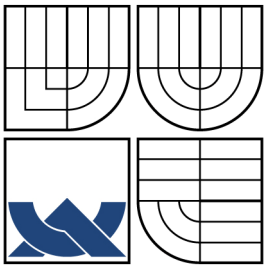
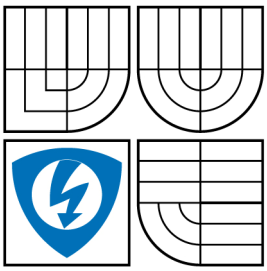


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

# IMPLEMENTACE A TESTOVÁNÍ HASHOVACÍHO ALGORITMU MD5

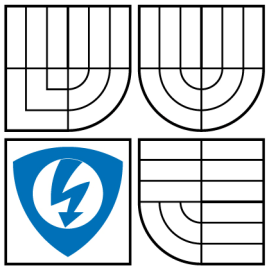
IMPLEMENTATION AND TESTING OF MD5 HASH ALGORITHM

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — MICHAL LEGEŇ
AUTHOR

VEDOUCÍ PRÁCE — ING. JAN MALÝ
SUPERVISOR

BRNO 2008



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**Fakulta elektrotechniky a komunikačních technologií**

**Ústav telekomunikací**

# Bakalářská práce

bakalářský studijní obor

**Teleinformatika**

***Student:*** Legeň Michal ***ID:*** 78474
***Ročník:*** 3 ***Akademický rok:*** 2007/2008

**NÁZEV TÉMATU:**

## Implementace a testování hashovacího algoritmu MD5

**POKYNY PRO VYPRACOVÁNÍ:**

Analyzujte hashovací algoritmus MD5 a implementujte ho v programovacím jazyce dle vlastního výběru (C/C++, pascal, Java). Na této implementaci dále proveďte simulace, které poukazují na slabá místa algoritmu a pokuste se navrhnout programové metody použitelné pro případný útok.

**DOPORUČENÁ LITERATURA:**

[1] Rivest, R.: The MD5 Message-Digest Algorithm, RFC 1321, 1992, dostupné z: http://www.faqs.org/rfcs/rfc1321.html
[2] Klíma, V.: Tunely v hašovacích funkcích: kolize MD5 do minuty, IACR ePrint archive Report 2006/105, 2006, dostupné z: http://cryptography.hyperlink.cz/2006/tunely.pdf

***Termín zadání:*** 11.2.2008 ***Termín odevzdání:*** 4.6.2008

***Vedoucí práce:*** Ing. Jan Malý

**prof. Ing. Kamil Vrba, CSc.**
*předseda oborové rady*

# LICENČNÍ SMLOUVA
# POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

**1. Pan/paní**

Jméno a příjmení: Michal Legeň

Bytem: Tajovského 1300, 95803, Partizánske

Narozen/a (datum a místo): 10.06.1986, Partizánske

(dále jen „autor“)

a

**2. Vysoké učení technické v Brně**

Fakulta elektrotechniky a komunikačních technologií

se sídlem Údolní 244/53, 602 00, Brno 2

jejímž jménem jedná na základě písemného pověření děkanem fakulty:

prof. Ing. Kamil Vrba, CSc.

(dále jen „nabyvatel“)

## Čl. 1
## Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - □ disertační práce
   - □ diplomová práce
   - □ bakalářská práce
   - □ jiná práce, jejíž druh je specifikován jako .....

   (dále jen VŠKP nebo dílo)

Název VŠKP: Implementace a testování hashovacího algoritmu MD5

Vedoucí/ školitel VŠKP: Ing. Jan Malý

Ústav: Ústav telekomunikací

Datum obhajoby VŠKP:

VŠKP odevzdal autor nabyvateli v*:

- □ tištěné formě – počet exemplářů 1
- □ elektronické formě – počet exemplářů 1

* hodící se zaškrtněte

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

## Článek 2
## Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - □ ihned po uzavření této smlouvy
   - □ 1 rok po uzavření této smlouvy
   - □ 3 roky po uzavření této smlouvy
   - □ 5 let po uzavření této smlouvy
   - □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

## Článek 3
## Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: …….

…….. ……

Nabyvatel Autor

# Abstract

Hash functions are one of basic structural components of modern cryptography. Their task is to create unique bit order for entry information called message digests. This way the inevitable integrity of information is secured because data transferring via different transfer media may be altered during the transfer. They provide a possibility of fast and effective encryption of passwords because of one-way as well.

The aim of this bachelor's thesis is to analyse one of the most well-known hash functions MD5 to implement it in programming language, to analyse its security risks and to try to use practical exhibitions to the benefit of an attacker. The bachelor's thesis is presenting theoretical and practical part on Implementation and testing of MD5 hash algorithm.

The introduction of the work is devoted to the basics of cryptography, to explanation of the concept of hash function and its basic attributes. The chapter 3 is focussed on the analysis of algorithm itself. The following part describes its implementation in programming language C++. The last two parts are about problems of security and weak sides of hash algorithm MD5. On the basic of existance of first order collisions, there is in practical demonstration presented creating two different programmes with the same message digest. The final part belongs to description of implementations of attacks on one-way algorithm MD5 and comparisons of their time possibilities during getting original information on the basis of their message digests.

**Keywords:** hash function MD5, collision, cryptography, message diggest

# Anotace

Hashovací funkce jsou jedním ze základních stavebních prvků moderní kryptografie. Jejich úkolem je pro vstupní zprávy vytvářet unikátní bitové posloupnosti, nazývané digitální otisky. Tím zabezpečují nezbytnou integritu zpráv, protože data přenášena přes různá přenosová média mohou být během přenosu pozměněna. Zároveň z důvodu jednosměrnosti poskytují možnost rychlého a účinného šifrování hesel.

Cílem této bakalářské práce je analyzovat jednu z nejznámějších hashovacích funkcí MD5, implementovat ji v libovolném programovacím jazyku, analyzovat její bezpečnostní rizika a pokusit se je na praktických ukázkách využít ve prospěch útočníka. Bakalářská práce prezentuje teoretickou a praktickou část na téma Implementace a testování hashovacího algoritmu MD5.

Úvod práce je věnován základům kryptografie, vysvětlení pojmu hashovací funkce a jejím základním vlastnostem. Kapitola č.3 je zaměřená na analýzu samotného algoritmu MD5. Následující část je věnovaná popisu její implementace v programovacím jazyku C++. Poslední dvě části jsou věnované problematice bezpečnosti a slabých míst hashovacího algoritmu MD5. Na základě existence kolizí prvního řádu je tu na praktické ukázce prezentován postup vytvoření dvou odlišných programů se stejným digitálním otiskem. Závěr patří popisu implementace útoků na jednosměrnost algoritmu MD5 a porovnání jejich časových možností při získávání původních zpráv na základě jejich digitálních otisků.

**Klíčová slova:** hashovací funkce MD5, kolize, kryptografie, digitální otisk

# Prohlášení

Prohlašuji, že svou bakalářskou práci na téma „Implementace a testování hashovacího algoritmu MD5“ 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 bakářske 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 jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne …..……...  ……...

(podpis autora)

# Poděkování

Děkuji vedoucímu bakalářské práce Ing. Janovi Malému, za velmi užitečnou metodickou pomoc a cenné rady při zpracování bakalářské práce.

V Brně dne …..……...  ……...

(podpis autora)

# Zoznam skratiek

*AES* - Advanced Encryption Standard

*DES* - Data Encryption Standard

*MD* - Message Diggest

*POV* - Point Of Verification

*RFC* - Request For Comment

*RSA* - Rivest, Shamir, Adleman

*SHA* - Secure Hash Algorithm

# Obsah

# Zoznam obrázkov



# Zoznam tabuliek

# 1 Úvod

Kryptografia je v dnešnej dobe dôležitou súčasťou každodenného života. Možno si to väčšina z nás ani neuvedomuje ale prichádza s ňou do styku skoro neustále. Umožňuje nám bezpečnú komunikáciu na internete, používa sa u finančných operácii, u digitálnych podpisoch alebo pri takých bežných veciach ako je prihlasovanie sa na počítač pod heslom. Kódovacie a šifrovacie algoritmy položili v minulosti základy pre bezpečnú komunikáciu pomocou kryptografických metód, ktoré sa dodnes zdokonalujú a vyvíjajú.

K jedným zo základných stavebných prvkov súčasnej kryptografie patria hashovacie funkcie. Sú to výpočetne efektívne funkcie, ktorých primárnou úlohou je vytvárať unikátnu bitovú postupnosť pre rozdielne vstupné správy, nazývanú digitálny otisk alebo hash. Tieto algoritmy sú navrhované tak, aby sa nedala získať pôvodná správa a taktiež aby bolo veľmi ťažké nájsť dve kolízne správy, čo sú správy ktorých digitálny otisk by bol rovnaký, hoci teoreticky je to možné.

Základnou úlohou v bakalárskej práci Implementace a testování hashovacího algoritmu MD5 je analyzovať jednu z najpoužívanejších a najznámejších hashovacích funkcií MD5. Začiatok práce je venovaný základom kryptografie, vlastnostiam hashovacích funkcií a podmienkam ktoré musia tieto funkcie spĺňať aby sme mohli ich bezpečne používať. V ďalšej kapitole je analyzovaný samotný algoritmus MD5. Štvrtá kapitola je venovaná popisu uskutočnenej implementácie hashovacej funkcie MD5, ktorá prakticky prezentuje získané teoretické poznatky ohľadne tohoto algoritmu. Ďalšia časť je zameraná na problematiku bezpečnosti hashovacej funkcie MD5. Stručne sú popísané metódy ako mnohonásobná modifikácia správ a metóda tunelovania, ktoré úspešne dokážu vygenerovať dve správy s rovnakým digitálnym otiskom. Následne je prezentovaná praktická ukážka zneužitelnosti existencie kolíznych správ na vytvorenie dvoch odlišných programov s rovnakým digitálnym otiskom. V poslednej kapitole je venovaná pozornosť na popis uskutočnených implementácií útokov na jednosmernosť hashovacej funkcie MD5, ako je útok hrubou silou a slovníkový útok s využitím databázového systému MySQL a porovnanie ich časových možností pri získavaní pôvodných správ pri znalosti ich digitálnych otiskov.

# 2 Úvod do kryptografie

Kryptológia je matematický vedný obor, ktorý sa zaoberá šifrovacími a kódovácími algoritmami.

*Delí sa na dve veľké skupiny:*

- Kryptografia – zaoberá sa návrhom šifrovacích algoritmov.
- Kryptoanalýza – snaží sa naopak šifrovacie algoritmy prelomiť.

Pojem kryptografia má svoj pôvod v gréckom slove „kryptos" a vyjadruje čo je jej cieľom, to znamená snaha o skrytie významu správy. Kryptografia nie je vynálezom poslednej doby. Už z dôb staroveku sú známe prvé pokusy o šifrovanie textu. Kryptografia sa stala veľmi obľúbenou vednou disciplínou používanou hlavne k vojenským a iným štátnickým účelom [7]. Kryptografia prechádzala vývojom celé storočia. Veľký skok vo vývoji spôsobila prvá a následne druhá svetová vojna. Ešte pred začiatkom druhej svetovej vojny sa krajiny snažili dosiahnuť čo najdokonalejšie spôsoby utajovania správ. Nemecká armáda vyvinula počas vojny asi najznámejší šifrovací stroj označovaný ako Enigma. S vývojom kryptografie postupoval aj vývoj kryptoanalýzy, keď znepriatelené strany usilovne pracovali na rozlúštení šifier druhej strany. Až do polovice sedemdesiatych rokov sa používali len tzv. symetrické metódy šifrovania. V roku 1976 Whitfield Diffie, Martin Hellman a Ralph Merkle publikovali článok o nových možnostiach kryptografie, a tak boli položené základy asymetrických kryptografických metód [7]. V roku 1978 vzniká asymetrický šifrovací algoritmus RSA, ktorý sa upravený používa dodnes. Dôležitým rokom v histórii kryptológie bol rok 1980, kedy sa konala prvá veľká konferencia venovaná tejto vednej disciplíne. Konferencia sa od tej doby koná dodnes a nesie názov „CRYPTO". V dnešnej dobe je kryptografia už nepostrádateľnou súčasťou každodenného života. Má svoje uplatnenie ako v profesionálnom tak i v súkromnom živote.

Hlavnou úlohou kryptografie je teda zaistenie dôvernosti chránených údajov. Nikto nepovolaný nesmie mať možnosť prečítať data, ktoré sú chránené kryptografickými prostriedkami a nie sú určené pre neho.

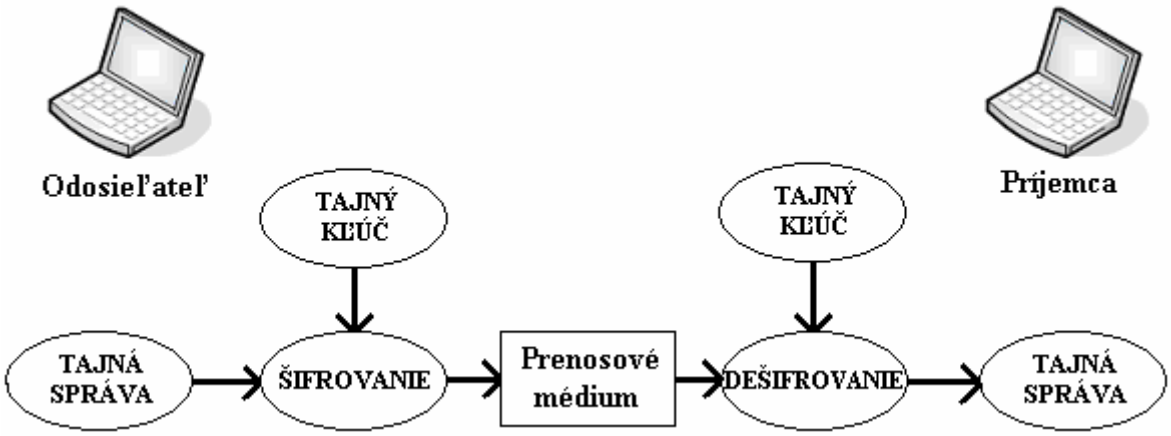
Kryptografické metódy obecne využívajú tzv. kľúč pomocou ktorého tajné data zakódujú a následne opäť rozkódujú.

*Podľa spôsobu práce s kľúčmi sa kryptografické metódy delia na:*

- Symetrické metódy kryptografie
- Asymetrické metódy kryptografie

## 2.1 Symetrické metódy kryptografie

V počiatkoch kryptografie boli šifrovacie algoritmy založené výlučne na symetrickom kľúči. Podstata je v tom pokiaľ chcú dvaja ľudia komunikovať, dohodnú si vhodnou cestou tajný kľúč [2]. Poznajú ho len oni dvaja a nikdy ho nikomu nesmú prezradiť. Vstupom je teda nejaká tajná správa a kľúč. Šifrovacia funkcia pomocou kľúča prevedie tajný text na kód, ktorý môže byť odoslaný adresátovi. Príjemca použije dešifrovaciu funkciu s rovnakým kľúčom a tým získa pôvodný tajný text. Dôležité je, že pre dešifrovanie musí mať príjemca k dispozícií rovnaký kľúč akým bol text zakódovaný. Z toho vyplýva, že je potrebné zaistiť bezpečný spôsob doručenia kľúča, tak aby sa nedostal do nepovolaných rúk.




Obr. 2.1: *Symetrické metódy kryptografie*

Pre šifrovanie sa používajú funkcie, u ktorých platí, že pri znalosti vstupného a zakódovaného textu je veľmi ťažké vygenerovať kľúč, napriek tomu že vlastné kódovanie a dekódovanie pomocou tohoto kľúča je pomerne rýchla záležitosť.

*Medzi najznámejšie symetrické metódy kryptografie patria napr. :*

- DES (Data Encryption Standard) – dĺžka kľúča 56 bitov.
- tripleDES – dĺžka kľúča 168 bitov.
- AES (Advanced Encryption Standard) – dĺžka kľúča 128, 168 alebo 256 bitov.

*Výhody symetrickej kryptografie:*

- Veľkou výhodou symetrických metod je ich rýchlosť.
- Kľúče symetrických šifrovacích systémov sú relatívne krátke.

