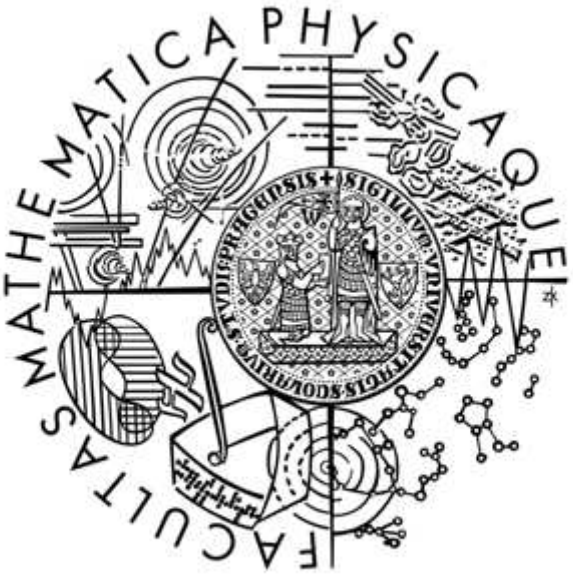
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Barbora Galaczová

Bezpečnost a použitelnost základních hashovacích funkcí, zejména MD-5, SHA-1 a SHA-2

Katedra algebry

Vedoucí diplomové práce: Doc. RNDr. Jiří Tůma, DrSc.

Studijní program: Matematika

Studijní obor: Matematické metody informační bezpečnosti

Praha 2011

Zde je vložena kopie „Zadání diplomové práce“.

Na tomto místě bych chtěla poděkovat svému vedoucímu diplomové práce Doc. RNDr. Jiřímu Tůmovi, DrSc a také svému konzultantovi Ing. Petru Budišovi, Ph.D. za cenné připomínky a rady a za trpělivost při konzultacích. Děkuji také své studijní referentce paní Marcele Všechovské za vstřícný přístup během celého mého studia. Děkuji také své rodině a přátelům za morální podporu při vytváření této práce.

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 2. srpna. 2011.

Název práce: *Bezpečnost a použitelnost základních hashovacích funkcí, zejména MD-5, SHA-1 a SHA-2*
Autor: *Galaczová Barbora*
Katedra: *Katedra algebry*
Vedoucí diplomové práce: *Doc. RNDr. Tůma Jiří, DrSc., Katedra algebry*
Konzultant: *Ing. Budiš Petr, Ph.D.*

Abstrakt: *V této práci se snažíme přehledně popsat základní hashovací funkce, zejména MD-5, SHA-1 a SHA-2. Uvádíme souhrn dosavadních útoků na tyto hashovací funkce, přičemž podrobně jsme se zaměřili především na postupy hledání kolizí funkce MD-5, neboť z těchto vycházejí útoky také na ostatní hashovací funkce. Dále popisujeme možnosti praktického využití kolizí hashovacích funkcí zejména v oblasti kvalifikovaných certifikátů a možné hrozby při jejich zneužití. V závěru uvádíme přehled nových hashovacích funkcí, jež mohou v budoucnu nahradit stávající. Součástí práce je také software na výpočet MD-5 hashe a hledání kolizí této hashovací funkce, založený na postupu českého kryptoanalytika Vlastimila Klímy.*

Klíčová slova: *hashovací funkce, kolize, kvalifikovaný certifikát, bezpečnost.*

Title: *Security and usability of standard hash functions, in particular MD-5, SHA-1 and SHA-2*
Author: *Galaczová Barbora*
Department: *Department of Algebra*
Supervisor: *Doc. RNDr. Tůma Jiří, DrSc., Department of Algebra*
Consultant: *Ing. Budiš Petr, Ph.D.*

Abstract: *In the present work we try to digestedly describe standard hash functions, in particular MD-5, SHA-1 and SHA-2. We describe resume of existing attacks on these hash functions. We closely focused on MD-5 collision attacks, because the other hash function collision attacks result from these. Next we describe posibilities of practical usage of hash function collisions, in particular into the qualified certificates area and possible threats. At the end to the present work we describe new hash functions, which could replace current hash functions. This work also contains software to calculate MD-5 hash and search it`s collisions. The software is based on method invented by Czech cryptoanalytist Vlastimil Klíma.*

Keywords: *hash function, collision, qualified certificate, security.*

Obsah

1. Úvod

V srpnu roku 2004 zveřejnili čínští matematici Xiaoyun Wang, Dengguo Feng, Xuejia Lai a Hongbo Yu zprávu o nalezení kolizí známé a hojně používané hashovací funkce MD5. Tato zpráva vzbudila pozornost nejen kryptologických odborníků, ale také laické veřejnosti.

Čínští matematici nalezli kolizi prvního řádu, tj. vytvořili dva různé vstupy se stejnou výstupní hashí. Tím dokázali, že hashovací funkce MD5 není tak bezpečná, za jakou byla považována. Zároveň veřejně upozorňovali, že na základě jejich výsledků by mělo být používání této hashovací funkce omezeno a následně zrušeno.

Na práci čínských matematiků navázal náš český kryptolog Vlastimil Klíma. Podařilo se mu snížit čas potřebný pro nalezení kolize, a to mnohonásobně. Čínští matematici nalezli kolizi zhruba za hodinu, Vlastimil Klíma toto dokázal za pouhých 17 vteřin. Svým výkonem poukázal na to, že útoky na hashovací funkci MD5 mohou představovat reálnou hrozbu.

Většina bankovních institucí, ale také jiné organizace, které potřebují chránit citlivé údaje, začaly pomalu přecházet na bezpečnější hashovací funkce, především SHA-1 a SHA-2. Avšak již od roku 2005 se začínají objevovat nové teoretické útoky na hashovací funci SHA-1. Možná nebude dlouho trvat a tato hashovací funkce také přestane vyhovovat bezpečnostním kritériím.

V této práci se zaměříme především na bezpečnost hashovacích funkcí MD5, SHA-1 a SHA-2. Pokusíme se zodpovědět otázku, zda je možné ještě těmto hashovacím funkcím důvěřovat (z bezpečnostního hlediska) a používat je.

2. Přehled hashovacích funkcí

2.1. Historický přehled

Než si uvedeme přehled používaných hashovacích funkcí, zamysleme se nad tím, proč se vlastně hashovací funkce začaly používat.

S rozvojem osobních počítačů v 70. a 80. letech 20. století, vznikla také potřeba bezpečnějších a výkonnějších šifer. Vznikly nové šifrovací algoritmy DES (Data Encryption Standard) a později AES (Advanced Encryption Standard). Tyto šifry patří do skupiny symetrických šifer, což znamená, že k šifrování a dešifrování zpráv se používá jeden a tentýž klíč. Jak rostl počet komunikujících stran a posílaných šifrovaných zpráv, tak rostl i počet klíčů, které bylo nutné doručit danému příjemci zprávy. Distribuce klíčů se stala velice časově i finančně náročná. Bylo nutné tuto situaci řešit.

Bylo jen otázkou času, kdy se objeví nový způsob šifrování. Asymetrické šifrování, neboli kryptografie s veřejným klíčem, úplně změnila dosavadní pohled na důležitost distribuce klíčů. Na kryptografickou scénu nyní vstupuje snad nejrozšířenější asymetrická šifra RSA (Rivest, Shamir, Adleman – podle svých autorů).

Asymetrická šifra, na rozdíl od symetrické používá dva klíče, veřejný a soukromý. Pokud chtějí dvě strany spolu komunikovat, obě si vygenerují dva klíče, veřejný a soukromý. Soukromý klíč je nutné držet v tajnosti. Pokud chce jedna strana – odesílatel poslat zprávu straně druhé – příjemci, zašifruje zprávu veřejným klíčem příjemce. V tomto případě má odesílatel naprostou jistotu, že zprávu si přečte pouze příjemce, neboť pouze on může zprávu dešifrovat svým soukromým klíčem.

Asymetrické algoritmy jsou však podstatně výpočetně složitější než algoritmy založené na symetrické kryptografii. Šifrování, či dešifrování delší zprávy asymetrickým algoritmem může trvat i několik minut. Výhodnější je tedy šifrovat kratší zprávy.

