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

HASHOVACÍ FUNKCE - CHARAKTERISTIKA, IMPLEMENTACE A KOLIZE

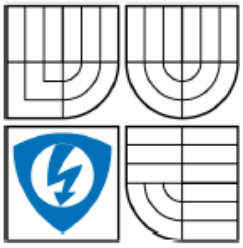
HASH FUNCTIONS - CHARACTERISTICS, IMPLEMENTATION AND COLLISIONS

DIPLOMOVÁ PRÁCE
MASTER´S THESIS

AUTOR PRÁCE — Bc. JAN KARÁSEK
AUTHOR

VEDOUCÍ PRÁCE — ING. PETRA LAMBERTOVÁ
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Jan Karásek	***ID:***	84319
Ročník:	2	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Hashovací funkce - charakteristika, implementace a kolize

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte základní vlastnosti hashovacích funkcí, popište možnosti jejich využití v praxi. Dále se detailně zaměřte na jeden hashovací algoritmus, který implementujte v programovacím jazyce dle vlastního výběru. Poukažte na nedostatky vybraného algoritmu a implementujte metody útoků, které vyvolají kolize.

DOPORUČENÁ LITERATURA:

[1] FERGUSON, Niels, SCHNEIER, Bruce. Practical Cryptography. [s.l.] : [s.n.], 2003. 432 s. ISBN 978-0-471-22357-3.

[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:	9.2.2009	***Termín odevzdání:***	26.5.2009

Vedoucí práce: Ing. Petra Lambertová

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Anotace

Hašovací funkce patří mezi prvky moderní kryptografie. Jejich úkolem je na vstupu očekávaná data převést do unikátní bitové posloupnosti. Hašovací funkce jsou používány v mnoha aplikačních oblastech, jako je ověřování integrity zpráv, autentizace informací, jsou používány v kryptografických protokolech, ke komparaci dat a dalších aplikacích.

Cílem diplomové práce je charakterizovat hašovací funkce, popsat jejich základní vlastnosti a využití. Dále se zaměřit na jednu hašovací funkci, konkrétně MD5, a tu náležitě popsat. Popsat její konstrukci, bezpečnost a možnosti útoků na tuto funkci. Posledním úkolem je tuto funkci implementovat a implementovat i kolize na ni.

V úvodních kapitolách je v práci popsána základní definice hašovací funkce, jsou popsány vlastnosti, jaké by funkce měla mít, zmíněny metody, kterými je možné předcházet jejich kolizím a zmíněny oblasti, ve kterých se hašovacích funkcí využívá. Další kapitoly jsou zaměřeny na charakteristiky druhů hašovacích funkcí. Těmito druhy jsou základní hašovací funkce postavené na základních bitových operacích, dokonalé hašovací funkce a kryptografické hašovací funkce.

Po dokončení charakteristiky hašovacích funkcí se dále věnuji praktickým záležitostem. Je popsán základní vzhled a ovládání programu, na který navazuje postupné popisování jednotlivých jeho funkcí, které jsou i dostatečně teoreticky vysvětleny. V dalším textu je popsána funkce MD5, kde se věnuji její konstrukci, bezpečnostním rizikům a samotné implementaci. Jako poslední navazuje kapitola, týkající se samotných útoků na hašovací funkce, ve které je popsána metoda tunelování hašovací funkce, metoda útoku brutální silou a slovníkový útok.

Klíčová slova

Hašovací funkce; dokonalé hašování; kryptografické hašovací funkce; haš; MD5; FNV; útoky; kolize; multikolize; kompresní funkce; tunelování; C#

Abstract

Hash functions belong to elements of modern cryptography. Their task is to transfer the data expected on the entry into a unique bite sequence. Hash functions are used in many application areas, such as message integrity verification, information authentication, and are used in cryptographic protocols, to compare data and other applications.

The goal of the master's thesis is to characterize hash functions to describe their basic characteristics and use. Next task was to focus on one hash function, in particular MD5, and describe it properly. That means, to describe its construction, safety and possible attacks on this function. The last task was to implement this function and collisions.

The introductory chapters describe the basic definition of hash function, the properties of the function. The chapters mention the methods preventing collisions and the areas were the hash functions are used. Further chapters are focused on the characteristics of various types of hash functions. These types include basic hash functions built on basic bit operations, perfect hash functions and cryptographic hash functions.

After concluding the characteristics of hash functions, I devoted to practical matters. The thesis describes the basic appearance and control of the program and its individual functions which are explained theoretically. The following text describes the function MD5, its construction, safety risks and implementation. The last chapter refers to attacks on hash functions and describes the hash function tunneling method, brute force attack and dictionary attack.

Keywords

Hash functions; perfect hashing; cryptographic hash functions; hash; MD5; FNV; attacks; collisions; multi-collisions; compression function; tunneling; C#

KARÁSEK, J. *Hashovací funkce - charakteristika, implementace a kolize.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. XY s. Vedoucí diplomové práce Ing. Petra Lambertová.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci na téma „Hashovací funkce – charakteristika, implementace a kolize“ vypracoval samostatně pod vedením Ing. Petry Lambertové a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové 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 Rosicích dne 25. 5. 2009 Bc. Jan Karásek

Poděkování

Děkuji vedoucí diplomové práce Ing. Petře Lambertové za velmi důležitou metodickou pomoc a cenné rady při zpracovávání diplomové práce.

Dále bych také rád poděkoval svým rodičům a prarodičům za stálou podporu během studia, jejich zájem a vytvoření pohodového domácího prostředí při psaní diplomové práce.

V Rosicích dne 25. 5. 2009 Bc. Jan Karásek

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

Hašovací funkce patří mezi prvky moderní kryptografie. Jejich úkolem je na vstupu očekávaná data převést do unikátní bitové posloupnosti. S hašovacími funkcemi přicházíme v počítačovém světě do styku každý den, aniž si to mnozí z nás vůbec uvědomují. Hašovací funkce jsou používány v mnoha aplikačních oblastech, například při přihlašování do operačního systému je vložené haslo hašované a je porovnávána tato haš s haší originálního hesla v databázi sytému, další aplikací může být každodenní podepisování emailových zpráv nebo porovnávání souborů na disku počítače při synchronizačních operacích. Aplikačních oblastí, ve kterých lze hašovací funkce využít je mnoho, je to proto velice zajímavé téma a proto jsem si je vybral pro zpracovávání jako diplomovou práci.

Cílem diplomové práce je charakterizovat hašovací funkce, popsat jejich základní vlastnosti a využití. Dále se zaměřit na jednu hašovací funkci, konkrétně MD5, a tu náležitě popsat. Popsat její konstrukci, bezpečnost a možnosti útoků na tuto funkci. Posledním úkolem je tuto funkci implementovat a implementovat i kolize na ni.

V úvodních kapitolách diplomové práce bych se chtěl zaměřit na obecný popis hašovací funkce, definovat její vlastnosti, základní možnosti jak lze přecházet kolizím a poukázat na aplikační oblasti, ve kterých se použití hašovacích funkcí uplatňuje. Poté by mělo být zřejmé, co to samotná hašovací funkce je, a budou rozebrány charakteristiky jednotlivých typů hašovacích funkcí. Jde o obecné hašovací funkce, založené na metodách: násobení, středních čtverců, dělení a rozkladu. Na základě těchto hašovací funkcí bude uveden konkrétní příklad Fibonacciho hašovací funkce, která je založena na metodě násobení a hašovací funkce FNV, široce prakticky používaná, a bude na ní ukázána i praktická implementace a kolizní test. Dokonalé hašovací funkce budou rozděleny na statické a dynamické a u každé skupiny budou uvedeny její nejznámější zástupci. Dále pak bude uveden výčet soudobých kryptografických hašovacích funkcí, ze kterých byla vybrána jedna, konkrétně MD5, jíž se budu dále věnovat.

Na funkci MD5 bude demonstrována konstrukce moderních hašovacích funkcí, bude poukázáno na její slabosti v podobě složitosti jednotlivých útoků a nakonec bude rozebráno, jak funguje, a prakticky implementována v jazyce C#. Po implementaci funkce budou nastíněny metody útoků na ni, implementovány a bude demonstrována jejich praktická ukázka.

2. Popis hašovací funkce

2.1. Obecný popis hašovací funkce

Obecně se hašovací funkce chápe jako zobrazení h, které přiřazuje libovolně dlouhé vstupní posloupnosti, podle předem definovaného algoritmu, výstupní posloupnost vždy pevné délky. Tato výstupní posloupnost charakterizující zprávu je označována jako otisk zprávy nebo hash zprávy, česky otisk nebo haš.

Základní vlastnosti hašovací funkce:

- Libovolné množství dat poskytuje vždy stejně dlouhý výstup (haš).
- Malou změnou vstupních dat docílíme velkou změnu dat na výstupu (výsledná haš se bude od původní haše významně lišit na první pohled).
- Vysoká pravděpodobnost, že dvě zprávy se stejnou haší jsou totožné.

2.2. Formální popis hašovací funkce

Formálně je hašovací funkce matematická funkce h [2.1], která převádí vstupní posloupnost D bitů (bytů) libovolné délky na posloupnost pevné délky R bitů (bytů).

$$h: \ D \rightarrow R, \quad \text{kde} \ \ |D| > |R| \qquad [2.1]$$

Z definice hašovací funkce plyne existence kolizí, to znamená dvojice vstupních dat (x, y), $x \neq y$ takových, že $h(x) = h(y)$. Slovně popsáno, dvojice různých vstupních dat může mít stejnou haš.

Kolize jsou nežádoucí, ale v principu se jim nelze úplně vyhnout, lze jen snižovat pravděpodobnost jejich výskytu. Cílem je tedy dosáhnout co nejvyšší pravděpodobnosti, že dvě zprávy se stejnou haší jsou totožné.

Pokud by měly kolize být zcela eliminovány, haš, by musela být stejné délky jako vstupní data, čímž by se ale ztratila efektivita, která spočívá v kompresi dat pro další případné zpracování. [10]

2.3. Vlastnosti hašovacích funkcí

Hašovací funkce byla v obecném měřítku označením funkce, která libovolně velkému vstupu přiřazovala výstup pevné délky. Nyní se pojem hašovací funkce používá v kryptografii pro kryptografickou hašovací funkci, která má oproti základním vlastnostem ještě navíc vlastnost jednosměrnosti a rezistence.

Kryptografická hašovací funkce má takové vlastnosti, které umožňují její použití v zabezpečujících aplikacích, např. autentizace nebo zaručení integrity dat.

Vlastnosti hašovací funkce určují obtížnost jejího napadení. Tím je myšlena výpočetní složitost, která by neměla být za současných technologických možností realizována v reálném čase. V literatuře o hašovacích funkcích se objevuje několik typů vlastností, které tyto pojmy upřesňují, např. Jaroslav Pinkava [31] uvádí:

V1. Praktická efektivnost: Pro dané x je výpočet $h(x)$ efektivně proveditelný (přesněji – je proveditelný v čase, který je omezen polynomiální funkcí délky vstupu x).

V2. Mixující zobrazení: Pro každý vstup x má výstupní hodnota "náhodný" charakter (Maova [26] definice je přesnější, vyžaduje však zavedení některých dalších pojmů).

V3. Rezistence vůči kolizím: Je z výpočetního hlediska neuskutečnitelné nalézt dva vstupy x, y $(x \neq y)$, aby $h(x) = h(y)$.

V4. Rezistence prvního vzoru: Pro danou hodnotu haše h je výpočetně neuskutečnitelné nalézt vstupní řetězec x tak, že $h = h(x)$.

V5. Rezistence druhého vzoru: Je výpočetně neuskutečnitelné pro daný vstup x nalézt druhý vstupní řetězec y tak, že $h(y) = h(x)$. Tato vlastnost se od vlastnosti V3 liší tím, že zde je jeden vstup již fixován.

V6. Rezistence vůči blízkým kolizím: Je z výpočetního hlediska neuskutečnitelné nalézt dva vstupy x, y $(x \neq\neq y)$ tak, že $h(x)$ a $h(y)$ se liší jen v malém počtu bitů. Blízké kolize jsou nejjednodušším příkladem zakázaných vztahů mezi výstupy hašovací funkce.

Na základě vlastností V1-V5 je možné definovat dva typy hašovacích funkcí:

OWHF (One-Way Hash Function) – jednosměrné hašovací funkce. Pro ně jsou splněny vlastnosti V1+V2+V4+V5; Jednosměrná hašovací funkce je jednoduše taková funkce, kterou lze snadno vyčíslit, ale je prakticky nemožné z výsledku funkce odvodit její výstup. Ze zadaného x tedy lze snadno získat $f(x)$, avšak výpočet inverzní funkce, získání x při znalosti $f(x)$, je prakticky neřešitelné.

CRHF (Collision Resistant Hash Function) – hašovací funkce rezistentní vůči kolizím. Splňuje podmínky V1+V2+V3. Hašovací funkce rezistentní vůči kolizím je vždy jednosměrnou hašovací funkcí. První tvrzení platí jednoduše, neboť z vlastnosti V3 plyne vlastnost V5. Jednosměrnost funkce rezistentní vůči kolizím se dokazuje sporem.

2.4. Předcházení kolizím

Předcházení kolizím je nelehká záležitost, na kterou musí být brán zřetel již při konstrukci algoritmu hašovací funkce. U kryptografických funkcí se tato problematika řeší komplexním postupem od zesílení řetězce přídavnými bity, přes výběr kvalitní blokové šifry jako kompresní funkce uvnitř hašovací funkce s konečným cílem dosáhnout výsledku s ideálním rozprostřením. Problematice kryptografických funkcí a kolizí se budu věnovat v další části textu. Zde bych rád nastínil obecné metody řešení kolizí, neboť se jedná o nutný teoretický základ k hašovacím tabulkám a s tím souvisejícím dokonalým šifrováním.

Kolize, jak již bylo řečeno v předchozím textu, vzniká v případě, když $h(x) = h(y)$ a $x \neq y$. To znamená, že výsledek hašovací funkce vstupu x, je stejný jako výsledek hašovací funkce vstupu y. V tomto případě řekneme, že ukazují na stejné místo v hašovací tabulce. Existuje několik základních způsobu, jak problém vyřešit:

2.4.1. Zřetězené hašování

V případě, že dojde ke kolizi a dva nebo více klíčů ukazují na místo v tabulce se stejnou adresou a je použito zřetězené hašování, vytvoří se na této adrese zřetězený seznam kolidujících klíčů, který se nachází mimo tabulku v dynamické paměti. Poslední přidaný klíč je vždy umístěn na začátek seznamu, aby se zachoval konstantní čas vkládání do tabulky.

2.4.2. Otevřené adresování a lineární vkládání

Tato metoda se zakládá na sekvenčním hledání volného místa v hašovací tabulce. Pokud dojde ke kolizi haše počítaného ze vstupu *x* funkcí *h(x)* a adresa, kam mají být data vložena, je již obsazena, použije se místo v paměti *h(x)+s*, kde *s* > 0 je krok, který je postupně iterován v závislosti na obsazené paměti.

Při této metodě je třeba dbát na to, aby tabulka nebyla zcela zaplněna a vkládání se tak nestalo nekonečným procesem. Čím více kolizí bude vznikat, tím budou tvořeny delší shluky obsazených buněk v tabulce. Metoda tedy není příliš vhodná pro vkládání hodnot, u kterých po hašování nedochází k širokému rozprostření.

2.4.3. Otevřené adresování a dvojité hašování

Metoda s dvojitým hašováním odstraňuje nedostatky lineárního vkládání. Principem je, že pokud dojde při vkládání ke kolizi, není tvořen shluk hodnot na dané pozici v tabulce, ale je spočítána sekundární hašovací funkce. Druhou hašovací funkci musíme zvolit nanejvýš opatrně, protože by program nemusel vůbec fungovat, pokud by stále docházelo ke kolizím.

2.5. Využití hašovacích funkcí

Hašovací funkce, díky jedné ze svých vlastností – bezkoliznosti[1], slouží jako funkce, která každé jakkoliv dlouhé posloupnosti dat přiřazuje jedinečný identifikátor stejné délky. Pomocí tohoto jedinečného identifikátoru lze tyto funkce efektivně využívat v mnoha oblastech informačních technologií. Dále bude uvedeno několik příkladů, kde hašovací funkce nacházejí své uplatnění.

2.5.1. Ověření integrity

Slovem integrita rozumíme „celistvost“ zprávy. Ověření integrity tedy znamená ověření takové, kdy přijaté informace jsou zcela shodné s informacemi odeslanými.

S ověřováním integrity úzce souvisí detekční a korekční kódy. Hlavním smyslem detekčních kódů je odhalení chyby v přenosu informací. Nejčastěji je využívána technika, kdy je vypočítán kontrolní součet[2] zprávy a ten je poslán s původní zprávou.

---

[1] Bezkoliznost neboli odolnost vůči kolizím je často nepochopeným pojmem. Možných zpráv je mnoho a hašovacích kódů je pouze 2^n, kde n je bitová délka výstupu hašovací funkce. Z uvedeného vyplývá existence množství zpráv, vedoucích na stejnou haš. Kolizí tedy existuje ohromné množství. Pointa je v tom, že nalezení byť jedné kolize je nad soudobé výpočetní možnosti.

[2] Kontrolní součet neboli Message Integrity Code (MIC).

Na straně příjemce je vypočítán ze zprávy nový kontrolní součet a je porovnán s původním. Pokud se liší, integrita byla narušena.

Korekční kód dokáže nejen detekovat chybu v přenosu informací, ale disponuje i vlastnostmi, které mohou data opravit.

Cyklický redundantní součet (CRC – Cyclic Redundancy Check)

Jedná se o speciální hašovací funkci, používanou k detekci chyb během přenosu či ukládání dat. Pro svou jednoduchost a dobré matematické vlastnosti jde o velmi rozšířený způsob realizace kontrolního součtu.

Kontrolní součet je doplňková informace, která bývá odesílána společně s daty, a slouží k ověření, zda při jejich přenosu nebo uchování mohlo dojít k chybě. Po převzetí dat je znovu nezávisle spočítán. Pokud jsou kontrolní součty odlišné, je zřejmé, že při přenosu resp. ukládání došlo k chybě. Pokud jsou shodné, téměř jistě k chybě nedošlo. V určitých případech je možné chybu opravit.

Jednoduchým příkladem kontrolního součtu je použití v daňovém přiznání, nebo ve statistických výkazech, kde je předávána řada čísel a kontrolním součtem je součet všech těchto čísel. CRC je vhodné použít pro zjišťování chyb vzniklých v důsledku selhání techniky, ale pro odhalení záměrné změny dat počítačovými piráty je příliš slabý. [2]

2.5.2. Autentizace informací

Autentizace je proces ověření proklamované identity subjektu. Stěžejním použitím hašovacích funkcí je právě oblast autentizace informací a jejich využití při tvorbě digitálních podpisů zpráv.

Místo toho, abychom ověřovali autentičnost celého obsahu zprávy, která může být v některých případech velice objemná, ověříme pouze autentičnost její haše. Komunikačním kanálem je tedy zpráva posílána nezabezpečená, a zabezpečuje se pouze její haš. Podepisovat lze samozřejmě i celou zprávu, otázkou je však efektivnost takovéhoto postupu. [2]

Autentizační kód zprávy (MAC)

Při autentizaci zprávy jsou využívány tzv. autentizační kódy zpráv, tvořené autentizační funkcí A. Výsledek funkce, autentizační kód, se nazývá MAC (Message Authentication Code) [3.1] a je tvořen podobně jako otisk hašovací funkce. Rozdíl je v tom, že funkce není jen výsledkem zpracovávaných dat Z, ale i tajného klíče K.

$$MAC\ (Z) = A\ (\ Z,\ K\ ) \qquad [2.2]$$

Výhodou kódu je skutečnost, že ho útočník nemůže ani vytvořit ani ověřit, protože nezná tajný klíč *K*. V případě haše tuto možnost má, a proto se zavádí šifrování. [2]

Hašovaný autentizační kód zprávy (HMAC)

Autentizační kód zprávy může být dále ještě vylepšen pomocí hašovací funkce. Výsledek této funkce se nazývá HMAC (Hash MAC). HMAC může být použit stejně jako MAC pro ověření datové integrity nebo autentizaci zprávy. Pro jeho výpočet lze použít funkci MD5 nebo SHA-1. HMAC je definován jako:

$$HMAC_K\ (Z)\ =\ h\ (\!(K \oplus\ opad)\,||\,h(\!(K \oplus ipad)\,||\,Z)\!) \qquad [2.3]$$

V rovnici [3.2], *h* je iterativní hašovací funkce, *K* je tajný šifrovací klíč zalamovaný nulami na velikost základního bloku hašovací funkce, *Z* je zpráva, nad kterou je kód počítán, operace || označuje zřetězení, ⊕ označuje exkluzivní bitový součet[3], *opad* je řetězec 64 bytů 0x5C a *ipad* je řetězec 64 bytů 0x36.