*Nevýhody symetrickej kryptografie:*

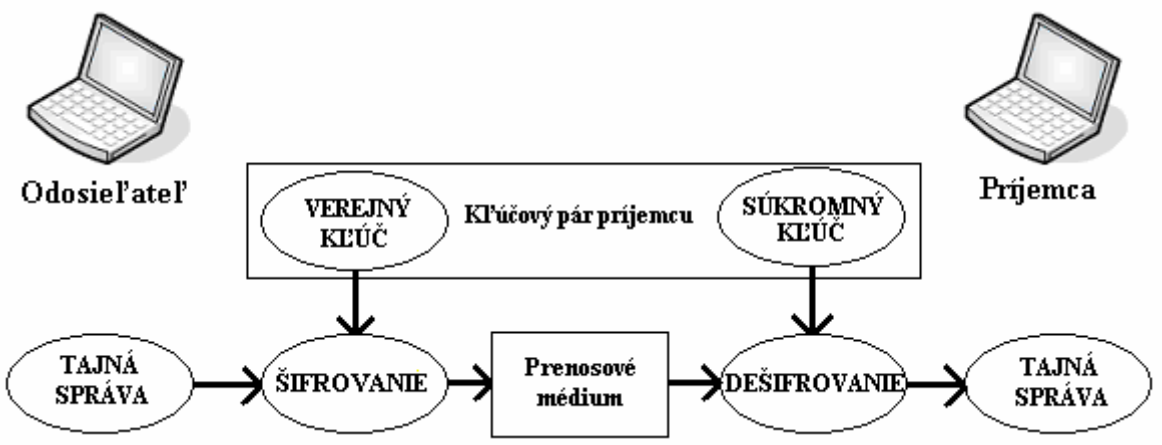
- Nevýhodou môže byť problém zabezpečenia kanálu ktorým si odosieľateľ a príjemca predávajú kľúč.
- Ďalšou nevýhodou je počet kľúčov [7]. Počet kľúčov *k* potrebných pre komunikáciu *n* osôb je

$$k = \frac{n*(n-1)}{2}. \tag{2.1}$$

S rastúcim počtom komunikujúcich strán tak počet kľúčov neúmerne rastie a tým aj náklady na ich správu. Pre ilustráciu, ak spolu komunikuje 50 ľudí je potrebných až 1225 kľúčov.

## 2.2 Asymetrické metódy kryptografie

Druhou skupinou kryptografických metód je tzv. asymetrická kryptografia. Jej základným znakom je existencia dvoch kľúčov pre každého užívateľa [2]. Dohromady sa obidva kľúče označujú ako kľúčový pár. Typickou vlastnosťou tohoto páru je fakt, že text ktorý sa zašifruje jedným kľúčom, je možné dešifrovať iba druhým kľúčom z rovnakého páru. Treba dodať, že text zašifrovaný jedným kľúčom je nemožné týmto kľúčom dešifrovať. Jeden z kľúčov sa označuje ako súkromný, druhý ako verejný. Dôležité je aby užívateľ uchovával súkromný kľúč v tajnosti. Verejný kľúč môže zverejniť.




Obr. 2.2: *Asymetrické metódy krytografie*

*Medzi najznámejšie asymetrické metódy kryptografie patria napr. :*

- RSA (iniciály autorov Rivest, Shamir, Adleman) – dĺžka kľúča 1024, 2048 bitov.
- El Gamal – využíva problém zložitosti výpočtu diskrétnych logaritmov v algebraickej štruktúre.

*Výhody asymetrickej kryptografie:*

- Hlavnou výhodou je fakt, že sa súkromný kľúč nemusí nikam posielať a tak nemôže dôjsť k jeho zneužitiu.
- Postačí menší počet kľúčov. Pri vzájomnej komunikácií niekoľkých užívateľov stačí jeden pár kľúčov pre každého užívateľa.

*Nevýhody asymetrickej kryptografie:*

- Asymetrické metódy sú podľa [9] až 1000 krát pomalšie ako metódy symetrické.
- Ďalšou nevýhodou je skutočnosť, že dĺžka kľúčov je omnoho väčšia ako u symetrických metód. V dnešnej dobe pracuje asymetrická kryptografia s 1024 alebo 2048 bitovými kľúčmi.

## 2.3 Hashovacie funkcie a ich vlastnosti

Hashovacia funkcia $h$ je výpočetne efektívna funkcia, ktorá musí spĺňať následujúce dve implementačné podmienky :

- Funkcia $h$ mení ľubovoľne dlhú konečnú bitovú postupnosť $x$ na digitálny otisk $h(x)$ presne definovanej bitovej dĺžky (spravidla 128, 160, 256 ,512 bitov).
- Pre zadanú funkciu $h$ a vstup $x$, výpočet hashu $h(x)$ musí byť dostatočne rýchly.

Druhou skupinou podmienok ktoré musí hashovacia funkcia spĺňať, aby sa dala bezpečne používať, sú podmienky kryptografické:

- Jednocestnosť
- Odolnosť voči kolíziam prvého radu
- Odolnosť voči kolíziam druhého radu

### 2.3.1 Podmienky kladené na hashovacie funkcie

- **Jednocestnosť**

Pre otisk *y* musí byť výpočetne nezvládnuteľné nájsť taký vstup *x*, aby platilo $h(x) = y$.

Táto podmienka hovorí, že hashovacia funkcia musí byť jednocestná (one-way). To znamená, že hashovacia funkcia nesmie byť dešifrovatelná, alebo aspoň nesmie byť výpočetne zvládnuteľné nájsť k výstupu teda digitálnemu otisku jeho vstup.

Pokiaľ by táto podmienka nebola splnená, je daná hashovacia funkcia absolútne nepoužiteľná, pretože útočník poznajúci otisk správy by mohol jeho analýzou získať pôvodnú správu.

- **Kolízie prvého radu**

Podmienka hovorí, že musí byť výpočetne nezvládnuteľné nájsť dva rôzne vstupy *x* a *y*, pre ktoré platí $h(x) = h(y)$.

Nesplnenie tejto podmienky za určitých podmienok nemusí vadiť. Závisí to od spôsobu použitia danej hashovacej funkcie. Je však potreba individuálne posúdiť či je toto použitie bezpečné alebo nie.

Pri splnení tejto podmienky sa hovorí, že hashovacia funkcia je odolná voči kolíziam prvého radu.

- **Kolízie druhého radu**

Pre vstup *x* musí byť výpočetne nezvládnuteľné nájsť iný vstup $y \neq x$, tak aby platilo $h(x) = h(y)$.

Pri nesplnení tejto podmienky je hashovací algoritmus nepoužiteľný. Útočník by mohol k správe ktorej hash získal, vytvoriť inú správu a vydávať ju za právu, práve na základe rovnakého digitálneho otisku oboch správ.

Pri splnení tejto podmienky sa hovorí, že daná funkcia je odolná voči kolíziám druhého radu.

Keď sa hovorí o kolíziach treba podotknúť, že správ je mnohonásobne viac ako digitálnych otiskov, kolízie preto musia teoreticky existovať [5]. V skutočnosti ich existuje obrovské množstvo. Z toho dôvodu sa hashovacie funkcie navrhujú tak, aby hľadanie kolízií bolo nad naše výpočetné schopnosti.

### 2.3.2 Narodeninový paradox

S nachádzaním kolízií prvého radu úzko súvisí narodeninový paradox, ktorý je základom pre posudzovanie bezpečnosti hashovacích funkcií. Hovorí o tom koľko správ by sa muselo hashovať, aby sa našla kolízia, teda dve rôzne správy s rovnakým digitálnym otiskom. Odpoveď sa skrýva v následujúcom tvrdení.

Je daná množina $A$ o $n$ rôznych prvkoch. Ak je $n$ dostatočne veľké a $k$ sa rovná približne hodnote $(2.n.\ln 2)^{0,5}$ potom v množine s $k$ prvkami, ktoré sa vyberajú z množiny $A$ náhodne, sa približne s 50% pravdepodobnosťou nájdu dva rovnaké prvky.

Narodeninovým paradoxom sa nazýva z dôvodu, že sa dá toto tvrdenie aplikovať na problém nájdenia dvoch ľudí s rovnakým dátumom narodenia. Pre $n$=365 postačí skupina náhodne vybraných 23 ľudí k tomu aby sa medzi nimi našla asi s 50% pravdepodobnosťou dvojica oslavujúca narodeniny v rovnaký deň v roku.

Podobne je to s hashovacími funkciami, kde $A$ je množina všetkých možných výsledkov danej hashovacej funkcie. Ak budeme vychádzať z hashovacej funkcie MD5 hashovací kód je 128-bitový, v tom prípade $n=2^{128}$ , a postačí hashovať $2^{64}$ správ aby sme s 50% pravdepodobnosťou našli kolíziu. Tento počet sa dá označiť za výpočetne nezvládnuteľný. Pokiaľ sa však dá povedať, že je možné nachádzať kolízie jednoduchšie ako pomocou narodeninového paradoxu, potom sa dá prehlásiť, že hashovacia funkcia je prelomená, pretože je výpočetne zvládnuteľné nájsť kolíziu prvého radu [3].

### 2.3.3 Náhodné orákulum

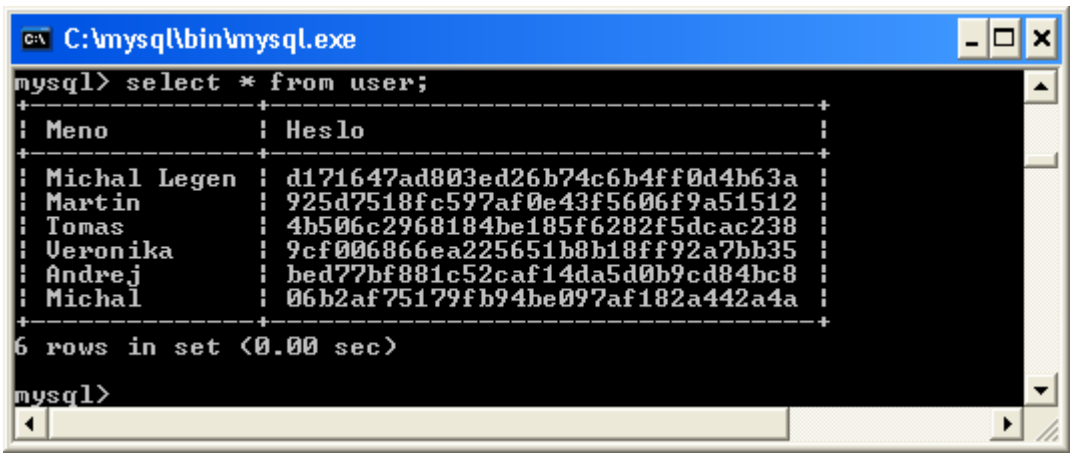
Náhodné orákulum je určitá matematická fikcia. Podstatou je, že pri zadaní vstupnej správy je výstupom náhodná hodnota. Toto priradenie si náhodné orákulum zapamätá. Pri zadaní iných správ vždy vygeneruje náhodnú hodnotu výstupu to znamená nezávisle na tom, aké hodnoty už vygeneroval. Pokiaľ však zadáme správu, pre ktorú náhodné orákulum už výstupnú správu generovalo, potom siahne do svojej pamäte a vygeneruje rovnakú hodnotu ako predtým. Ideálna hashovacia funkcia by mala mať matematicky zhodné vlastnosti aké má náhodné orákulum.

### 2.3.4 Použitie hashovacích funkcií

Hashovacie funkcie majú bohaté využitie v počítačovej bezpečnosti, ktoré vyplýva z ich typických vlastností.

- **Šifrovanie hesiel**

Jednocestnosť hashovacích funkcií umožnuje jednoduché ale účinne šifrovanie hesiel. Pretože je zbytočne riskantné ukladať heslá napr. do databáz ako čistý text, namiesto toho sa ukladajú vo forme ich digitálneho otisku. Z tohoto otisku vďaka jednocestnosti nie je možné určiť pôvodné heslo. Takto jednocestný hashovací algoritmus s minimálnym úsilím poskytuje lepšie zabezpečenie. Obrázok *2.3* zobrazuje typický príklad uloženia hesiel v podobe ich digitálneho otisku MD5 v databázi MySQL.




Obr. 2.3: *Uloženie hesiel v podobe ich digitálneho otisku*

- **Digitálny podpis**

V niektorých prípadoch je nutné zaistiť identifikáciu odosielateľa. Pretože v digitálnom svete nie je možné podpísať sa klasicky, použitím vlastnoručného podpisu, treba postupovať inak. Túto úlohu zaisťuje digitálny podpis. Je vytváraný pomocou súkromného kľúča asymetrickej šifrovacej metódy [2]. Jeho správnosť je overovaná verejným kľúčom ktorý k danému súkromnému kľúču patrí. Za povšimnutie stojí, že je to presne naopak ako u asymetrického šifrovania kde sa verejným kľúčom šifrovalo a súkromným dešifrovalo. Rozdiel vyplýva z odlišnej potreby, pretože pri podpisovaní je cieľom aby si každý mohol overiť platný podpis.

Podstatným problémom asymetrickej kryptografie je veľká náročnosť na výpočetnú kapacitu. Keďže sa pri digitálnom podpise predpokladá podpisovanie pomerne veľkých objemov dát, bolo nutné tento problém nejako vyriešiť.

Ideálnym riešením je použitie niektorej z hashovacích funkcií, ktoré aj z veľkých objemov dat vytvoria digitálny otisk o presne definovanej dĺžke. Výhodou digitálneho otisku je jeho veľmi malá veľkosť. Teraz sa asymetrická kryptografia nemusí zaoberať šifrovaním veľkých megabajtových súborov, stačí ak zašifruje rádovo stovky bitov dlhý digitálny otisk. To je možné práve z dôvodu bezkolíznosti hashovacej funkcie, ktorá zaisťuje, že nie je možné nájsť dva dokumenty s rovnakým digitálnym otiskom.

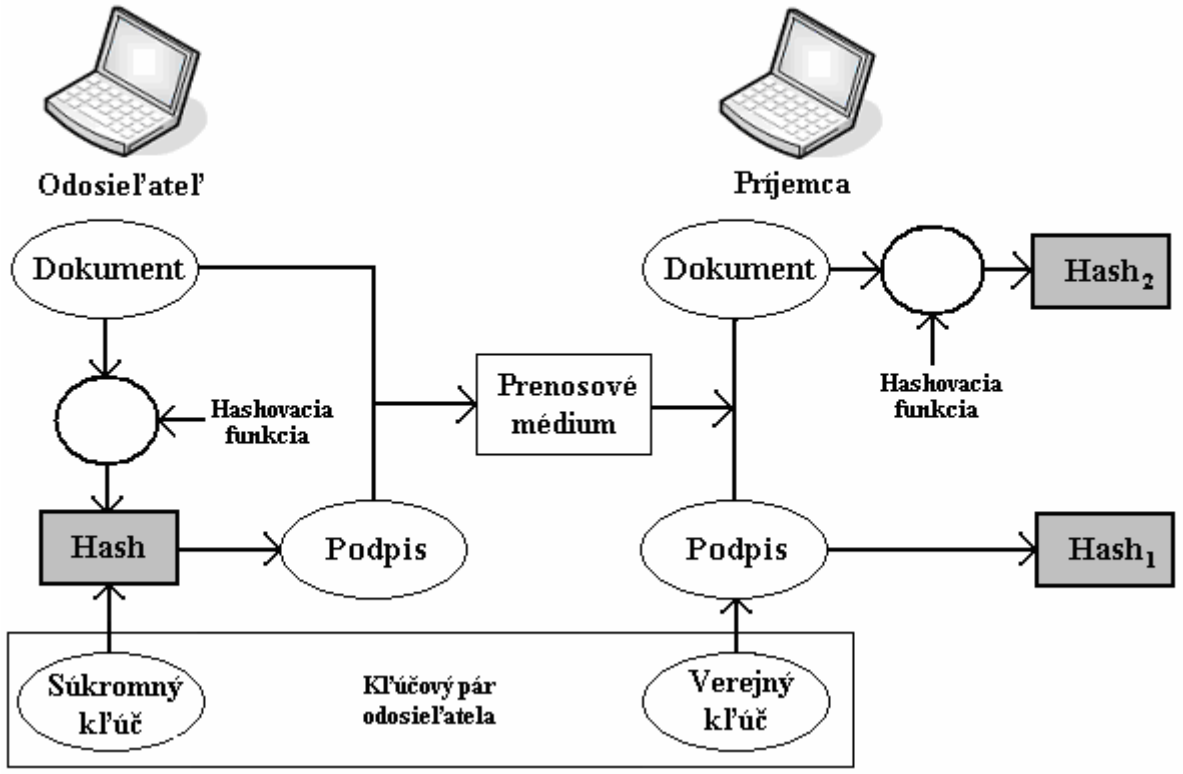
Zároveň tieto funkcie umožnujú dôležitú funkciu zvanú integritu dát. Pokiaľ chceme zaistiť bezpečný prenos dat, napríklad ochranu voči modifikácií správy útočníkom alebo prípadnej strate, potrebujeme na strane príjemcu nejako tieto dáta overiť. Najlepším riešením je vytvoriť na strane odosielateľa digitálny otisk správy, ktorý by po kontrole na strane príjemcu pomocou rovnakej hashovacej funkcie dával rovnaký výsledok.