Hashovací funkce se začaly používat spolu s asymetrickými šiframi, především pro vytváření digitálních podpisů, kde není podstatné skrývat obsah zprávy, ale podstatné je zajistit jednoznačnou identifikaci podepisující osoby (osoby vytvářející digitální podpis zprávy). Vstupem hashovací funkce je řetězec znaků (bitů) libovolně dlouhý a výstupem je jiný řetězec znaků (bitů) určité konstantní délky, který se pak zašifruje asymetrickou šifrou. Hashovací funkce však nemůže být libovolná kompresní funkce, musí splňovat určité předpoklady, které uvedeme a podrobně rozebereme v následujících kapitolách.

2.2. Přehled používaných hashovacích funkcí

V této sekci shrneme a krátce představíme nejpoužívanější hashovací funkce, především MD5, SHA-1 a SHA-2 (tj. SHA-224, SHA-256, SHA-384, SHA-512).

- **MD4 a MD5.** Hashovací funcke MD4 a MD5 jsou zástupci funkcí třídy MD. Autorem obou hashovacích funkcí je Ronald Rivest (písmeno R v RSA). Funkce MD4 byla zkonstruována v roce 1990 a MD5 krátce po ní, jako její silnější a bezpečnější varianta. Ronald Rivest také vytvořil hashovací funkci MD2 (1989), ta se však nepoužívá (byla konstruována pro osmibitové procesory). Písmena „MD“ zastupují slova „*message digest*“, přehled, shrnutí zprávy (jiné prameny uvádějí, že označení „MD“ zastupuje iniciály jmen autorů konstrukce, na které jsou tyto hashovací funkce založeny, Merkleho a Damgarda). Hashovací funkce

třídy MD rozdělují vstupní zprávu do 512-ti bitových bloků. Délka výstupní hashe je 128 bitů.

- **SHA-0 a SHA-1.** SHA (Secure Hash Algorithm) byl původně vyvinut pro používání spolu s DSS (Digital Signature Standard) v roce 1993. Autorem algoritmu SHA je organizace NIST (National Institute of Standards and Technology), která publikovala několik hashovacích funkcí rodiny SHA jako normu FIPS (Federal Information Processing Standards). První verze SHA funkce, označována jako SHA-0 (norma FIPS 180, 1993) vychází z funkce MD4, délka výstupu je 160 bitů. V roce 1995 objevil NIST slabinu ve funkci SHA-0 a vydal tedy její opravenou variantu s názvem SHA-1 (norma FIPS 180-1, 1995).
- **SHA-224, SHA-256, SHA-384, SHA-512.** V roce 2002 NIST vydal tři nové hashovací funkce z rodiny funkcí SHA pod názvy SHA-256, SHA-384 a SHA- 521. Později, roku 2003 byla ještě přidána funkce SHA-224. Hashovací funkce SHA-256 a SHA-224 zpracovávají zprávu vždy po 512-bitových blocích a jejich výstup je dlouhý 256 bitů pro SHA-256 a 224 bitů pro SHA-224. Funkce SHA-512 zpracovává zprávu po 1024-bitových blocích a její výstupní hash je dlouhá 512 bitů. SHA-384 je triviální modifikací funkce SHA-512 s výstupní hashí délky 384 bitů.
- **Whirpool.** Hashovací algoritmus Whirpool byl navržen v rámci projektu Cryptonessie, jeho tvůrci jsou Paulo Barreto a Vincent Rijmen. Tento hashovací algoritmus je založen na principech blokové šifry AES [1], jeho výstupem je hash dlouhá 512 bitů.

Pro úplnost ještě dodejme, hashovací funkci HAVAL, kterou v roce 1992 navrhli Zheng, Pieprzyk a Seberry. HAVAL má několik různých variant s možnými výstupními hashi délky 128, 160, 192, 224 a 256 bitů.

Nakonec zmiňme tři relativně známé hashovací algoritmy *Snefru*, *Subhash* a *Tiger*.

Všechny hashovací algoritmy uvedené v této sekci i s dalšími informacemi může čtenář najít např. v [2].

Pro přehlednost shrňme ještě jednou uvedené, nejčastěji používané, hashovací funkce do tabulky Tab. 1.

Tab. 1: Přehled nejčastěji používaných hashovacích funkcí

Název	Rok standardizace	Autor	Velikost vstupního bloku (bity)	Velikost výstupní hashe (bity)
MD2	1989	Rivest	512	128
MD4	1990	Rivest	512	128
MD5	1992	Rivest	512	128
HAVAL	1992	Zheng, Pieprzyk a Seberry	1024	128, 160, 192, 224, 256
SHA-0	1993	NIST/NSA	512	160
SHA-1	1995	NIST/NSA	512	160
SHA-256	2002	NIST/NSA	512	256
SHA-384	2002	NIST/NSA	1024	384
SHA-512	2002	NIST/NSA	1024	512
SHA-224	2004	NIST/NSA	512	224
Whirpool	2003	Barreto, Rijmen	512	512

3. Vlastnosti hashovacích funkcí

Obecně se hashovací funkce chápe jako zobrazení h, které přiřazuje zprávě jako vstupu výstup označovaný slovem hash (hodnota hashe), resp. je to zobrazení, které řetězci libovolné délky přiřazuje řetězec pevné délky (většinou menší, než je délka původního řetězce). Samozřejmě v takovém případě je existence kolizí (dvojic vstupů se stejným výstupem) nevyhnutelná. Kryptografickou hashovací funkcí se pak rozumí funkce, která má navíc i určitou bezpečnostní vlastnost právě ve vztahu k možnostem vyhledávání kolizí. Předpokládá se, že popis hashovací funkce je veřejně znám, tj. samotný algoritmus není utajován.

Terminologie k hashovacím funkcím není v literatuře jednotná a mnohdy jsou vlastnosti hashovacích funkcí uváděny neformálně. Popišme nyní stručně základní vlastnosti, které by měla kryptografická hashovací funkce splňovat.

- **Praktická efektivnost.** Pro daný vstup x je výpočet výstupní hodnoty $h(x)$ efektivně proveditelný (je proveditelný v čase, který je omezen polynomiální funkcí délky vstupu x).
- **Mixující zobrazení.** Pro každý vstup x má výstupní hodnota $h(x)$ „náhodný“ charakter.
- **Rezistence vůči kolizím.** Je z výpočetního hlediska efektivně neproveditelné nalézt dva různé vstupy x, y, $x \neq y$ tak, aby $h(x) = h(y)$.Tato podmínka bezkoliznosti je nejsilnější požadavek na hashovací funkce. Složitost hledání kolize hrubou silou je u dobré hashovací funkce díky *narozeninovému paradoxu* rovna $2^{n/2}$, kde n délka výstupní hodnoty hashovací funkce v bitech. Výpočet a podrobnější informace lze nalézt např. v [3].
- **Rezistence vzoru (jednosměrnost).** Pro danou hodnotu hashe h je efektivně neproveditelné nalézt vstupní řetězec x tak, že $h = h(x)$.

Jednosměrnost (rezistence vzoru) je stěžejní vlastností hashovacích funkcí. Uveďme si tedy korektní definici jednosměrnosti.

Definice. Funkci $f : X \to Y$ nazveme *jednosměrnou*, pokud je pro libovolné $x \in X$ snadné (efektivně proveditelné) vypočítat hodnotu $y = f(x) \in Y$, ale pro libovolně zvolené $y \in Y$ je efektivně neproveditelné najít takovou hodnotu $x \in X$, že $y = f(x)$.

Některé prameny uvádějí ještě další vlastnosti hashovacích funkcí.

- **Rezistence druhého vzoru.** Je výpočetně neuskutečnitelné pro daný vstup x nalézt druhý vstupní řetězec y tak, že $h(x) = h(y)$. Tato vlastnost se od rezistance vůči kolizím liší tím, že zde je jedena vstupní hodnota (vstup x) již fixována
- **Rezistence vůči blízkým kolizím.** Je z výpočetního hlediska efektivně neproveditelné nalézt dva různé vstupy x, y, $x \neq y$ tak, že $h(x)$ a $h(y)$ se liší jen v malém počtu bitů. Blízké kolize jsou nejjednodušším příkladem zakázaných vztahů mezi výstupními hodnotami hashovací funkce.