Konstrukce kódu HMAC byla poprvé publikována roku 1996 Mihirem Bellarem, Ranem Canattim a Hugem Krawczykem. Je uložena jako RFC 2104 a její varianty HMAC-SHA-1 a HMAC-MD5 se používají v protokolech IPSec[4] a SSL/TLS[5]. [2]

2.5.3. Kryptografické protokoly

V kryptografii se hašovací funkce používají zejména při vytváření certifikátů a digitálních dopisů, ověřování identity uživatelů nebo přímo jako součást komunikačního protokolu pro zabezpečení komunikace v celé délce jejího trvání.

Hašovací funkce, v podobě HMAC, je použita např. v protokolu IPSec jako zabezpečení procedury pro výměnu klíčů IKE (Internet Key Exchange) a dále také pro

---

[3] Exkluzivní součet, operace se též nazývá nonekvivalence, exkluzivní OR nebo XOR. Jde o logickou operaci, při níž je hodnota pravda, právě když se hodnoty liší.

[4] IPSec (Internet Protocol Security) je průmyslový standard, který je určen k zajištění bezpečnosti informací v komunikačních sítích založených na přenosovém protokolu IP. Zajišťuje autentičnost přenášených paketů a důvěrnost přenášených informací.

[5] Protokol Transport Layer Security (TLS) a jeho předchůdce, Secure Sockets Layer (SSL), jsou kryptografické protokoly, poskytující možnost zabezpečené komunikace na Internetu pro služby jako www, elektronická pošta, internetový fax a další.

kontrolu integrity zprávy. Důležitá je zde odolnost vůči nalezení druhého vzoru. Dále jsou hašovací funkce použity např. v protokolech TLS a SSL nebo SSH[6].

Např. v protokolu S/MIME (Secure Multipurpose Internet Mail Extensions), který je přímo určen pro zabezpečení přenosu emailových příloh, se hašovací funkce používá spolu s kryptografickým systémem založeným na veřejném a tajném klíči. Nebo se využívá přímo v protokolu PGP (Pretty Good Privacy) při šifrování samotných zpráv. [2]

2.5.4. Pseudonáhodné generátory

Pseudonáhodný generátor je efektivní deterministický program, který generuje posloupnosti znaků, které nejsou v ideálním případě statistickými testy rozlišitelné od náhodných posloupností.

Hašovací funkci k tvorbě náhodného, resp. pseudonáhodného šifrovacího klíče z hesla umožňuje využít standard PKCS#5. Při tomto postupu se využívá techniky zasolení a několikanásobného hašování. Výsledkem je krátký klíč *K*, kterého lze využít lépe než původní heslo. Má pevnou délku, můžeme využít tolik bitů, kolik potřebujeme, a má také lepší statistické vlastnosti než heslo původní.

Typické použití hašovacích funkcí jako pseudonáhodných funkcí je v případech, kdy máme k dispozici krátký řetězec dat tzv. *seed*. Může se jednat například o 256 bitový náhodný šifrovací klíč, záznam náhodného pohybu myši na displeji, časový profil náhodných stisků kláves apod. Přitom potřebujeme z tohoto vzorku získat pseudonáhodnou posloupnost o velké délce, například 1 GB. A k promítnutí původního krátkého vzorku do delší posloupnosti se používají právě hašovací funkce. [2]

Například standard PKCS#1 v2.1 definuje pseudonáhodný generátor MGF1 (Mask Generation Function) pomocí hašovací funkce *H* s počátečním, většinou náhodným nastavením *seed* takto:

$$H(seed \,||\, 0x00000000), H(seed \,||\, 0x00000001), H(seed \,||\, 0x00000002)... \qquad [2.4]$$

2.5.5. Komparace souborů

Jiným příkladem použití hašovacích funkcí je porovnávání souborů, bez znalosti jejich obsahu – jsou porovnávány jen jejich haše. Toto využití nachází své uplatnění např. při

[6] SSH (Secure Shell) je program a síťový protokol umožňující bezpečnou komunikaci mezi dvěma počítači pomocí transparentního šifrování a volitelné komprimaci přenášených dat.

verifikaci hesla, jako součást autorizačního procesu nebo při vyhledávání souborů ve známých peer-to-peer sítích, při vyhledávání v databázích apod.

2.5.6. Ukládání hesel

V autentizačních systémech, resp. jejich databázích, se hesla neukládají přímo jako text, ale ukládají se jen jejich haše. Případnému útočníku, tedy zůstává skryto, kdo jaké heslo používá. Tato aplikace haše využívá i tzv. zasolení. Jde o funkci, ve které je k heslu přidán náhodný řetězec a poté je tento celek hašován. Takto jednoduše lze zabránit slovníkovému útoku, nebo útoku hrubou silou.

Ve většině výše uvedených příkladů jsou využity starší, dnes již bezpečnostně nevyhovující, hašovací funkce jako MD2, MD4, MD5, SHA-1. Tyto funkce se stále využívají, protože vznikly, stejně jako samotné protokoly, před delší dobou, kdy ještě nebyla dokázána jejich nespolehlivost.

2.5.7. Hašovací tabulka

Hašovací tabulka je datová struktura, která asociuje hašovací klíč s odpovídajícími daty (hodnotami). Hašovací klíč je výsledkem tzv. hašovací funkce. Používá se pro rychlé vyhledání položky v poli nebo jiném homogenním datovém typu. Homogenní datový typ je takový, kde jsou všechny položky stejného typu.

Pomocí hašovací funkce přiřazujeme hodnotě klíče index (ukazatel) do homogenní datové struktury. Při zápisu obsahu položky zapíšeme položku na odpovídající místo. Pokud je místo obsazeno, pomocí vhodného algoritmu přiřadíme položce další vhodný index v pořadí. Při vygenerování položky spočteme s pomocí klíče index hledané položky. Pokud již bylo odpovídající místo přepsáno položkou s jiným klíčem, opět pomocí vhodného algoritmu prohledáme další položky. Při správném zvolení velikosti homogenní datové struktury a zvolené hašovací funkce má tento algoritmus konstantní složitost[7]. [2]

[7] Konstantní složitost spadá do kategorie asymptotických složitostí. Asymptotická složitost je způsob klasifikace počítačových algoritmů. Určuje operační náročnost algoritmu tak, že zjišťuje, jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat.

3. Charakteristika obecné hašovací funkce

Obecná definice hašovací funkce již byla zmíněna na začátku práce. V této kapitole budou nastíněny základní metody, na kterých je možné hašovací funkci vybudovat a podle kterých lze funkce vzájemně posuzovat. Bude uvedena metoda dělení, metoda středních čtverců, metoda násobení a metoda rozkladu. Jako poslední bude uvedena metoda Fibonacciho hašování, která se zakládá na metodě násobení. [11]

Hašovací funkce můžeme posuzovat podle:

1. Rozložení výsledků hašovací funkce

   Jde o měření, kde se sleduje, jak určitá hašovací funkce rozprostírá svoje výsledky v množině možných výsledků. Analýza s tímto měřením vyžaduje znalost počtu kolizí, které mohou nastat na určené množině dat. Pokud se pro vyřešení kolizí použilo zřetězení[8], je uvažována průměrná délka řetězce.

2. Časová složitost hašovací funkce

   Hašovací funkce by měla být velmi rychlá a deterministická operace. Když je analyzován hašovací algoritmus, obecně se předpokládá, že složitost výpočtu hašovací funkce je konstantní (hašovací tabulky), to ovšem není možné zaručit. Např. při použití v Red-Black stromech[9] je složitost logaritmická. Časová složitost je tedy další vhodný parametr pro posuzování hašovacích funkcí.

[8] Pokud během tohoto způsobu hašování dojde ke kolizi a dva či více klíčů ukazují na stejnou adresu, pak se na této adrese vytvoří zřetězený seznam kolidujících klíčů, který se nachází mimo tabulku v dynamické paměti, přičemž se později přidaný klíč umisťuje na začátek seznamu, aby se zachoval konstantní čas vkládání.

[9] Red-Black strom je binární vyhledávací strom. Jde o datovou strukturu používanou při implementaci asociativního pole. Asociativní pole je pole, kde nejsou prvky indexovány pomocí posloupnosti čísel, ale pomocí klíčů (hašů).

Hašovací funkce jsou typicky definovány způsobem, jakým jsou vytvářeny. Existuje několik základních metod, pomocí kterých je možné vytvořit hašovací funkci:

3.1. Metoda dělení (Division method)

Metoda dělení je základní a nejjednodušší hašovací funkce. Metoda se zakládá na celočíselném dělení čísla x číslem M, zbytek po dělení je použit jako výsledek hašovací funkce.

V některých případech je vhodné zvolit hodnotu M zvláště opatrně. Např. je vhodné zvolit M jako liché číslo, ale to by znamenalo, že pokud je M liché, je i výsledek funkce $h(x)$ lichý, stejně tak pokud je M sudé, je výsledek $h(x)$ sudý. Tato vlastnost by nebyla problém, pokud by byly všechny výsledky hašovací funkce stejně pravděpodobné. Pokud jsou ovšem např. sudé výsledky pravděpodobnější než liché, nebudou výsledky hašovací funkce rovnoměrně rozprostřeny, čili některé budou pravděpodobnější než jiné. Podobně by tomu bylo, pokud by M byla mocnina dvou ($M = 2^k$), kde $k > 1$. V tomto případě hašovací funkce extrahuje spodních k-bitů z binární reprezentace čísla x. Tuto funkci je sice jednoduché vypočítat, ale její výsledek není žádoucí, protože nezávisí na všech bitech původního čísla x.

Z tohoto důvodu jsou jako M volena prvočísla, což zvyšuje pravděpodobnost rovnoměrného rozložení výsledku hašovací funkce, protože dělení závisí na všech bitech čísla x, nikoliv pouze na spodních k-bitech.

Výhodou této metody je, že dělitel nemusí být v čase kompilace konstantní číslo, ale může být vyjádřen až za běhu, časová složitost algoritmu je konstantní. Potenciální nevýhodou může být např. použití v hašovací tabulce, generování za sebou jdoucích mapovacích klíčů, které sice nebudou kolidovat mezi sebou, ale po sobě jdoucí místa v paměti již budou alokovaná. Je tedy vhodné zvážit počet klíčů, jenž bude využit.

V rovnici [4.1] M představuje předem známý dělitel, x je předem známý klíč, který bude hašován, a mod je funkce modulo[10]. [7]

$$h(x) = x \bmod M \qquad [3.1]$$

[10] Funkce modulo nebo také zbytek po dělení je početní operace související s celo-číselným dělením. Např. $7/3 = 2$ se zbytkem 1. Můžeme tedy říci, že $7 \bmod 3 = 1$.

3.2. Metoda středních čtverců (Midsquare method)

Protože dělení je obvykle pomalejší než násobení, můžeme nahrazením dělení za násobení vylepšit časovou náročnost hašovacích algoritmů. Tato metoda pracuje následovně: Mějme M, které je k-tou mocninou dvou ($M = 2^k$), pro $k >= 1$. Potom výsledek hašovací funkce $h(x)$ je výsledkem funkce:

$$h(x) = \left\lfloor \frac{M}{W} \left(x^2 \bmod M \right) \right\rfloor \qquad [3.2]$$

V rovnici [4.2] M i W jsou mocniny dvou a jejich poměr $W/M = 2^{w-k}$ je také mocninou dvou. Abychom mohli vynásobit výraz (x^2 *mod* W) výrazem M/W, jednoduše posuneme bity doprava o w-k. Ve skutečnosti vyextrahujeme k-bitů ze středu mocniny klíče. Odtud také pochází jméno metody.

Metoda středních čtverců si vede dobře, pokud jsou všechny klíče stejně pravděpodobné. Metoda je také charakteristická tím, že rozprostře po sobě jdoucí klíče. Protože metoda počítá jen s podmnožinou bitů uprostřed klíče, klíče mající velké množství nul od počátku mohou vzájemně kolidovat. [27]

3.3. Metoda násobení (Multiplication method)

Velmi jednoduchá variace metody středních čtverců, která zmírňuje její nedostatky, se nazývá metoda násobení. Místo násobení klíče x sebou samým násobíme klíč x pečlivě zvolenou konstantou a, potom vyextrahujeme prostředních k-bitů výsledku.

$$h(x) = \left\lfloor \frac{M}{W} \left((a \cdot x) \bmod M \right) \right\rfloor \qquad [3.3]$$

Jak je vhodné zvolit konstantu a? Jestliže se chceme vyhnout problému, který je spojen s klíči generovanými metodou středních čtverců, tedy problému s velkým počtem nul na začátku nebo na konci, musíme zvolit a takové, které nemá žádné nuly na začátku nebo na konci. Mimoto, jestli vybereme a takové, které je nesoudělné s W, pak existuje číslo: a'. Jinými slovy, a' je inverzní výsledku (a *mod* W). Toto počítaní má tu

vlastnost, že pokud vezmeme klíč x a vynásobíme jej a, dostaneme ax, můžeme obnovit původní klíč x násobením výsledku ax inverzní hodnotou a', viz rovnice [4.4].

$$axa' = aa'x = 1x \qquad [3.4]$$

Existuje mnoho konstant, které mají tyto požadované vlastnosti, avšak jedna hodnota, která se využívá pro 32 bitová čísla ($W = 2^{32}$) je a = 2654435769, binárně = 10011110001101110111100110111001. Toto číslo nemá žádné nuly na počátku ani na konci. Dále, hodnota a a $W = 2^{32}$ jsou nesoudělná a hodnota a' = 340573321. [28]

3.4. Metoda rozkladu (Folding method)

Metoda rozkladu dělí klíč x do několika skupin (a,b,c) tak, že každá část má stejný počet číslic jako požadovaný hašovací klíč. Suma všech částí tohoto klíče je dělena M, což je velikost hašovacího prostoru (tabulky). Výsledkem je hašovací klíč, viz rovnice [4.5]. [17]

$$H(x) = (a + b + c) \bmod M \qquad [3.5]$$

3.5. Fibonacciho hašování

Ve skutečnosti je Fibonacciho hašování založené na metodě násobení, která využívá speciální hodnoty pro konstantu a. Hodnota a úzce souvisí se zlatým řezem[11]. Zlatý řez může být definován následovně. Máme dvě čísla x a y a poměr $\Phi = x/y$ je zlatý řez, pokud poměr x ku y je stejný jako $x+y$ ku x. Zlatý řez lze pomocí rovnice [4.6] vyjádřit takto:

$$\begin{aligned} \frac{x}{y} = \frac{x+y}{x} &\Rightarrow 0 = x^2 - xy - y^2 \\ &\Rightarrow 0 = \phi^2 - \phi - 1 \\ &\Rightarrow \phi = \frac{1+\sqrt{5}}{2} \end{aligned} \qquad [3.6]$$

[11] Zlatý řez (Golden ratio), je poměr o hodnotě přibližně 1 : 1,618. Zlatý řez vznikne rozdělením úsečky na dvě části tak, že poměr malé části k větší je stejný jako poměr větší části k celé úsečce. Zlatý řez se využívá v mnoha vědních oblastech.

Ve vzorci je úzká souvislost mezi zlatým poměrem a Fibonacciho čísly[12]. Fibonacciho čísla získáme ze vztahů [4.7], [4.8], [4.9]:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \qquad [3.7]$$

$$\phi = \frac{\left(1+\sqrt{5}\right)}{2} \quad \text{a} \quad \hat{\phi} = \frac{\left(1-\sqrt{5}\right)}{2!} \qquad [3.8], [3.9]$$

V tomto kontextu, kde se jedná o Fibonacciho hašování, nás nebude zajímat hodnota Φ, ale její inverzní hodnota Φ^{-1}, kterou lze spočítat podle rovnice [4.10]:

$$\phi^{-1} = \frac{2}{1+\sqrt{5}} = \left(\frac{2}{1+\sqrt{5}}\right)\cdot\left(\frac{\sqrt{5}-1}{\sqrt{5}-1}\right) = \frac{\sqrt{5}-1}{2} \approx \underline{0{,}618033887} \qquad [3.10]$$

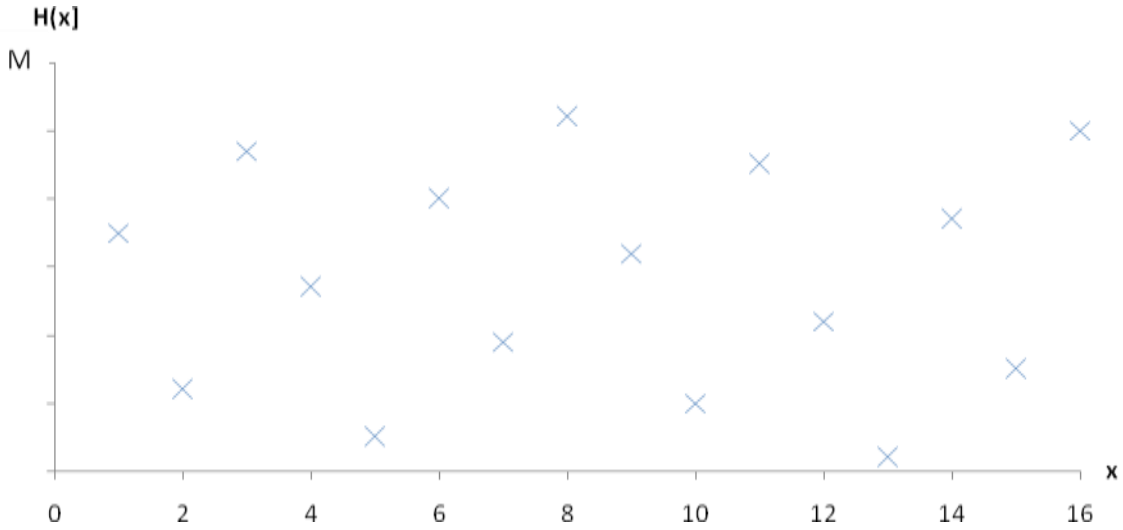
Fibonacciho hašování je v zásadě metoda násobení, ve které je konstanta a vybrána jako celé číslo nesoudělné s W, které je nejblíže $\Phi^{-1}W$. Následující tabulka ukazuje vhodné hodnoty konstanty a pro různé délky slova W.

W	a
2^{16}	40503
2^{32}	2654435769
2^{64}	11400714819323198485

Tab. 1: Hodnoty a odpovídající délkám slova W

Pokud pečlivě zvolíme konstantu a, bude výrazně ovlivněno rozprostření výsledných hodnot hašovací funkce. Jak je vidět z grafu 1, klíče jsou rovnoměrně rozloženy po celé množině. Ve skutečnosti, pokud využijeme, že $a = \Phi^{-1}W$, výsledná hodnota hašovací funkce padne mezi dva již vytvořené haše. Právě zde se uplatňuje vlastnost zlatého poměru. Každá následující zhašovaná hodnota dělí interval, do kterého padne, přesně podle tohoto poměru. [14]

[12] Fibonacciho posloupnost je v matematice nekonečná posloupnost přirozených čísel začínající 0, 1, 1, 2, 3, 5, 8, 13, 21, ... (čísla nacházející se ve Fibonacciho posloupnosti se někdy nazývají Fibonacciho čísla), kde každé číslo je součtem dvou předchozích.



Graf 1: Rozložení Fibonacciho hašování

3.6. FNV hašovací funkce

Jednoduché hašovací funkce, postaveny na zmiňovaných metodách budou později v práci demonstrovány i prakticky. Tyto funkce jsou používány pro hašování dat v hašovacích tabulkách a strukturách programovacích jazyků. Nejsou kryptograficky bezpečné a jsou náchylné ke kolizím.

Jako zástupce těchto funkcí uveďme funkci FNV (Fowler/Noll/Vo). Základní myšlenka byla převzata z metody posílání recenzentských komentářů pro IEEE POSIX P1003.2, kterou navrhli Glenn Fowler a Phong Vo. Funkce byla následně ještě upravena do dnešní podoby Landonem Nollem. Funkce je velice rychlá a generuje jen nízký počet kolizí v porovnání s ostatními funkcemi tohoto typu.

Praktické využití funkce můžeme nalézt v implementacích doménových serverů, v systémech řízení báze dat, při indexování databáze, při indexování datových struktur obecných i při praktickém použití, např. v unixových implementacích souborového systému NFS, dále třeba v PHP 5.x při hašování cache a také Microsoft používá tuto funkci k implementaci hašových map v jazyku Visual C++ 2005. [8], [28]

4. Charakteristika dokonalé hašovací funkce

Jedná se o specifickou variantu hašovacích funkcí. Předpokládejme, že existuje, množství klíčů S. Potom bychom mohli najít takovou hašovací funkci, která pro danou množinu nebude mít ani jednu kolizi. Minimální dokonalé hašování garantuje, že n klíčů bude mapováno od 0 až $(n-1)$ haší bez kolize.