Obrázok *2.4* zobrazuje priebeh vytvárania digitálneho podpisu a overovania digitálneho podpisu. Najprv sa pomocou hashovacej funkcie vypočíta zo vstupného textu digitálny otisk. Tento otisk sa následne zašifruje súkromným kľúčom, a tým získame podpis. Ten sa priloží k vstupnému textu a pošle príjemcovi.

Príjemca najprv urobí to isté. Vypočíta pomocou hashovacej funkcie digitálny otisk správy ktorú obdržal. Následne rozšifruje prijatý digitálny podpis verejným kľúčom odosielateľa a obe hodnoty digitálnych otiskov porovná.

Pokiaľ sa obidve hodnoty rovnajú, všetko je v poriadku. V prípade, že príjemca dospel k dvom rozdielnym digitálnym otiskom tak podľa [2] nastala jedna z týchto situácií:

- Správu niekto po ceste modifikoval.
- Správa bola podpísaná súkromným kľúčom, ktorý nepatrí verejnému kľúču, ktorý použil príjemca.
- Digitálny podpis si niekto vymyslel.




Obr. 2.4: *Princíp digitálneho podpisu*

# 3 Hashovacia funkcia MD5

Hashovací algoritmus MD5 bol vynájdený v roku 1992 Ronom Rivestom na univerzite MIT (Massachusetts Institute of Technology). Definovaný je v RFC 1321 [10]. MD5 je vylepšenou verziou svojho predchodcu algoritmu MD4, ktorý bol v tom čase síce veľmi rýchlym algoritmom ale jeho bezpečnosť bola narušená úspešnými kryptografickými útokmi. Pri tvorbe MD5 nebol kladený taký veľký dôraz na rýchlosť výpočtu ale na bezpečnosť.

Hashovacia funkcia MD5 je definovaná pomocou kompresnej funkcie *f* a inicializačnej hodnoty $H_o$, ktorá je tvorená štyrmi 32-bitovými registrami *A*, *B*, *C*, *D*.

Hashovacia funkcia MD5 spracováva správu ľubovoľnej dĺžky. Predpokladajme teda, že máme vstupnú správu dlhú *n* bitov. Kde *n* je:

- Ľubovoľne veľké kladné číslo.
- Môže byť aj nulové.
- Nemusí byť násobkom ôsmich.

## 3.1 Pridanie zarovnávacích bitov

Správa dlhá *n* bitov sa doplní tzv. „padding bitami" (zarovnávacie bity), tak aby platilo:

$$(n + \text{počet zarovnávacích bitov}) \text{ MOD } 512 = 448, \qquad (3.1)$$

kde MOD znamená operáciu modulo. Inými slovami povedané, dĺžka po pridaní zarovnávacích bitov musí byť presne o 64 bitov menšia ako násobok 512. Pripojenie týchto zarovnávacich sa uskutočnuje vždy, dokonca aj v prípade že dĺžka pôvodnej správy *n* bez zarovnávacích bitov spĺňa podmienku *3.1*.

Pridávanie zarovnávacich bitov sa u moderných hashovacích funkcií uskutočnuje následovne. Najprv sa k správe pridá jeden bit 1. Následne sa pridávajú už len samé nuly, tak aby počet bitov doplnenej správy spĺňal vyššie uvedenú podmienku. V podstate to znamená, že vždy sa najmenej pridá jeden bit a najviac sa pridá 512 bitov.

Zarovnanie musí byť také aby umožnovalo jednoznačné odtrhnutie, inak by vznikali jednoduché kolízie. Napríklad keby sa doplnenie správy uskutočnilo samými nulovými bitmi, by sa nedalo rozoznať, koľko bitov bolo doplnených a či niektoré nie sú platnými bitmi správy, pokiaľ by správa nulovými bitmi končila [3].

## 3.2 Pridanie dĺžky

Po doplnení zarovnávacími bitmi má správa dĺžku presne o 64 bitov menšiu ako je násobok 512 bitov. Zvyšných 64 bitov sa vyplní 64-bitovým vyjadrením čísla *n*, to znamená počtu bitov pôvodnej správy bez zarovnávacích bitov. Dôležité je poznamenať, že týchto 64 bitov sa pridáva vo forme dvoch 32 bitových slov s low-order poradím bajtov. To znamená, že najmenej významný bajt bude ako prvý a najvýznamnejší bajt ako posledný. Doplnenie dĺžky pôvodnej správy sa nazýva Damgard-Merklovo zosilnenie, ktoré bolo nezávisle navrhnuté oboma autormi na konferencii CRYPTO 1989 [3].

Je veľmi nepravdepodobné, že by vstupná správa bola taká dlhá že by 64 bitov nestačilo na jej vyjadrenie. Presnejšie musela by byť dlhšia ako 2^64 bitov. V prípade ale, že by taká situácia nastala, použije sa dolných 64 bitov tohoto vyjadrenia [10]. Po pridaní zarovnávacích bitov a čísla *n* (dĺžka pôvodnej správy) je dĺžka správy presným násobkom 512.

- **Príklad pridávania zarovnávacích bitov a dĺžky**

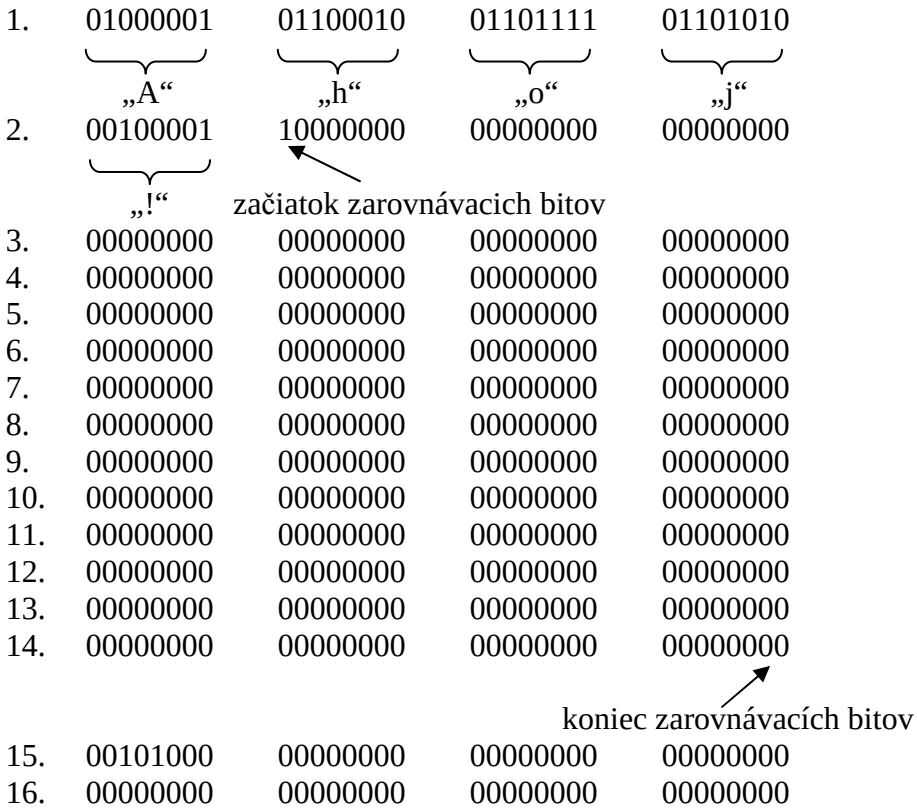
Pre ilustráciu a lepšie pochopenie tu bude uvedený názorný príklad pridávania zarovnávacích bitov a pridania dĺžky pre vstupnú správu „Ahoj!“. Uvažujeme 8-bitové ASCII kódovanie, v ktorom majú jednotlivé znaky vyjadrenia zobrazené v tabuľke *3.1*.

| ZNAK | DEC | BIN |
|---|---|---|
| A | 65 | 01000001 |
| h | 104 | 01100010 |
| o | 111 | 01101111 |
| j | 106 | 01101010 |
| ! | 33 | 00100001 |

Tab. 3.1: *ASCII vyjadrenie správy „Ahoj!“*

Dĺžka správy je *n* = 8 bitov * 5 znakov = 40 bitov. Pre splnenie rovnice *3.1* je treba k správe pridať 408 zarovnávacích bitov. Z toho prvý bit bude 1 a zvyšných 407 budú bity nulové.

Na následujúcich riadkoch je zobrazená celá 512-bitová správa.




Správa „Ahoj!“ zaberá prvých 5 bajtov. Ďalej nasledujú zarovnávacie bajty, ktoré začínajú šiestym bajtom. V riadkach 15 a 16 sú bajty vyjadrujúce dĺžku pôvodnej správy v low-order poradí.

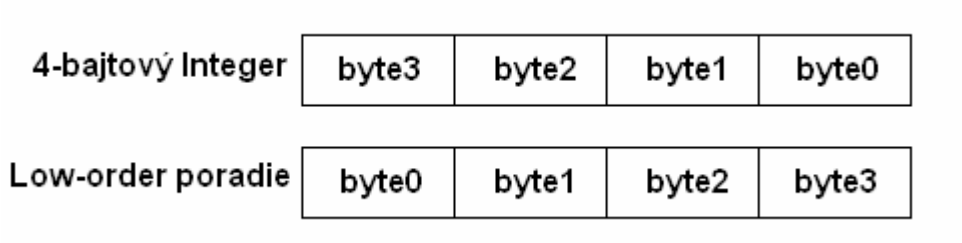
## 3.3 Inicializácia kontextu

Algoritmus MD5 pracuje s tzv. kontextom *H*. Kontext je realizovaný štyrmi 32 bitovými registrami označovanými ako *A*, *B*, *C*, *D*. Na začiatku je do týchto registrov uložený inicializačný vektor algoritmu MD5 vyjadrený v hexadecimálnom tvare:

- Register A = 0x67452301.
- Register B = 0xefcdab89.
- Register C = 0x98badcfe.
- Register D = 0x10325476.

## 3.4 Spracovanie správy po blokoch

MD5 algoritmus spracováva správu po blokoch o veľkosti presne 512 bitov. Každých následujúcich 512 bitov sa vyjadrí ako šestnásť 32-bitovych slov a uložia sa do postupnosti *X*[0,1,2,...,15]. Slovo je v tomto prípade chápané ako 4-bajtové číslo s poradím low-order. To znamená, že najmenej významný bajt bude prvý a najvýznamnejší bajt bude posledný. Príklad low-order poradia bajtov je zobrazený na obrázku *3.1*.




Obr. 3.1: *Low-order poradie bajtov*

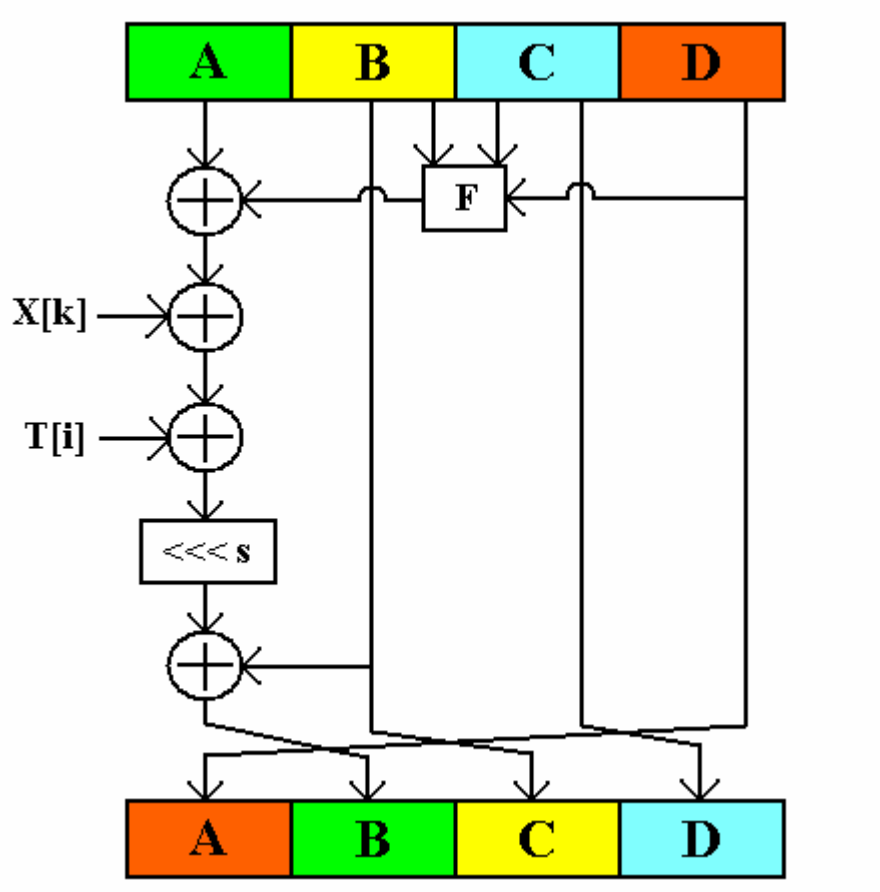
Týchto 16 slov spolu s kontextom *H* následne spracuje kompresná funkcia na nový kontext *H*. Z toho je vidieť, že názov kompresná funkcia nie je náhodný, pretože funkcia spracováva širší vstup, tvorený 512 bitovým blokom a kontextom, na omnoho kratší výstup v podobe nového kontextu. Kompresná funkcia je veľmi zložitá, aby zaistila miešanie bitov správy s kontextom a jednocestnosť. Kontext tak postupne vstrebáva jednotlivé bity a bloky správy a ukladá ich v sebe zložitým spôsobom [5]. Celkom sa nad každým 512-bitovým blokom uskutoční 64 kôl. V každom kole dochádza k strate informácie zo správy a súčasne k jej zložitému včleneniu do kontextu.

V každom z týchto kôl je nad kontextom vykonávaná určitá funkcia tvorená bitovými operáciami. Táto operácia sa mení po 16 kolách, taktiež každých 16 kôl sa mení postupnosť *X*[0,1,2,...,15] na inú permutáciu. Prehľad operácii vykonávaných nad kontextom je znázornený v tabuľke *3.2*.

| Označenie funkcie | Použitie funkcie v kolách | Vykonávaná operácia |
|---|---|---|
| F(X,Y,Z) | 0-15 | XY v not(X) Z |
| G(X,Y,Z) | 16-31 | XZ v Y not(Z) |
| H(X,Y,Z) | 32-47 | X xor Y xor Z |
| I(X,Y,Z) | 48-63 | Y xor (X v not(Z)) |

Tab. 3.2: *Operácie algoritmu MD5 v jednotlivých kolách*

Následne sa k výsledku tejto operácie pripočíta aktuálne 32-bitové slovo správy *X*[*k*] pre dané kolo, ďalej je pripočítaná určitá konštanta *T*[*i*] pre dané kolo a uskutoční sa bitový posun doľava o príslušný počet bitov. Výsledný kontext je vstupom pre ďalšie kolo. Priebeh spracovania kontextu v jednom kole je znázornený na obrázku *3.2*.