Většina hashovacích funkcí (také funkce, kterými se zde budeme zabývat především, MD5, SHA-1 a SHA-2) vychází z využití tzv. komprestní funkce, která má pevnou délku

vstupu a zpracovává shodně každý blok zprávy. Někdy se takovýmto hashovacím funkcím říká také iterované hashovací funkce. Vstup do hashovací funkce je doplněn tak, aby délka vstupu byla násobkem délky bloku. Tedy pro hashovací funkci, která na vstupu zpracovává bloky délky K bitů, bude vstupní zpráva doplněna tak, aby měla nK bitů, kde n je přirozené číslo. Potom je vstup x rozdělen na jednotlivé bloky h_i, $i = 1,...,t$ délky K bitů a hash je počítán iterativně (C je iniciační konstanta).

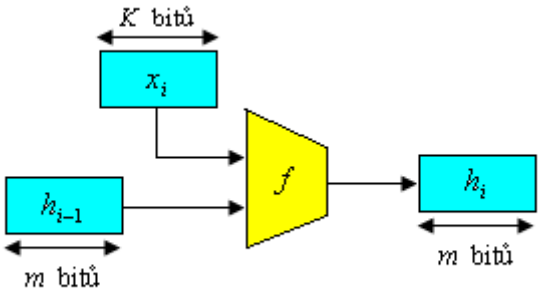
$$h_0 = C,$$

$$h_i = f(x_i, h_{i-1}),\ i = 1,...,t,$$

$$h(x) = h_t.$$

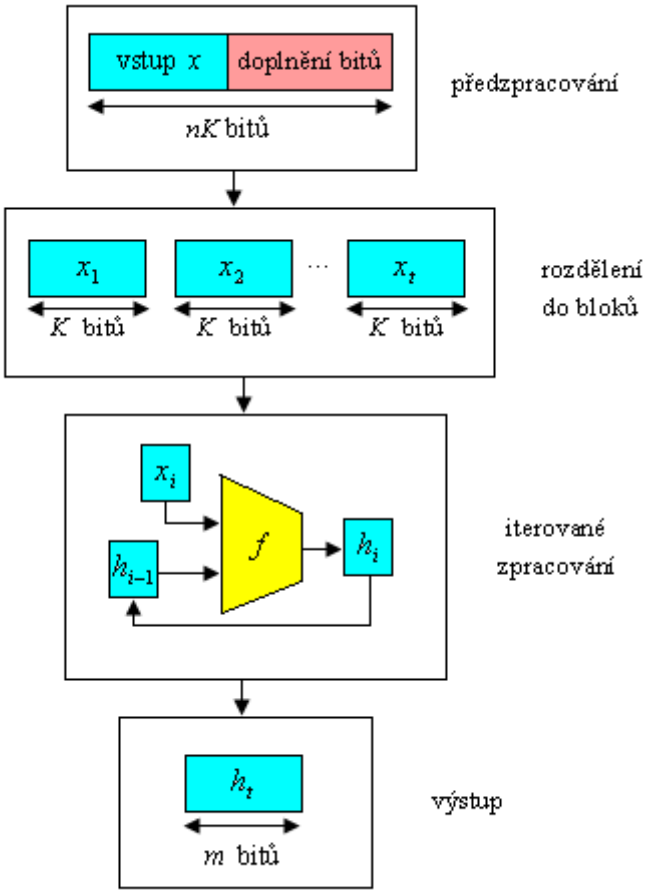
Zde f je kompresní funkce (iterovaná hashovací funkce) viz obrázek Obr. 1. Délka výsledné hodnoty hashe $h(x)$ je m bitů.

Obr. 1: Kompresní funkce



Na obrázku Obr. 2 vidíme obecný postup hashování pro většinu hashovacích funkcí. Tomuto postupu se také říká Merkle-Damgardova konstrukce (podle svých autorů). Doplnění bitů musí být prováděno tak, aby nemohly existovat dvě různé zprávy, které jsou po provedeném doplnění identické. Doplněk obsahuje v sobě také informaci o délce vstupního řetězce.

Obr. 2: Merkle-Damgardova konstrukce



4. Hashovací funkce MD5

Hashovací funkce MD5 je nejmladší funkcí navrženou Ronaldem Rivestem. Tato funkce byla navržena v roce 1991 a uznána v květnu 1992 jako doporučení RFC 1321 [5]. Funkce MD5 vychází z funkce MD4, jako její opravená, bezpečnější varianta. Obě funkce jsou třídy MD, tj. jsou založeny na Merkle-Damgardově konstrukci.

4.1. Popis algoritmu MD5

Než přistoupíme k popisu samotného algoritmu, osvětleme některé pojmy, které dále použijeme. Posloupnost bitů může být interpretována přirozeným způsobem jako posloupnost bytů, kde každá skupina 8 po sobě jdoucích bitů představuje jeden byte. V tomto případě první bit zleva je nejvíce významný bit (v příkladu označen červeně).

Příklad:

$$\mathtt{01010011\ 11110000} = 01010011_2\ 11110000_2 = 53_{16}\ f0_{16}$$

Podobně, posloupnost bytů můžeme interpretovat jako posloupnost 32-bitových slov. *Slovo* je po sobě jdoucí posloupnost čtyř bytů (tj. 32 bitů), kde první byte zleva je nejméně významný byte (toto uspořádání se nazývá *little endian* – little end first).

Přílad: Pokud chceme z posloupnosti bytů $\mathtt{ef}_{16}$, $\mathtt{cd}_{16}$, $\mathtt{ab}_{16}$, 89_{16} *vytvořit slovo, dostaneme* $\mathtt{89abcdef}_{16}$.

Následující označení je inspirováno prací [38]. Funkce MD5 používá 32-bitová slova v little endian uspořádání. Definujme bitovou reprezentaci pomocí číslic opatřených znaménky (BSDR – z anglického názvu *binary signed digit representation*) pro 32-bitové slovo X následujícím způsobem:

$$X = \sum_{i=0}^{31} 2^i k_i, \quad k_i \in \{-1,0,+1\}.$$

Zkráceným způsobem značíme $X = (k_i)_{i=0}^{31}$. Pro jedno slovo X může existovat více různých BSDR zápisů. Definujme *váhu* $w(Y)$ BSDR zápisu slova $Y = (k_i)_{i=0}^{31}$ jako součet nenulových hodnot k_i, kde $0 \le i \le 31$, tedy $w(Y) = \sum_{i=0}^{31} |k_i|$.

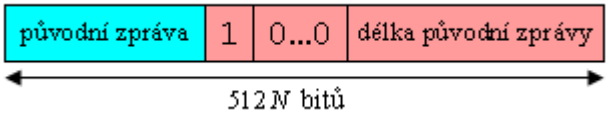
Poměrně užitečný BSDR zápis je ve tvaru tzv. nesousedící formy (NAF – z anglického názvu *Non-Adjacent Form*), kde žádné dvě nenulové hodnoty k_i nejsou sousední. Mezi všemi různými BSDR zápisy jednoho slova X je právě NAF zápis s nejnižší váhou.

Popišme si nyní, jak funguje vlastní algoritmus hashovací funkce MD5. Funkce MD5 zpracovává vstup po jednotlivých blocích délky 512 bitů. Nejprve se tedy celý vstupní řetězec předzpracovává a upravuje, aby mohl být následně rozdělen do 512-bitových bloků.

Doplnění bitů (*padding*) probíhá následovně. Na konec zprávy se přidá jeden bit rovný jedné (tj. přidá se bit `1`). Pak jsou doplněny nuly tak, aby vznikl soubor o délce, která je o 64 bitů kratší než násobek 512. Tím získáme zprávu o délce kongruentní 448 modulo 512. Zbývajících 64 bitů doplníme bitovým vyjádřením délky původní zprávy (tj. zprávy bez paddingu) v uspořádání little endian. Doplňování bitů je zobrazeno na obrázku Obr. 3.