Dokonalá hašovací funkce se dělí na statické a dynamické, podle toho, zda se množina S v době existence dokonalé hašovací funkce mění.

U statických metod dokonalého hašování existuje stálá, neměnná množina klíčů, které budou hašovány. Statické metody hašování se proto využívají například v takových aplikacích, kde existuje instalace programu na pevném disku počítače a nějaká externí databáze údajů a počet těchto údajů je předem znám. Jako zástupce statických funkcí bude uvedeno Cormackovo a Larson-Kalja hašování.

U dynamických hašovacích funkcí není předem známo, kolik klíčů bude hašováno, tento počet se může v čase libovolně měnit. Je tedy složitější zajistit unikátnost zhašovaných hodnot. Jako zástupci dynamických funkcí budou uvedeny rozšiřitelné a lineární hašovací metody, které jsou základem pro další. [32]

Vlastnosti dokonalých hašovacích funkcí:

- Nenastávají kolize
- Používají se pro hašování klíčových slov kompilátorů, slovníků apod.
- Je možné realizovat, pokud přesně víme, jaká množina znaků bude hašována

4.1. Cormackovo hašování

Cormackovo hašování se v dnešní době uplatňuje ve spoustě aplikací. Je založeno na existenci primární hašovací funkce $h(x)$ a celé třídě sekundárních hašovacích funkcí $h_i(x,r)$. Funkce $h(x)$ musí mít obor hodnot roven velikosti adresáře. Položky se ukládají do primárního souboru (pevné velikosti) způsobem, který bude popsán za pomoci adresáře (také pevné velikosti). Protože primární soubor i adresář mají pevnou velikost, řadí se Cormackovo hašování do statických metod dokonalého hašování.

Při vytváření klíčů se nejprve vypočítá primární hašovací funkce. Pokud již existuje nějaká položka se stejným klíčem, snažíme se najít takovou sekundární hašovací funkci, která vrátí rozdílný hašovací klíč pro nově vkládanou položku a pro všechny ostatní položky v datovém bloku odkazů do primárního souboru.

Při vyhledávání začínáme analogicky vypočtením hodnoty primární hašovací funkce a sledujeme počet odpovídajících položek. Odpovídá-li danému klíči více položek v primárním souboru, použije se sekundární hašovací funkce daného bloku a tímto způsobem je nalezen odpovídající klíč primárního souboru. Často používanou sekundární hašovací funkcí je:

$$H_i(x,r) = (x \bmod (2i + 100r + 1)) \bmod r, \quad x >> 2i + 100r + 1 \qquad [4.1]$$

V rovnici [5.1] je, x vstupní klíč, i je číslo použité hašovací funkce a r je počet již existujících položek stejné haše v primárním souboru a mod je funkce modulo. [3]

4.2. Larson-Kalja hašování

Hašovací funkce je založena na otevřeném adresování s dvojitým hašováním[13]. Používá tabulku pomocných informací, která je podstatně menší než u Cormackova hašování. Vyžaduje $M*d$ bitů, kde M je počet stránek adresného prostoru a d je velikost pomocné datové struktury. Dále se používá M hašovacích funkcí h_i, které pro každý klíč vrací postupně adresy těch stránek, na kterých je možné hledat daný záznam. Dalších M hašovacích funkcí s_i generuje výsledné klíče dat k.

Pokud chceme vložit záznam k, pomocí $h_i(k)$ zjistíme umístění stránky. Potom záznam zhašujeme pomocí funkce $s_i(k)$, čímž dostaneme výsledný klíč k. Porovnáme adresy stránek s výsledkem funkce $h_i(k)$, pokud najdeme takovou, kam je možné klíč vložit, vložíme jej. Tyto klíče budou seřazeny od největšího po nejmenší. V případě, že by mělo být záznamů na stránce větší množství než je velikost stránky, budou vyjmuty záznamy postupně od nejvyššího klíče, který je dán funkcí $s_i(k)$. Jejich klíč bude nastaven jako separátor nové stránky a vybrané hodnoty budou znovu zhašovány.

[13] Otevřené adresování s dvojitým hašováním - princip je v podstatě podobný jako u lineárního vkládání s tím rozdílem, že pokud dojde ke kolizi, neposuneme se na následující pozici v tabulce, jak je tomu u lineárního vkládání, ale použijeme druhou hašovací funkci.

4.3. Rozšířitelné hašování

Rozšířené hašování, vyvinuté týmem Ronalda Fagina v roce 1979, je založeno na vstupu x a hašovací funkci $h(x)$, u které je předpoklad, že bude hašovaná data rozprostírat rovnoměrně. Dále máme datovou strukturu - adresář o velikosti d, která je nejvýše rovna logaritmu z maximální hodnoty výstupu hašovací funkce. Adresář obsahuje na jednom řádku položku d bitů a položku s ukazatelem na stránku s daty.

Při vyhledávání dat spočítáme haš, podíváme se na prvních d bitů výsledku a najdeme odpovídající řádek v adresáři, kterých může být i více, a nalezneme na něm ukazatel ukazující na patřičnou stránku s daty. Pokud vkládáme data, může nastat situace, kdy na některé ze stránek dojde místo, čili stránka bude mít stejnou velikost jako adresář. V takovém případě zdvojnásobíme délku adresáře přidáním dalšího bitu. Stránku, kde došlo k přetečení, rozdělíme do dvou nových stránek, a aktualizujeme ukazatele v adresáři. Naopak při mazání záznamu, lze stránky, které se liší pouze v posledním bitu, slévat. [13]

4.4. Lineární hašování

Lineární hašování vyvinuté Witoldem Litwinem roku 1980 je nepoužívanější dynamická metoda používaná v hašovacích tabulkách. Základní odlišností od jiných metod je, že metoda nepoužívá pomocnou datovou strukturu – adresář, ale tzv. oblast přetečení.

Na začátku jsou všechna data ukládána do jedné primární stránky a do oblasti přetečení, která je přítomna u každé stránky. Po provedení n operací vkládání stránku rozdělíme na dvě podle posledního bitu výsledku hašovací funkce. Po dalších n operacích vkládání bude plná stránka rozdělena na další dvě podle posledních dvou bitů. Uveďme tedy příklad dělení stránek: nejprve se první primární stránka rozdělí na dvě, a to na 0, 1. Stránka 0 bude poté rozdělena na 00 a 10, stránka 1 na 01 a 11 a tímto způsobem bude rozklad pokračovat.

Hledání je provedeno následovně. Máme aktuálně m stránek, vezmeme posledních $\lfloor \log_2 m \rfloor + 1$ bitů, pokud je výsledek $> m$, zanedbáme horní bit a dostaneme číslo stránky. Pokud na stránce data nejsou, musíme zkontrolovat oblast přetečení. [23]

5. Charakteristika kryptografické hašovací funkce

Vlastnosti kryptografických hašovacích funkcí byly již popsány v první kapitole. Pro připomenutí zdůrazněme, že kryptografickou hašovací funkcí je taková funkce, mezi jejíž vlastnosti patří jednosměrnost a rezistence vůči kolizím.

V této kapitole charakterizuji soudobé i historicky důležité hašovací funkce, které nějakým způsobem ovlivnily, nebo byly použity jako základ pro další funkce. Tyto charakteristiky poslouží v dalším zpracování diplomové práce tak, že bude vybrána jedna z uvedených funkcí, na které bude podrobně vysvětlena její konstrukce, implementace a metody kolizí. Implementace funkce a kolizí bude provedena prakticky a náležitě popsána v dalších kapitolách.

5.1. MD funkce

5.1.1. MD

Jde o řešení, vyvinuté Ronaldem Rivestem, profesorem Massachusettského technologického institutu a jedním z vynálezců krypto-systému RSA. Funkce nikdy nebyla veřejně publikována a ani nenašla široké využití. Zkratka MD představuje „Message Digest“, čili česky podpis zprávy, otisk nebo termín použitý v této diplomové práci – haš.

5.1.2. MD2

Jedná se o první publikovaný algoritmus z rodiny funkcí MD, určený pro 8bitové procesory s malou pamětí. Hodí se pro embedded zařízení. Byl navržen roku 1989 Ronaldem Rivestem a publikován v dubnu 1992 jako RFC 1319.

Hašování tímto algoritmem probíhá ve třech fázích. V první fázi je řetězec, který má být hašován, zarovnán na délku dělitelnou 16ti. Ve druhé fázi se počítá 16 bytový kontrolní součet, který je připojený ke konci zarovnané zprávy. Ta je poté rozdělena do 16bytových bloků. Třetí fáze sestává z opakování kompresní funkce, která počítá nové hodnoty pro zřetězení. Zřetězuje se vždy hodnota aktuální proměnné a blok zprávy, který je na řadě. Inicializační hodnota proměnné, která je v každém cyklu aktualizována, je pevná a je součástí algoritmu. Délka výstupního klíče je 128 bitů.

V roce 1997 byl pány Rogierem a Chauvaudem popsán útok na kompresní funkci algoritmu. Ten ale nebyl rozšířen na celý algoritmus. Až v roce 2004 byl demonstrován útok s časovou složitostí 2^{104} a od té doby hašovací funkce již nemůže být považována za bezpečnou. [4], [34]

5.1.3. MD3

Algoritmus nebyl nikdy publikován a byl téměř okamžitě nahrazen algoritmem MD4.

5.1.4. MD4

Čtvrtý ze série algoritmů MD, navržený v říjnu roku 1990 profesorem Ronaldem Rivestem z univerzity MIT. Funkce pracuje rychleji než MD2, používá 32 bitové operace. Návrh tohoto algoritmu ovlivnil i v pořadí následující hašovací funkce, jako např. MD5, SHA a RIPEMD. Algoritmus se využívá k výpočtu haše hesla v systémech Microsoft Windows NT, XP a Vista. Varianta tohoto algoritmu se používá k poskytování jedinečných identifikátorů pro soubory ve výměnných sítích, jako jsou eDonkey2000, eMule a dříve byl tento algoritmus použit i v protokolu rsync[14].

Na vstupu přijímá funkce blok dat do velikosti 2^{64} bitů. Vstup je nejdříve zarovnán a poté rozdělen na bloky o délce 512 bitů. Algoritmus pracuje s bloky dat o délce 128 bitů, rozdělenými na čtyři, stejně dlouhá bitová slova. V každém cyklu hašovací funkce je zpracováván 512 bitový vstup. K výpočtu je využito 3 rund, kde každá z nich obsahuje 16 podobných matematických operací, založených na nelineární funkci *f*, bitovém exkluzivním součtu a levé bitové rotaci. Nakonec je vyprodukována haš o délce 128 bitů.

Slabina algoritmu MD4 byla demonstrována Denem Boerem v roce 1991. První útok kolizí byl proveden Hansem Dobbertinem v roce 1996 a v roce 2004 Xiaoyun Wang našel velmi efektivní způsob útoku kolizí na celou řadu hašovacích funkcí postavených na designu MD4. Později byl útok ještě zdokonalen týmem profesora Sasakiho a generování kolizí se stalo tak snadným jako jejich ověřování. [35]

[14] rsync je počítačový program původně vyvinutý pro unixové systémy, který synchronizuje soubory a adresáře mezi různými lokacemi za použití co nejmenšího přenosu dat. Pokud je to možné, přenáší pouze rozdíly mezi soubory.

5.1.5. MD5

Jedna z nejpoužívanějších hašovacích funkcí, navržena již v roce 1991 jako náhrada za MD4 a publikována v dubnu roku 1992. Algoritmus se široce využívá ve spoustě různých aplikací.

Funkce má podobnou konstrukci jako funkce MD4. Na vstupu přijímá vstupní data do velikosti 2^{64} bitů. Vstup je zarovnán a rozdělen na bloky dat o velikosti 512 bitů. Stejně jako předešlá funkce, pracuje i tato s bloky dat o délce 128 bitů rozdělenými do čtyř 32 bitových slov. K výpočtu je zde použito 4 rund, každá má 16 operací, které jsou založeny na nelineární funkci *f*, exkluzivním bitovém součtu a levé bitové rotaci. Výsledná haš má stejně jako MD4 délku 128 bitů.

První útok na hašovací funkci MD5 byl proveden roku 1993 Denem Boerem a Antoonem Bosselaersem. Publikovali výsledek hledání pseudo-kolizí na MD5 kompresní funkci, kde dva rozdílné inicializační vektory mohou produkovat stejnou haš. O tři roky později profesor Dobertin oznámil kolizi kompresní funkce MD5. I když nešlo o útok na celou hašovací funkci, bylo doporučeno používat alternativní hašovací funkce jako Whirlpool, SHA-1 a RIPEMD-160. V roce 2004 byly nalezeny další nedostatky pod vedením doktorky Wangové, které ještě více zpochybnily použitelnost MD5. V roce 2005 Wangová, Lanstra a Weger ukázali vytvoření dvou certifikátů X.509 s různými veřejnými klíči a stejnou MD5 haší. Dále pak na výzkum Dr. Wangové navázal český kryptolog Vlastimil Klíma a publikoval 18. března 2006 algoritmus, který byl schopen tyto kolize hledat do minuty na obyčejném notebooku. Využil při tom metodu, kterou nazval tunelování. [18-22] [36]

5.2. SHA funkce

5.2.1. SHA-0 a SHA-1

První specifikace algoritmu známého jako SHA-0 byla zveřejněna v roce 1993 jako Secure Hash Standard, FIPS PUB 180. Tuto specifikaci zveřejnila agentura NIST (National Institute of Standards and Technology) ze Spojených států amerických. Tato specifikace byla krátce po své publikaci označena NSA (National Security Agency) za nedostatečně bezpečnou a byla v roce 1995 nahrazena dalším standardem FIPS PUB 180-1, který je znám pod zkratkou SHA-1. SHA-1 se liší od SHA-0 jen jednou bitovou rotací v kompresní funkci. Tuto opravu navrhla NSA, aby byla plně zachována kryptografická bezpečnost šifry, ale nepublikovala žádnou zprávu, či bližší vysvětlení o slabině hašovací funkce. Zanedlouho byla nahlášena slabina i v upravené hašovací

funkci SHA-1. Funkce jsou založeny na podobných principech jako MD4 a MD5 navržené Ronaldem Rivestem.

Oba algoritmy produkují klíč o velikosti 160 bitů, který je vytvořen ze zprávy o nejvyšší přípustné délce $2^{64} - 1$ bitů. Vstup je nejprve zarovnán, rozdělen do bloků o délce 512 bitů. Funkce pracuje s bloky o velikosti 160 bitů, které jsou rozděleny na 5 bitových slov, tedy o velikosti po 32 bitech. Algoritmus v každém bloku zpracovává 512 bitový vstup, pomocí kterého modifikuje kontext. Vstupní blok dat je v každém cyklu rozšířen na 80x32 bitových slov. Každý blok se skládá ze 4 rund, kde každá obsahuje 20 podobných operací založených na nelineární funkci f, exkluzivním bitovém součtu a levé bitové rotaci. [15]

5.2.2. SHA-2

Institut NIST publikoval další čtyři hašovací funkce v rodině SHA, všechny jsou označovány zkratkou SHA-2 – tato zkratka nikdy nebyla standardizována. Jednotlivé varianty, označené podle bitové délky jejich haší jsou SHA-224, SHA-256, SHA-384 a SHA-512. Poslední tři byly navrženy a schváleny v roce 2001 jako standard FIPS PUB 180-2 a veřejně publikovány jako oficiální standard v roce 2002. V únoru roku 2004 byla specifikace doplněna o funkci SHA-224, která měla splňovat kriteria pro použití v šifře TripleDES. Tyto funkce byly patentovány a uvolněny pod bezplatnou licencí.

Hašovací funkce SHA-256 pracuje s 32 bitovými slovy, hašovací funkce SHA-512 s 64 bitovými slovy. Při výpočtu využívají různou velikost posuvu a počet rund, jinak jsou v podstatě totožné. Další dvě funkce SHA-224 a SHA-384 jsou jednoduše zkrácené verze prvních dvou funkcí.

Funkce SHA-256 resp. SHA-224 přijímá na vstupu blok dat do velikosti 2^{64} bitů. Vstup je zarovnán a rozdělen na bloky o velikosti 512 bitů. Algoritmus pracuje s bloky o velikosti 256 bitů, které jsou rozděleny do osmi bitových slov. V každém bloku je zpracován 512 bitový vstup, jímž je modifikován kontext a k výpočtu je využito 80ti podobných operací, založených na nelineárních funkcích. Podobně je tomu u funkcí SHA-512 resp. SHA-384, které se liší tím, že přijímají na vstupu data do velikosti 2^{128} bitů, používají 64 bitová slova, jiné nelineární funkce, konstanty a inicializační vektory.

U funkcí SHA-2 dochází k daleko komplikovanějšímu výpočtu a expanzi vstupního bloku zprávy, než je tomu u funkcí SHA-1 či MD5. Dosud nejsou známé žádné úspěšné útoky na tyto funkce, a jsou tak považovány za bezpečné. [16]

5.3. RIPEMD

Jedná se o rodinu hašovacích funkcí navrženou Hansem Dobbertinem, Antoonem Bosselaersem a Bartem Preneelem. Zkratka RIPEMD znamená Race Integrity Primitives Evaluation Message Digest. Původně byla vyvinuta funkce RIPEMD v roce 1992 a její algoritmus byl velice podobný hašovací funkci MD4. Tato původní funkce produkuje výstup o délce 128 bitů. V roce 1996 byly navrženy další funkce rozšiřující tuto rodinu.

RIPEMD-128 byla navržena kvůli kompatibilitě s aplikacemi, které pracují s délkou klíče 128bitů. Už v té době byly známy slabiny hašovacích funkcí s takovouto délkou klíče a byl doporučen přechod na 160 bitovou verzi algoritmu. RIPEMD-160 měla nahrazovat již nedostatečné 128bitové hašovací funkce, jako jsou MD4, MD5. RIPEMD-256 a RIPEMD-320 jsou rozšíření funkce a mohou být použity v aplikacích, které vyžadují delší klíč pro větší bezpečnost.

Obecně funkce RIPEMD-xxx na vstupu přijímá blok dat do maximální velikosti 2^{64} bitů, který je doplněn na určitou velikost a rozdělen na bloky o stejné délce, která činí 512 bitů. Funkce RIPEMD-128/256 pracují s bloky dat o velikosti 256 bitů a funkce RIPEMD-160/320 s bloky o velikosti 320 bitů. Algoritmy všech funkcí zpracovávají paralelně dva výpočty, které se na konci každého bloku sečtou. [37]

5.4. Haval

Haval je hašovací algoritmus, který navrhli roku 1992 Yuliang Zheng, Josef Pieprzyk a Jennifer Seberry. Poslední úpravy na tomto algoritmu byly provedeny v roce 1997, kdy byla nalezena kolize na algoritmu s délkou klíče 128 bitů. Od té doby nebyl nahlášen ani jeden úspěšný útok na tyto funkce. Algoritmus podporuje 15 různých úrovní zabezpečení, které se liší délkou klíče (128/160/192/224/256 bitů) a volitelným počtem výpočetních cyklů funkce (3, 4, 5). Jako výchozí je považován algoritmus s délkou klíče 256 bitů a 5 cykly výpočtu.

Funkce předpokládá jako vstupní blok data menší než 2^{128} bitů, poté je blok dat doplněn na určitou délku v závislosti na tom, jaká funkce je použita, tj. kolika-bitový výstup funkce produkuje a takto upravená data jsou rozdělena do bloků o délce 512 bitů. Algoritmus poté pracuje s bloky dat o velikosti 256 bitů a výsledná haš vznikne výpočtem n cyklů. [12]

5.5. Tiger

Hašovací funkce navržena roku 1995 Rossem Andersonem a Eli Bihamem. Je speciálně určena pro 64 bitové platformy. Velikost výstupu je 192 bitů. Má dvě další verze, vytvářející 128 a 160 bitový výstup, známe jako Tiger/128 a Tiger/160, kde je původní haš zkrácena na požadovanou délku.

Funkce na vstupu očekává blok dat o maximální možné délce 2^{64} bitů. Jak již bylo zmíněno, standardní výstupní délka haše je 192 bitů. Je založena na podobném principu jako MD5, což je iterativní výpočet s nelineárními funkcemi. V roce 2006 byl publikován výzkum, který odhalil kolize pro funkci omezenou 16-ti průchody a později 20-ti průchody, kde bylo možné nalézt pseudo-kolize. Poté již funkce nebyla prolomena.

Existuje varianta Tiger2, která užívá stejné zarovnání hašované hodnoty jako MD5 nebo SHA. Bohužel tato varianta doposud nebyla veřejně publikována. [39]

5.6. Snefru

Snefru je kryptografická hašovací funkce vyvinutá Ralphem Merklem pro firmu Xerox, která podporuje 128bitový a 256bitový výstup. Funkce nese jméno po egyptském faraonovi, stejně jako blokové šifry Khufu a Khafre od tohoto autora.

