]. 1996 [cit. 2014-12-14]. Dostupné z: http://www.cl.cam.ac.uk/~rja14/Papers/tiger.pdf

[18] Cryptanalysis of the Tiger Hash Function. (ED)., Kaoru Kurosawa). *Advances in cryptology– ASIACRYPT 2007 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007: proceedings*. Berlin: Springer, 2007. ISBN 9783540769002.

[19] ANDERSON, Ross a Eli BIHAM. *Generation of the S boxes of Tiger* [online]. 1996 [cit. 2014-12-14]. Dostupné z: http://www.cl.cam.ac.uk/~rja14/Papers/tigersb.pdf

[20] SCV TOOLS. *SCV Cryptomanager* [online]. [cit. 2015-05-30]. Dostupné z: http://cryptomanager.com/

[21] *Project NESSIE - New European Schemes for Signature, Integrity, and Encryption. *Kuleuven.be* [online]. [cit. 2015-05-31]. Dostupné z:

https://www.cosic.esat.kuleuven.be/nessie/testvectors/bc/des/Des-64-64.test-vectors

[22] Hash test vectors. In: *Http://www.febooti.com* [online]. [cit. 2015-05-31]. Dostupné z: `http://www.febooti.com/products/filetweak/members/hash-and-crc/test-vectors/#febooti`

[23] Test vectors for SHA-1, SHA-2 and SHA-3. In: [online]. [cit. 2015-05-31]. Dostupné z: `http://www.di-mgt.com.au/sha_testvectors.html`

[24] ANR SAPHIR 2. Sphlib 3.0. In: [online]. [cit. 2015-06-01]. Dostupné z: `http://www.saphir2.com/sphlib/`

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

3DES Triple Data Encryption Standard

AES Advanced Encryption Standard

DES Data Encryption Standard

HMAC Hash-based message authentication code

IDEA International data encryption algorithm

MAC Message Authentication Code

MD5 Message Digest algorithm 5

RC4 Rivest Cipher 4

RSA Šifra pojmenovaná po iniciálech autorů: Rivest, Shamir, Adleman

SHA Secure Hash Algorithm

SEZNAM PŘÍLOH

A PŘÍLOHA 3DES

Tab. A.1: Funkce S_1,...,S_8

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	12	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S_3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S_5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

$\mathbf{S}_7$

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

$\mathbf{S}_8$

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

B TESTOVACÍ VEKTORY

B.1 DES

Tab. B.1: Data v Test_round

vstupní hodnota r	0xDA0307D5C257
vstupní hodnota k	0x0
výstupní hodnota	0xCFAA0575ABA7

Tab. B.2: Data v Test_encryption

prostý text	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
klíč	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
kryptogram	0x99 0x4D 0x4D 0xC1 0x57 0xB9 0x6C 0x52

Tab. B.3: Data v Test_decryption

kryptogram	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
klíč	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
prostý text	0x99 0x4D 0x4D 0xC1 0x57 0xB9 0x6C 0x52

B.2 TDES

Tab. B.4: Data v Test_encryption

prostý text	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
klíč	0x87 0x25 0xDD 0xD9 0x43 0xC7 0x24 0xDD 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x11
kryptogram	0x99 0x4D 0x4D 0xC1 0x57 0xB9 0x6C 0x52

Tab. B.5: Data v Test_decryption

kryptogram	0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
klíč	0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x87 0x25 0xDD 0xD9 0x43 0xC7 0x24 0xDD 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01
prostý text	0x99 0x4D 0x4D 0xC1 0x57 0xB9 0x6C 0x52

B.3 Blowfish

Tab. B.6: Data v Test_function_f

vstupní hodnota	0x886a3f24
výstupní hodnota	0xa8634c5c

Tab. B.7: Data v Test_encryption

prostý text	0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x11
klíč	0x12 0x34 0x56 0x78 0x91 0x23 0x45 0x67 0x89 0x12 0x34 0x56 0x78 0x91 0x23 0x45 0x67 0x89
kryptogram	0x37 0x5C 0xC1 0xC2 0xAA 0xB4 0x45 0xAC

Tab. B.8: Data v Test_decryption

kryptogram	0x11 0x11 0x11 0x11 0x11 0x11 0x11 0x11
klíč	0x12 0x34 0x56 0x78 0x91 0x23 0x45 0x67
prostý text	0xE2 0x4A 0x0B 0xFD 0xAB 0x09 0x88 0x61

B.4 SHA-3

Tab. B.9: Testovací řetězce SHA3(512)

zpráva	""
hash	"a69f73cca23a9ac5c8b567dc185a756e97c982164fe25859e0d1dcc1475c80 a615b2123af1f5f94c11e3e9402c3ac558f500199d95b6d3e301758586281dcd26"

zpráva	"abc"
hash	"b751850b1a57168a5693cd924b6b096e08f621827444f70d884f5d0240d27 12e10e116e9192af3c91a7ec57647e3934057340b4cf408d5a56592f8274eec53f0"

zpráva	"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklm nopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
hash	"afebb2ef542e6579c50cad06d2e578f9f8dd6881d7dc824d26360feebf18a4 fa73e3261122948efcfd492e74e82e2189ed0fb440d187f382270cb455f21dd185"

Tab. B.10: Testovací řetězce SHA3(384)

zpráva	""
hash	"0c63a75b845e4f7d01107d852e4c2485c51a50aaaa94fc61995e71bbee983a2 ac3713831264adb47fb6bd1e058d5f004"

zpráva	"abc"
hash	"ec01498288516fc926459f58e2c6ad8df9b473cb0fc08c2596da7cf0e49be4b 298d88cea927ac7f539f1edf228376d25"

zpráva	"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklm nopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
hash	"79407d3b5916b59c3e30b09822974791c313fb9ecc849e406f23592d04f625d c8c709b98b43b3852b337216179aa7fc7"

Tab. B.11: Testovací řetězce SHA3(256)

zpráva	""
hash	"a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a"

zpráva	"abc"
hash	"3a985da74fe225b2045c172d6bd390bd855f086e3e9d525b46bfe24511431532"

zpráva	"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklm nopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
hash	"916f6061fe879741ca6469b43971dfdb28b1a32dc36cb3254e812be27aad1d18"

Tab. B.12: Testovací řetězce SHA3(224)

zpráva	""
hash	"6b4e03423667dbb73b6e15454f0eb1abd4597f9a1b078e3f5b5a6bc7"

zpráva	"abc"
hash	"e642824c3f8cf24ad09234ee7d3c766fc9a3a5168d0c94ad73b46fdf"

zpráva	"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklm nopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
hash	"543e6868e1666c1a643630df77367ae5a62a85070a51c14cbf665cbc"

B.5 Tiger

Tab. B.13: Testovací řetězce Tiger

zpráva	""
hash	"3293ac630c13f0245f92bbb1766e16167a4e58492dde73f3"

zpráva	"The quick brown fox jumps over the lazy dog"
hash	"6d12a41e72e644f017b6f0e2f7b44c6285f06dd5d2c5b075"

zpráva	"Test vector from febooti.com"
hash	"382599758b759db703d4940c08c3393182adad7e9a7e590f"

C OBSAH PŘILOŽENÉHO CD

Na přiloženém CD je uložena tato práce v elektronické podobě a celá knihovna Aalg. Knihovna je rozčleněna na několik balíků, v balíku aalg.crypto se nacházející třídy pro zadané algoritmy, v balíku aalgTest jsou k dispozici testovací třídy.

Pro zpracování byl užit program Eclipse verze Luna Service Release 1 (4.4.1) s JDK 1.7 a Java Cryptography Extension (JCE) 7.