Z celého postupu plyne, že funkce MD5 zpracovává zprávy, jejichž délka je menší než 2^{64} bitů. Takto upravený vstup se nyní rozdění na N 512-bitových bloků $M_1, M_2, ..., M_N$.

Obr. 3: Doplnění bitů do původní zprávy



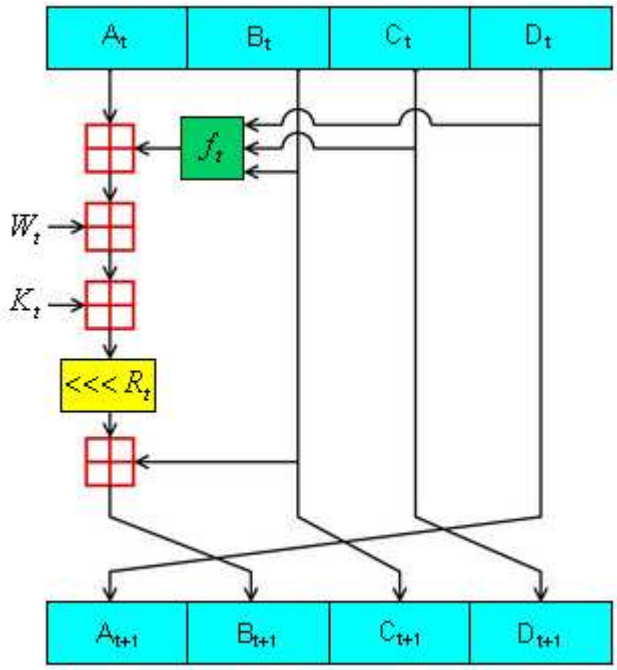
Hlavní část algoritmu MD5 pracuje s kontextem o velikosti 128 bitů rozděleným na čtyři 32-bitová slova A, B, C, D. Na začátku algoritmu jsou tato slova naplněna inicializačními hodnotami, které jsou veřejné a všeobecně známé. Hodnoty jednotlivých slov jsou uvedeny v tabulce Tab. 2, v little endian uspořádání (nejméně významný byte je první z leva).

Tab. 2: Inicializační hodnoty slov funkce MD5

Slovo	Hexadecimální hodnota
A_0	01234567
B_0	89abcdef
C_0	fedcba98
D_0	76543210

Algoritmus zpracovává vždy 512-bitový blok vstupu, výsledkem je nový 128-bitový kontext (skupina čtyř slov). Zpracování 512-bitového vstupu probíhá ve čtyřech cyklech. Každý cyklus se skládá z 16 operací založených na nelineární funkci, modulárním součtu a levé bitové rotaci. Tedy celkem proběhne 64 kroků (číslujeme od 0 do 63) v jedné rundě, ve které se zpracuje právě jeden 512-bitový blok vstupní zprávy. Obrázek Obr. 4 zachycuje detail komprestní funkce algoritmu. Symbol $\boxplus$ označuje sčítání slov (tj. sčítání modulo 2^{32}) a symbol $<<< R_t$ označuje rotaci bitů vlevo o R_t bitových pozic.

Obr. 4: Kompresní funkce MD5 algoritmu



Popišme si nyní, jak probíhá jedna runda. V závislosti na prováděném kroku t ($t = 0,...,63$) se použije aditivní konstanta K_t a konstanta rotace R_t. Tyto konstanty jsou definovány následovně.

$$K_t = \lfloor 2^{32} |\sin(t+1)| \rfloor,\ 0 \le t < 64$$

Přičemž t je vyjádřeno v radiánech.

$$(R_t, R_{t+1}, R_{t+2}, R_{t+3}) = \begin{cases} (7,12,17,22) & \text{pro } t = 0,4,8,12, \\ (5,9,14,20) & \text{pro } t = 16,20,24,28, \\ (4,11,16,23) & \text{pro } t = 32,36,40,44, \\ (6,10,15,21) & \text{pro } t = 48,52,56,60. \end{cases}$$

Nelineární funkce f_t také závisí na prováděném kroku t.

$$f_t(X,Y,Z) = \begin{cases} F(X,Y,Z) = (X \wedge Y) \vee (\overline{X} \wedge Z) & \text{pro } 0 \le t < 16, \\ G(X,Y,Z) = (Z \wedge X) \vee (\overline{Z} \wedge Y) & \text{pro } 16 \le t < 32, \\ H(X,Y,Z) = X \oplus Y \oplus Z & \text{pro } 32 \le t < 48, \\ I(X,Y,Z) = Y \oplus (X \vee \overline{Z}) & \text{pro } 48 \le t < 64. \end{cases}$$

Tento soubor vzorců pro f_t neříká nic jiného, než že pro kroky 0, …, 15 se použije funkce F, pro kroky 16, …, 31 se použije funkce G, pro kroky 32, …, 47 se použije funkce H a pro kroky 48, …, 63 se použije funkce I. Proměnné X, Y, Z, reprezentují postupně vstupní slova B_t, C_t, D_t funkce f_t, jak je vidět na obrázku Obr. 4. Všechny operace se provádějí na bitové úrovni, přičemž $\overline{X}$ je bitový doplněk (operace NOT) a $\oplus$ značí bitový součet (operace XOR).

Vstupní 512-bitový blok zprávy se rozdělí na šestnáct 32-bitových slov $m_0, m_1, ..., m_{15}$ (v little endian uspořádání). Tato slova jsou expandována do celkového počtu 64 slov $(W_t)_{t=0}^{63}$, pro každý krok t, užitím následujících relací (zjednodušeně, každé slovo se v jedné rundě použije čtyřikrát).

$$W_t = \begin{cases} m_t & \text{pro} \quad 0 \le t < 16, \\ m_{(1+5t) \bmod 16} & \text{pro} \quad 16 \le t < 32, \\ m_{(5+3t) \bmod 16} & \text{pro} \quad 32 \le t < 48, \\ m_{(7t) \bmod 16} & \text{pro} \quad 48 \le t < 64. \end{cases}$$

Hodnoty všech aditivních konstant K_t, rotačních konstant R_t a expandovaných slov W_t pro každý jednotlivý krok $t = 0,...,63$ v rámci jedné rundy jsou přehledně uvedeny v tabulce Tab. 3.

Po proběhnutí celé první rundy získáme hodnoty čtyř registrů A_{64}, B_{64}, C_{64} a D_{64} (32-bitová slova). K těmto hodnotám přičteme postupně vstupní hodnoty těchto registrů A_0, B_0, C_0 a D_0 a získáme 128-bitovou mezihodnotu hashe

$$H = (A_0 + A_{64}, B_0 + B_{64}, C_0 + C_{64}, D_0 + D_{64}),$$

která se použije jako iniciační hodnota čtyř registrů při zpracovávání dalšího 512-bitového bloku. Výsledná hashová hodnota se objeví ve čtyřech registrech (128-bitů dlouhý řetězec) po proběhnutí poslední rundy, tj. po zpracování celé vstupní zprávy.

Popis kompresní funkce MD5 algoritmu pomocí čtyř registrů je názorný a snadno pochopitelný, avšak v dalších kapitolách lépe užijeme popis pomocí jednoho registru. Pokud se podíváme na kompresní funkci MD5 algoritmu podrobněji, můžeme si všimnout, že v průběhu každého kroku rundy se přepočítá pouze jeden registr, ostatní se pouze prohodí. Uveďme si nyní také vzorce pro popis kompresní funkce pomocí jednoho registru.

Pro $t = 0,...,63$ máme hodnoty čtyř slov Q_t, Q_{t-1}, Q_{t-2} a Q_{t-3}. Tato slova jsou inicializována hodnotami $(Q_0, Q_{-1}, Q_{-2}, Q_{-3}) = (B_0, C_0, D_0, A_0)$. Dále pro $t = 0,...,63$ platí:

$$F_t = f_t(Q_t, Q_{t-1}, Q_{t-2}),$$

$$T_t = F_t + Q_{t-3} + K_t + W_t,$$

$$RL_t = ROTL(T_t, R_t),$$

$$Q_{t+1} = Q_t + RL_t.$$

Kde F_t je pouze zjednodušené označení pro nelineární funkci f_t, T_t je kompresní funkce algoritmu a $ROTL(T_t, R_t)$ značí levou bitovou rotaci řetězce T_t o R_t pozic.