Nalezení kolizí prokázali, pomocí diferenční kryptoanalýzy, izraelští kryptologové Eli Biham a Adi Shamir. Algoritmus funkce byl poté modifikován zvýšením počtu iterací hlavního algoritmu ze dvou na osm průchodů. Ačkoliv diferenční kryptoanalýza může prolomit upravený algoritmus rychleji než útok brutální silou, je stále vyžadováno $2^{88,5}$ operací, a tak tento útok je prozatím v praxi nepoužitelný.

Zpráva je rozdělena do minimálně (512 - m) bitových slov a každé z nich je smícháno s výslednou hodnotou hašovací funkce *H*. Funkce bere jako vstup 512 bitový vzorek dat a produkuje *m*-bitový výstup. Funkce *H* je založena na reverzibilní funkci *E* (vstup i výstup je 512bitů) a vrací výsledek funkce XOR prvních *m* bitů vstupu a posledních *m* bitů výstupu. Funkce *E* generuje náhodná data v několika průchodech. Každý z průchodů se skládá z 64 rund[15], kde v každém z nich je použit jiný byte dat jako vstup do další funkce, jejíž výstup je XORován se sousedními uzly. [6]

[15] Jako runda je označován elementární blokový šifrátor, který vstupním bitům na základě šifrovacího klíče přiřazuje výstupní blok bitů.

5.7. Whirlpool

Whirlpool je hašovací funkce navržena Vincentem Rijmenem a Paulem S. L. M. Barretem. Funkce byla postupně vyvíjena a měla tři verze. První, Whirlpool-0 byla podrobena analýze v projektu NESSIE[16]. Druhou, vylepšenou verzí byla Whirlpool-T, která byla zahrnuta do portfolia kryptografických primitiv tohoto projektu. V této druhé verzi byla ovšem nalezena chyba, která se týkala schopnosti rozptylu výsledků funkce. Ta byla posléze opravena a výsledná hašovací funkce nese prosté jméno Whirlpool. Tato výsledná funkce byla též přijata jako standard organizací ISO pod označením ISO/IEC 10118-3:2004.

Funkce pracuje velmi podobně jako algoritmus AES[17]. Operuje se zprávami nejvýše do velikosti 2^{256} bitů a produkuje klíč o celkové délce 512 bitů. Funkce využívá blokovou šifru, kde je nejprve vstupní řetězec doplněn sekvencí jedniček, poté sekvencí nul a nakonec je zarovnán vstupními daty. Zarovnaná zpráva je rozdělena do bloků o délce 512 bitů ($m_1, m_2, \ldots, m_n$), které jsou pak využity pro generování pomocných hašovacích klíčů ($H_0, H_1, \ldots, H_n$). H_0 je definován jako řetězec nulových bitů. Klíč H_i se vypočítá takto: bloková šifra zašifruje blok m_i klíčem H_{i-1} a výsledek XORuje s oběma vstupy do blokové šifry, tedy m_i a H_{i-1}. Výsledek hašovací funkce je H_n. [39]

5.8. Panama

Panama je kryptografická funkce, která může být užita stejně dobře jako hašovací funkce nebo jako proudová šifra. Byla navržena Joanem Daemenem a Craigem Clappem a prezentována v roce 1998 na konferenci FSE (Fast Software Encryption).

Funkce pracuje na principu posuvného registru s posouváním bitového slova o velikosti 32x32 bitů a s kruhovou mixující funkcí pracující se slovy o velikosti 17x32 bitů. Na vstupu funkce přijímá blok dat o velikosti 2^{64} bitů a výsledkem funkce je blok dat o velikosti 256 bitů.

Na konferenci FSE v roce 2007 byl představen praktický útok na funkci, který se složitostí 2^6 umožnil najít kolizní pár, a tato funkce se stala nevhodnou k praktickému použití. [30]

---

[16] NESSIE (New European Schemes for Signatures, Integrity and Encryption) byl Evropský výzkumný projekt v letech 2000-2003, který měl prověřit bezpečnost kryptografických primitiv.

[17] AES – Advanced Encryption Standard. Jedná se o symetrickou blokovou šifru, nahrazující standard DES, vyvinutou Belgičany. Vyznačuje se vysokou rychlostí šifrování, prací s bloky o délce 128 bitů a délkou klíče 128/192/256 bitů.

5.9. GOST

Hašovací funkce publikovaná v roce 1994, definována standardy GOST R 34.11-94 a GOST 34.311-95. Původně byla definována jako ruský národní standard a je jedinou kryptografickou hašovací funkcí, která může být užita v ruském digitálním podpise. Je také zmíněna v několika RFC dokumentech a implementována v rozličných kryptografických aplikacích (například jako instance OpenSSL).

GOST je hašovací funkce, pracující na principu iterace a produkující 256bitový výsledný klíč. Na rozdíl od nejpoužívanějších hašovacích funkcí, jako je MD5 a SHA funkce, GOST definuje kromě běžné struktury iterace i výpočet kontrolního součtu nad všemi vstupními bloky zprávy. Tento kontrolní součet je součástí výpočtu finálního klíče. GOST standard také specifikuje blokovou šifru, která je hlavním stavebním blokem hašovací funkce. [25]

5.10. HAS-160

Funkce pracuje dohromady s algoritmem užitým v korejském digitálním podpise. Jedná se v podstatě o derivát funkce SHA-1 s rozmanitými změnami, které by měly zvýšit její bezpečnost. Produkuje 160bitový výstup. Bohužel není mnoho dokumentů dostupných o této funkci, ať už v anglickém nebo jiném světovém jazyku. [1]

5.11. LM Hash

LM Hash neboli Lan Manager Hash je jeden z formátů pro ukládání hesel, který před příchodem Windows Vista používala firma Microsoft. Samozřejmě je kvůli zpětné kompatibilitě integrován i ve Windows Vista. Ovšem musí být explicitně povolen.

LM Hash je počítán následovně. Uživatelské heslo v textové podobě je převedeno na velké znaky. Poté je doplněno nebo zarovnáno na 14 bytů a rozděleno na dvě poloviny. Hodnoty jsou použity pro vytvoření dvou DES[18] klíčů, konverzí 7 bytů na bity a vložením nuly za každou sedmici bitů dá dohromady 64 bitů potřebných pro DES klíč. Každý z těchto klíčů je použit pro zašifrování konstanty „KGS!@#$%“. Z toho vyplynou 8bytové hodnoty, které jsou spojeny do 16bytového klíče, který představuje LM hash.

[18] DES – Data Encryption Standard. Symetrická šifra, vyvinutá v sedmdesátých letech původně pro šifrování v civilních státních organizacích USA. Používá délku klíče 56 bitů, není tudíž dostatečně odolná a je možné ji prolomit hrubou silou za méně než 24 hodin.

Protože je funkce založena na použití šifry DES, která je již dnes lehce prolomitelná, je i samotná hašovací funkce prolomitelná. První slabinou je, že každé heslo delší sedmi znaků je rozděleno na dvě části, tak může být každá z těchto částí dešifrována samostatně. Druhou slabinou je, že všechny malé znaky jsou převedeny na velké, což redukuje množinu symbolů, ze které je haš vytvořena, na polovinu. [24]

5.12. N-Hash

Kryptografická hašovací funkce založená na algoritmu FEAL (Fast Data Encipherment Algorithm), což je bloková šifra navržená jako alternativa k DES. V dnešní době není považována za bezpečnou. Byla navržena v roce 1990 týmem Shojiho Miyaguchiho. O rok později byla publikována zpráva o jejích slabinách Eli Bihamem a Adi Shamirem.

Funkce pracuje se zprávou rozdělenou do 128bitových bloků, kde každý blok je kombinován s výstupem hašovací funkce, která obsahuje 8 rund. Výsledkem funkce je 128bitový výstup. [5]

5.13. Grindahl

Hašovací funkce byla představena poprvé na konferenci FSE (Fast Software Encryption) v roce 2007. Jejími autory jsou Lars R. Knudsen, Christian Rechberger a Søren S. Thomsen. Funkce je založena na blokové šifře Rijndael. Funkce Grindahl existuje ve dvou variantách, a to jako Grindahl-256 a Grindahl-512. Funkce jsou zpracovány stejným způsobem a jsou odlišné jen velikostí hašovacího klíče.

Základním stavebním blokem hašovací funkce je pole 4x13 bytů, které představuje stav této funkce. S tímto polem jsou dále prováděny následující transformace: SubBytes, ShiftRows, MixColumns a AddRoundKey. SubBytes je nelineární substituční funkce převzatá ze specifikace šifry Rijndael. ShiftRows cyklicky posouvá byty řádku v závislosti na jeho délce. Další funkce, MixColumns, je také převzata z šifry Rijndael a provádí transformaci sloupců. Poslední funkce AddRoundKey, která je původem také z šifry Rijndael, může být zaměněna s funkcí AddConstant, která vnáší asymetrii do každé rundy změnou posledního bytu pole. Nakonec je ještě připojeno zakončení, podobně jako v MD5, a je provedeno dalších 8 cyklů funkce, které zajišťují dostatečné rozprostření hašovacího klíče. [9]

5.14. Radiogatún

Funkce Radiogatún byla poprvé představena v roce 2006 na konferenci Second Cryptographic Hash Workshop v Santa Barbaře. Je založena na známé hašovací funkci Panama, a byla navržena tak, aby odolala všem známým útokům a opravila slabiny této funkce.

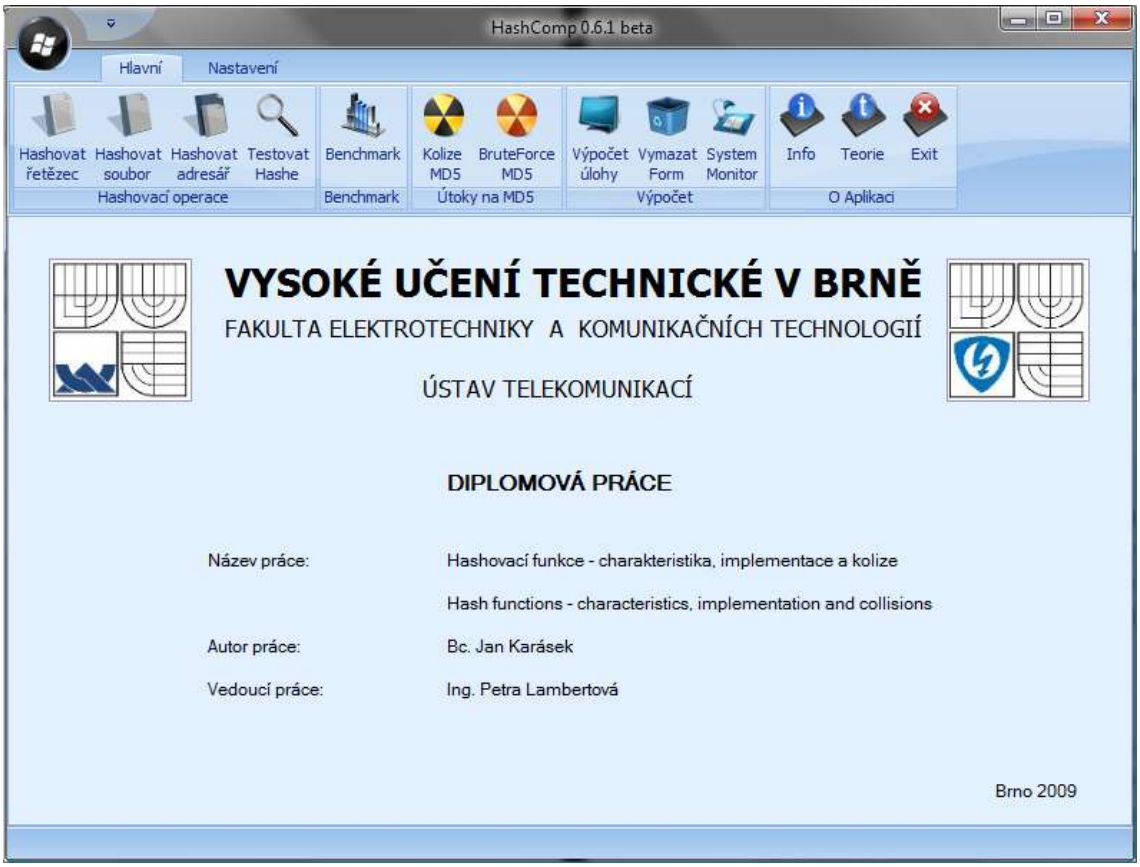
Radiogatún obsahuje dva základní prvky, které vycházejí z konstrukce Panamy, a jsou to Belt a Mill, do češtiny překládáme jako pás a mlýn. Operace, které jsou využity v této hašovací funkci, jsou standardní bitové operace jako AND, XOR, NOT a cyklický posuv bitových slov. Velikost zpracovávaného bitového slova může být od 1 do 64 a velikost výstupního klíče je 256 bitů. Protože bezpečnost hašovací funkce závisí na délce bitového slova, optimální využití nalezne tato funkce na 64bitových platformách. Dá se ovšem použít i na 32bitových. Jednou z výhod je vysoká výkonnost a díky kompaktnosti funkce je možná i její implementace přímo do hardwaru. [33]

6. Základní popis programu

Dříve než přejdu ke složitější problematice hašovací funkce MD5, její implementaci, tvrzení o bezpečnosti a implementaci kolizí, popíšu na tomto místě základní rozhraní vytvářeného programu, jeho základní možnosti tak, abych postupně v dalším textu mohl navazovat kapitolami zabývajícími se implementačními částmi a přímou demonstrací této implementace na zmiňovaném programu.

Jako implementační jazyk jsem zvolil programovací jazyk C#, se kterým jsem operoval v prostředí Microsoft Visual Studio 2008. V tomto jazyku jsem implementoval i svou bakalářskou práci a ze všech programovacích jazyků je mi nejbližší. Při vytváření grafického rozhraní jsem využil grafických komponent firmy DevExpress[19] ve verzi 2009.1.

Začněme spuštěním programu. Na obr. 1 je vidět základní obrazovka, která se objeví vždy po spuštění programu.



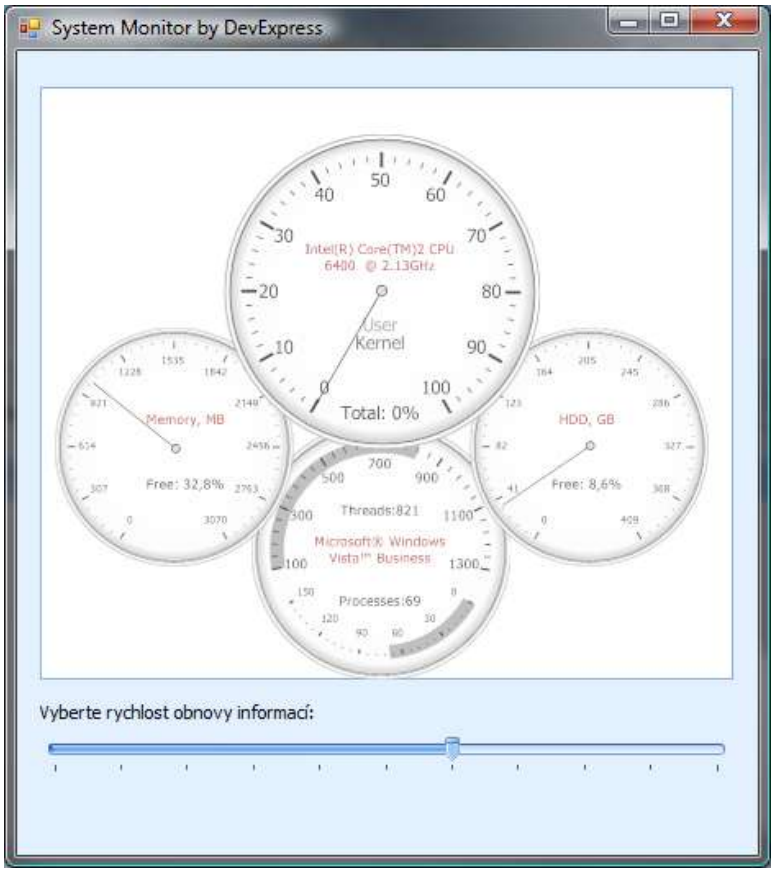
Obr. 1: Základní obrazovka programu

[19] Komponenty DevExpress je možné nalézt na adrese http://www.devexpress.com

Po spuštění programu je možné zvolit operaci z horního menu, tzv. ribbon menu (viz obr. 2), kterou chce uživatel provést. Základní obrazovka byla předvedena a nyní v dalších kapitolách bude postupně rozebrána, teoreticky i prakticky a s názornou ukázkou na programu, problematika hašovacích funkcí a jejich kolizí.

Obr. 2: Menu aplikace

V této části bych se rád ještě zmínil o monitoru systémových prostředků (viz obr. 3), který je použit jako komponenta z knihoven prvků DevExpress. Tento monitor v reálném čase znázorňuje vytížení procesoru, počet běžících vláken, vytížení pamětí a disku, což může být zajímavé hlavně při počítání kolizí nad hašovací funkcí MD5.



Obr. 3: Monitor systémových prostředků

7. Hašovací funkce FNV

Hašovací funkce patří mezi obecné hašovací funkce. Teorie ohledně funkce byla popsána v kapitole 3.6. Ač se nejedná o kryptografickou hašovací funkci, je její konstrukce natolik kvalitní, že funkce našla široké využití v mnoha programech, programovacích jazycích, datových strukturách a dokonce i při implementaci souborového systému.

7.1. Implementace funkce

Základní funkce jsou implementovány jako statické metody ve třídě *BaseHaseFunctions*. Jednou z metod této funkce je i výpočet FNV funkce založené na operaci bitové negace a násobení prvočíslem.

Na vstupu funkce přijímá řetězec. Je definováno prvočíslo, jehož dekadická hodnota je 2166136261 a inicializační hodnota haše nula. V těle metody probíhá cyklus o počtu průchodů rovných počtu znaků vstupního řetězce. V každém průchodů je haš (výsledek z předchozího cyklu) násobena prvočíslem a poté je XORována s i-tým znakem vstupu. Výsledkem metody je 64bitová celočíselná hodnota. [8], [28]

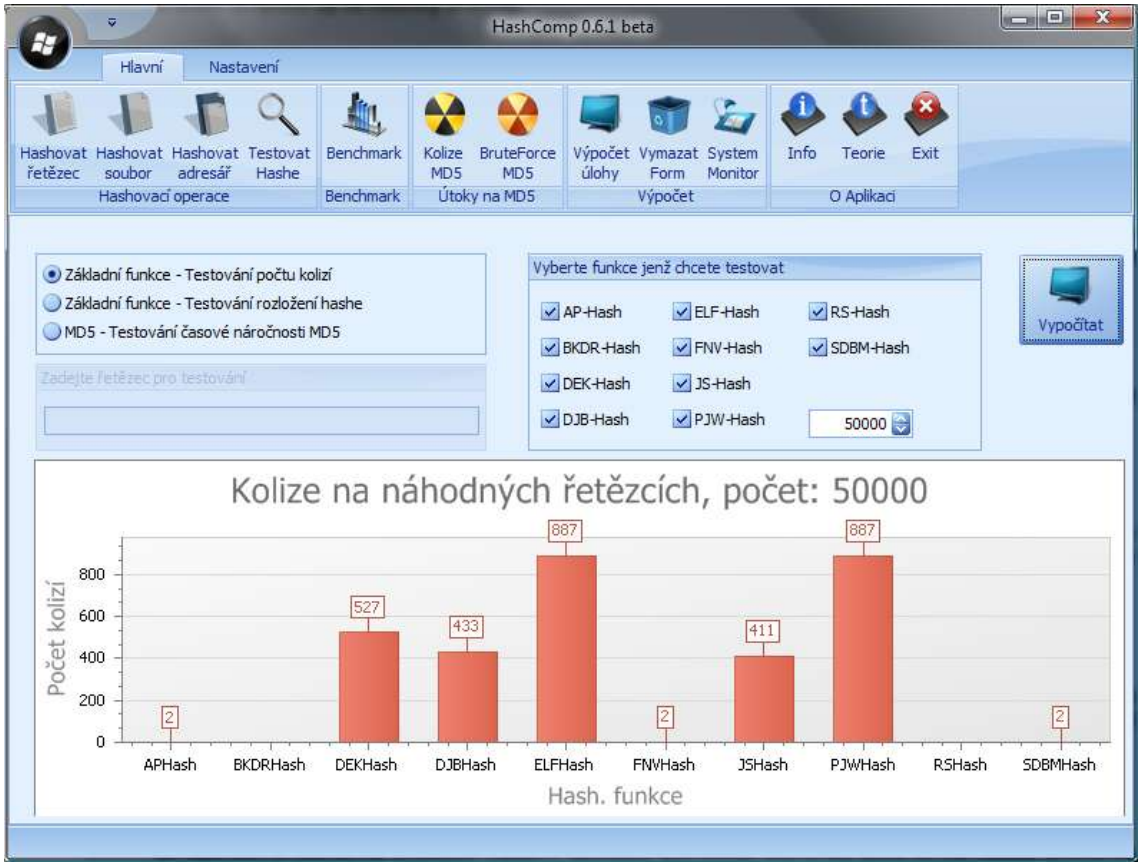
```
public static long FNVHash(String str)
{
      uint fnv_prime = 0x811C9DC5;
      uint hash = 0;

      for (int i = 0; i < str.Length; i++)
      {
            hash *= fnv_prime;
            hash ^= (char)str.ToCharArray().GetValue(i);
      }

      return hash;
}
```

Výsledek hašovací funkce je zobrazen na obr. 8 v kapitole 8.3, kdy jsou testovány všechny základní hašovací funkce i s funkcí MD5 na řetězci „message digest“.