Obr. 3.2: *Priebeh spracovania kontextu v jednom kole algoritmu MD5*

Ešte treba povedať čo je to *T*[*i*], kde *i* = 1, 2,...,64. Jedná sa o 64 prvkovú postupnosť, ktorej hodnoty sú vypočítané z funkcie sínus. Ak *T*[*i*] je i-tý prvok tejto postupnosti tak jeho hodnota sa rovná :

$$T[i] = 4294967296 * |(\sin(i)|, \qquad (3.2)$$

kde *i* je v radiánoch a *T*[*i*] je celočíselná hodnota [10].

Po poslednom kole je k výstupnému kontextu pripočítaný pôvodný kontext pred začatím spracovania daného bloku. Tento výsledok sa následne považuje za inicializačný vektor ďalšieho 512-bitového bloku, v prípade že nejaký existuje.

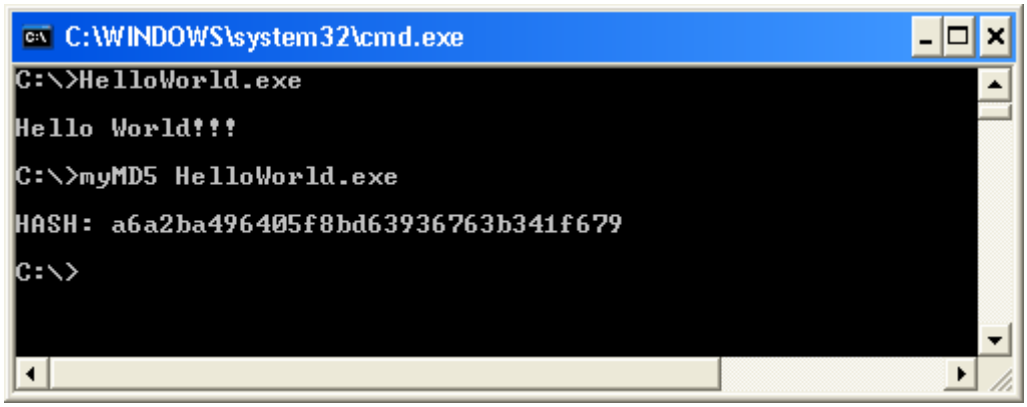
## 3.5 Výstup hashovacej funkcie MD5

Po spracovaní posledného 512-bitového bloku je výsledný kontext *H*, čiže registre *A*, *B*, *C*, *D* hashom správy. Spolu 128 bitov je vo väčšine prípadov vyjadrených v hexadecimálnom tvare v presne definovanom poradí. Začína sa najnižším bajtom registra *A* a končí najvyšším bajtom registra *D*.

# 4 Implementácia hashovacej funkcie MD5

V tejto kapitole bude podaný stručný popis implementácie hashovacej funkcie MD5. Pre implementáciu bol zvolený programovací jazyk C++. Programovanie prebiehalo v programovacom prostredí Dev-C++, v ktorom bol tento program ale aj ostatné programy v rámci práce skompilovaný.

Program je písaný ako konzolová aplikácia, ktorej vstupným parametrom je názov súboru ktorý sa bude hashovať. Výstup programu pri hashovaní jednoduchého programu „HelloWorld.exe“ je zobrazený na obrázku *4.1*.




Obr. 4.1: *Výstup programu*

## 4.1 Popis funkcií

Pri implementácií sa pre manipuláciu s jednotlivými 32-bitovými slovami využívali premenné typu *unsigned int*, ktoré majú veľkosť 4 bajty. Pomocou príkazu *typedef* sa vytvorila skratka *UINT*:

```
typedef unsigned int UINT;
```

Nasledujú prototypy použitých funkcií:

```
UINT F (UINT x, UINT y, UINT z);
UINT G (UINT x, UINT y, UINT z);
UINT H (UINT x, UINT y, UINT z);
UINT I (UINT x, UINT y, UINT z);
UINT KOLO(UINT a, UINT b, UINT c, UINT d, UINT x, int s, UINT T, int
&oj);
UINT TURN (UINT x);
void MD5 (unsigned char buffer [64], int i);
void ONEBLOCK (UINT x [16]);
```

Funkcie ktoré sú vykonávané nad kontextom viď. tabuľka *3.2* sú definované:

```
UINT F (UINT x, UINT y, UINT z)
{
   return (((x & y)) | (~x & z));
}
UINT G (UINT x, UINT y, UINT z)
{
   return (((x & z)) | (y & ~z));
}
UINT H (UINT x, UINT y, UINT z)
{
   return (x ^ y ^ z);
}
UINT I (UINT x, UINT y, UINT z)
{
   return (y ^ (x | ~z));
}
```

Ďalšou dôležitou funkciou je funkcia *ONEBLOCK*. To je funkcia, ktorá ako vstupný paramater príjma 512 bitový blok rozdelených do 16 slov s ktorými vykoná 64 operácií. Na začiatku funkcie sú do pomocných premenných *aa,bb,cc,dd* uložené hodnoty aktuálneho kontextu *a,b,c,d* pred spracovaním daného 512 bitového bloku. Kontext je deklarovaný globálne to znamená, že jeho hodnota je platná v každej funkcii a nie je teda potreba ho predávať parametrom funkcie. Je vidieť, že na konci funkcie dochádza k spočítaniu pôvodného kontextu a výstupného kontextu po spracovaní daného bloku.

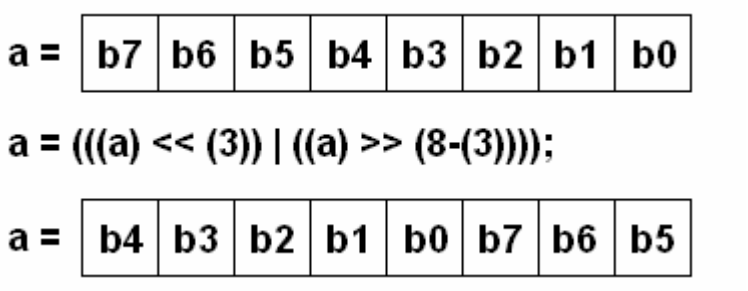
```
void ONEBLOCK (UINT x [16])
{
   int j = 0;
   UINT aa = a;
   UINT bb = b;
   UINT cc = c;
   UINT dd = d;
   a = KOLO(a, b, c, d, x[0], S11, 0xd76aa478,j); //0
   d = KOLO(d, a, b, c, x[1], S12, 0xe8c7b756,j); //1
   c = KOLO(c, d, a, b, x[2], S13, 0x242070db,j); //2
   .
   .
   .
   d = KOLO(d, a, b, c, x[11], S42, 0xbd3af235,j); //61
   c = KOLO(c, d, a, b, x[2], S43, 0x2ad7d2bb,j); //62
   b = KOLO(b, c, d, a, x[9], S44, 0xeb86d391,j); //63
   a+=aa;
   b+=bb;
   c+=cc;
   d+=dd;
}
```

Funkcia *ONEBLOCK* 64-krát volá funkciu *KOLO*.

```
UINT KOLO(UINT a, UINT b, UINT c, UINT d, UINT x, int s, UINT T, int &oj)
 {
   if ((0<=oj) && (oj<= 15)) a += F(b, c, d) + x + T;
   if ((16<=oj) && (oj<= 31)) a += G(b, c, d) + x + T;
   if ((32<=oj) && (oj<= 47)) a += H(b, c, d) + x + T;
```

```
    if ((48<=oj) && (oj<= 63)) a += I(b, c, d) + x + T;
    a = (((a) << (s)) | ((a) >> (32-(s))));
    oj++;
    return a + b;
}
```

Kde *a,b,c,d* sú registre kontextu, *x* je aktuálne 32-bitové slovo, parameter *s* určuje počet pozícií o koľko dôjde k bitovému posunu doľava, a *T* je jedinečná konštanta pre dané kolo. Parameter *oj* určuje ktorá operácia sa prevedie nad príslušným kontextom. Je predávaný vo forme odkazu to znamená, že jeho hodnota ostáva rovnaká aj mimo funkcie. Bitový posun doľava v tomto prípade neprebieha klasicky, to znamená nahradením bitových pozícii zprava nulami ale nahradia sa bitami ktoré z dôvodu posunu doľava z 32-bitového rozsahu vypadnú. Dochádza tak k rotácii 32-bitového čísla o *s* pozícii smerom doľava. Príklad rotácie 8-bitového čísla o 3 pozície zobrazuje obrázok *4.2*.




Obr. 4.2: *Príklad rotácie 8-bitového čísla o 3 pozície*

## 4.2 Hlavná časť programu

Hlavná časť programu začína otvorením súboru, ktorý chceme hashovať, pre čítanie v binárnom režíme.

```
int main(int argc,char *argv[])
{
   FILE *f;
   int i=0;
   unsigned char buffer [64];
   f = fopen(argv[1], "rb");
   while ((i = fread(buffer, 1, sizeof(buffer), f)) >= 0)
   {
         if (koniec) break;
         MD5(buffer,i);
   }
```

Do pola buffer sa vždy uloží 512 bitov a predajú sa funkcii *MD5*. Táto funkcia sa tu z dôvodu veľkosti neuvádza. Jej úlohou je deliť vstupné 512 bitové bloky do 16 slov, zabezpečovať ich low-order poradie, pridávať zarovnávacie bity a dopĺnať dĺžku pôvodnej správy. Každých 16 slov následne predáva funkcii *ONEBLOCK*.

## 4.3 Záver programu

Po spracovaní posledného 512 bloku je v registroch *a,b,c,d* uložený výsledok. Pre zobrazenie v správnom poradí bajtov a ich vyjadrenie v hexadecimálnom tvare je každý z registrov predaný funkcii *TURN*:

```
a = TURN(a);
b = TURN(b);
c = TURN(c);
d = TURN(d);
```

Funkcia *TURN* zabezpečí prehodenie prvého a posledného, a následne aj druhého a tretieho bajtu registra, a tým zabezpečí požadované poradie bajtov. Využíva pri tom nulovanie nepotrebných bitov pomocou bitovej operácie AND a následný posun bitov na požadované miesto v registri.

```
UINT TURN (UINT x)
{
  UINT pom = x;
  UINT nul[4]={0x000000ff,0x0000ff00,0x00ff0000,0xff000000};

  x = ((pom & nul[0]) << 24);
  x += ((pom & nul[1]) << 8);
  x += ((pom & nul[2]) >> 8);
  x += ((pom & nul[3]) >> 24);
  return x;
}
```

# 5 Kolízie prvého radu u MD5

V následujúcej kapitole bude podaný stručný úvod do problematiky slabých miest hashovacej funkcie MD5. MD5 tak ako mnohí jej predchodcovia nie je odolná voči kolíziám prvého radu.

## 5.1 Princíp hľadania kolízie prvého radu

Princíp útoku pochádza z roku 2004, kedy na konferencii CRYPTO 2004 Xiaoyun Wangová a kolektív predložili dve kolidujúce správy. Metóda pomocou ktorej sa k týmto správam dostali však publikovaná nebola. Zverejnená bola v roku 2005 spolu s postačujúcimi podmienkami zaručujúce kolíziu. Neskôr sa ukázalo, že tieto podmienky neboli úplne správne a bola predložená úplná sada nových podmienok, ktorá je pravdepodobne správna a konečná [6]. Popísaný bude základný postup hľadania kolízií.

Dve kolidujúce správy sa skladajú z dvoch 512-bitových blokov ($M_1$, $N_1$) a ($M_2$, $N_2$), kde $M_1$, $M_2$ sú prvé 512-bitové bloky a $N_1$, $N_2$ sú druhé 512-bitové bloky oboch kolidujúcich správ. Pričom aby boli kolidujúce musí platiť:

$$h(M_1, N_1) = h(M_2, N_2), \tag{5.1}$$

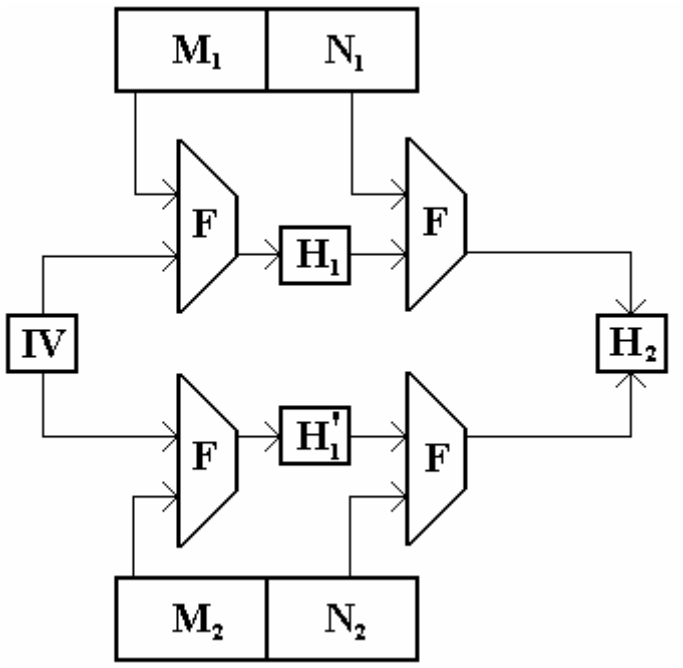
kde $h$ je hashovacia funkcia, v tomto konkrétnom prípade MD5.

Postup spracovania bloku je podobný u oboch blokov, popísaný bude postup u prvého. Blok $M$ má 512 bitov a je spracovávaný po 32-bitových slovách v 64 krokoch kompresnej funkcie. V prvom kroku vzniká medzipremenná $Q[1]$, v druhom $Q[2]$ a tak ďalej až $Q[64]$. Premenné $Q[-3, -2, -1, 0]$ sú inicializačným vektorom. Po 64 krokoch je k premenným $Q[61, 62, 63, 64]$ pripočítaná hodnota inicializačného vektoru a dostávame výsledok spracovania prvého bloku $IHV[0, 1, 2, 3]$. $IHV$ následne vstupuje do druhého bloku $N$ v podobe inicializačného vektora a postup sa opakuje.

Na premenné $Q[0, 1, 2, \ldots, 64]$ *a* $IHV[0, 1, 2, 3]$ sú kladené tzv. postačujúce podmienky. Tie určujú, že niektoré bity týchto premenných musia byť rovnaké, niektoré rôzne, iné musia byť nuly a jedničky a tie zostávajúce môžu byť ľubovoľné. V prvom bloku $M_1$ a v druhom bloku $N_1$ sú postačujúce podmienky rôzne. V prvom bloku ich je viac, lebo sa týkajú premenných $Q$ aj $IHV$. V prípade, že správa ($M_1$, $N_1$) dané postačujúce podmienky spĺňa, tak potom pre správu ($M_2$, $N_2$) platí :

$$(M_2, N_2) = (M_1, N_1) + C, \tag{5.2}$$

kde *C* je predom definovaný konštantný vektor ktorý má len šesť bitov nenulových [6]. Výsledný otisk takto vzniknutých správ je rovnaký. Celý princíp útoku zobrazuje obrázok *5.1*. Za pozornosť stojí predovšetkým fakt, že po spracovaní prvých 512-bitových blokov oboch kolidujúcich správ kompresnou funkciou *F* sa ich digitálne otisky líšia. V schéme na obrázku *5.1* sú označené $H_1$ a $H'_1$. Pri spracovaní druhých blokov sa tento rozdiel vyrovná a výsledný digitálny otisk $H_2$ je rovnaký u oboch správ.