Po proběhnutí první rundy získáme mezihodnotu hashe:

$$H = (A_0 + Q_{61}, B_0 + Q_{64}, C_0 + Q_{63}, D_0 + Q_{62}).$$

Tab. 3: Přehled aditivních a rotačních konstant a expandovaných slov

t	K_t	R_t	W_t
0	$\texttt{d76aa478}_{16}$	7	m_0
1	$\texttt{e8c7b756}_{16}$	12	m_1
2	$\texttt{242070db}_{16}$	17	m_2
3	$\texttt{c1bdceee}_{16}$	22	m_3
4	$\texttt{f57c0faf}_{16}$	7	m_4
5	$\texttt{4787c62a}_{16}$	12	m_5
6	$\texttt{a8304613}_{16}$	17	m_6
7	$\texttt{fd469501}_{16}$	22	m_7
8	$\texttt{698098d8}_{16}$	7	m_8
9	$\texttt{8b44f7af}_{16}$	12	m_9
10	$\texttt{ffff5bb1}_{16}$	17	m_{10}
11	$\texttt{895cd7be}_{16}$	22	m_{11}
12	$\texttt{6b901122}_{16}$	7	m_{12}
13	$\texttt{fd987193}_{16}$	12	m_{13}
14	$\texttt{a679438e}_{16}$	17	m_{14}
15	$\texttt{49b40821}_{16}$	22	m_{15}

t	K_t	R_t	W_t
16	$\texttt{f61e2562}_{16}$	5	m_1
17	$\texttt{c040b340}_{16}$	9	m_6
18	$\texttt{265e5a51}_{16}$	14	m_{11}
19	$\texttt{e9b6c7aa}_{16}$	20	m_0
20	$\texttt{d62f105d}_{16}$	5	m_5
21	$\texttt{2441453}_{16}$	9	m_{10}
22	$\texttt{d8a1e681}_{16}$	14	m_{15}
23	$\texttt{e7d3fbc8}_{16}$	20	m_4
24	$\texttt{21e1cde6}_{16}$	5	m_9
25	$\texttt{c33707d6}_{16}$	9	m_{14}
26	$\texttt{f4d50d87}_{16}$	14	m_3
27	$\texttt{455a14ed}_{16}$	20	m_8
28	$\texttt{a9e3e905}_{16}$	5	m_{13}
29	$\texttt{fcefa3f8}_{16}$	9	m_2
30	$\texttt{676f02d9}_{16}$	14	m_7
31	$\texttt{8d2a4c8a}_{16}$	20	m_{12}

t	K_t	R_t	W_t
32	$\texttt{fffa3942}_{16}$	4	m_5
33	$\texttt{8771f681}_{16}$	11	m_8
34	$\texttt{6d9d6122}_{16}$	16	m_{11}
35	$\texttt{fde5380c}_{16}$	23	m_{14}
36	$\texttt{a4beea44}_{16}$	4	m_1
37	$\texttt{4bdecfa9}_{16}$	11	m_4
38	$\texttt{f6bb4b60}_{16}$	16	m_7
39	$\texttt{bebfbc70}_{16}$	23	m_{10}
40	$\texttt{289b7ec6}_{16}$	4	m_{13}
41	$\texttt{eaa127fa}_{16}$	11	m_0
42	$\texttt{d4ef3085}_{16}$	16	m_3
43	$\texttt{04881d05}_{16}$	23	m_6
44	$\texttt{d9d4d039}_{16}$	4	m_9
45	$\texttt{e6db99e5}_{16}$	11	m_{12}
46	$\texttt{1fa27cf8}_{16}$	16	m_{15}
47	$\texttt{c4ac5665}_{16}$	23	m_2

t	K_t	R_t	W_t
48	$\texttt{f4292244}_{16}$	6	m_0
49	$\texttt{432aff97}_{16}$	10	m_7
50	$\texttt{ab9423a7}_{16}$	15	m_{14}
51	$\texttt{fc93a039}_{16}$	21	m_5
52	$\texttt{655b59c3}_{16}$	6	m_{12}
53	$\texttt{8f0ccc92}_{16}$	10	m_3
54	$\texttt{ffeff47d}_{16}$	15	m_{10}
55	$\texttt{85845dd1}_{16}$	21	m_1
56	$\texttt{6fa87e4f}_{16}$	6	m_8
57	$\texttt{fe2ce6e0}_{16}$	10	m_{15}
58	$\texttt{a3014314}_{16}$	15	m_6
59	$\texttt{4e0811a1}_{16}$	21	m_{13}
60	$\texttt{f7537e82}_{16}$	6	m_4
61	$\texttt{bd3af235}_{16}$	10	m_{11}
62	$\texttt{2ad7d2bb}_{16}$	15	m_2
63	$\texttt{eb86d391}_{16}$	21	m_9

4.2. Vývoj útoků na funkci MD5

Funkce MD5 se rozšířila mezi veřejnost a byla používána v různých bezpečnostních aplikacích. Byla standardizována v roce 1992 a již o rok později dochází k prvním útokům. Bert den Boer a Antoon Bosselaers [8] ukázali slabinu ve funkci MD5, když našli „pseudo kolizi“ pro MD5. Útok spočíval v použití stejné zprávy s odlišnými iniciačními hodnotami slov.

Roku 1996 publikoval Hans Dobbertin [9] částečné kolize, kdy použil dvě různé 512-bitové zprávy a zvolené iniciační hodnoty slov. Nedokázal však nalézt reálný případ kolize funkce MD5.

Funkce MD5 vrací výstupní hash délky 128 bitů, což je dostatečně malá hodnota pro útok hrubou silou, například narozeninovým útokem (založeném na narozeninovém paradoxu) řádu 2^{64}. O takovýto útok hrubou silou se pokusil distribuovaný výpočetní projekt MD5CRK, který začal v březnu roku 2004. Nicméně tento projekt skončil v srpnu stejného roku, kdy Wangová a kol. [10] zveřejnili nalezené kolize pro více hashovacích funkcí, mimo jiné také pro funkci MD5. Tedy projekt MD5CRK zůstal nedokončen a o jeho potenciálním úspěchu se můžeme dnes pouze dohadovat.

Později roku 2005 Wangová a Yu [11] představili zásadní metodu na konstrukci kolizí použitím diferenciální kryptoanalýzy. Složitost tohoto útoku byl odhadnut na 2^{39} volání kompresní funkce algoritmu MD5. Tento způsob útoku byl poté několikrát zdokonalen japonskými [12] a také českými [13] matematiky. Nejrychlejší úprava algoritmu Vlastimila Klímy [14] dokáže najít kolizi u této funkce již do jedné minuty na běžném PC.

Popišme si nyní podrobněji kolizní metodu dr. Wangové a některá vylepšení, pro zrychlení hledání kolizí touto metodou.

4.2.1. Kolize podle Wangové a kol.

Tato část vychází z práce dr. Wangové a kol. [11].

Cílem týmu dr. Wangové bylo nalézt algoritmus pro hledání kolizí funkce MD5. Zaměřili se na hledání kolizí krátkých zpráv o dvou 512-bitových blocích. Speciálně hledali dvojici zpráv (M_0, M_1) a (M_0', M_1') pro které platí:

$$(A,B,C,D) = \mathrm{MD5}(A_0, B_0, C_0, D_0, M_0),$$

$$(A',B',C',D') = \mathrm{MD5}(A_0, B_0, C_0, D_0, M_0'),$$

$$\mathrm{MD5}(A,B,C,D,M_1) = \mathrm{MD5}(A',B',C',D',M_1'),$$

kde A_0, B_0, C_0, D_0 jsou iniciační hodnoty slov funkce MD5, (A,B,C,D) je mezihodnota hashe po zpracování 512-bitového bloku M_0 a (A',B',C',D') je mezihodnota hashe po zpracování 512-bitového bloku M_0'. Tento algoritmus se dá zobecnit na dvojici zpráv jakékoliv větší délky ve smyslu hledání kolize dvou sousedních 512-bitových bloků zpráv.