7.2. Odolnost vůči kolizím

Ukažme si nyní zajímavější věc, a to na obr. 4, odolnost vůči kolizím, a srovnání s ostatními základními hašovacím funkci. Tento test bude proveden na 50 000 náhodných řetězcích.

Ve výsledku kolizního testu vidíme, že ne všechny hašovací funkce jsou tak dobře navrženy jako funkce FNV. Této funkci může konkurovat pouze funkce AP (navržena softwarovým inženýrem Arashem Patrowem), BKDR Hash (navržena Brianem Kerniganem v knize „The C Programming Language"), RS Hash (navržena Robertem Sedwickem v knize „C book" a SDBM Hash (používaná v open source projektech řízení báze dat). [8]



Obr. 4: Testování kolizí na obecných hašovacích funkcích

8. Hašovací funkce MD5

Po zpracování charakteristik hašovacích funkcí byla pro další činnosti vybrána funkce MD5. Jedná se o hašovací funkci založenou na moderní konstrukci, jejíž popis a implementace jsou zveřejněny v RFC 1321. Byla již několikrát prolomena, což z ní činí ideální funkci k demonstrování požadovaných úkolů v zadání diplomové práce.

Před samotnou implementací funkce se nejprve podívejme, jak je moderní hašovací funkce konstruována. Konstruování funkce konkrétně vysvětlím na konstrukci funkce MD5. Přispěje to k lepšímu pochopení implementace.

8.1. Konstrukce hašovací funkce

U moderních hašovacích funkcí může být zpráva velmi dlouhá až $D = 2^{64}-1$ bitů. Je zřejmé, že takovou zprávu musíme zpracovávat po částech. Odtud vyplývá skutečnost, že hašování zprávy probíhá sekvenčně po blocích určité délky, u MD5 jde o blok délky 512 bitů. Ze zpracování po blocích také plyne nutnost zarovnání vstupní zprávy na délku celistvých bloků. [22]

8.1.1. Zarovnání zprávy

Zarovnání musí být takové, aby umožňovalo jeho jedinečné odejmutí. Například vezměme tři stejné zprávy, které se budou lišit jen na konci v počtu nul, pokud by zarovnání bylo realizováno samými nulami, zarovnané zprávy by byly totožné a jednoduše by došlo ke kolizi. Aby šlo zarovnání jednoduše určit, přidává se ke zprávě jako první bit 1 a dále je zpráva doplněna bity 0. Dle [36] musí platit rovnice [8.1]:

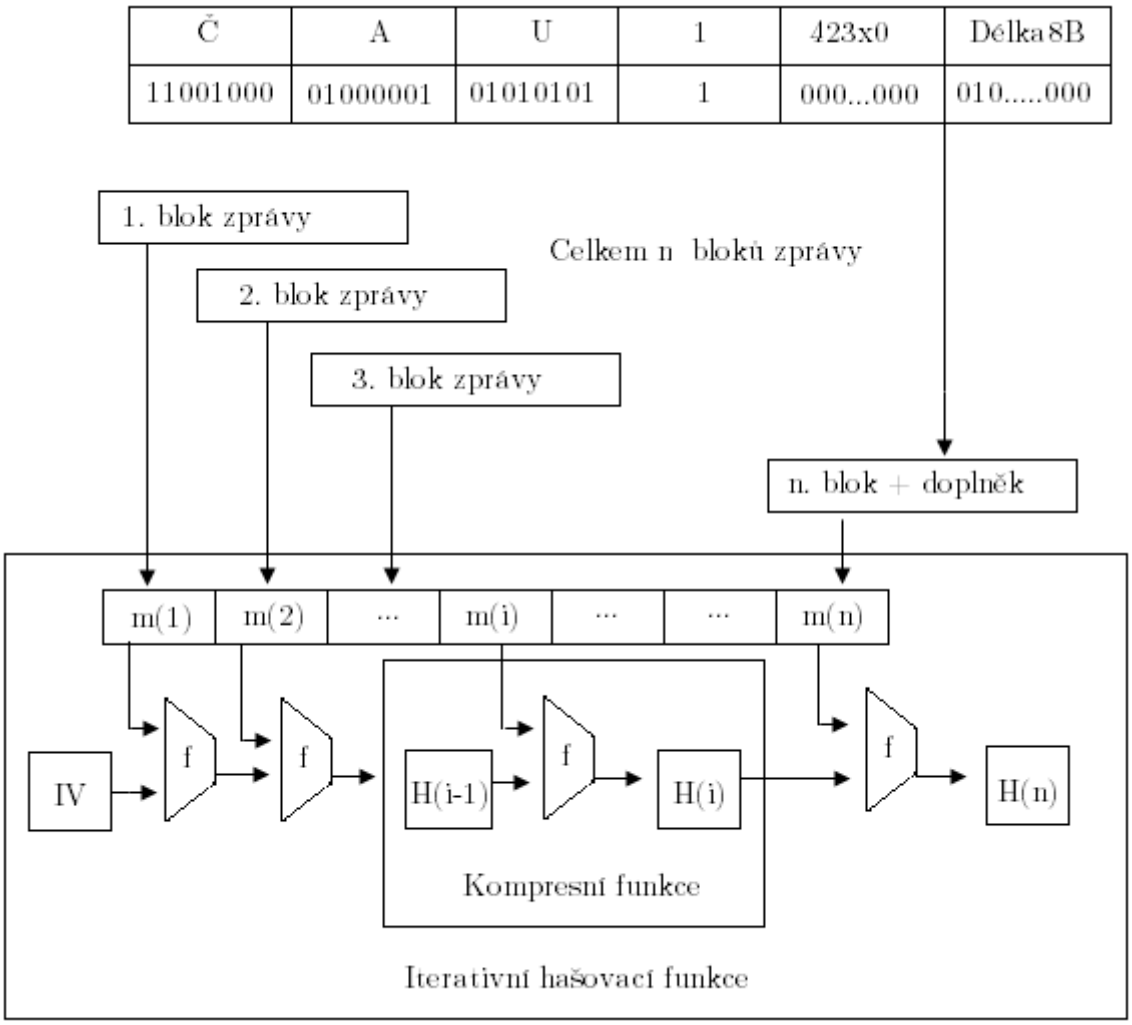
$$(\text{délka bloku zprávy} + \text{délka zarovnání}) \bmod 512 = 448 \qquad [8.1]$$

8.1.2. Damgard-Merklovo zesílení

Za potřebným počtem bitů se ještě pro zesílení bezpečnosti přidává délka původní zprávy. Tomuto doplnění se říká Damgard-Merklovo zesílení, které bylo nezávisle navrženo oběma autory na konferenci Crypto 1989. Po doplnění zarovnávacími bity zbývá do celistvého bloku ještě 64 bitů, právě těchto 64 bitů je vyplněno 64bitovým vyjádřením velikosti původní zprávy. Odtud je tedy podmínka, že funkce může hašovat zprávy do délky $D = 2^{64}-1$ bitů. [22]

V podstatě všechny současné prakticky používané funkce používají Damgard-Merklův princip iterativní hašovací funkce s využitím kompresní funkce. Na obr. 5 je názorně ukázáno, jak postupně po blocích hašování probíhá.

Počáteční hodnota kontextu H_0 se nazývá inicializační vektor (IV). Dále je na obrázku patrné rozdělení zpráv do bloku m_i, ty vstupují do kompresní funkce společně s výstupní hodnotou předešlé kompresní funkce H_{i-1}, nebo v případě první kompresní funkce s inicializačním vektorem H_0. Průběh hašování je iterativní, hašují se postupně všechny bloky zprávy, dokud není zhašován i poslední blok m_n, jehož výsledkem je očekávaný haš H_n. [22]



Obr. 5: Doplňování, kompresní funkce a iterativní hašovací funkce

8.1.3. Konstrukce kompresní funkce

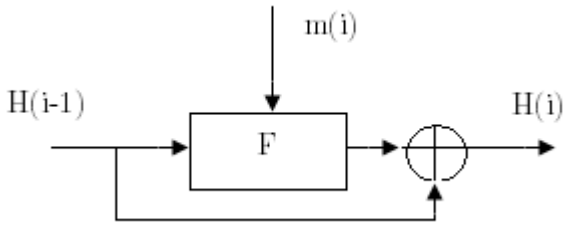
Kompresní funkce je jádrem hašovací funkce. Měla by být robustní, aby zajistila dokonalé promíchání bitů zprávy a jednosměrnost. Kompresní funkce by se měla chovat jako náhodné orákulum. Jako náhodné orákulum je nazýváno zařízení, které na každý vstup odpovídá náhodně vybraným výstupem z určité množiny hodnot. Navíc je definována podmínka, pokud se na vstupu objeví jistá hodnota vícekrát, náhodné orákulum na něj odpoví vždy stejným výstupem. Cílem je, aby se kompresní funkce chovala jako náhodné orákulum.

Kompresní funkce máme možnost sestrojit na bázi známých jednosměrných funkcí. Zde se uplatňují znalosti z blokových šifer. Kvalitní bloková šifra $E_k(x)$ se má při pevném klíči k také chovat jako náhodné orákulum. Dále zaručuje, že známe-li jakoukoli množinu vstupů – výstupů, nemůžeme odtud určit klíč k. Vzhledem ke klíči je tak bloková šifra jednosměrná. Možnost konstrukce kompresní funkce viz rovnice [8.2]:

$$H_i = E_{mi}(H_{i-1}) \qquad [8.2]$$

Moderní hašovací funkce používají navíc ještě jednu operaci, která zesiluje vlastnost jednosměrnost ještě přidáním vzoru před výstupem. Tato konstrukce se nazývá Davies-Meyerova. Davies-Meyerova konstrukce je znázorněna v rovnici [8.3] a na obr. 6. [22]

$$H_i = E_{mi}(H_{i-1}) \text{ XOR } H_{i-1} \qquad [8.3]$$



Obr. 6: Davies-Meyerova kompresní funkce

8.2. Popis hašovací funkce MD5

Začněme předpokladem, že na vstupu je b bitová zpráva, a chceme vypočítat její haš. Nechť b je libovolné nezáporné celé číslo, může být i nula, nemusí být násobkem osmi a může být libovolně dlouhé. Představme si bity zprávy psané takto: $m_0, m_1 \ldots m_{b-1}$. V následujících čtyřech krocích vypočteme žádanou haš zprávy:

8.2.1. Zarovnání zprávy (viz kapitola x.x)

8.2.2. Damgard-Merklovo zesílení (viz kapitola x.x)

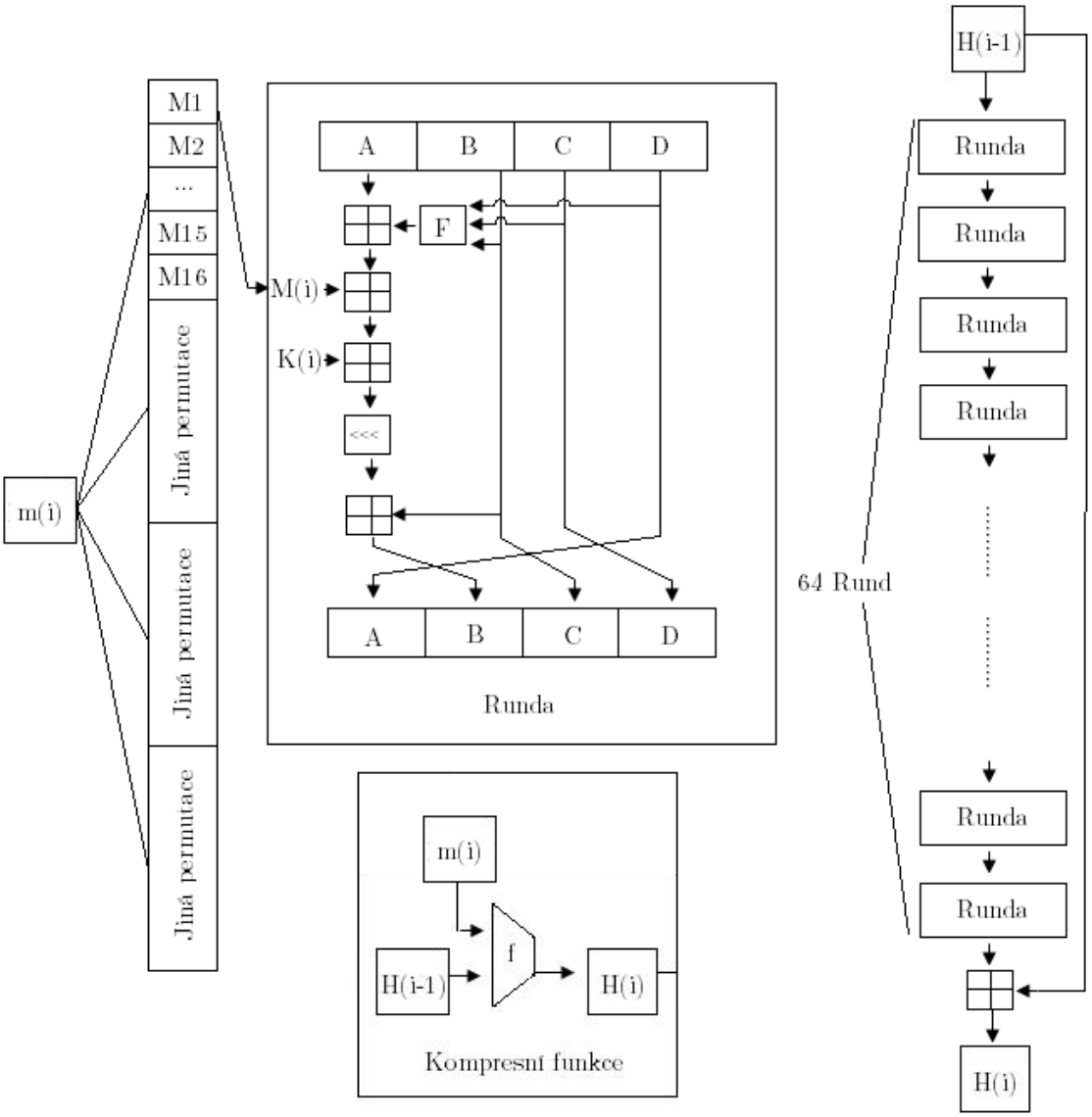
8.2.3. Inicializace bufferu funkce

Funkce MD5 používá při výpočtu buffer rozdělený na čtyři 32bitová slova. Tyto části bufferu jsou označovány jako A, B, C, D. Tyto buffery jsou na začátku výpočtu inicializovány těmito hexadecimálními hodnotami:

A = 0x67452301,
B = 0xEFCDAB89,
C = 0x98BADCFE,
D = 0x10325476.

8.2.4. Blokové zpracování zprávy

Na obr. 7 znázorněme, jak probíhá zpracování jednoho bloku zprávy. Algoritmus MD5 zpracovává bloky o velikosti 512 bitů. Tento blok je rozdělen do 16ti 32bitových slov $M_0, M_1, M_2 \ldots M_{15}$ a je uložen postupně do paměti s nejméně významným bitem na začátku. Tento proces je proveden čtyřikrát za sebou v jiných permutacích. Na obrázku vidíme, že v kompresní funkci se kontext „zašifruje“ vždy jedním 32bitovým slovem M_i. Na místě dílčí funkce F se po 16ti rundách střídají nelineární funkce F, G, H, I a v každé rundě se využívá jiná konstanta K_i. Po 64 rundách dojde ještě k přičtení původního kontextu (H_{i-1}) k výsledku předešlé kompresní funkce. V MD5 je jako kompresní funkce využita Davies-Mayerova konstrukce, operace XOR je nahrazena aritmetickým součinem modulo 2^{32}. Tak vznikne nový kontext H_i. Pokud by zpráva měla jen jeden blok, byl by obsah bufferu A, B, C, D celkovým výsledkem. Pokud byla zpráva rozdělena do více bloků, pokračuje se stejným způsobem v hašování do posledního bloku. Výsledkem funkce je 128bitová haš. [36], [18-22]



Obr. 7: Schéma zpracování uvnitř funkce MD5

8.3. Implementace hašovací funkce MD5

V jazyku C# je pro vlastní implementace hašovacích funkci připravena bázová třída. Z této třídy, *HashAlgorithm*, dědí všechny vestavěné hašovací funkce a měly by z ní dědit i veškeré vlastní hašovací funkce, aby byla zachována hierarchie stromu objektů a rozhraní, přes které se tato nově implementovaná funkce bude volat. Implementace tedy začne dědičností, vlastní třídu nazvěme příhodně *MD5*, je zařazena v jiném jmenném prostoru (*namespace*), takže ke kolizi jmen s již vestavěnou třídou *MD5* nedojde.

```
namespace DiplomovaPrace.BusinessLogic.HashFunctions
{
      public class MD5 : HashAlgorithm
      {
      …
      }
}
```

Bázová třída *HashAlgorithm* vyžaduje po třídě MD5, aby implementovala tři metody, první z nich je metoda *Initilize()*, druhá *HashCore(...)* a třetí *HashFinal()*.