Obr. 5.1: *Princíp útoku na MD5*

Nevýhodou postačujúcich podmienok je v tom, že ich je veľmi veľa, zhruba 250, a ako uvádza [6] zasahujú príliš ďaleko do premenných *Q*. Metóda mnohonásobnej modifikácie správ spočívala v tom, že sa zvolila náhodná správa a jej modifikáciou sa postupne spĺňali podmienky na *Q*[0, 1, 2, ... , 64]. Ako uvádza [6] tento proces končil vo väčšine prípadov najprv na *Q*[18], potom na *Q*[19], najďalej sa bolo možné dostať k splneniu podmienky na *Q*[24]. Tento bod nazývame bodom verifikácie POV (Point Of Verification), pretože v tomto bode ostávalo len overiť, či zostávajúce podmienky na *Q*[25, 26, 27,..., 64] a *IHV*[0, 1, 2, 3] sú splnené náhodne. MD5 má týchto zostávajúcich podmienok 29, preto zložitosť metódy mnohonásobnej modifikácie správ spočíva v nájdení $2^{29}$ bodov POV. Rýchlosť hľadania kolízie teda závisí od počtu podmienok, ktoré sú splnené len náhodne.

## 5.2 Tunely v hashovacej funkcií MD5

Metóda tunelovania začína na rozdiel od mnohonásobnej modifikácie správ v bode POV [6]. Niekoľkými tunelmi sa z jedného bodu POV geometrickou radou postupne vytvorí dostatočné množstvo ďalších bodov POV. Použitím tunelu o sile *n* vytvoríme z jedného bodu POV $2^n$ nových bodov POV. Existujú tunely rôznych typov, ktoré sa dajú medzi sebou kombinovať, takže z pôvodného bodu POV sa dá jedným tunelom o sile $n_1$ vytvoriť $2^{n1}$ bodov POV a z každého z nich iným tunelom o sile $n_2$ získať ďalších $2^{n2}$ bodov, spolu teda $2^{n1+n2}$ bodov POV. V práci [6] je ukázaných niekoľko tunelov, ktoré spojením dávajú tunel o sile 24. To znamená, že každý originálny bod POV je možné rozmnožiť na $2^{24}$ nových bodov POV. Tým pádom stačí vygenerovať $2^5$ POV oproti $2^{29}$ bodom v prípade najúspešnejšej metódy mnohonásobnej modifikácie správ u hashovacej funkcie MD5. V ideálnom prípade však stačí jeden bod POV. Trik spočíva tiež v tom, že jeden výchozí bod POV nemusíme získavať metódou mnohonásobnej modifikácie správ ale napríklad náhodne s minimálnou zložitostou. Úplne tak odpadá fáza prípravy bodov POV. Ďalej je možné z diferenčného schématu odstrániť prebytočné extra podmienky, ktoré si k existujúcim postačujúcim podmienkam pridáva každá metóda mnohonásobnej modifikácie správ pre zvýšenie svojej účinnosti. Ako uvádza [6], metóda tunelovania umožnuje do značnej mieri nahradiť metódy mnohonásobnej modifikácie správ a s využitím tunelov sa skracuje čas hľadania kolízií hashovacej funkcie MD5 z pôvodných osem hodín na minútu na bežných počítačoch.

- **Popis tunelov**

Podobne ako v predchádzajúcej kapitole sa bude predpokladať, že je daný 512-bitový blok *M*, ktorý je zpracovávaný po 32-bitových slovách $M=(x[0], ..., x[15])$ v 64 krokoch hashovacej funkcie MD5. V týchto kolaćh vznikajú medzipremenné $Q[1]$ až $Q[64]$. Pre lepšie pochopenie významu týchto medzipremenných v obrázku *3.2* sú to premenné ukladané do registra *B* podľa vztahu:

$$B = B + \mathrm{RL}\,(\,F(B,C,D) + A + x[k] + T[i],\, s\,), \qquad (5.3)$$

kde funkcia *F* je jedna zo štyroch funkcií definovaných v tabuľke *3.2* a $\mathrm{RL}(x, s)$ značí bitovú rotáciu slova *x* o *s* pozícií smerom doľava.

- **Tunel Q9**

Z rovníc pre *Q*[11] a *Q*[12] na obrázku *5.2* je vidieť, že prípadná zmena na i-tom bite by sa nemusela v týchto rovniciach vôbec prejaviť, pokiaľ bude i-tý bit *Q*[10] nula a i-tý bit *Q*[11] jednička, to znamená $Q[10]_i = 0$ a $Q[11]_i = 1$.

```
Q[ 7] = Q[ 6] + RL( F(Q[ 6], Q[ 5], Q[ 4]) + Q[ 3] + x[ 6] + 0xA8304613, 17);
Q[ 8] = Q[ 7] + RL( F(Q[ 7], Q[ 6], Q[ 5]) + Q[ 4] + x[ 7] + 0xFD469501, 22);
Q[ 9] = Q[ 8] + RL( F(Q[ 8], Q[ 7], Q[ 6]) + Q[ 5] + x[ 8] + 0x698098D8,  7);
Q[10] = Q[ 9] + RL( F(Q[ 9], Q[ 8], Q[ 7]) + Q[ 6] + x[ 9] + 0x8B44F7AF, 12);
Q[11] = Q[10] + RL( F(Q[10], Q[ 9], Q[ 8]) + Q[ 7] + x[10] + 0xFFFF5BB1, 17);
Q[12] = Q[11] + RL( F(Q[11], Q[10], Q[ 9]) + Q[ 8] + x[11] + 0x895CD7BE, 22);
Q[13] = Q[12] + RL( F(Q[12], Q[11], Q[10]) + Q[ 9] + x[12] + 0x6B901122,  7);
Q[14] = Q[13] + RL( F(Q[13], Q[12], Q[11]) + Q[10] + x[13] + 0xFD987193, 12);
Q[15] = Q[14] + RL( F(Q[14], Q[13], Q[12]) + Q[11] + x[14] + 0xA679438E, 17);
```

Obr. 5.2: *Rovnice pre tunel Q9*

V prípade vyššie uvedenej podmienky funkcia *F* nezávisí na hodnote i-tého bitu *Q*[9]. Vyplýva to z definície funkcie F(x,y,z) = (x AND y ) OR ((NON(x) AND z).

| **x** | **y** | **z** | **F(x,y,z)** |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |

Tab. 5.1: *Pravdivostná tabuľka funkcie F pre Q[11]*

Tabuľka *5.1* zobrazuje pravdivostnú tabuľku funkcie *F* pre rovnicu *Q*[11]. Premenná *x* reprezentuje hodnotu $Q[10]_i$, ktorá má pre tunel Q9 konštantnú hodnotu nula. Premenná *z* predstavuje $Q[8]_i$, ktorá môže nadobúdať ľubovoľné hodnoty a tunel Q9 na nu nekladie žiadne podmienky. Pravdivostná tabuľka potvrdzuje tvrdenie, že zmena i-tého bitu Q[9] teda premennej *y* neovplyvní výslednú hodnotu funkcie *F*.

Podobne na tom je aj rovnica pre *Q*[12]. Jej pravdivostná tabuľka je zobrazená v tabuľke *5.2*. V tomto prípade premenná $x = Q[11]_i$ ktorá má na základe tunelu Q9 stálu hodnotu 1 a $y = Q[10]_i$ ktorá má hodnotu 0. V prípade, že obidve premenné spĺnaju tieto podmienky tak výsledná hodnota funkcie *F* nezávisí na premennej *z* reprezentujúcu hodnotu bitu $Q[9]_i$.

| x | y | z | F(x,y,z) |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 |

Tab. 5.2: *Pravdivostná tabuľka funkcie F pre Q[12]*

Z predchádzajúcich tvrdení teda vyplýva, že pre tie bity, kde $Q[10]_i = 0$ a súčasne $Q[11]_i = 1$ môžeme zmeniť hodnotu $Q[9]_i$, pričom táto zmena sa neprejaví v rovniciach pre $Q[10]$ a $Q[11]$. Prejaví sa v rovniciach pre $Q[9]$, $Q[10]$ a $Q[13]$. Tieto zmeny sa však kompenzujú zmenou hodnoty správy, to znamená zmenou premenných $x[8]$, $x[9]$ a $x[12]$, zatiaľ čo hodnoty premenných $Q[9]$, $Q[10]$ a $Q[13]$ zostávajú nezmenené, ako je vidieť na obrázku *5.3*. Dôležité je, že zmeny hodnôt $x[8]$, $x[9]$ a $x[12]$ sa prejavia až za bodom verifikácie POV. Tým pádom sa zmenia všetky premenné $Q[25]$ až $Q[64]$, a to dosť zložitým a náhodným spôsobom [6].

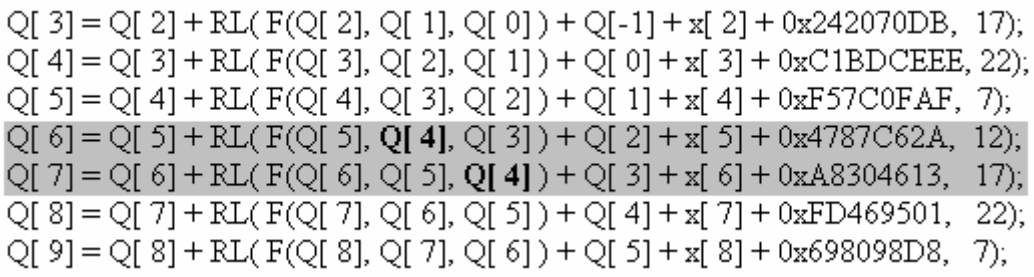
```
Q[ 7] = Q[ 6] + RL( F(Q[ 6], Q[ 5], Q[ 4]) + Q[ 3] + x[ 6] + 0xA8304613, 17);
Q[ 8] = Q[ 7] + RL( F(Q[ 7], Q[ 6], Q[ 5]) + Q[ 4] + x[ 7] + 0xFD469501, 22);
Q[ 9] = Q[ 8] + RL( F(Q[ 8], Q[ 7], Q[ 6]) + Q[ 5] + x[ 8] + 0x698098D8,  7);
Q[10] = Q[ 9] + RL( F(Q[ 9], Q[ 8], Q[ 7]) + Q[ 6] + x[ 9] + 0x8B44F7AF, 12);
Q[11] = Q[10] + RL( F(Q[10],      , Q[ 8]) + Q[ 7] + x[10] + 0xFFFF5BB1, 17);
Q[12] = Q[11] + RL( F(Q[11], Q[10],      ) + Q[ 8] + x[11] + 0x895CD7BE, 22);
Q[13] = Q[12] + RL( F(Q[12], Q[11], Q[10]) + Q[ 9] + x[12] + 0x6B901122,  7);
Q[14] = Q[13] + RL( F(Q[13], Q[12], Q[11]) + Q[10] + x[13] + 0xFD987193, 12);
Q[15] = Q[14] + RL( F(Q[14], Q[13], Q[12]) + Q[11] + x[14] + 0xA679438E, 17);
```

Obr. 5.3: *Tunel Q9*

Cieľom je samozrejme získať čo najsilnejší tunel. Preto pokiaľ je možné je potrebné nastaviť čo najviac dvojíc bitov $Q[10]_i = 0$ a $Q[11]_i = 1$. Tieto podmienky nie sú súčasťou postačujúcich podmienok, iba urýchlujú hľadanie kolízií pomocou metódy tunelovania. Nazývajú sa podobne ako u metódy mnohonásobnej modifikácie správ extra podmienky. Naneštastie počiatočné podmienky v súčasnom diferenčnom schémate umožnujú pre tento účel použiť len tri pozície bitov, ostatné nie sú volné. Tým získavame tunel o sile 3.

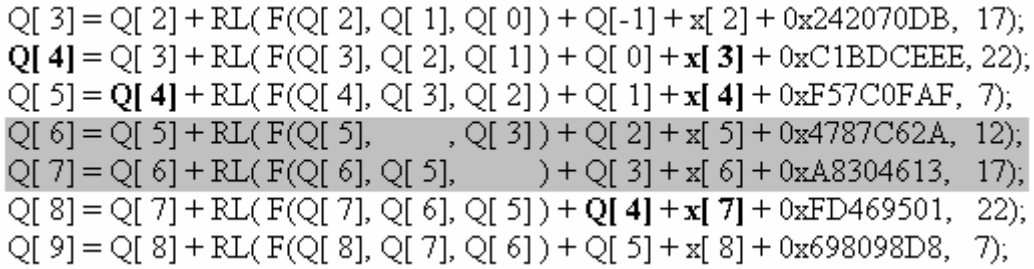
- **Tunel Q4**

Princíp tohoto tunelu je rovnaký ako tunelu Q9. Využívajú sa rovnice pre $Q[6]$ a $Q[7]$ zobrazené na obrázku *5.4*.




Obr. 5.4: *Rovnice pre tunel Q4*

Pravdivostné tabuľky sú rovnaké ako v prípade tunelu Q9 preto sa tu neuvádzajú. Pre tie bity *i*, kde $Q[6]_i = 1$ a súčasne $Q[5]_i = 0$ získavame tunel podľa obrázku *5.5*. Na bitoch *i* sa teda môže zmeniť hodnota bitu $Q[4]_i$ , pričom táto zmena sa neprejaví v rovniciach pre *Q*[6] a *Q*[7]. Prejaví sa v rovniciach pre *Q*[4], *Q*[5] a *Q*[8]. Tieto zmeny sa kompenzujú zmenou hodnoty správy *x*[3], *x*[4] a *x*[7].




Obr. 5.5: *Tunel Q4*

V tomto prípade ale tieto zmeny neovplyvnujú len hodnoty *Q*[25] až *Q*[64] za bodom verifikácie POV. Presnejšie premenná *x*[4] sa vyskytuje pred bodom verifikácie POV v rovnici pre *Q*[24]. Tu môže spôsobiť narušenie už splnenej postačujúcej podmienky, ktorá pre súčasné diferenčné schéma je jednobitová. Tento tunel je príkladom tunelu pravdepodobnostného. Z jedného bodu POV nemusí tunel stopercentne viesť k ďalším bodom POV, ale len s určitou pravdepodobnosťou. Niektoré tunely môžu mať pravdepodobnosť nižšiu, napríklad polovičnú alebo štvrtinovú.

V prípade MD5 je tento tunel použitý ako veľmi úzky, pretože počiatočné podmienky ponechávajú voľný iba jeden bit. Zmena tohoto bitu ovplyvnuje jednobitovú počiatočnú podmienku v *Q*[24] len s veľmi malou pravdepodobnosťou, takže tento tunel má silu skoro 1 [6].

- **Iné tunely**

V práci [6] sú popísané ešte iné typy tunelov. Tie uvádzané nebudú, vzhľadom na to, že cieľom nie je popis všetkých existujúcich tunelov. Myšlienka tunelov je použitelná obecne i pre iné hashovacie funkcie. Musia byť pochopitelne nájdené odpovedajúce konkrétne tunely a diferenčné schémata [6]. Tunely, ktoré sú popísané v MD5, sú príliš umelé a složité, pretože dané diferenčné schéma nebolo pre tento účel vytvorené. Z významu aký majú tunely pre hľadanie kolízií je jasné, že sa vyplatí zmeniť diferenčné schéma tak, aby už v sebe zahrnovalo možnosť tunelov. Rozhodne to ale nie je triviálna úloha. V práci [4] sa autor zamýšla, prečo teda nevytvoriť nové diferenčné schémata, ktorých hlavným cieľom bude jeden 32-bitový alebo dva 16-bitové tunely. Diferenčná schéma, ktorá by umožnila tunely Q4 alebo Q9 v šírke 16 až 32 bitov, by urýchlila tvorbu kolízií $2^{16}$ až $2^{32}$ krát.