Metoda hledání kolizí týmu dr. Wangové je založena na útoku pomocí diferenciální kryptoanalýzy [6]. Diferenciální kryptoanalýza spočívá v pozorovnání vlivu malé změny ve dvojici otevřeného textu na změnu ve dvojici výsledného šifrového textu. Většina diferenciálních útoků používá exkluzivní or (xor) jako měřítko rozdílu, útok dr. Wangové

používá však také modulární celočíselné odčítání (modulo 2^{32}). Tento typ rozdílu se nazývá *modulární rozdíl.*

Kombinací obou druhů rozdílů (modulární a xor) můžeme získat více informací než použitím pouze jednoho z nich. Pomocí obou rozdílů se zkonsturují dvě *diferenční cesty*. Tyto diferenční cesty podrobně popisují promítnutí rozdílu ve slovech vstupních zpráv do kompresní funce algoritmu MD5. Následně se určí *postačující podmínky* pro hodnoty jednotlivých bitů některých slov registru. Nakonec se upraví vstupní zprávy tak, aby byly splněny tyto postačující podmínky.

Modulární rozdíl a XOR rozdíl

Modulární rozdíl dvou slov X a X' označíme $\delta X = X' - X$. Použijeme také modulární rozdíl mezi každým párem bitů $X[i]$ a $X'[i]$ pro $0 \le i \le 31$. Získáme tak hodnoty (-1, 0 nebo +1) pro každý bit modulárního rozdílu $\Delta X = (k_i)$, kde $k_i = X'[i] - X[i]$ pro $0 \le i \le 31$. Ve zřejmém smyslu také platí $\delta X \equiv \Delta X$.

Pro příklad předpokládejme, že hodnota modulárního rozdílu je $\delta X = X' - X = 2^6$. Pak máme více možností pro hodnotu XOR rozdílu $X' \otimes X$.

- *Rozdíl v jednom bitu.* Rozdíl na pozici 6, tedy $X'[6] = 1,\ X[6] = 0$, ostatní bity jsou v obou slovech stejné. Pak platí $\delta X = 2^6$ a $X' \otimes X = \texttt{00000040}_{16}$.
- *Rozdíl ve dvou bitech.* Rozdíl na pozicích 6 a 7 způsobený *přenosem*, tedy $X'[6] = 0$, $X[6] = 1$ a $X'[7] = 1,\ X[7] = 0$. Pak platí $\delta X = -2^6 + 2^7 = 2^6$ a $X' \otimes X = \texttt{000000c0}_{16}$.
- *Rozdíl v n bitech.* Rozdíl na pozicích 6 až $6+n-1$ způsobený $n-1$ přenosy, kde $3 \le n \le 26$. Tedy $X'[i] = 0,\ X[i] = 1$ pro $i = 6,...,6+n-2$ a $X'[6+n-1] = 0$, $X[6+n-1] = 1$. Pak platí $\delta X = -2^6 - 2^7 - ... - 2^{6+n-2} + 2^{6+n-1} = 2^6$.
- *Rozdíl ve 26 bitech.* Rozdíl na pozicích 6 až 31 způsobených 26 přenosy, tedy $X'[i] = 0$ a $X[i] = 1$ pro $i = 6,...,31$. Pak platí $\delta X = -2^6 - 2^7 - ... - 2^{31} = 2^6 \pmod{2^{32}}$.

Počáteční podmínky

Dvě diferenční cesty se zkonstruují pomocí počátečních rozdílů jednotlivých bloků vstupních zpráv.

$$\delta H_0 \xrightarrow{(M_0, M_0')} \delta H_1 \xrightarrow{(M_1, M_1')} \delta H = 0$$

První diferenční cesta je založena na počátečních rozdílech prvních bloků zpráv.

$$\delta M_0 = M_0' - M_0 = \delta m_i = (0, 0, 0, 0, 2^{31}, 0, 0, 0, 0, 0, 0, 2^{15}, 0, 0, 2^{31}, 0), 0 \le i \le 15$$

Druhá diferenční cesta je založena na opačných rozdílech druhých bloků zpráv.

$$\delta M_1 = M_1' - M_1 = \delta m_i = (0, 0, 0, 0, -2^{31}, 0, 0, 0, 0, 0, 0, -2^{15}, 0, 0, -2^{31}, 0), 0 \le i \le 15$$

Navíc platí $-2^{31} \equiv 2^{31} \bmod 2^{32}$, tedy ve skutečnosti δm_4 a δm_{14} se neliší ve znaménku.

Hodnota δH_0 je rozdíl jednotlivých slov registru (v tomto případě naplněn iniciačními hodnotami) před zpracováním prvního bloku zpráv, tedy je rovna 0. Hodnota δH_1 je rozdíl

jednotlivých slov registru po zpracování prvního bloku zpráv. Podlední hodnota δH je rozdíl výsledních hashí, jež požadujeme, aby byl nulový.

$$\delta H_0 = (\delta A_0, \delta B_0, \delta C_0, \delta \mathrm{D}_0) = 0,$$

$$\delta H_1 = (\delta A, \delta B, \delta C, \delta \mathrm{D}) = (2^{31}, 2^{31} + 2^{25}, 2^{31} + 2^{25}, 2^{31} + 2^{25}).$$

Poznamenejme ještě, že počáteční podmínky rozdílu vstupních zpráv byly autory pečlivě voleny pro zajištění vysoké pravděpodobnosti očekávaného výsledku a snížení složitosti algoritmu pro vyhledávání kolizí.

Diferenční cesty

Diferenční cesty pro oba bloky M_0 a M_1 vstupní zprávy jsou sestaveny tak, aby došlo k předpokládané kolizi. Tyto diferenční cesty podrobně popisují jakým způsobem se rozdíly ve vstupních blocích ∂M_0 a ∂M_1 postupně distribuují kompresní a rotační funkcí algoritmu MD5, viz tabulky Tab. MD5-1 a Tab. MD5-2 v příloze. Konkrétně je diferenční cesta popsána hodnotami ∂m_t, ∂Q_t a ∂T_t, pro každý krok $t = 0,...,64$.

První diferenční cesta začíná s hodnotou $\delta H_0 = 0$ a první nenulová hodnota se objeví v kroku $t = 4$ pro $\delta m_4 = 2^{31}$. Druhá diferenční cesta začíná již s nějakou danou nenulovou hodnotou δH_1. Pokud si obě tabulky prohlédmeme pozorněji, zjistíme, že v obou případech dojde k vynulování všech hodnot právě v kroku $t = 25$ a počínaje krokem $t = 34$ obsahují obě diferenční cesty stejné hodnoty jen v opačných znaménkách. Tedy po zpracování obou bloků zpráv získáme požadovanou hodnotu rozdílu výsledných hashí $\delta H = 0$.

Tým dr. Wangové sestavoval obě diferenční cesty ručně, přičemž zde uplatnili své vynikající zkušenosti a nespornou intuici. Právě nad automatizací konstrukce diferenčních cest visel po značnou dobu velký otazník. V tomto období byla zveřejněna skvělá práce [15], která se podrobně zabývá ručním nalezením diferenčních cest a ukazuje také pravděpodobný postup čínských matematiků.

Postačující podmínky

Postačující podmínky lze použít pro efektivnější hledání bloků zpráv takových, pro které jsou splněny diferenční cesty uvedené výše. Tyto postačující podmínky zaručí, že dojde k nezbytným bitovým přenosům a k zachování rozdílů některých bitů v kompresní funci algoritmu MD5.

Poznámka. Značení čínských matematiků je těžkopádné a zbytečně složité. Použijeme tedy stejné značení jako v práci [7], jež je jednodušší na pochopení i vysvětlení.

Každá podmínka určuje hodnotu i-tého bitu $Q_t[i]$ daného slova registru Q_t v kroku t. Tato hodnota je dána přímo – 1, 0, případně žádná hodnota (o této hodnotě nelze nic říct) nebo nepřímo – hodnota je stejná (resp. opačná) jako hodnota stejného bitu v předchozím kroku. Tyto informace jsou přehledně uvedety v tabulce Tab. 4.