První z metod inicializuje hašovací funkci, resp. její počáteční hodnoty v bufferu, v předchozí kapitole šlo o čtyři části bufferu *A, B, C a D*. Tento buffer je v programu implementovaný jako 4 prvkové pole typu uint (unsigned integer – 32bitové celé číslo) a jmenuje se *memoryRegister*. Dále je inicializována hodnota, která vyjadřuje velikost zpracovávaného bloku, což je u MD5 512 bitů a hodnota představující výsledná haš zprávy, který je u MD5 o velikosti 128 bitů. Dále jsou inicializovány ještě proměnné vyjadřující velikost zpracovávané zprávy, paměťový buffer aktuálně zpracovávaného bloku a další.

```
override public void Initialize()
{
      …
      memoryRegisters[0] = 0x67452301;           // Pametovy registr A
      memoryRegisters[1] = 0xEFCDAB89;           // Pametovy registr B
      memoryRegisters[2] = 0x98BADCFE;           // Pametovy registr C
      memoryRegisters[3] = 0x10325476;           // Pametovy registr D
}
```

Druhá metoda v pořadí, *HashCore(...)*, je jádrem hašovací funkce, na vstupu přebírá pole bytu, které bude hašováno, offset od kterého bude hašováno a počet bytů v poli, které budou hašovány od zadaného offsetu. Tato metoda má za úkol v cyklu hašovat celistvé bloky dat o délce 512 bitů. K tomuto účelu je volána vlastní metoda jménem *ProcessBlock*(...), která výpočet řeší na základě teorie zmíněné v kapitole 8.3, znázorněno též na obr. 7. Metoda *ProcessBlock(...)* využívá k výpočtu nelineárních funkcí, z nichž zde alespoň jednu pro ilustraci uvedu.

```
override protected void HashCore (byte[] array, int ibStart, int cbSize)
{
      //v metodě dochází k cyklickému volání metody ProcessBlock(…),
      // dokud není dosaženo posledního bloku, tehdy se vrátí řízení
      // nadřazené třídě HashAlgorithm, a je volána metoda HashFinal()
}
```

```
protected void ProcessBlock(byte[] inputBuffer, int inputOffset)
{
      //uchování paměťových registrů pro výpočet na konci
      uint a = memoryRegisters[0];
      uint b = memoryRegisters[1];
      uint c = memoryRegisters[2];
      uint d = memoryRegisters[3];

      //převod hodnot ze zprávy na číselné vyjádření
      uint[] processBuffer = ByteToUInt(inputBuff, inputOff, BlockSize);

      //první čtyři permutace, na obrázku x.x odpovídá M1 – M4
      a = this.FF(a, b, c, d, processBuffer[0],   7, 0xD76AA478);
      d = this.FF(d, a, b, c, processBuffer[1],  12, 0xE8C7B756);
      c = this.FF(c, d, a, b, processBuffer[2],  17, 0x242070DB);
      b = this.FF(b, c, d, a, processBuffer[3],  22, 0xC1BDCEEE);
      …
      //poslední čtyři permutace, na obrázku x.x odpovídá M61-M64
      a = this.II(a, b, c, d, processBuffer[4],   6, 0xF7537E82);
      d = this.II(d, a, b, c, processBuffer[11], 10, 0xBD3AF235);
      c = this.II(c, d, a, b, processBuffer[2],  15, 0x2AD7D2BB);
      b = this.II(b, c, d, a, processBuffer[9],  21, 0xEB86D391);

      //konečná haš bloku, označovaná jako H(i)
      memoryRegisters[0] += a;
      memoryRegisters[1] += b;
      memoryRegisters[2] += c;
      memoryRegisters[3] += d;
}
```

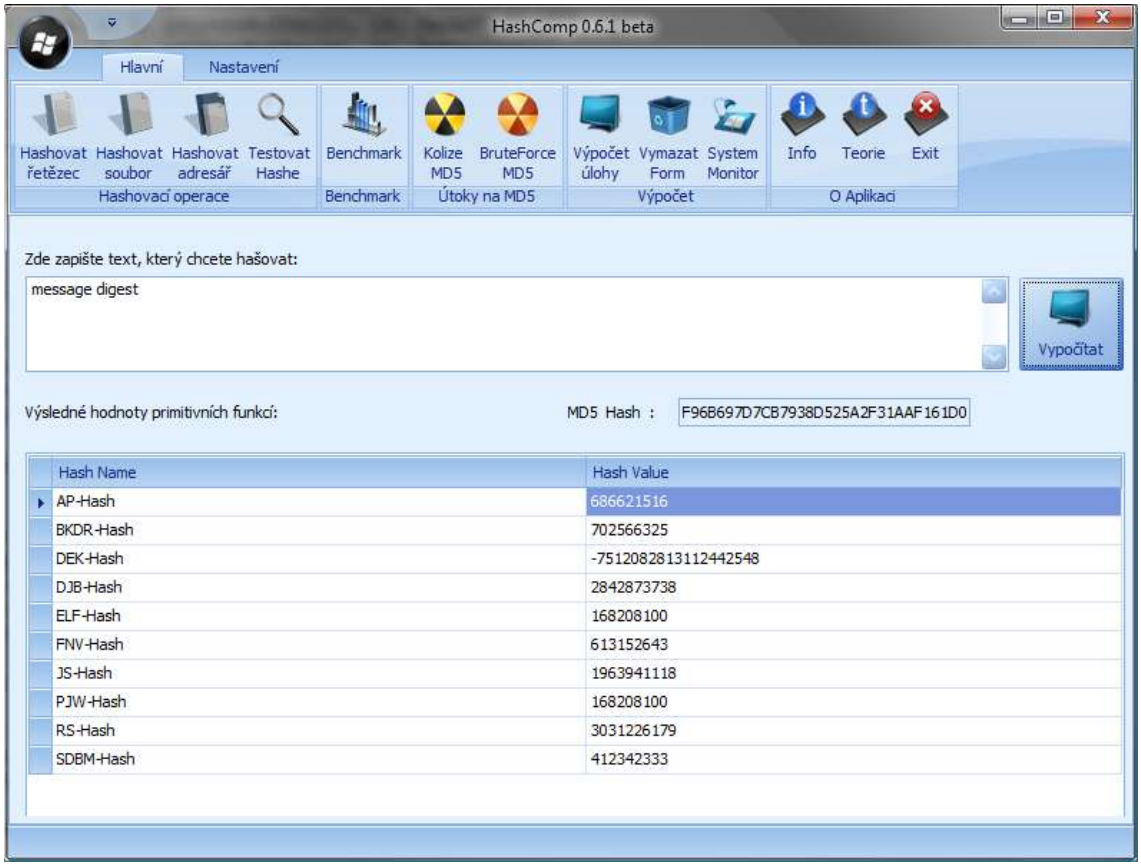
Ukázka implementace jedné z nelineárních funkcí, a to metoda *F(...)*, která je použita v permutaci $FF(...)$ společně s levou bitovou rotací $RL(...)$.

```
private uint FF(uint a, uint b, uint c, uint d, uint k, int s, uint t)
{
      return b + this.RotateLeft((a + ((b & c) | (~(b) & d)) + k + t), s);
}
```

Pokud již byla zhašována celá zpráva a zbývá poslední blok, je volána metoda *HashFinal()*, která je poslední z povinně implementovaných. Tato metoda má za úkol před zhašováním doplnit chybějící bity a provést poslední hašování. K tomuto účelu byla ještě vytvořena metoda *ProcessFinalBlock(...)*.

Dále jsou pak v algoritmu využívány metody jako levá bitová rotace a různé převody typu byte a na celé číslo o různém rozsahu a obráceně. Většina z těchto metod může být použita i v jiných částech programu a byly tak implementovány jako statické do třídy *Utilities*.

Nyní si ukažme, jak hašovací funkce funguje. Ověřit ji můžeme např. takto: z [36] víme, že haš řetězce "message digest" je f96b697d7cb7938d525a2f31aaf161d0. Otevřu tedy program a v menu kliknu na tlačítko „Hashovat řetězec“. Výsledek je vidět na obr. 8, výsledné haše se shodují.

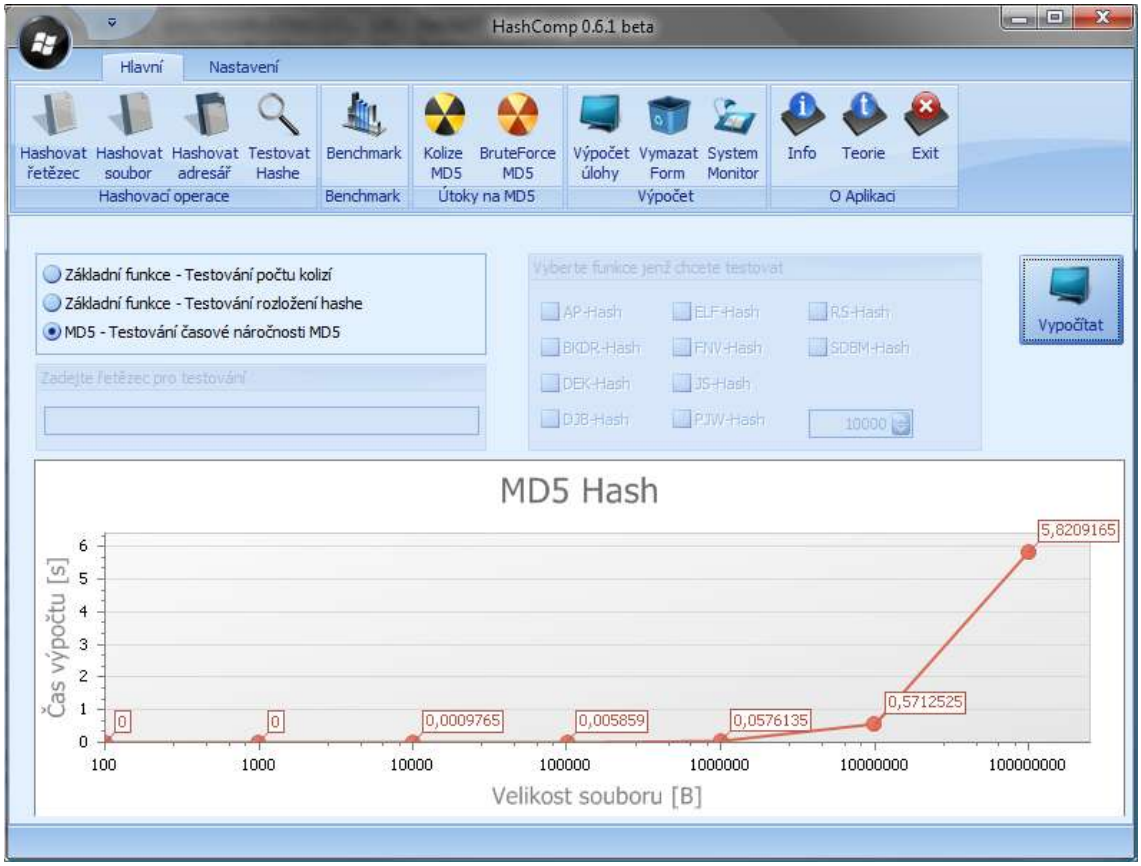


Obr. 8: Hašování řetězce

Jak je vidět na obr. 8, hašovací funkce MD5 není jedinou implementovanou funkcí, v tabulce jsou vidět i haše pro obecné hašovací funkce. Tyto funkce byly v kapitole 7.2 podrobeny výkonnostním testům na počty kolizí v n jedinečných řetězcích.

Hašovací funkci MD5 je možné použít nejen na hašování řetězců, jak je vidět na obrázku, ale i na hašování souborů a adresářů, nebo je možné otestovat haš libovolného souboru vůči již známé haši.

Pro hašovací funkci MD5 byl pro zajímavost vytvořen také jednoduchý test, který na názorném grafu ukazuje rychlost hašovací funkce při hašování souborů. Na obr. 9 je vidět výsledek hašování 6 souborů od velikosti 100 B do velikosti 100 MB.



Obr. 9: Výsledek hašování souborů

9. Bezpečnost funkce MD5

9.1. Složitost nalezení kolize

Pokud se bude hašovací funkce chovat jako náhodné orákulum (viz 8.1.3) bude složitost nalezení vzoru rovna 2^n, kde n je počet bitů výstupu hašovací funkce. Pokud je nalezena cesta, jak vzory nalézat jednodušeji, hovoříme o prolomení hašovací funkce, a ta již nemůže být dále považována za bezpečnou.

Z definice hašovací funkce plyne, že kolize zákonitě existují. Jak velká tedy musí být množina náhodných zpráv, aby v ní s nezanedbatelnou pravděpodobností existovaly dvě různé zprávy se stejnou haší? Tuto otázku, zodpovídá narozeninový paradox, který říká, že pro n bitovou hašovací funkci nastává kolize s asi 50% pravděpodobností v množině $2^{n/2}$ zpráv namísto očekávaných $\frac{1}{2}\cdot 2^n$ zpráv. [22]

Narozeninový paradox

Představme si množinu M o počtu m vzorů. Kolik z těchto vzorů musíme vybrat, abychom v množině obrazů získali kolizi? Tento problém se nazývá narozeninový paradox, protože se aplikuje na narozeniny.

Mějme ve třídě množinu lidí a množinu obrazů, kterou jsou narozeniny. Kolik lidí je nutno dát dohromady, aby se v množině k prvků našli dva, kteří slaví narozeniny ve stejný den? Rovnice [9.1] vyjadřuje, s jakou pravděpodobností nastane kolize.

$$p(m,k) = 1 - \left(\frac{365}{365}\cdot\frac{365-1}{365}\ldots\frac{365-k}{365}\right) = 1 - e^{\frac{k^2}{2m}} \qquad [9.1]$$

Pro velké m se ve výběru k rovná odmocnině z m (viz rovnice [9.2]):

$$k = (2\cdot m\cdot \log_e 2)^{\frac{1}{2}} = m^{\frac{1}{2}} = \sqrt{m} \qquad [9.2]$$

S 50% pravděpodobností stačí $\sqrt{m}$ lidí, aby nastala kolize, pro 365 dní je to 23 lidí. [22]

9.2. Složitost nalezení multikolize

Multikolize je r-násobná kolize, kdy r zpráv vede na stejnou haš. K tomu, abychom mezi odpověďmi náhodného orákula na N dotazů našli jednu odpověď r krát, postačí s dostatečnou pravděpodobností vztah [9.3]:

$$N = 2^{n\cdot(r-1)/r}, \qquad [9.3]$$

což je pro větší r přibližně 2^n, a pro $r = 2$ dostáváme narozeninový paradox a složitost $2^{n/2}$. Pokud se hašovací funkce bude chovat jako náhodné orákulum, bude složitost nalezení kolize právě $2^{n/2}$ a bude-li nalezena cesta, jak kolize hledat jednodušeji, hovoříme o prolomení hašovací funkce. [22]

9.3. Kolize kompresní funkce

Kolize kompresní funkce f spočívá v nalezení inicializační hodnoty H a dvou různých bloků B_1 a B_2 tak, že $f(H, B_1) = f(H,B_2)$.

Pokud je hašovací funkce bezkolizní, vyplývá odtud i bezkoliznost kompresní funkce. Důkazem je haš pro zprávy o velikosti jednoho bloku. Toto tvrzení obráceně neplatí, čili, pokud je bezkolizní kompresní funkce, nemusí být bezkolizní i hašovací funkce. Avšak u Damgard-Merklovy konstrukce bylo prokázáno, že bezkoliznost kompresní funkce implikuje bezkoliznost iterované hašovací funkce. A tak se tato konstrukce stala základem pro všechny moderní hašovací funkce. [22]

10. Útoky na funkci MD5

První útok na hašovací funkci MD5 byl proveden roku 1993 Denem Boerem a Antoonem Bosselaersem. Publikovali výsledek hledání pseudo-kolizí na MD5 kompresní funkci, kde dva rozdílné inicializační vektory mohou produkovat stejnou haš. O tři roky později profesor Dobertin oznámil kolizi kompresní funkce MD5. I když nešlo o útok na celou hašovací funkci, bylo doporučeno používat alternativní hašovací funkce jako Whirlpool, SHA-1 a RIPEMD-160. V roce 2004 byly nalezeny další nedostatky pod vedením doktorky Wangové, které ještě více zpochybnily použitelnost MD5. V roce 2005 Wangová, Lanstra a Weger ukázali vytvoření dvou certifikátů X.509 s různými veřejnými klíči a stejnou MD5 haší. Dále pak na výzkum Dr. Wangové navázal český kryptolog Vlastimil Klíma a publikoval 18. března 2006 algoritmus, který byl schopen tyto kolize hledat do minuty na obyčejném notebooku. Využil při tom metodu, kterou nazval tunelování. [21]

10.1. Hledání kolizí prvního řádu

V roce 2004 Xiaoyun Wangová a kolektiv publikovali na konferenci CRYPTO 2004 výsledky výzkumu, kde předložili dvě zprávy se stejnou haší. Metoda, kterou k tomuto výsledku došli, ovšem nebyla publikována až do roku 2005. V této publikaci byly zveřejněny i tzv. postačující podmínky zaručující kolizi, které se později ukázali jako chybné. Jejich částečnou úpravu postupně zveřejnilo nezávisle na sobě několik dalších kryptologů, ale až na konferenci Eurocrypt 2005 byla předložena úplně nová sada podmínek, která je pravděpodobně správná a konečná. Tuto sadu podmínek používá ve své demonstrační aplikaci i český kryptolog Vlastimil Klíma, který vynalezl metodu tunelování, umožňující vyhledat kolize na běžném počítači do půl minuty.

Nejprve byly práce hledající kolize založeny na metodě mnohanásobné modifikace zprávy, které vedly k naplnění množiny postačujících podmínek od začátku zprávy až do dosažení bodu, za nímž již nebylo možno nic změnit. Tento bod se nazývá bod verifikace (POV – point of verification). U MD5 jde o bod Q[24]. Těchto bodů je nutné získat velké množství, protože splnění všech postačujících podmínek za tímto bodem nejsme schopni ovlivnit. Jsou splněny náhodně. U MD5 je to 29 postačujících podmínek, a proto je potřeba vygenerovat 2^{29} bodů POV. Jeden z nich pak bude náhodně splňovat i zbývající postačující podmínky, a tudíž bude použit ke kolizi.

Metody mnohanásobné modifikace zpráv modifikují zprávu tak, aby se naplnily úvodní postačující podmínky a získali jsme bod POV.

Metoda tunelování Vlastimila Klímy naopak začíná právě v bodě POV. Několika tunely z něj geometrickou řadou postupně vytvoří dostatečné množství dalších POV, aniž by narušila počáteční podmínky před body POV. V ideálním případě potřebujeme pouze jeden bod POV. S použitím tunelu „o síle“ n z něj vytvoříme 2^n bodů POV. Tunely se dají dále kombinovat, takže z jednoho tunelu o síle n_1 vytvoříme dalších 2^{n1} bodů POV. U MD5 existuje hned několik tunelů, které dohromady dávají tunel o síle 24, tzn. že postačí vygenerovat pouze 2^5 = 32 originálních POV oproti 2^{29} bodům u současné nejúspěšnější metody mnohanásobné modifikace zpráv. [18-21]

10.1.1. Základní postup hledání kolizí

Jelikož se v další kapitole budu držet ukázkové implementace Vlastimila Klímy, bude popsán postup hledání kolizí s takovými proměnnými, které jsou použity v algoritmu. Písmeno *H* nahrazuje znak *, neboť v názvu proměnné v programu nelze tento znak použít. Např. proměnná *Q** bude tedy nahrazena proměnnou *HQ*.

Kolidující zprávy se skládají ze dvou 512bitových bloků *(M, N)* a *(HM, HN)*, přičemž *MD5(M, N) = MD5(HM, HN)*. Jsou zpracovávány po 32bitových slovech *M = (x[0], ..., x[15])* v 64 krocích. V prvním kroku vzniká meziproměnná *Q*[1], ve druhém kroku *Q*[2] atd. až *Q*[64] podle rovnic v obr. 10. Vektory *Q*[-3, -2, -1, 0] obsahují hodnoty inicializačního vektoru *IV*. Výsledek celého bloku je pak uložen do proměnných *IHV*. Vektor *IHV* dále vstupuje do druhého bloku *N* jako jeho inicializační vektor a celý postup se opakuje.

```
Q[ 1]=Q[ 0]+RL(F(Q[ 0],Q[-1],Q[-2])+Q[-3]+x[ 0]+0xd76aa478, 7);  0 p.
Q[ 2]=Q[ 1]+RL(F(Q[ 1],Q[ 0],Q[-1])+Q[-2]+x[ 1]+0xe8c7b756,12);  0 p.
Q[ 3]=Q[ 2]+RL(F(Q[ 2],Q[ 1],Q[ 0])+Q[-1]+x[ 2]+0x242070db,17); 17 p.
Q[ 4]=Q[ 3]+RL(F(Q[ 3],Q[ 2],Q[ 1])+Q[ 0]+x[ 3]+0xc1bdceee,22); 21 p.
Q[ 5]=Q[ 4]+RL(F(Q[ 4],Q[ 3],Q[ 2])+Q[ 1]+x[ 4]+0xf57c0faf, 7); 32 p.
      :
      :
      :
Q[61]=Q[60]+RL(I(Q[60],Q[59],Q[58])+Q[57]+x[ 4]+0xf7537e82, 6);  2 p.
Q[62]=Q[61]+RL(I(Q[61],Q[60],Q[59])+Q[58]+x[11]+0xbd3af235,10);  2 p.(+1m.)
Q[63]=Q[62]+RL(I(Q[62],Q[61],Q[60])+Q[59]+x[ 2]+0x2ad7d2bb,15);  2 p.
Q[64]=Q[63]+RL(I(Q[63],Q[62],Q[61])+Q[60]+x[ 9]+0xeb86d391,21);

IHV[0] = IV[0]+Q[61];
IHV[3] = IV[3]+Q[62]; 1 p.
IHV[2] = IV[2]+Q[63]; 3 p.
IHV[1] = IV[1]+Q[64]; 4 p.

        IV[0]=0x67452301;   IV[1]=0xefcdab89;   IV[2]=0x98badcfe;  IV[3]=0x10325476;
```

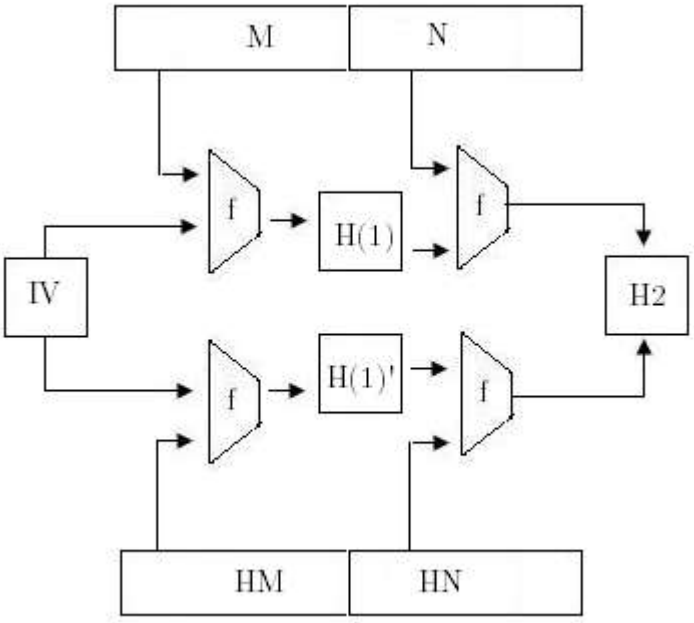
Obr. 10: Výpočet meziproměnných *Q*[x] a výsledku *IVH*

Na obr. 10 je dále vidět využití nelineárních funkcí *F, I* a levé bitové rotace *RL*. Na pravé straně vedle rovnic je vidět, která rovnice musí splňovat kterou postačující podmínku. Tyto postačující podmínky říkají, že některé bity zmíněných proměnných musí být stejné, některé různé, některé nuly a některé jedničky, zbývající bity mohou být libovolné. Detailnější pohled na výpočetní algoritmus je znázorněn v literatuře [21].