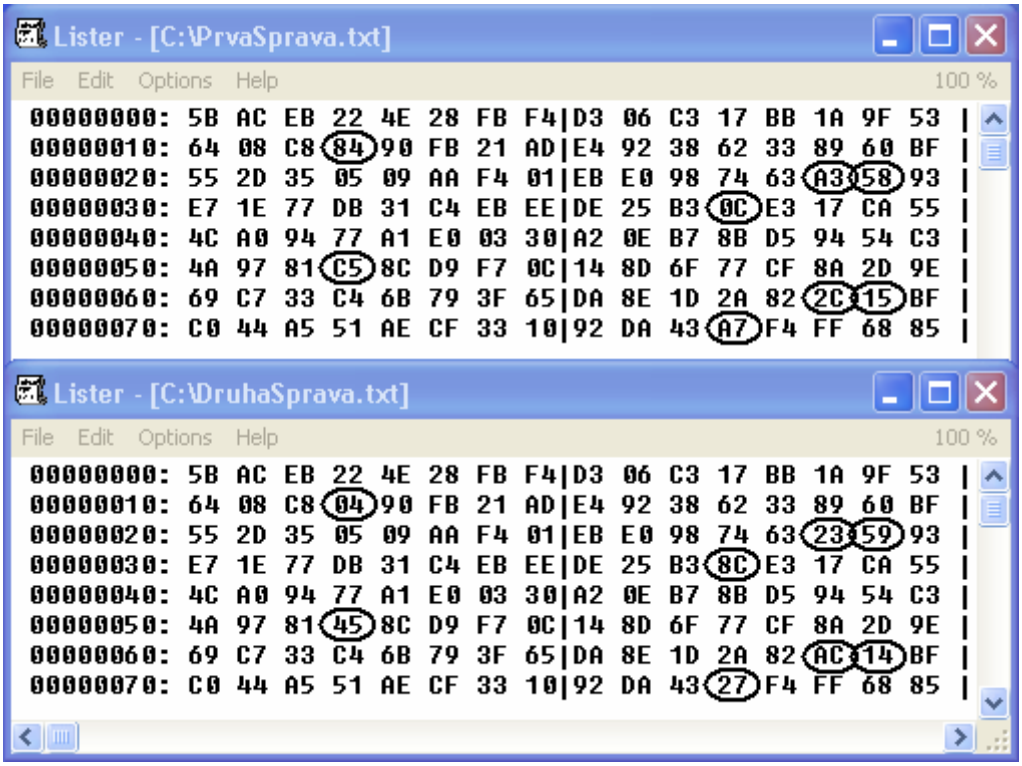
- **Experimentálne výsledky**

Program Vlastimila Klímy, ktorého zdrojový kód je k dispozícií k staženiu na autorových stránkach bol napísaný pre experimentálne overenie metódy tunelovania a k získaniu časových odhadov. Bol testovaný na notebooku (Acer TraveMate 450Lmi, Intel Pentium 1.6 GHz ). S aplikáciou tunelov v prvom aj v druhom bloku bola priemerná doba hľadania kolízie cca 31 sekúnd, pričom 30 sekúnd trvalo nájdenie kolízie prvého bloku a 1 sekundu kolízia druhého bloku. Na počítači s Intel Pentium 4 ( 3.2 GHz ) bola priemerná doba hľadania kolízie prvého radu 17 sekúnd [6].

## 5.3 Simulácia kolízie

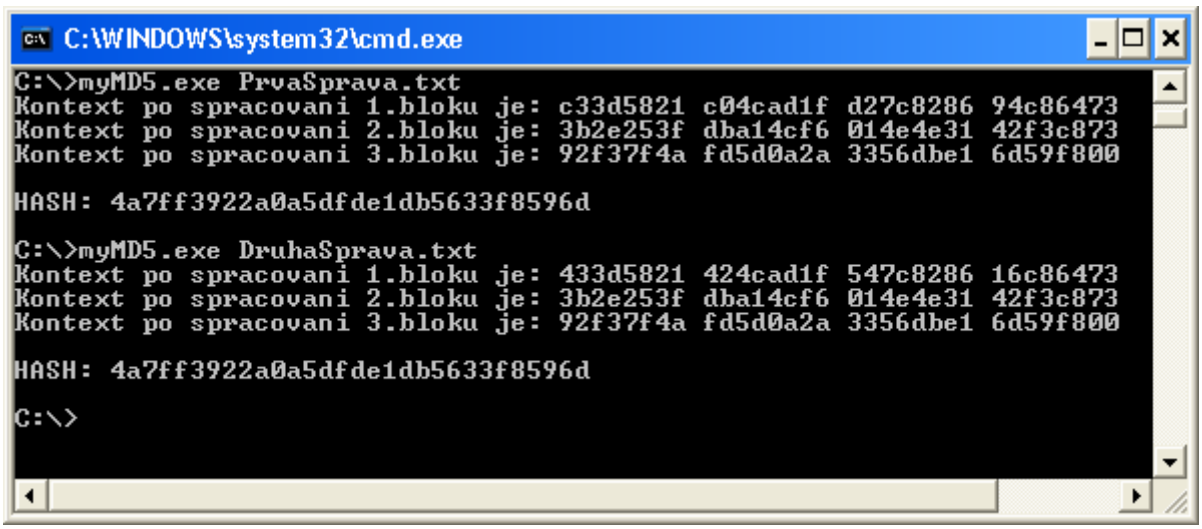
V tejto kapitole bude uskutočnená vzorová simulácia kolízie dvoch 1024 bitových správ vygenerovaných programom Vlastimila Klímy pre inicializačnú hodnotu hashovacej funkcie MD5.

Naprogramovaná implementácia hashovacej funkcie MD5 z kapitoly *4* je upravená tak, aby jej výstupom bol nielen výsledny digitálny otisk ale aj jednotlivé kontexty po spracovaní každého z 512-bitových blokov. Kolízne správy sú zobrazené na obrázku *5.6*. Sú v ňom vyznačené rozdielne bajty oboch správ.




Obr. 5.6: *Kolízne správy*

Obidve správy následne zahashujeme implementáciou MD5. Výstup je zobrazený na obrázku *5.7*. Jasne je vidieť, že po spracovaní prvého 512-bitového bloku je ešte kontext obidvoch správ odlišný. Počas spracovávania druhého bloku sa tento rozdiel vyrovná a kontext kolíznych správ je rovnaký. Kedže tretí blok majú obidve správy rovnaký, je tvorený zarovnávacími bitmi a dĺžkou pôvodnej správy, ovplyvní tento blok spracovanie kontextu u oboch správ rovnakým spôsobom. Výsledný hash je teda rovnaký pre obidve kolízne správy.




Obr. 5.7: *Vystup programu v prípade dvoch kolíznych správ*

Treba podotknúť, že tieto kolízne správy sú kolízne iba pre inicializačný vektor algoritmu MD5 (viac kapitola *3.3*). Pre akýkoľvek iný inicializačný vektor bude ich výstupný digitálny otisk odlišný.

## 5.4 Využitie kolízie prvého radu

V predchádzajúcej časti bola rozobraná problematika nachádzania kolízii prvého radu u hashovacej funkcie MD5. Bola uskutočnená simulácia, ktorá poukazuje na schopnosť nachádzať kolízie prvého radu omnoho jednoduchšie a rýchlejšie ako v prípade narodeninového paradoxu. Ide teda o porušenie základnej vlastnosti hashovacej funkcie odolnosti voči kolíziam prvého radu. Stále ale nie je možné nájsť k danému dokumentu iný s rovnakým digitálnym otiskom, teda nájsť kolíziu druhého radu. Sme schopní nájsť iba dva rôzne dokumenty s rovnakým digitálnym otiskom.

V práci [1] je prezentovaná ukážka zneužitelnosti tejto skutočnosti na vytvorenie falošného certifikátu. Využíva sa možnosť nájdenia kolízie pre akúkoľvek inicializačnú hodnotu. To umožní v podstate vytvoriť správy s ľubovolným začiatkom správy ktorý je doplnený odlišnými 128-bajtovými blokmi a pokračujúce ľubovoľným obsahom. Dôležité je aby začiatok bol zarovnaný na 64 bajtov. Práve odlišných 128 bajtov sa uloží do miesta v certifikáte, kde je verejný kľúč, ktorý potvrdzuje certifikačná autorita svojím digitálnym podpisom v certifikáte. Výsledkom je, že útočník si pripraví dva rôzne kľúče na ktoré bude mať vydaný jeden platný certifikát.

Fakt, že sme schopní vydávať dva rôzne dokumenty za jeden na základe rovnakého digitálneho otisku hashovacej funkcie je možné využiť rôznymi spôsobmi. Zaújmavá je možnosť existencie dvoch rôzne sa chovajúcich programov, pričom ich digitálny otisk je rovnaký, z čoho by malo teoreticky vyplývať že ide o rovnaké programy. Obidve verzie programov majú rovnaký digitálny otisk tým pádom sa na jeho základe nedá zistiť odlišnosť. V rámci tejto kapitoly bude demonštrovaná ukážka praktického využitia kolízie prvého radu hashovacej funkcie MD5 na vytvorenie dvoch rôznych programov s rovnakým digitálnym otiskom.

### 5.4.1 Dva rôzne programy s rovnakým digitálnym otiskom

Na demonštráciu tohoto útoku využívajúceho existenciu kolízie prvého radu využijeme fakt, že pokiaľ do daného programu vhodne vložíme dva kolízne bloky, ktoré začínajú na mieste v pamäti zarovnanom na 512 bitov, potom sme schopní vytvoriť dva rôzne programy s rovnakým digitálnym otiskom. Tieto programy sa tým pádom líšia len v bitoch v ktorých sa líšia kolízne bloky. Útočník si je vedomý miest na ktorých sa tieto rozdiely nachádzajú a preto je na základe týchto rozdielov schopný riadiť beh programu. Tým vlastne docieli odlišné chovanie jedného z programov.

V praxi to môže vyzerať tak, že útočník najprv vytvorí primárny program, ktorý splňuje všetky podmienky ktoré sú na kladené na tento program. Následne je vytvorený sekundárny program, ktorý obsahuje presne ten istý kód ako v predchádzajúcom prípade ale s iným kolíznym blokom. Oba programy obsahujú v niektorej svojej časti zdrojového kódu rozhodnutie poprípade podmienku, ktorá na základe odlišných bitov v kolíznych blokoch, vykoná operáciu zodpovedajúcu primárnemu respektíve sekundárnemu programu. Následne môže byť napríklad verejne vystavený primárny program s jeho digitálnym otiskom aby si mohli užívatelia otestovať správnu funkčnosť tohoto programu [5]. Potom je vo vhodnom okamžiku podstrčená sekundárna verzia programu, ktorá môže vykonávať okrem správnej funkcie prvého programu aj nejakú škodlivú činnosť na základe podmienky testujúcej konkrétny bit kolidujúceho vektoru. Pri zamaskovaní tejto škodlivej činnosti užívateľ rozdiel nespozná, pretože sa spolieha na overenie integrity digitálnym otiskom MD5, ktorý majú obidva programy rovnaký.

### 5.4.2 Scenár

Pre demonštráciu využitia existencie kolízii prvého radu pre implementáciu programov s rovnakým digitálnym otiskom bude vytvorený jednoduchý fiktivný scenár, na ktorom budú ukázané postupy potrebné k vytvoreniu podobného útoku. Pri implementácii je využitý jazyk C++. Jedná sa opäť o konzolovú aplikáciu, ktorá načíta údaje zo vstupného textového súboru a výstupom je taktiež textový súbor s výstupnými údajmi.

Hlavnú úlohu v scenári hrá pán M.Útočník, ktorý ako zamestnanec veľkej organizácie pracuje vo funkcii správcu výpočtovej techniky. Vedenie firmy zadá projekt na vytvorenie automatizovaného pracoviska, ktorý by spracovával vstupné parametre a výsledkom by bol údaj o mesačnej mzde jednotlivých zamestnancov. Útočník teda dostal za úlohu vytvoriť

program, ktorý spracuje vstupný súbor obsahujúci údaje o zamestnancov na výstupný súbor obsahujúci informáciu o hrubej mzde. Útočník si je vedomý, že firma používa na overovanie integrity dát digitálne otisky hashovacej funkcie MD5. Rozhodne sa využiť túto príležitosť na vytvorenie dvoch programov s rovnakým digitálnym otiskom za účelom zvýšenia svojho platu.

### 5.4.3 Postup

Prvé čo je nutné urobiť je oboznámiť sa a uskutočniť implementáciu skutočného programu, ktorý je požadovaný. Vstupný textový dokument má presne definovaný formát uloženia dat. Obsahuje informácie o zamestnancoch v porádí: meno, identifikačné číslo, počet odpracovaných hodín, hodinová mzda, počet odpracovaných nadčasov a odmena navrhovaná nadriadenou osobou. Príklad vstupného formátu je následovný:

M.Homer 7844 160 100 8 1000
M.Útočník 78474 155 110 15 500
R.Hula 5622 165 95 32 500
R.Flaming 78475 129 130 10 2000
J.Simson 7842 148 100 3 1500
R.Fray 784 158 100 30 1600

V prvej fáze sa teda najprv naprogramuje požadovaný program. V zdrojovom kóde sa musí vytvoriť miesto na dva 512-bitové kolízne bloky. Zatiaľ samozrejme ich presný obsah nie je známy, musí sa však pre tieto bloky alokovať miesto tak aby začínalo na adrese delitelnej bezozbytku 512. Príklad alokácie miesta kolíznych blokov:

```
unsigned char BlokPreAlokaciu [] = {
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,
0x0E,0x0F,0x8F,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,
0x1B,0x1C,0x1D,0x1E,0x1F,0x9F,0x21,0x22,0x23,0x24,0x25,0x26,0x27,
0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0xAF,0x31,0x32,0x33,0x34,
0x35,0x36,0x37,0x38,0x39,0x3A,0x3B,0x3C,0x3D,0x3E,0x3F,0xBF,0x41,
0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,
0x4F,0xCF,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0x5B,
0x5C,0x5D,0x5E,0x5F,0xDF,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,
0x69,0x6A,0x6B,0x6C,0x6D,0x6E,0x6F,0xEF,0x71,0x72,0x73,0x74,0x75,
0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,0xFF,};
```

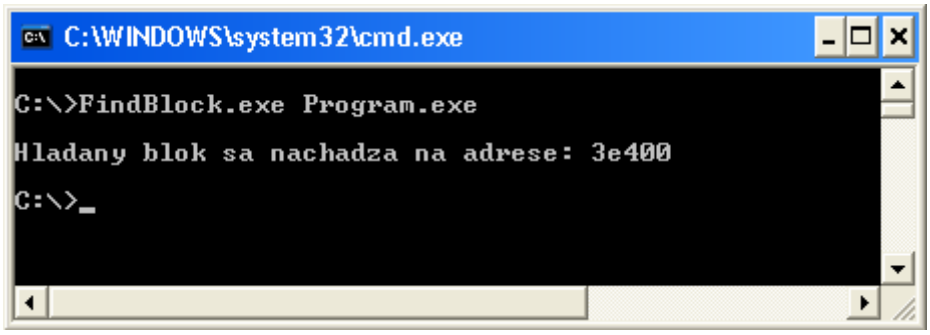
V tomto prípade sa nepredpokladá, že niekto bude študovať zdrojový kód programu. Pokiaľ by tak hrozilo je potreba tieto kolízne bloky vhodným spôsobom zamaskovať, napríklad nejakými užitočnými datami alebo vyplniť *BlokPreAlokaciu* nejakým zmysluplným textom. Práve na týchto alokovaných miestach sa budú nachádzať rozdielne bloky, okrem ktorých budú

programy úplne rovnaké. Útočník si je vedomý pozícii týchto rozdielov respektíve odlišných bitov a na ich základe vytvorí program s odlišným chovaním. Docieli sa to jednoduchou podmienkou *if-else* alebo tak aby sa výsledna mzda počítala pomocou hodnôt ležiacich na odlišných bajtoch kolíznych blokoch. V tomto prípade je to riešené pomocou podmienky:

```
if (((int(BlokPreAlokaciu[19]) & 128) == 128) && (osoba->ID==78474)) ...
```

Podmienka je založená na poznatku, že práve jeden z odlišných bitov oboch 1024-bitových správ je bit číslo 8 v 20-tom bajte [6]. Na základe jeho odlišnosti a samozrejme v prípade, že identifikačné číslo je číslo útočníka, je hrubá mzda vypočítana podľa iného vzorca, zaručujúceho zvýšenie mzdy.