Tab. 4: Postačující podmínky – způsob značení

Symbol	Podmínka na $Q_t[i]$	přímá/nepřímá
`.`	žádná	přímá
`0`	$Q_t[i] = 0$	přímá
`1`	$Q_t[i] = 1$	přímá
`^`	$Q_t[i] = Q_{t-1}[i]$	nepřímá
`!`	$Q_t[i] = \overline{Q_{t-1}[i]}$	nepřímá

Postačující podmínky se stanoví pouze pro bloky vstupní zprávy (M_0, M_1), jež jsou pak použity pro hledání bloků druhé zprávy (M_0', M_1') tak, aby byly splněny obě diferenční cesty. Přehled všech podmínek pro jednotlivé bity $Q_t = b_{31}...b_0$ jsou uvedeny v tabulce Tab. MD5-3 pro první blok M_0 a v tabulce Tab. MD5-4 pro druhý blok M_1. Lze také sestavit další sady postačujících podmínek, v závislosti na použití dané diferenční cesty.

Poznamenejme, že postačující podmínky tak, jak je uvádí Wangová a kol. [11], ve skutečnosti postačující nejsou a dokonce nejsou ani zcela korektní. Například v práci [16] autoři Yajima a Shimoyama navrhli částečnou korekci postačujících podmínek. Autoři práce [12] koncem roku 2005 ukázali nesprávnost některých postačujících podmínek Wangové a kol. a předložili novou úplnou sadu postačujících podmínek, která se zdá již být konečná a správná.

Hlavní důvod tolika korekcí je ten, že při sestavování postačujících podmínek je potřeba provést precizně velké množství jednodušších výpočtů a úvah, při jejichž ručním provádění se člověk může jednoduše splést. Automatické strojové zpracování však zatím zřejmě narazilo na nějaké problémy a není k dispozici.

Poznámka: V tabulkách Tab. MD5-3 a Tab. MD5-4 jsou černě uvedeny hodnoty postačujících podmínek podle Wangové a kol. [11] a modře další podmínky nalezené Stevensem [7].

Modifikace zpráv

Použitím těchto postačujících podmínek, lze efektivně hledat oba bloky vstupní zprávy (M_0, M_1). Jednoduše lze zajistit, že náhodně zvolené bloky M_0 a M_1 budou splňovat všechny postačující podmínky v první rundě (tj. v prvních 16-ti krocích). K tomu tým dr. Wangové používá tzv. *jednoduchou modifikaci zpráv.*

Příklad: Abychom zajistili platnost podmínek pro Q_3, tedy $Q_3[6] = 0$, $Q_3[11] = 0$, $Q_3[19] = 0$ (viz Tab. MD5-3), provedeme následující modifikaci vstupního slova m_2:

$$\tilde{Q}_3 \leftarrow Q_3 - Q_3[6] \cdot 2^6 - Q_3[11] \cdot 2^{11} - Q_3[19] \cdot 2^{19},$$

$$\tilde{m}_2 \leftarrow RR(\tilde{Q}_3 - Q_3, 17) + m_2,$$

kde $RR(A, x)$ značí pravou rotaci slova A o x pozic.

Po drobnější úpravě získáme nakonec jednoduchý vzorec:

$$\tilde{m}_2 \leftarrow m_2 - Q_3[6] \cdot 2^{21} - Q_3[11] \cdot 2^{26} - Q_3[19] \cdot 2^2.$$

Z čehož je zřejmé, že ve vstupním slově m_2 upravujeme skutečně pouze tři bity, a to právě ty, které nás zajímají. Podobně lze upravit všechna slova vstupního bloku M_0 (resp. M_1). Takto získáme platnost všech postačujících podmínek v první rundě.

Zbývající postačující podmínky jsou splněny jen s určitou pravděpodobností použitím *mnohonásobné modifikace zpráv*.

Příklad: Pro splnění první diferenční cesty požadujeme, aby platilo $Q_{17}[31] = 0$ *(viz Tab. MD5-3). Předpokládejme, že máme* $Q_{17}[31] = 1$. *Tuto chybu můžeme opravit modifikací slov* m_1, m_2, m_3, m_4, m_5*:*

$$\tilde{m}_1 \leftarrow \left(m_1 + 2^{26}\right), \text{tím dojde ke změně } Q_2 \text{ na } \tilde{Q}_2,$$

$$\tilde{m}_2 \leftarrow (RR(Q_3 - \tilde{Q}_2, 17) - Q_{-1} - F(\tilde{Q}_2, Q_1, Q_0) - K_2),$$

$$\tilde{m}_3 \leftarrow (RR(Q_4 - Q_3, 22) - Q_0 - F(Q_3, \tilde{Q}_2, Q_1) - K_3),$$

$$\tilde{m}_4 \leftarrow (RR(Q_5 - Q_4, 7) - Q_1 - F(Q_4, Q_3, \tilde{Q}_2) - K_4),$$

$$\tilde{m}_5 \leftarrow (RR(Q_6 - Q_5, 12) - \tilde{Q}_2 - F(Q_5, Q_4, Q_3) - K_5).$$

V tomto souboru změn je nejdůležitější hned první řádek. Přidáním změny $+2^{26}$ ve slově m_1 získáme změnu $+2^{31}$ pro bit $Q_{17}[31]$, čímž dostaneme požadovanou hodnotu $Q_{17}[31] = 0$. Další čtyři řádky naznačují jednoduchou změnu slov m_2, m_3, m_4 a m_5, které zajistí, že se potřebná změna ve slově m_1 nepromítne do hodnot registrů Q_3 až Q_{16}. Neboť nemáme žádné podmínky pro Q_2, všechny předchozí podmínky zůstanou také nezměněny.

Algoritmus útoku

Následující algoritmus popisuje, jak nalézt kolizi zpráv o velikosti dvou 512-ti bitových bloků. Máme tedy zprávu (M_0, M_1), kterou v průběhu výpočtu postupně upravujeme a hledáme takovou zprávu (M_0', M_1'), že hodnoty hashe obou zpráv budou stejné.

1. Opakujte následující kroky dokud není nalezen první blok M_0', splňující podmínky (c) a (d):
   (a) vyberte náhodně blok zprávy M_0,
   (b) upravte blok M_0 pomocí modifikace zpráv, popsané v předcházejícím úseku,
   (c) pak bloky M_0 a $M_0' = M_0 + \Delta M_0$ produkují první diferenční hodnotu meziheshe $\Delta H_1 = H_1' - H_1$ s pravděpodobností 2^{-37},
   (d) ověřte, zda jsou splněny všechny charakteristiky (první diferenční cesta a postačující podmínky) pro kompresní funkci použitou na bloky M_0 a M_0'.
2. Opakujte následující kroky dokud není nalezena kolize:
   (a) vyberte náhodně blok zprávy M_1,
   (b) upravte blok M_1 pomocí modifikace zpráv, popsané v předcházejícím úseku,
   (c) pak bloky M_1 a $M_1' = M_1 + \Delta M_1$ produkují výslednou diferenční hodnou hashe $\Delta H = H' - H = 0$ s pravděpodobností 2^{-30},
   (d) ověřte, zda takto získané zprávy generují hledanou kolizi.

Složitost první části algoritmu, tj. hledání bloků M_0 a M_0' nepřevyšuje 2^{39} MD5 operací. Složitost druhé částli algoritmu, tj. hledání bloků M_1 a M_1' nepřevyšuje 2^{32} MD5 operací.

V tabulce Tab. 5 je uveden příklad dvou párů kolidujících zpráv (dvě různé zprávy se stejnou výslednou hashí) pro MD5. *H* je hodnota hashe v little-endian uspořádání bez použití paddingu a *H** je hodnota hashe v big-endian uspořádání s použitím paddingu. K tomuto příkladu poznamenejme jen, že obě kolidující zprávy začínají stejným prvním blokem ($M_0 = M_0'$), čímž je zajištěno, že všechny podmínky pro první blok zprávy jsou splněny. Pak je možné najít více různých 512-ti bitových bloků M_1 a M_1', které vedou ke kolizi.