V prvním bloku *M* a druhém bloku *N* jsou postačující podmínky různé, v bloku *M* je jich mnohem více než v bloku *N*. Pokud tyto bloky splňují postačující podmínky, tak pro bloky *HM* a *NM* platí rovnice [10.1].

$$(HM, NM) = (H, N) + C \qquad [10.1]$$

V rovnici [10.1] je *C* předem definovaný konstantní vektor, o který se zprávy liší. Tento vektor se postupem času zruší a obdržíme dvě zprávy, jejichž výsledná haš je stejná. Hlavní princip útoku je znázorněn na obr. 11.



Obr. 11: Princip útoku na funkci MD5

10.1.2. Tunelování hašovacích funkcí

Začněme opět na předpokladu, že je zpracováván 512bitový blok *M= (x[0], ..., x[15])* v 64 krocích. Při výpočtu opět vznikají meziproměnné *Q*[1..64] jako v předcházející kapitole. I jejich výpočet a vstupní hodnoty jsou stejné. Nyní bude uveden konkrétní příklad tunelu Q9 pro lepší pochopení problému.

Tunel Q9

Podíváme-li se na obr. 12, kde jsou vyznačeny rovnice $Q[11]$ a $Q[12]$, vidíme, že případná změna $Q[9]$ na i-tém bitu by se nemusela v těchto rovnicích vůbec projevit za předpokladu, že i-tý bit $Q[10]$ bude nula a i-tý bit $Q[11]$ bude jedna. Tato skutečnost vyplývá z funkce F (viz rovnice [10.2]).

```
Q[ 7]=Q[ 6]+RL(F(Q[ 6],Q[ 5],Q[ 4])+Q[ 3]+x[ 6]+0xa8304613,17); 32 p.
Q[ 8]=Q[ 7]+RL(F(Q[ 7],Q[ 6],Q[ 5])+Q[ 4]+x[ 7]+0xfd469501,22); 29 p.
Q[ 9]=Q[ 8]+RL(F(Q[ 8],Q[ 7],Q[ 6])+Q[ 5]+x[ 8]+0x698098d8, 7); 28 p.
Q[10]=Q[ 9]+RL(F(Q[ 9],Q[ 8],Q[ 7])+Q[ 6]+x[ 9]+0x8b44f7af,12); 18 p.
Q[11]=Q[10]+RL(F(Q[10],Q[ 9],Q[ 8])+Q[ 7]+x[10]+0xffff5bb1,17); 19 p.
Q[12]=Q[11]+RL(F(Q[11],Q[10],Q[ 9])+Q[ 8]+x[11]+0x895cd7be,22); 15 p.
Q[13]=Q[12]+RL(F(Q[12],Q[11],Q[10])+Q[ 9]+x[12]+0x6b901122, 7); 14 p.
Q[14]=Q[13]+RL(F(Q[13],Q[12],Q[11])+Q[10]+x[13]+0xfd987193,12); 15 p.
Q[15]=Q[14]+RL(F(Q[14],Q[13],Q[12])+Q[11]+x[14]+0xa679438e,17);  9 p.
```

Obr. 12: Rovnice tunelu Q9

$$F(x,y,z) = (x \text{ AND } y) \text{ XOR } ((\text{NON}(x) \text{AND}(z))) \qquad [10.2]$$

Pro ty bity i, kde $Q[10]_i=0$ a současně $Q[11]_i=1$, máme tunel podle následujícího obrázku (obr. 13). Na všech těchto bitech i můžeme změnit hodnotu $Q[9]_i$, přičemž tato změna se neprojeví v rovnicích pro $Q[11]$ a $Q[12]$. Projeví se v rovnicích $Q[9]$, $Q[10]$ a $Q[13]$. Tyto změny však kompenzujeme změnou hodnot zprávy, tj. $x[8]$, $x[9]$ a $x[12]$, zatímco $Q[10]$, $Q[11]$, $Q[12]$ a $Q[13]$ zůstávají nezměněny.

```
Q[ 7]=Q[ 6]+RL(F(Q[ 6],Q[ 5],Q[ 4])+Q[ 3]+x[ 6]+0xa8304613,17); 32 p.
Q[ 8]=Q[ 7]+RL(F(Q[ 7],Q[ 6],Q[ 5])+Q[ 4]+x[ 7]+0xfd469501,22); 29 p.
Q[ 9]=Q[ 8]+RL(F(Q[ 8],Q[ 7],Q[ 6])+Q[ 5]+x[ 8]+0x698098d8, 7); 28 p.
Q[10]=Q[ 9]+RL(F(Q[ 9],Q[ 8],Q[ 7])+Q[ 6]+x[ 9]+0x8b44f7af,12); 18 p.
Q[11]=Q[10]+RL(              Q[ 8] +Q[ 7]+x[10]+0xffff5bb1,17); 19 p.
Q[12]=Q[11]+RL(       Q[10]        +Q[ 8]+x[11]+0x895cd7be,22); 15 p.
Q[13]=Q[12]+RL(F(Q[12],Q[11],Q[10])+Q[ 9]+x[12]+0x6b901122, 7); 14 p.
Q[14]=Q[13]+RL(F(Q[13],Q[12],Q[11])+Q[10]+x[13]+0xfd987193,12); 15 p.
Q[15]=Q[14]+RL(F(Q[14],Q[13],Q[12])+Q[11]+x[14]+0xa679438e,17);  9 p.
```

Obr. 13: Tunel Q9

Změny hodnot zprávy $x[8]$, $x[9]$ a $x[12]$ se projeví až za bodem verifikace (POV), který leží za bodem $Q[24]$, názorná ukázka je vidět v [21]. Všechny podmínky před ním zůstávají splněny. Změněny budou všechny proměnné $Q[25]$ až $Q[64]$, a to dosti složitým a náhodným způsobem, jak bylo ověřeno v [21].

Abychom získali co nejsilnější tunel, máme zájem na nastavení co nejvíce dvojic bitů $Q[10]_i=0$ a současně $Q[11]_i=1$. Tyto podmínky nejsou součástí postačujících

podmínek, pouze urychlují hledání kolizí. Nazýváme je, stejně jako u metody mnohonásobné modifikace zpráv, extra podmínky. Bohužel počáteční podmínky v současném diferenčním schématu umožňují jen volbu třech těchto pozic (i = 22, 23, 240, ostatní nejsou volné. Tím získáme tunel o síle 3.

V práci [21] byly publikovány i další tunely, které zde ovšem nebudu dále rozepisovat, protože pro vysvětlení principu, jak tato metoda funguje, uvedené vysvětlení stačí. Vlastimil Klíma v [21] uvádí, že myšlenka tunelů je obecně použitelná i v jiných hašovacích funkcích, pro které musí být pochopitelně zkonstruované jiné konkrétní tunely.

10.1.3. Implementace vyhledávání kolizí

Vyhledávání kolizí bylo implementováno na základě zveřejněného algoritmu Vlastimila Klímy. Jedná se o velice rozsáhlý kód čítající téměř dva tisíce řádků, takže zde bude popsán jen velice stručně. Pro další studium kódu doporučuji A2, kde je implementovaný celý algoritmus.

Algoritmus vyhledávání kolizí je rozdělen do dvou základních metod, nazvaných *B1(...)* a *B2()*, kde každá z těchto metod operuje s jedním ze dvou bloků zprávy. Metoda *B1(...)* přijímá jako vstupní parametr celé číslo, které naznačuje, kolik kolizí se má vyhledat. Tuto funkci jsem do algoritmu přidal, neboť v původní verzi bylo použito nekonečné smyčky, ve které byly kolize vyhledávány.

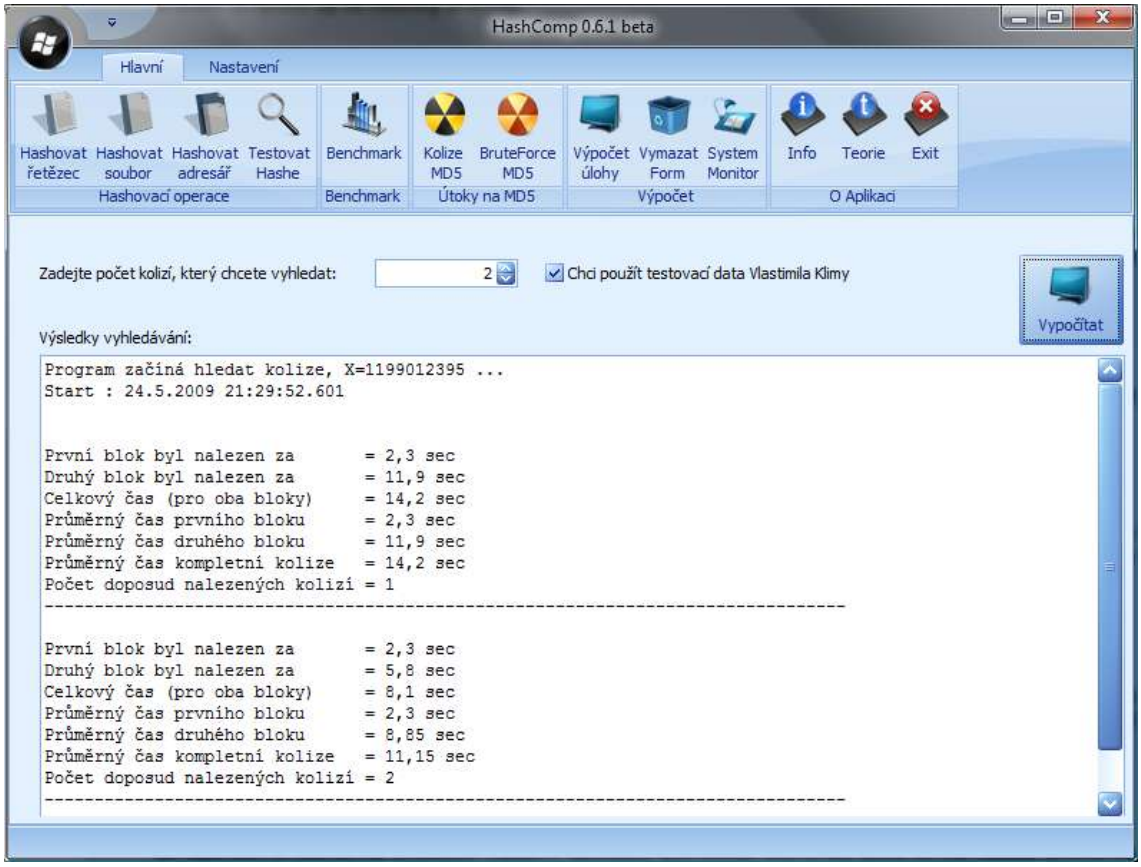
Metoda B1 počítá postupně všechny rovnice $Q[1..24]$. Jak je známo z kapitoly o tunelování, po těchto výpočtech je dosaženo bodu verifikace (POV). Následně jsou počítány tzv. dynamické tunely $Q[10]$, $Q[20]$, $Q[13]$ a tunely $Q[4]$ a $Q[9]$ a je modifikována zpráva na pozicích $x[8]$, $x[9]$ a $x[12]$. V dalším kroku jsou poté vypočítány rovnice $Q[25..64]$ a je zapsán výsledek ve formě vektorů *IHV*. V tomto místě je cyklicky volána funkce *B2()* a celý výpočet je prováděn znovu pro druhý blok zprávy.

Ukázka hledání kolizí je znázorněna na obr. 14, kde bylo nastaveno, že se mají vyhledat dvě kolize, a pro jistotu byla použita inicializační data Vlastimila Klímy, která zaručují, že první kolize bude nalezena do půl minuty.

Jak je z obrázku patrné, první kolize byla nalezena za 14,2 s, z toho první blok trvalo nalézt 2,3 s a druhý blok 11,9 s. U druhé kolize jsou data obdobná, celkový čas hledání druhé kolize je 11,15 s.

I když byla tato metoda implementována a demonstrována, žádný velký praktický užitek nepřinesla, protože zjistit původní vzor je zatím nemožné.

V následující kapitole bude ukázána konkrétní aplikace kolizí na dva různé soubory, které budou mít ve výsledku stejnou haš. [21]



Obr. 14: Implementace kolizí podle Vlastimila Klímy

10.1.4. Praktické využití útoku

Současné útoky na hašovací funkci MD5 neumí k danému dokumentu nalézt jiný se stejnou haší. Umíme pouze nalézt dva různé dokumenty (soubory) se stejnou haší. Útočník tedy nemůže napadnout jiný dokument, útok může být proveden pouze tak, že vygeneruje dvě kolidující zprávy. Jako příklad uveďme smlouvu, která bude pro útočníka nevýhodná, ovšem bude existovat ještě jedna, kterou útočník vygeneruje výhodnou pro sebe.

Dalším příkladem může být vytvoření dvou programů, z nichž jeden bude uživateli nějakým způsobem užitečný, bude digitálně podepsán, ale bude vytvořen útočníkem s úmyslem uživateli uškodit. Útočník tedy do daného programu vloží dva kolizní bloky, které začínají v paměti na místě zarovnaném na 512 bitů. Potom bude útočník schopen vytvořit uživatelem požadovaný program a záškodný program se stejným hašem. Tyto programy se budou lišit jen v bitech ve kterých se liší kolizní bloky. Tento útok byl demonstrován např. Ondrejem Miklem na Cryptofestu 2005.

V praxi tento příklad můžeme demonstrovat následovně. Vytvoříme dva programy.

Program A:

```
static void Main(string[] args)
{
      Console.WriteLine ("Užitečný program");
}
```

Program B:

```
static void Main(string[] args)
{
      Console.WriteLine ("Zlý program, který formátuje harddisk");
}
```

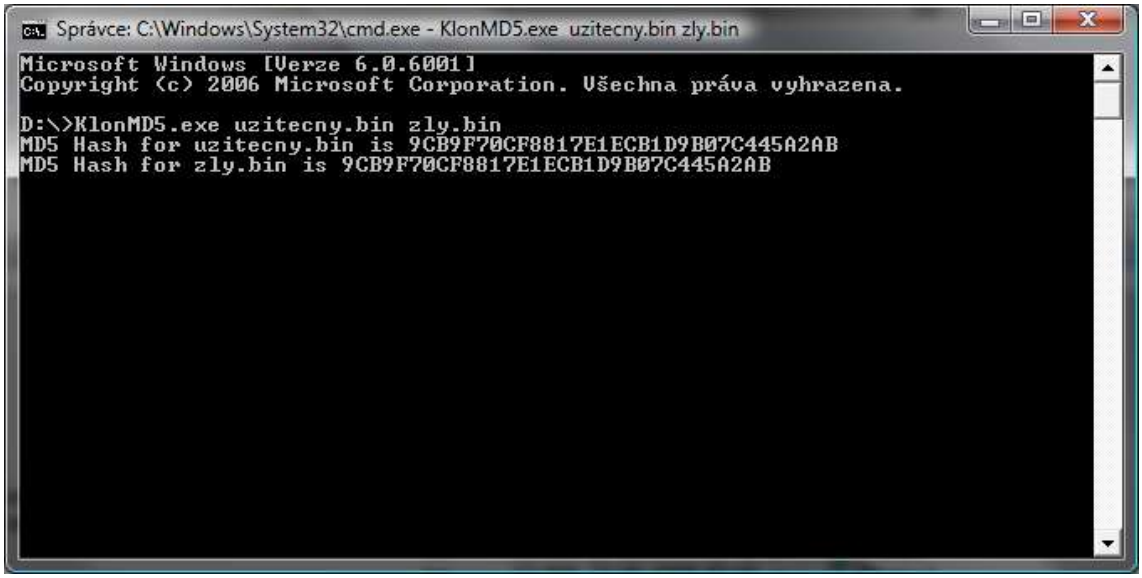
Dále musíme ještě vytvořit program, který nám zajistí stejné haše u těchto dvou programů. Implementace tohoto programu je inspirována [44].

Hlavní funkce programu C vypadá následovně:

```
static void Main(string[] args)
{
      byte[] uzitecny   = ReadBinaryFile(args[0]);
      byte[] zly        = ReadBinaryFile(args[1]);

      WriteBinary("uzitecny.bin", vec1, uzitecny, zly);
      WriteBinary("zly.bin", vec2, uzitecny, zly);

      ShowMD5("uzitecny.bin");
      ShowMD5("zly.bin");
}
```

Funkce *ReadBinaryFile(...)* převádí soubor na pole bytů, aby jej mohla vzápětí funkcí *WiriteBinary(...)* převést na binární soubor, se zakomponovanými připravenými vektory. Nakonci programu je volána funkce *ShowMD5(...)*, která vypíše MD5 haš.

Vektory je možné vidět ve zdrojovém kódu (viz příloha A1), nebo v literatuře [44]. Výstup programu je možné vidět na obr. . Haše pro oba programy se zcela shodují.



Obr. 15 Ukázka dvou souborů se stejnou haší

10.2. Útoky na jednosměrnost

Jednosměrná hašovací funkce je jednoduše taková funkce, kterou lze snadno vyčíslit, ale je prakticky nemožné z výsledku funkce odvodit její výstup. Ze zadaného x tedy lze snadno získat $f(x)$, avšak výpočet inverzní funkce, získání x při znalosti $f(x)$, je prakticky neřešitelné. Ukažme si tedy praktické metody, které mohou být použity k odhalení původního vstupu, zde hesla, do hašovací funkce.

10.2.1. Útok hrubou silou

Útok hrubou silou, často v češtině nazýván brute-force útok, je založen na testování všech možných řetězců. Na začátku se nastaví požadovaná délka od – do, která se má testovat, a množina znaků, ze které má být generován řetězec hesla. Tyto znaky mohou být velká či malá písmena, číslice nebo speciální znaky.

Pro jednoduchost si řekněme, že hledané heslo má čtyři znaky, a budeme předpokládat, že může být tvořeno velkými či malými písmeny a číslicemi. Počet generovaných řetězců tak bude vycházet z rovnice [10.3], kde n je počet znaků hesla. Pro tento konkrétní případ je celkový počet znaků množiny, ze které budou generována potenciální hesla 62 a počet znaků hesla $n = 4$. Z každého vygenerovaného

potenciálního hesla tak bude vytvořena haš a porovnána s obrazem původní haše, což bude prováděno tak dlouho, dokud nebude nalezena shoda.

$$\sum_{i=1}^{n}(pocet\_malych\_znaku + pocet\_velkych\_znaku + pocet\_cislic)^i \qquad [10.3]$$

10.2.2. Implementace útoku hrubou silou

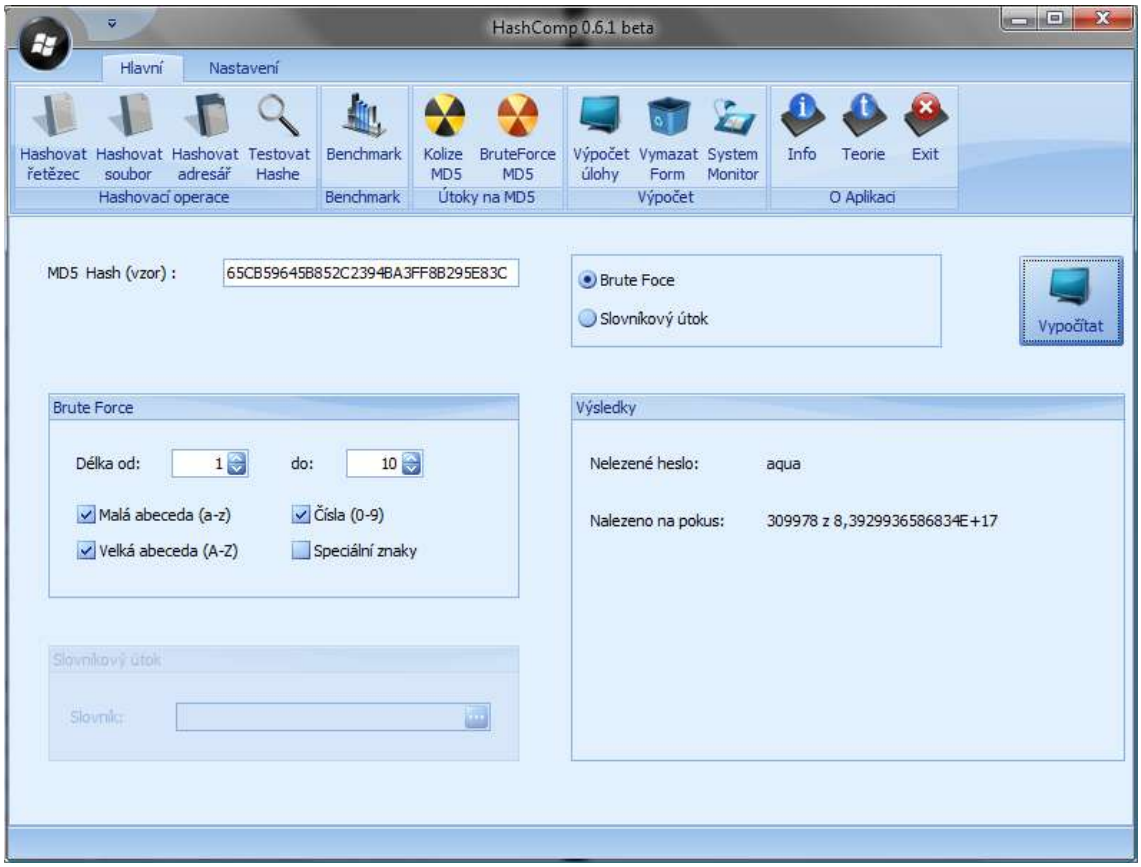
V programu je tento útok implementován následovně. Jelikož jsou vstupy pro metodu zadávány na formuláři, byla metoda pro jednoduchost na formuláři také implementována. Metoda *ComputeBruteForce()* generuje postupně všechny možné kombinace znaků z požadovaných množin v rozmezí zadaných hodnot délky hesla od – do. Tyto hodnoty jsou vždy po vygenerování hašovány vlastní implementací funkce MD5 a její výsledek je porovnán s obrazem původní haše. Tuto funkčnost obstarává metoda *UpdatePassword().*

Aktuální hodnota potenciální haše je vždy ukládána do pole *passwordArray*, typu string, o celkové délce maximální možné zprávy. Jednotlivé prvky tohoto pole obsahují znaky postupně generované funkcí *ComputeBruteForce()*. Celý výpočet je prováděn v několika cyklech, kdy hlavní cyklus běží dokud není nalezeno původní heslo, a dva vnořené cykly postupně generují a testují hesla.