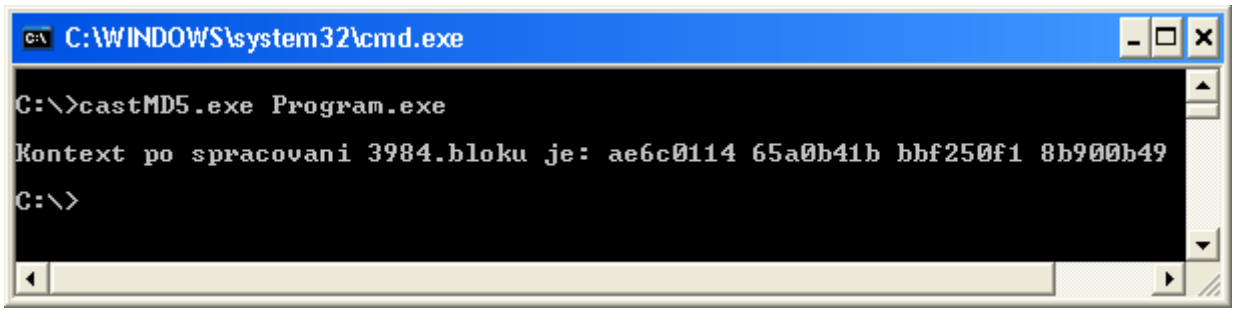
V ďalšej fázi môže prísť ku skompilovaniu zdrojového kódu a tým získaniu funkčného programu. Pre ďalší postup je potreba z tohoto skompilovaného programu získať adresu, na ktorej sa nachádza alokované miesto pre uloženie kolíznych blokov. Pre tento účel bola uskutočnená implementácia programu *FindBlock*, ktorý má v zdrojovom kóde uložený rovnaký blok aký sa použil pre alokáciu v pôvodnom programe, to znamená blok *BlokPreAlokaciu*. Následne sa parametrom predá tomuto programu názov programu v ktorom sa požaduje tento blok nájsť. Výsledkom je potrebná adresa, ktorá sa použije v ďalšom postupe. Pokiaľ získaná adresa nie je bezozbytku delitelná 512 bitmi, je potrebné ukutočniť zarovnanie, napríklad doplnením bloku *BlokPreAlokaciu* dostatočným počtom bajtov. Výstup programu *FindBlock* zobrazuje obrázok *5.8*.




Obr. 5.8: *Výstup programu FindBlock*

Následne je nutné získať čiastočný digitálny otisk skompilovaného programu práve po miesto kde sa nachádza alokácia miesta pre kolízne bloky, teda po adresu získanu programom *FindBlock*. Po zistení umiestnenia blokov je z tejto adresy určený počet 512-bitových blokov, ktoré boli spracované do miesta, kde sa nachádza blok *BlokPreAlokaciu*. Implementácia

z kapitoly *4* je upravená tak aby, jej výstupom nebol digitálny otisk výsledný, ktorý momentálne nie je dôležitý, ale len jednotlivé registre kontextu po spracovaní vypočítaného počtu 512-bitových blokov. Kolízne bloky začínajú na adrese 0x3E400 čo zodpovedá počtu 3984 blokov o veľkosti 512 bitov. Preto práve kontext po spracovaní 3984-tého bloku sa použije ako inicializačná hodnota pre nájdenie kolíznych blokov. Výstup upravenej implementácie je možno vidieť na obrázku *5.9*.

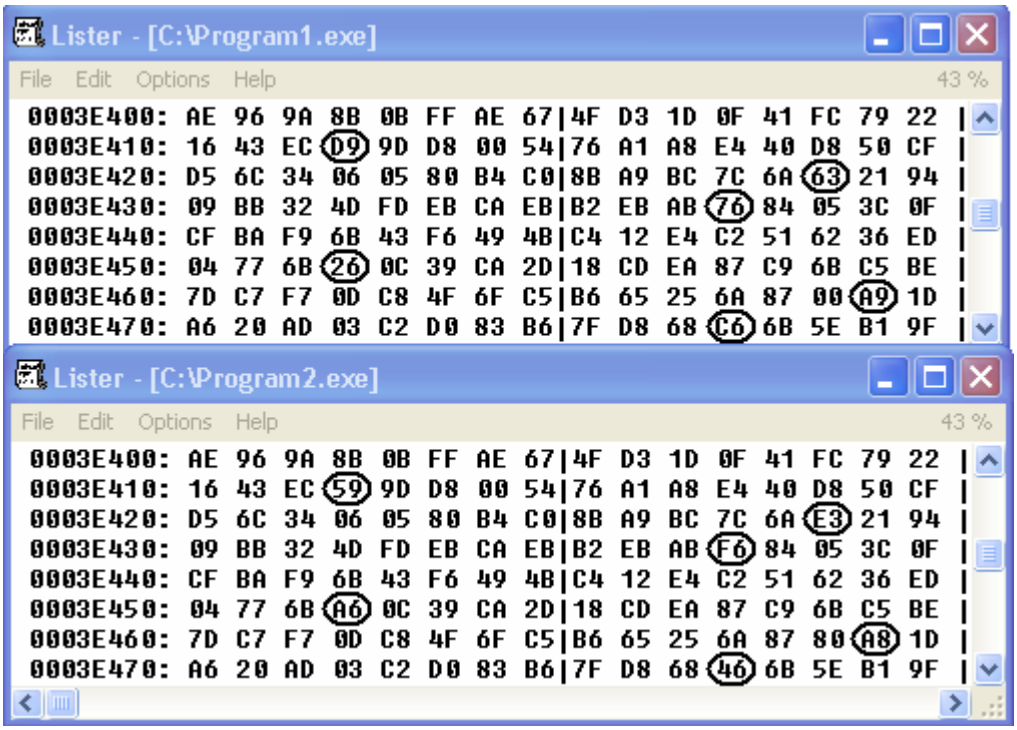



Obr. 5.9: *Čiastočný digitálny otisk*

K nájdeniu kolízie sa použije program Vlastimila Klímy, pričom nájdenie kolíznych reťazcov je otázkou pár desiatok sekúnd. Teraz je všetko pripravené na vloženie kolíznych blokov do programu. Treba poznamenať, že na vkladanie blokov sa musí použiť program z ktorého sme získali čiastočný digitálny otisk. Dôvod je jednoduchý. Po opätovnom skompilovaní zdrojového kódu vznikne síce ten istý program ale s iným digitálnym otiskom. Líšil by sa hlavne aj digitálny otisk na adrese kde začínajú kolízne bloky. Príčinou je kompilátor, ktorý do programu vkladá svoje vlastné data. Preto je nutné použiť pôvodne skompilovaný program, z ktorého sa vytvoria dve kópie a do každej sa nahrá jeden z kolíznych reťazcov.

Na tento účel bol naprogramovaný program *CreateCollisionPrograms*. Do zdrojového kódu tohoto programu sa vložia dva získané kolízne bloky *KoliznyBlok1*, *KoliznyBlok2* a do premennej *MiestoUlozenia* sa uloží adresa miesta kde sa majú kolízne bloky nahrať. V tomto konkrétnom prípade je to adresa 0x3E400. Ako parameter sa predá názov pôvodného programu. Program automaticky vytvorí dve kópie programu predaného parametrom, s tým rozdielom, že na miesta definované v premmenej *MiestoUlozenia* nahrá kolízne bloky. Blok *KoliznyBlok1* nahrá do programu *Program1* a blok *KoliznyBlok2* nahrá do programu *Program2*. Pomocou programu *CreateCollisionPrograms* tak odpadá nutnosť ručného vkladania týchto kolíznych blokov do pôvodného programu, ktoré môže byť do značnej miery ovplyvnené ľudským faktorom, a prezentovaný postup sa stáva plne automatickým.

Takto získané programy *Program1* a *Program2* sú hľadané kolízne programy. Obrázok *5.10* zobrazuje rozdiely medzi programami na bajtovej úrovňi. Kolízne bloky začínajú na adrese 0x3E400 a končia na 0x3E47F. Na obrázku sú vyznačené odlišné bajty kolíznych reťazcov a tým aj celých programov.

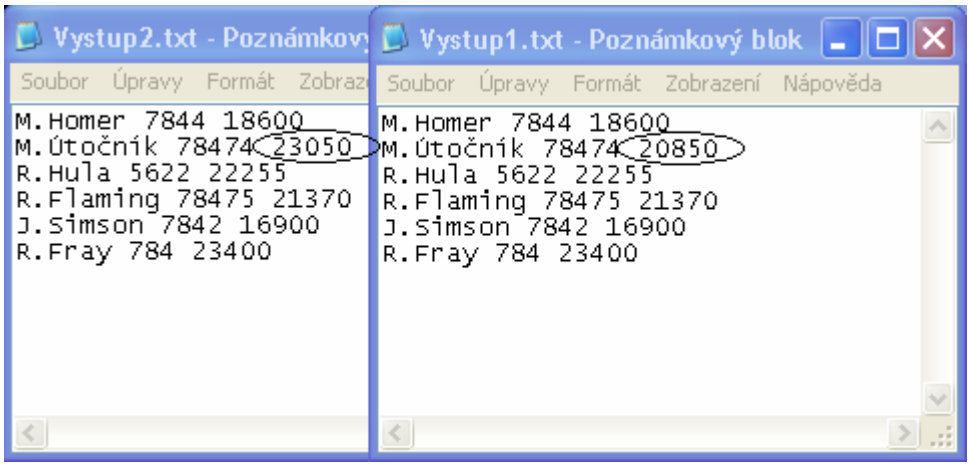



Obr. 5.10: *Porovnanie kolíznych blokov u oboch programov*

### 5.4.4 Výsledky

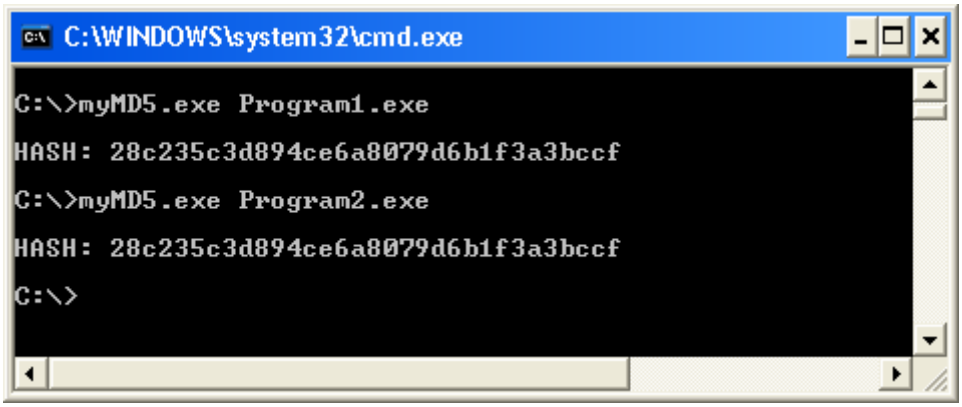
Na overenie správnej funkčnosti programov sa použili vstupné parametre z úvodu kapitoly *5.4.3*. Obrázok *5.11* zobrazuje porovnanie výstupov v prípade obidvoch programov. Výstupný súbor obsahuje informácie v poradí: meno, idenifikačné číslo a vypočítaná mesačná mzda zamestnanca.

Je vidieť, že jeden z programov pre útočníka vypočíta mzdu podľa iného vzorca zaručujúceho jej zvýšenie. Samozrejme podľa tohoto ide o dva rôzne sa chovajúce programy, tým pádom by mali mať rozdielne digitálne otisky. Ako si je však možné všimnúť na obrázku *5.12* obidva programy majú digitálne otisky rovnaké. Pokiaľ sa teda niekto spolieha na overenie integrity programov na základe digitálnych otiskov MD5, môže byť podobným spôsobom oklamaný.




Obr. 5.11: *Porovnanie výstupov*

Táto kapitola bola zameraná na využitie kolízií prvého radu na vytvorenie odlišných programov s rovnakým digitálnym otiskom. Na záver treba podotknúť, že samotné kolízne reťazce sú sami o sebe nepoužiteľne. Sú vytvárané náhodne a preto pravdepodobnosť použitia v texte je minimálna. Na druhej strane ich vhodným umiestnením do existujúcich programov sa dá docieliť odlišné chovanie programov a vytvoriť tak zmysluplný útok využívajúci kolízie prvého radu.




Obr. 5.12: *Porovnanie digitálnych otiskov*

# 6 Útoky na jednosmernosť

Hashovacie funkcie sú jednosmerné. Už z názvu vyplýva, že zo známej správy vytvoríme hash veľmi jednoducho a rýchlo, ale opačne to nesmie platiť. To znamená, pokiaľ je k dispozícií iba digitálny otisk neznámej správy musí byť odhalenie tejto správy poprípade hesla nemožné. Ako sa spomínalo v kapitole *2.3.4* hashovacie funkcie sa používajú na šifrovanie hesiel. Preto nie je nič nezvyčajné, že útočník v prípade úspešného vniknutia do databáze hesiel sa môže stretnúť s heslami v tvare digitálneho otisku. Z princípu jednosmernosti by malo byť nemožné získanie pôvodnej správy. U hashovacích funkcií je to pravda iba z časti. V podstate existujú dve možnosti na odhalenie skrytej správy. Pokiaľ tieto zlyhajú, nie je prakticky šanca na získanie pôvodného reťazca.

## 6.1 Prehľad možných útokov

- **Útok hrubou silou**

Známy je pod názvom brute-force attack. Tento útok je založený na testovaní všetkých možných reťazcov so zvolenou dĺžkou nad zvolenou znakovou sadou. To môžu byť napríklad veľke a malé písmena, číslice ale aj špeciálne znaky. Za predpokladu generovania reťazcov dĺžky 1 až 6 tvorených iba malými písmenami anglickej abecedy, ktorých je 26, tak celkový počet vygenerovaných reťazcov bude $26^1 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6$. Tým pádom vyjde obrovské číslo. Z každého takto vygenerovaného reťazca je vytvorený digitálny otisk a ten je porovnávaný s otiskom neznámej správy.

- **Slovníkový attack**

Alebo tiež dictionary attack je útok podobný útoku hrubou silou až na to, že reťazce respektíve ich digitálne otisky nie sú generované, ale sú brané z nejakého externého súboru dat. Ten sa nazýva slovník a väčšinou sa jedná o klasický textový súbor poprípade môže ísť o databázu, ktorá na každom riadku obsahuje jeden reťazec a jeho digitálny otisk. To znamená, že digitálne otiski sa nemusia generovať z náhodne skúšaných reťazcov ale priamo sa vyhľadávajú v tabuľke. Veľkosť tabuliek je rôzna. Úspešnosť nájdenia požadovaného otisku tým pádom závisí predovšetkým na slovníku aký sa použije. Existujú rôzne programy, ktoré umožnujú generovať vlastné slovníky presne na mieru, to znamená možnosť výberu rozsahu,

znakovej sady poprípade aj typ hashovacej funkcie. Na internete sa dá taktiež stretnúť s tzv. MD5 online crackingom. Ide o webovú aplikáciu, do ktorej užívateľ vloží digitálny otisk a aplikácia mu vráti pôvodný reťazec. Samozrejme v prípade, že ho má uložený vo svojej databáze, čo je u kratších správ pomerne pravdepodobné.

Následujúce kapitoly budú zamerané na popis implementácie útoku hrubej sily, slovníkového útoku a porovnania ich časových možností pri získavaní pôvodných reťazcov pri znalosti ich digitálnych otiskov.

## 6.2 Implementácia útoku hrubej sily

Táto kapitola popisuje implementáciu útoku hrubej sily na hashovaciu funkciu MD5. Vychádza sa pri tom z implementácie hashovacej funkcie z kapitoly *4*.

Zdrojový kód je doplnený o funkciu *Crack*, ktorej hlavnou úlohou je postupné generovanie všetkých možných reťazcov zvolenej dĺžky nad zvolenou znakovou sadou a zadanou dĺžkou neznámeho reťazca. Následne z každého z týchto reťazcov je získaný digitálny otisk hashovacej funkcie MD5. A výsledne registre *A, B, C, D* sú porovnané s regitrami *HladA, HladB, HladC, HladD* získanými z digitálneho otisku neznámej správy. Tie sa zadávajú priamo v zdrojovom kóde. Funkcii *Crack* je ako parameter predávaná informácia o maximálnej dĺžke hľadaného reťazca, aby hľadanie nebolo nekonečné v prípade nenájdenia reťazca, a tiež počet znakov z ktorých sa skladá znaková sada.

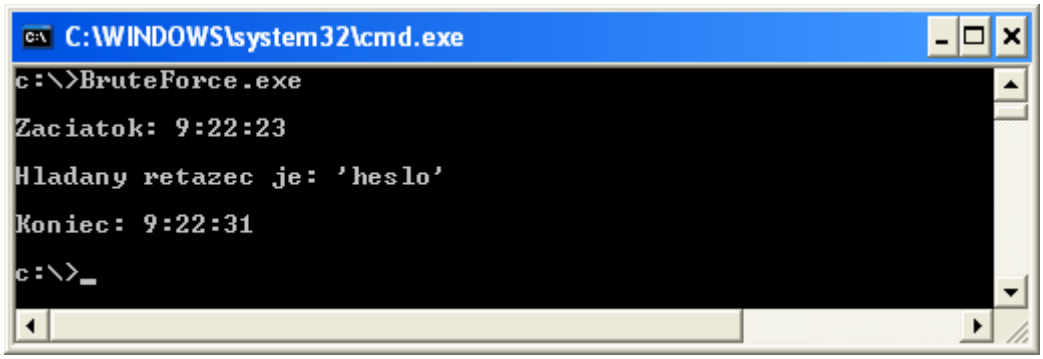
Funkcia *Crack* funguje tak, že udržuje pole *pole*, ktoré má veľkosť rovnú dĺžke hľadaného reťazca. Jednotlivé prvky tohoto pola nesú informáciu o znaku na príslušnej pozícii aktuálne generovaného reťazca, z ktorého je následne vygenerovaný digitálny otisk. Ten sa porovná s hľadaným digitálnym otiskom a ak sa zhoduje tak tento aktuálne generovaný reťazec je považovaný za výsledný. Pokiaľ nie sú zhodné, generovanie pokračuje s tým, že dôjde k aktualizácii *pole* tak aby v následujúcom kroku vygenerovalo nový reťazec.

Hlavná funkcia je doplnená o funkcie získavajúce informáciu o aktuálnom čase. Aktuálny čas je získaný na začiatku hľadanie pôvodnej správy a na konci výpočtu. Z týchto dvoch údajov sa dá ľahko odčítať celková doba útoku hrubou silou. Ukážka zdrojového kódu hlavnej funkcie:

```
time_t cas;
tm * ut;
int MaxDlzkaHesla = 7, PocetZnakov = 0;

while(ZnakovaSada[PocetZnakov]!='\0')  PocetZnakov++;
```

```
time (&cas);
ut = gmtime(&cas);
cout <<"Zaciatok: "<<ut->tm_hour<<":"<<ut->tm_min<<":"<<ut->tm_sec;

Crack(MaxDlzkaHesla, PocetZnakov);

time (&cas);
ut = gmtime(&cas);
cout <<"Koniec: "<<ut->tm_hour<<":"<<ut->tm_min<<":"<<ut->tm_sec<<;
```

Obrázok *6.1* zobrazuje výstup implementácie pri hľadaní správy „heslo“. Dĺžka reťazca bola teda 5 a znaková sada pozostávala z malých písmen anglickej abecedy, ktorých je 26. Z časových údajov je taktiež na prvý pohľad vidieť, že nájdenie pôvodnej správy trvalo presne 8 sekúnd. Všetky testy v rámci tejto kapitoly boli uskutočnené na notebooku s Intel Pentium M 1.6GHz, operačnou pamätou 512 MBytes a operačným systémom Windows XP.




Obr. 6.1: *Výstup implementácie útoku hrubej sily*

Naprogramovaná implementácia generuje reťazce nad zvolenou sadou postupne. To znamená že, pokiaľ je daná znaková sada zložená z číslic 0 až 9, tak na každej pozícií generovaných reťazcov sa postupne menia znaky od 0 až po 9. Samozrejme existujú aj špecializované programy na útok hrubou silou na hashovaciu funkciu MD5, ktoré umožnujú náhodný začiatok generovania poprípade aj voľbu počiatočného reťazca.

## 6.3 Implementácia slovníkového útoku

Pre ukážku slovníkového útoku sa využije databázový systém MySQL. Bude sa predpokladať databáza reťazcov dĺžky 0 až 7 a znaková sada zložená z číslic 0 až 9, spolu teda 10 znakov. Pre generáciu reťazcov sa použije naprogramovaný program *Generate*, ktorý vygeneruje súbor do ktorého umiestni všetky reťazce spolu s ich digitálnymi otiskami v tvare

vhodnom pre následný presun do databáze. MySQL k tomuto účelu obsahuje príkaz, pomocou ktorého sa dajú nahrať data z textového súboru priamo do databáze. Pomocou príkazu

```
LOAD DATA INFILE “Vstup.txt” into table MD5 FIELDS TERMINATED BY
',' ENCLOSED BY '"' LINES TERMINATED BY '||';
```

sa nahrajú všetky data zo súboru *Vstup.txt* do tabuľky s názvom *MD5*, ktorá sa skladá z dvoch stĺpcov. Prvý bude obsahovať pôvodný reťazec, druhý jeho 32-znakový digitálny otisk.

Indexy se používajú za účelom rychlejšieho vyhľadávánia dat v tabuľkách. Je to pomocná dátová štruktúra, ktorá určuje pozíciu dat v tabuľke na základe ich hodnôt. Inak povedané indexy sú usporiadané zoznamy dat jedného alebo niekoľkých stĺpcov tabuľky a riadku na ktorom sa daný záznam nachádza. Index trvá nejaký čas vytvoriť a vkladanie dat do tabuľky zabere viac času. Naopak vyhľadávanie podľa indexov je mnohonásobne rýchlejšie. Je tiež umožnené vytvoriť index len na niekoľko začiatočných znakov stĺpca. Prístup k datám cez index s obmedzenou dĺžkou je pomalší ale na druhej strane sa ušetrí miesto a zrýchli sa pridávanie dat. V tomto prípade bude index aplikovaný na stĺpec obsahujúci digitálny otisk, vzhľadom na to, že vyhľadávanie bude prebiehať práve na základe otisku. Konkrétne bude index aplikovaný najprv na jeho prvých 5 znakov a následne pre porovnanie na prvých 10 znakov. Do existujúcej tabuľky sa index nadefinuje príkazom:

```
alter table MD5 add index(hash(5));
```

Príkaz vytvorí index na prvých 5 znakov stĺpca hash, ktorý obsahuje 32 znakový digitálny otisk, v tabuľke s názvom MD5.

## 6.4 Výsledky

Pri porovnaní časových možností útoku hrubej sily a slovníkového útoku sa bude uvažovať 5 náhodných hesiel. Jednotlivé heslá budú postupne dekódované obidvoma typmi útokov, s tým že do testu je zahrnutý aj slovníkový útok v ktorom sa neuvažovala aplikácia indexov. Získané údaje sú zobrazené v tabuľke *6.1*.

V prípade slovníkového útoku je vidieť, že doba dekódovania nie je závislá na dĺžke hesla. Bez použitia indexov sa doba pohybovala okolo 23 sekúnd, pretože MySQL musela prejsť celú databázu. Naopak pri aplikácii indexov na prvých 5 znakov digitálneho otisku sa doba zkrátila v priemere na 1,5 sekundy, pretože MySQL pozná umiestnenie vyhľadávaných otiskov a nemusí tak prehľadávať celú databázu. V tabuľke, ktorá mala index vytvorený nad

10 znakmi, sa doba vyhľadávania pohybovala už na veľmi malých hodnotách. V prípade, že by sa index aplikoval na všetkých 32 znakov digitálneho otisku, a nie len na 5 respektíve 10 ako je to v tomto prípade, sa dá očakávať ešte zníženie doby potrebnej k nájdeniu príslušného reťazca. To by sa však naplno prejavilo v databázach omnoho väčších ako je v tomto ukážkovom prípade. Pri dekódovaní útoku hrubou silou bola doba závislá na dĺžke pôvodného reťazca. Preto je logické, že čím dlhšie heslo sa zvolí, tým ťažšie je ho útokom hrubej sily prelomiť. Navyše pri znakovej sady pozostávajúcej nielen s číslic 0 až 9 ale aj malých, veľkých písmen a prípadne aj nealfanumerických znakov a dĺžke hesla minimálne 8 znakov by nebolo možné digitálny otisk takéhoto hesla v rozumnom čase dekódovať. Najvýhodnejší spôsob sa javí použitie slovníkového útoku založeného na databázovom systéme MySQL s využitím indexov. Ten však počas testovania mal isté nevýhody. Nevýhodou je obrovská veľkosť jednotlivých databáz, ktoré s narastajúcou znakovou sadou a dĺžkou reťazcov veľmi rýchlo rastú. S tým súvisí aj narastajúca doba potrebná k indexovaniu takejto databáze. Pre ilustráciu v tomto prípade, kedy sa uvažovala znaková sada zložená len z 10 znakov a dĺžky hesiel 0 až 7, bola veľkosť databáze 500MB a doba potrebná pri jej indexovaní na 5 znakov činila 2 hodiny.

| **Heslo** | **Dĺžka hesla** | **Doba dekódovania** | | | |
|---|---|---|---|---|---|
| | | **Slovníkový útok** | | | **Útok hrubou silou** |
| | | bez indexov | index na 5 znakov | index na 10 znakov | |
| 12344 | 5 | 23.7s | 1.72s | 0.06s | 1s |
| 811255 | 6 | 23.4s | 1.54s | 0.02s | 3s |
| 1551258 | 7 | 23.7s | 1.59s | 0.02s | 8s |
| 5421549 | 7 | 23.5s | 1.41s | 0.03s | 18s |
| 9565324 | 7 | 23.7s | 1.36s | 0.03s | 30s |

Tab. 6.1: *Namerané doby dekódovania pri jednotlivých útokov*

# 7 Záver

Pri tvorbe tejto bakalárskej práce som čerpal informácie predovšetkým z literatúry a internetových zdrojov, venovaných kryptografii a problematike hashovacích funkcií predovšetkým algoritmu MD5. Táto problematika nie je ľahko zmapovateľná, pretože neustále prichádzajú nové techniky a postupy, ktoré túto atraktívnu tématiku posúvajú každým dňom na vyššiu úroveň.

Hashovacie funkcie patria v súčasnosti k základným stavebným prvkom modernej kryptografie. Sú to výpočetne efektívne funkcie, ktoré menia ľubovoľne dlhú konečnú bitovú postupnosť na unikátny digitálny otisk. Vďaka svojím typickým vlastnostiam, ku ktorým patria jednocestnosť a bezkolíznosť, nám umožnujú napríklad jednoduché ale účinné šifrovanie hesiel a sú základom digitálneho podpisu.

Hlavným cieľom tejto bakalárskej práce je analyzovať hashovací algoritmus MD5. Úvod práce patrí základom kryptografie, vysvetleniu pojmu hashovacie funkcie a ich vlastnostiam. Ďalšia kapitola je venovaná samotnému algoritmu MD5. Dopodrobna je tu analyzovaný postup, ktorým MD5 spracováva vstupné správy a vytvára pre ne jedinečné otisky. Následne bola uskutočnená jeho implementácia v programovacom jazyku C++, ktorá zhŕňa získané teoretické poznatky. Pri vzniku MD5 na rozdiel od jeho predchodcu bol kladený dôraz na bezpečnosť. Napriek tomu v súčasnej dobe vieme vygenerovať dve správy s rovnakým digitálnym otiskom. To znamená, že algoritmus MD5 nie je odolný voči kolíziám prvého radu. V ďalšej praktickej časti je ukázaná možnosť využitia tohoto poznatku. Kolízne správy sú vytvárané náhodne a preto pravdepodobnosť použitia v texte je minimálna. Pri realizácii sa vychádzalo z poznatku, že vhodným umiestnením kolíznych správ do existujúcich programov sa dá docieliť odlišné chovanie týchto programov pri zachovaní rovnakého digitálneho otisku. Preto ak sa niekto spolieha na overenie integrity programov pomocou hashovacej funkcie MD5, môže byť podobným spôsobom oklamaný. Posledná kapitola je zameraná na možnosti útokov na jednosmernosť, a to konkrétne útoku hrubou silou a slovníkového útoku. Najlepšie časové výsledky boli dosiahnuté v prípade slovníkového útoku s využitím databázového systému MySQL s aplikáciou indexov. Táto metóda však vyžaduje použitie vhodného slovníka, ktoré môžu nadobúdať veľmi veľké objemy. Pokiaľ je tento útok neúspešný, neostáva nič iné len sa spoľahnúť na útok hrubou silou a postupné generovanie všetkých možností. V tomto smere má hashovacia funkcia MD5 stále čo ponúknuť a pri dodržaní správnych zásad prí výbere hesla, nemusí byť použitie hashovacej

funkcie MD5 na ukladanie hesiel rizikové. Za najväčší prínos práce považujem zdokumentovanie poznatkov a bezpečnostných rizík súvisiacich s hashovacím algoritmom MD5.

Hashovacia funkcia MD5 sa stala obľúbenou hlavne vďaka svojej jednoduchosti a rýchlosti. Napriek tomu, postupom času bola jej bezpečnosť narušená a dá sa očakávať, že v blízkej budúcnosti bude kompletne nahradená hashovacou funkciou SHA-1 respektíve novými triedami funkcií označovanými ako SHA-2.

# Literatúra


[1] Arjen Lenstra, Xiaoyun Wang, Benne de Weger. *Colliding X.509 Certifikates* [online]. [cit. 15.3.2008]. Dostupné z URL: < http://eprint.iacr.org/2005/067.pdf >.

[2] Doseděl, T. *Počítačová bezpečnost a ochrana dat*. Brno:Computer Press, 2004. 190 s. ISBN 80-251-0106-1.

[3] Klíma, V. *Hašovací funkce, principy, příklady a kolize* [online].[cit. 1.11.2007]. Dostupné z URL: < http://cryptography.hyperlink.cz/2005/cryptofest_2005.htm >.

[4] Klíma, V. 2006. *Kolize MD5 do minuty aneb co v odborných zprávach nenajdete*. In: Crypto-World [online]. 2006, vol.8, no.4 [cit. 14.3.2008] p.2. Dostupné z URL: < http://www.crypto-world.info/casop8/crypto04_06.pdf>.

[5] Klíma,V. 2004. *Nedůvěřujte kryptologům*. In: Crypto-World [online].2004, vol. 6, no.11 [cit. 1.11.2007] p.24. Dostupné z URL : < http://www.crypto-world.info/casop6/crypto11s_04.pdf >

[6] Klíma,V. 2006. *Tunely v hašovacích funkcích: kolize MD5 do minúty* [online]. [cit.1.11.2007]. Dostupné z URL:< http://cryptography.hyperlink.cz/2006/tunely.pdf >.

[7] *Kryptografie* [online]. [cit. 1.11.2007]. Dostupné z URL : < http://krypto.krokonet.com >.

[8] *MD5 cracking* [online]. [cit. 20.3.2008]. Dostupné z URL: < http://www.security-portal.cz/clanky/md5-cracking.html >.

[9] Přibyl, J. 2004. *Informační bezpečnost a utajování zpráv*. 1.vyd. Praha: ČVUT, 2004. 239 s. ISBN 80-01-02863-1.

[10] Rivest, R. *The MD5 Message-Digest Algorithm*, RFC 1321, 1992. Dostupné z URL: < http://www.faqs.org/rfcs/rfc1321.html >.

[11] Zimola, O. 2006. *Kryptografické útoky na MD5*. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2006. 45 s. Vedoucí diplomové práce Ing. Daniel Cvrček, Ph.D.

# Príloha – Obsah priloženého CD

| | |
|---|---|
| **Implementácia MD5** | obsahuje skompilovaný program implementácie MD5 myMD5.exe a jeho zdrojový kód. |
| **Kapitola 5** | obsahuje programy CreateCollisionPrograms.exe, FindBlock.exe, castMD5.exe z kapitoly 5 a ich zdrojové kódy. |
| **Kolízne programy** | obsahuje programy s rovnakým digitálnym otiskom, Program1.exe a Program2.exe, vytvorené v rámci kapitoly č.5 a pôvodný program Program.exe so zdrojovým kódom. |
| **Metadata** | obsahuje metadata. |
| **Text práce** | obsahuje elektronický text bakalárskej práce. |
| **Útok hrubou silou** | obsahuje skompilovaný program útoku hrubou silou a jeho zdrojový kód. |