Na Obr. 16 je vidět rozhraní aplikace a nalezené heslo. Pro testovací případ jsem použil heslo „aqua“, jehož haš je 65CB59645B852C2394BA3FF8B295E83C. Předpokládejme tedy znalost haše a neznalost původního hesla. Tuto haš použijeme jako vzor a budeme požadované heslo hledat útokem hrubou silou.

V programu tedy nastavíme rozsah hesla, můžeme nastavit např. od 1 do 10 znaků. A jako požadované množiny, ze kterých se má heslo generovat, zvolíme číslice a malá a velká písmena. Nakonec stiskneme tlačítko „Vypočítat“ a budeme čekat na výsledek, který by se pro naše heslo měl dostavit již do několika sekund.

Jak je vidět na obr. 16, hledání bylo úspěšné, pro danou haš je prvním nalezeným heslem řetězec „aqua“ a byl nalezen na 3099878mý pokus z celkového počtu 8,3929936586834e17.



Obr. 16: Implementace útoku hrubou silou

10.2.3. Slovníkový útok

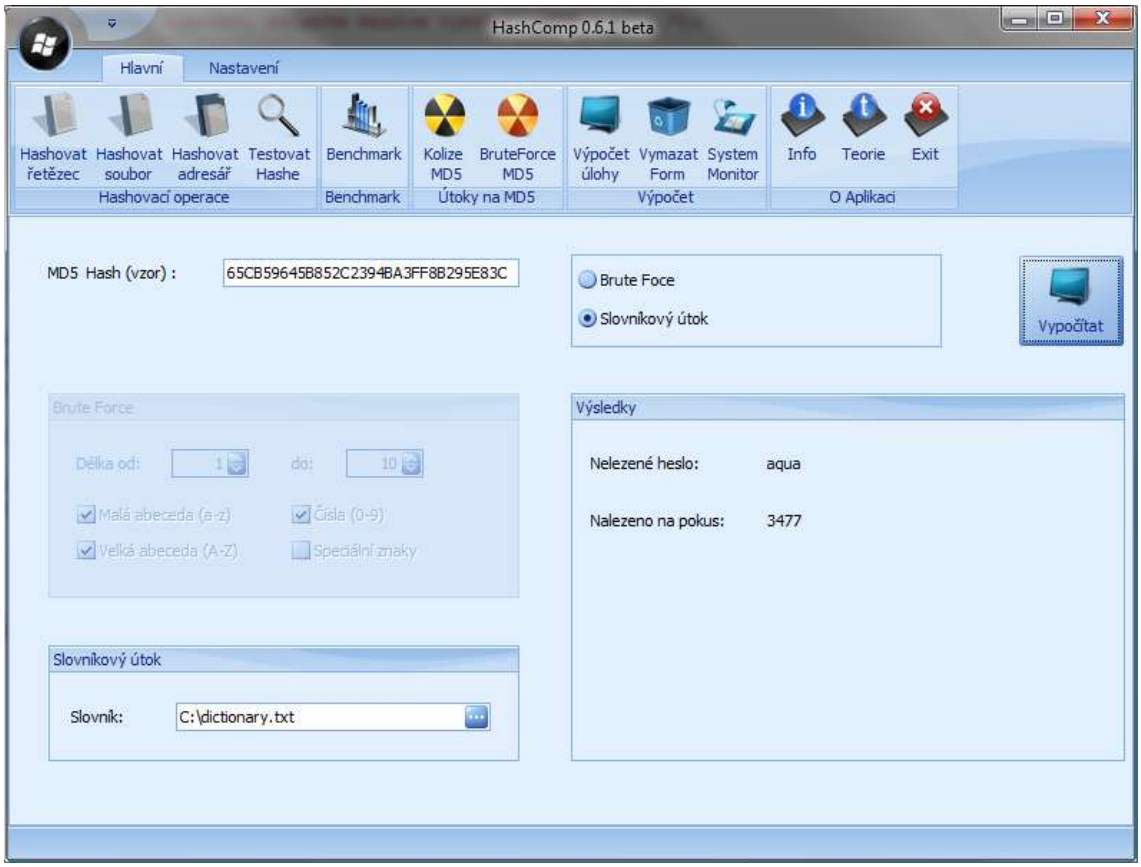
Slovníkový útok je velice podobný metodě hrubé síly až na to, že se negenerují postupně všechny možné řetězce, ale je použit seznam nejčastěji používaných hesel. Tento seznam se postupně prochází, řetězce jsou hašovány a zase se porovnávají s obrazem původní haše, dokud není nalezena shoda. Tato metoda nemusí dojít správného výsledku, pokud použitý seznam nebude obsahovat dané heslo. To se může stát velmi lehce, neboť většinou obsahuje nejpoužívanější jména, názvy nebo známé kombinace číslic a písmen.

10.2.4. Implementace slovníkového útoku

Implementace slovníkového útoku je mnohem jednodušší než u metody hrubé síly. Zde stačí vytvořit slovník popř. stáhnout z internetu, čímž odpadá složitý algoritmus postupného generování řetězců. Zde si v implementaci vystačíme s jednoduchou metodou, která načte daný slovník z disku – v tomto případě implementovanou jako reakci (událost) na stisk tlačítka. Dále už jen stačí zadat předlohu haše (vzor) a můžeme stisknout výpočetní tlačítko.

Celý algoritmus funguje v jednom cyklu, kdy postupně vyčítá řádky ze souboru až do konce. Tyto řádky postupně hašuje a porovnává se vzorem, dokud není nalezeno heslo nebo dosaženo konce souboru s potenciálními hesly.

Demonstrace je uvedena na obr. 17, kde opět použijeme haš nám známého hesla „aqua“. Heslo bylo opět prolomeno a bylo 3477tým heslem ve slovníku.



Obr. 17: Implementace slovníkového útoku

11. Závěr

Diplomová práce byla inspirována především výzkumy českého kryptologa Vlastimila Klímy. Zdroje, ze kterých jsem čerpal, jsou většinou sborníky z kryptografických konferencí publikované na internetu, nebo technické a implementační specifikace konkrétních hašovacích funkcí.

Hlavním cílem diplomové práce bylo popsat hašovací funkce, jejich základní vlastnosti a oblasti aplikace. Charakterizovat různé typy hašovacích funkcí a popsat metody, na kterých jsou založeny. Následně měla být z těchto funkcí jedna vybrána, aby byla demonstrativně implementována, a byly na ní provedeny různé metody útoků. Byla vybrána funkce MD5, na které jsem ukázal konstrukci moderní hašovací funkce, poukázal na nedostatky výpočetní složitosti kolizí a možné útoky na funkci. Nakonec byla samotná metoda i útoky implementovány.

V prvních kapitolách diplomové práce jsou shrnuty základní informace týkající se hašovacích funkcí. Jsou uvedeny jejich základní vlastnosti a možnosti jejich využití v reálné praxi. Ve třetí kapitole jsou charakterizovány základní metody, které slouží k vytváření obecných hašovacích funkcí a je uveden praktický příklad Fibonacciho hašování, který je založen na jedné z těchto metod, a to na metodě násobení. Dále je také uvedena hašovací funkce FNV, která je široce používána v různých systémech a programech. Ve čtvrté kapitole se zabývám dokonalým hašováním, statickým i dynamickým a rozebírám základní implementační metody. Pátá kapitola pojednává o hašovacích funkcích kryptografických. Jsou zde uvedeny nejpoužívanější soudobé kryptografické hašovací funkce, a je uvedena jejich stručná charakteristika.

V sedmé a osmé kapitole se zabývám implementací hašovacích funkcí FNV a MD5. V příslušných kapitolách jsou uvedeny implementační detaily a zobrazeny výsledky, jejichž korektnost byla ověřena oproti jejich technické dokumentaci.

Devátá kapitola se zabývá bezpečností hašovací funkce MD5 a složitostí hledání jejich kolizí. Na tuto kapitolu úzce navazuje kapitola desátá, ve které jsou rozebrány útoky podrobněji. Je vysvětlena metoda tunelování hašovacích funkcí, kterou vynalezl Vlastimil Klíma, metoda útoku brutální silou a slovníkový útok. Všechny tyto metody jsou popsány prakticky a následně implementovány a ověřeny v praxi.

V souvislosti s metodou hledání kolizí je uvedena praktická ukázka, kdy jsou generovány dvě stejné haše pro různé programy a tím je definitivně dokázáno, že hašovací funkce MD5 již nadále nemůže být považována za bezpečnou.

Literatura

[1] A Description of HAS-160. [Online] 30. 12 2003. [Citace: 10. 12 2008.] Dostupné z <http://www.randombit.net/text/has160.html>.

[2] Burda, K. *Bezpečnost Informačních systému. Skriptum.* Brno: VUT BRNO - FEKT, 2005. 104s

[3] Cormackovo hashování. [Online] 11. 8 2008. [Citace: 21. 10 2008.] Dostupné z: <http://cs.wikipedia.org/wiki/Cormackovo_hashov%C3%A1n%C3%AD>.

[4] Chauvaud., N. Rogier and P. *The compression function of MD2 is not collision free. Presented at Selected Areas in Cryptography '95.* Ottawa, Canada : Carleton University, 1995.

[5] *Differential Cryptanalysis of Feal and N-Hash.* Biham, E., Shamir, A. Dostupné z: <http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/E91/1.PDF>

[6] Differential Cryptanalysis of Sneferu, Khafe, REDOC-II, LOKI and Lucifer. Biham, E., Shamir, A. <http://*dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C91/156.PDF*>

[7] Division Method. *Divison Method.* [Online] [Citace: 19. 10 2008.] Dostupné z: <http://www.brpreiss.com/books/opus4/html/page211.html>.

[8] General Purpose Hash Function Algorithms. *General Purpose Hash Function Algorithms.* [Online] [Citace: 19. 10 2008.] Dostupné z: <http://www.partow.net/programming/hashfunctions/>.

[9] *Grindahl - a family of hash functions.* Knudsen, L.R., Rechberger, Ch., Tomsen, S.S. <http://www2.mat.dtu.dk/people/Lars.R.Knudsen/grindahl/grindahl.pdf>

[10] Hash function. *Wikipedia.* [Online] Wikipedia, 15. 9 2008. [Citace: 15. 10 2008.] Dostupné z: <http://en.wikipedia.org/wiki/Hash_function>.

[11] Hashing Methods. *Hashing Methods.* [Online] [Citace: 19. 10 2008.] Dostupné z: <http://www.cs.usask.ca/content/resources/tutorials/csconcepts/1998_1 /gen_hashing /2-0.html>.

[12] Haval. [Online] Calyptix Labs. [Citace: 26. 11 2008.] Dostupné z: <http://labs.calyptix.com/haval.php>.

[13] Fagin, R. *Extendible Hashing – A Fast Access Method for Dynamic File*s. [Online] Dostupné z: < http://www.almaden.ibm.com/cs/people/fagin/tods79.pdf>

[14] Fibonacci Hashing. *Fibonacci Hashing.* [Online] [Citace: 19. 10 2008.] Dostupné z: <http://www.brpreiss.com/books/opus4/html/page214.html>.

[15] FIPS 180-1 Secure Hash Standar. [Online] NIST, 11. 3 1993. [Citace: 22. 11 2008.] Dostupné z: <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.

[16] FIPS 180-2. [Online] NIST, 1. 8 2002. [Citace: 23. 11 2008.] Dostupné z: <http://www.governmentsecurity.org/articles/articles2/fips-180-2.pdf_fl/>

[17] Folding Method. *Folding Method.* [Online] [Citace: 19. 10 2008.] Dostupné z: <http://www.cs.usask.ca/content/resources/tutorials/csconcepts/1998_1/gen_hashing/2-3.html>.

[18] Klíma, V. *Hašovací funkce, principy, příklady a kolize* [Online][Citace: 1.5.2009]. Dostupné z: < http://cryptography.hyperlink.cz/2005/cryptofest_2005.htm >.

[19] Klíma, V. 2006. *Kolize MD5 do minuty aneb co v odborných zprávách nenajdete*. In: Crypto-World [Online]. 2006, [Citace: 4.5.2009] Dostupné z URL: <http://www.crypto-world.info/casop8/crypto04_06.pdf>.

[20] Klíma,V. 2004. *Nedůvěřujte kryptologům*. In: Crypto-World [Online].2004, vol. 6, no.11 [Citace 9.5.2009] <http://www.crypto-world.info/casop6/crypto11s_04.pdf >

[21] Klíma,V. 2006. *Tunely v hašovacích funkcích: kolize MD5 do minuty* [Online]. [Citace: 18.1.2009]. Dostupné z: < http://cryptography.hyperlink.cz/2006/tunely.pdf >.

[22] Klíma, V. *Cryptofest*. Cryptofest. [Online] 19. 3 2005. [Citace: 24. 1. 2009]

[23] Litwin, W., LINEAR HASHING : A NEW TOOL FOR FILE AND TABLE ADDRESSING. [Online] Dostupné z URL: <http://www.cs.cmu.edu/afs/ cs.cmu.edu/user/christos/www/courses/826-resources/PAPERS+BOOK/linear-hashing.PDF>

[24] LM Hash. *Wikipedia.* [Online] Wikipedia, 12. 10 2008. [Citace: 15. 12. 2008] <http://en.wikipedia.org/wiki/LM_hash>.

[25] Mendel, F., a kolektiv. Cryptoanalysis of the GOST Hash Function. Dostupné z: *<https://online.tugraz.ac.at/tug_online/voe_main2.getVollText?pDocumentNr=80200&p CurrP k=36649*

[26] Mao, W. *Modern Cryptology*, *Theory and Practice*. Prentice Hall : 2003.

[27] Middle Square Method. *Middle Square Method.* [Online] [Citace: 19. 10 2008.] <http://www.brpreiss.com/books/opus4/html/page212.html>.

[28] Multiplication Method. *Multiplication Method.* [Online] [Citace: 19. 10 2008.] <http://www.brpreiss.com/books/opus4/html/page213.html>.

[29] Noll, L., *FNV Hash*. [Online] FNV Hash. [Citace: 1. 5. 2009] Dostupné z URL: <http://isthe.com/chongo/tech/comp/fnv/history>

[30] Panama. *Panama.* [Online] [Citace: 5. 12 2008.] Dostupné z: <http://www.quadibloc.com/crypto/co4821.htm>.

[31] Pinkava, J. *Hashovací funkce v roce 2004.* Praha: PVT a.s., říjen 2004.

[32] Pokorný, J., Perfektní hašování. *Perfektní hašování.* Sv. <http://www.ksi.ms.mff.cuni.cz/~pokorny/vyuka/pokorny/cont.htm>

[33] RadioGatun. [Online] 10. 12 2008. [Citace: 2008. 13 2008.] <http://radiogatun.noekeon.org/>.

[34] RFC1319. [Online] 8 1992. [Citace: 15. 11 2008.] Dostupné z: <http://www.rfc-editor.org/rfc/rfc1319.txt>.

[35] RFC1320. [Online] 8 1992. [Citace: 16. 11 2008.] Dostupné z: <http://www.rfc-editor.org/rfc/rfc1320.txt>.

[36] RFC1321. [Online] 5 1992. [Citace: 16. 11 2008.] Dostupné z: <http://www.rfc-editor.org/rfc/rfc1321.txt>.

[37] The RIPMD-160. [Online] 25. 8 2004. [Citace: 25. 11 2008.] Dostupné z: <http://homes.esat.kuleuven.be/~bosselae/ripemd160.html>.

[38] The Whirlpool Hash Function. [Online] 25. 11 2008. [Citace: 2. 12 2008.] Dostupné z: <http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html>.

[39] Tiger: A Fast New Hash Function. [Online] [Citace: 30. 11. 2008] Dostupné z: <http://www.cs.technion.ac.il/~biham/Reports/Tiger/>.

[40] William G. Griswold, Gregg M Townsend. Software parctice and experience. *The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon.* http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue4/spe816.pdf.

[41] Denis St. T., Johnson S., *Cryptography for Developers*. Syngres. ISBN-10:1-59749-104-7,ISBN-13: 13-978-1-59749-104-4

[42] Schneier B., Applied *Cryptography: Protocols, Algorithms, and Source Code in C*, Wiley 1996. ISBN: 0471128457, 9780471128458

[43] Ferguson, N., Schneier B. *Cryptography*, Wiley 2003, ISBN 0471223573, 9780471223573

[44] Kaminsky, D. MD5 To Be Considered Harmful Someday.[Online] 6.12. 2004 [Citováno: 15.5.2009] Dostupné z: <http://www.doxpara.com/md5_someday.pdf>

Použité zkratky

CRC	Cyclic Redundancy Check, Cyklický redundantní součet
CRHF	Collision Resistant Hash Function, Bezkolizní hašovací funkce
DES	Data Encryption Standard, Šifra
FEAL	Fast Data Encipherment Algorithm, Bloková šifra
FIPS	Federal Information Processing Standard
FNV	Fowler/Noll/Vo hašovací funkce
FSE	Fast Software Encryption, Rychlé softwarové šifrování
GB	Giga byte, Jednotka objemu dat
HMAC	Hash MAC, Hašovaný autentizační kód zprávy
IEEE	Institute of Electrical and Electronics Engineers
IKE	Internet Key Exchange, Infrastruktura veřejných klíčů
IPSec	Internet Protocol Security, Kryptografický protokol
LM	Lan Manager, Síťový správce (hesel)
MAC	Message Authentication Code, Autentizační kód zprávy
MB	Giga byte, Jednotka objemu dat
MD	Message Digest, Rodina hašovacích funkcí
MGF	Mask Generation Function, Psudonáhodný generátor
NFS	Network File Systém, Souborový systém
NIST	National Institut of Standard and Technology
NSA	National Security Agency, Bezpečnostní agentura v USA
OWHF	One Way Hash Function, Jednosměrná hašovací funkce
PGP	Pretty Good Privacy, Kryptograficka aplikace
POV	Point Of Verification, Bod verifikace – používá se při tunelování hašů
RFC	Request For Comments
S/MIME	Secure Multipurpose Internet Mail Exchange, Kryptografický protokol
SHA	Secure Hash Algorithm, Rodina hašovacích funkcí
SSH	Secure SHell, Kryptografický protokol
SSL	Security Socket Layer, Kryptografický protokol
TLS	Transport Layer Security, Kryptografický protokol

Přílohy

A. CD

A.1	**Instalační balík programu**	**(Adresář**	\Instalace\)
A.2	Zdrojové kódy programu	**(Adresář**	\ZdrojoveKody\)
A.3	Diplomová práce	(Soubor	\xkaras21.pdf)
A.3	Slovník hesel	(Soubor	\dictionary.txt)