Tab. 5: Dva páry kolidujících zpráv pro funkci MD5 (odlišnosti jsou označeny modře).

M_0	02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
M_1	d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 0cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35
M_0'	02dd31d1 c4eee6c5 069a3d69 5cf9af98 07b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a41f125 e8255108 9fc9cdf7 72bd1dd9 5b3c3780
M_1'	d11d0b96 9c7b41dc f497d8e4 d555655a 479a7335 0cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c154c79 ddcb74ed 6dd3c55f 580a9bb1 e3a7cc35
H	9603161f a30f9dbf 9f65ffbc f41fc7ef
H^*	a4c0d35c 95a63a80 5915367d cfe6b751
M_0	02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
M_1	313e82d8 5b8f3456 d4ac6dae c619c936 b4e253dd fd03da87 06633902 a0cd48d2 42339fe9 e87e570f 70b654ce 1e0da880 bc2198c6 9383a8b6 2b65f996 702af76f
M_0'	02dd31d1 c4eee6c5 069a3d69 5cf9af98 07b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a41f125 e8255108 9fc9cdf7 72bd1dd9 5b3c3780
M_1'	313e82d8 5b8f3456 d4ac6dae c619c936 34e253dd fd03da87 06633902 a0cd48d2 42339fe9 e87e570f 70b654ce 1e0d2880 bc2198c6 9383a8b6 ab65f996 702af76f
H	8d5e7019 61804e08 715d6b58 6324c015
H^*	79054025 255fb1a2 6e4bc422 aef54eb4

Všechny následující metody představují vylepšení a zefektivnění útoku dr. Wangové a kol. Za nejvíce inovativní lze považovat metotu českého matematika Vlastimila Klímy tunelování. Při popisu těchto metod nebudeme zbytečně zabíhat do detailů, zaměříme se především na přehledné a snadno pochopitelné vysvětlení jejich podstaty.

4.2.2. Další postačující podmínky

Tato část vychází z práce M. M. J. Stevense [7].

Jak již bylo uvedeno dříve, postačující podmínky dr. Wangové a kol. ve skutečnosti postačující nejsou. Při důkladném rozboru práce dr. Wangové, jež provedli autoři článku [15], lze nalézt další nutná omezení pro hodnoty funkce T_t. Tato omezení jsou podstatná pro správný přechod od hodnot δT_t k hodnotám δRL_t pomocí levé bitové rotace. Hledání kolizí může být daleko efektivnější při splnění těchto dalších podmínek.

Použitím dalších bitových podmínek pro hodnoty registrů Q_{t+1} a Q_t, jak jsou uvedeny v tabulce Tab. 4, lze dosáhnout splnění nutných omezení pro hodnoty funkce T_t. Následující vztah nám umožňuje řídit hodnotu konkrétních bitů funkce T_t:

$$Q_{t+1} - Q_t = RL_t = ROTL(T_t, R_t).$$

Při sestavování diferenčních cest tým dr. Wangové nepřihlížel ke splnění nutných omezení pro hodnoty funce T_t, a při bližším zkoumání také zjistíme, že tato omezení s velkou pravděpodobností splněna nejsou. Lze nalézt podmínky pro hodnoty registrů Q_{t+1} a Q_t, jež jsou postačující pro splnění nutných omezení pro T_t, $t = 0,...,19$. Pro vyšší hodnoty kroků $t = 20,...,63$ je výhodnější přímo ověřit platnost omezení pro T_t, místo používání podmínek pro Q_{t+1} a Q_t.

Všechna omezení pro hodnoty T_t pro oba bloky zprávy lze nalézt v práci [15] spolu s vysvětlením, proč jsou podstatné pro diferenční cestu. Uveďme zde pouze jednoduchý příklad získání dodatečných podmínek pro Q_{t+1} a Q_t. Všechny kroky detailně popsané lze nalézt ve Stevensově práci [7]. V tabulkách Tab. MD5-3 a Tab. MD5-4 jsou přehledně uvedeny všechny postačující podmínky, jak týmu dr. Wangové (černě), tak dodatečné (modře, podtržené).

Příklad: Nezbytné omezení: $\Delta T_4 = -2^{31}$.

Toto omezení je podstatné pro zaručení hodnoty $\delta RL_4 = -2^6$. *Neboť platí* $RL_4 = ROTL(T_4, R_4)$ *a* $R_4 = 7$ *(viz tabulka Tab. 3) víme, že* $\delta RL_4 = -2^{31+7} = -2^6$ *a bit 31 slova* T_4 *odpovídá bitu 6 slova* RL_4. *Tedy nutně potřebujeme, aby platilo* $RL_4[6] = T_4[31] = 1$. *Toto zajistíme přidáním podmínek* $Q_4[4] = Q_4[5] = 1$ *a* $Q_5[4] = 0$ *k původním podmínkám* $Q_4[6] = Q_5[6] = 0$ *a* $Q_5[5] = 1$. *Následující tabulkla zobrazuje bity 4, 5 a 6 pro slova* Q_5, Q_4 *a* RL_4.

$Q_5[6,5,4]$		010 …
$Q_4[6,5,4]$	–	011 …
$RL_4[6,5,4]$	=	11. …

Odvozování dalších podmínek pro Q_t

Odvozování podmínek pro Q_t, které zaručí požadované omezení pro T_t lze ve většině případů intuitivně odhadnout. Pro každý krok t se jednoduše podíváme na bit 31 a okolní bity a na hodnoty R_t a R_{t+1} pro registry Q_t a Q_{t+1} (viz tabulka Tab. 3 – rotační konstanty). Dobrým pomocníkem může být program, který na vstupu dostane podmínky pro hodnoty některých bitů registrů $Q_1,...,Q_{k+1}$ a určí pravděpodobnosti správné rotace (přechodu od δT_t k δRL_t) pro jednotlivé kroky $t = 1,...,k$. Toto je užitečné především pokud se rotace ovlivňují navzájem. Přesto proces hledání dalších podmínek pro Q_t nelze provádět zcela automaticky, neboť se zde vyskytují dva podstatné problémy:

- Podmínky zaručující správný přechod od δT_t k δRL_t mohou narušit správnou rotaci od δT_{t+1} k δRL_{t+1}.

- Pro splnění správné rotace pro nějakou hodnotu δT_t můžeme nalézt více stejně dobrých řešení (tj. nelze vybrat řešení jehož pravděpodobnost správné rotace je výrazně vyšší než u ostatních řešení) skládající se z mnoha podmínek.

4.2.3. Stanovení počátečních podmínek

Tato část vychází z práce M. M. J. Stevense [7].

Připomeňme označení mezihodnoty hashe H_k pro $k = 0,...,n$, kde n je počet bloků vstupní zprávy. Pak H_0 je rovno inicačnímu vektoru a H_n je rovno výsledné hashi po zpracování celé vstupní zprávy.

Jak již bylo řečeno dříve, popisovaný útok se nemusí omezovat pouze na zprávy délky dvou bloků (tj. 512 bitů), ale v obecné rovině lze útok použít na zprávy libovolné délky $M = M_0,...,M_k,M_{k+1},...,M_n$ se zaměřením na dva sousední bloky M_k a M_{k+1}. Na základě tohoto označení, budeme mezihodnotu hashe H_k (tj. hodnota před zpracováním bloku M_k zprávy M) nazývat počáteční hodnota útoku.

Je tedy zřejmé, že počáteční hodnota útoku nemusí být nutně rovna iniciačnímu vektoru funkce MD5, může to být hodnota spočítána při zpracování předchozích bloků zprávy. Pro efektivnější vedení útoky by měly být tyto počáteční podmínky voleny pečlivě.

Mezihodnota hashe $H_{k+1} = (A_{k+1}, B_{k+1}, C_{k+1}, D_{k+1})$ jako výsledek po zpracování bloku M_k představuje vstupní hodnotu registru pro zpracování bloku M_{k+1} zprávy M. Pro dodržení druhé diferenční cesty máme pak nutné podmínky: $C_{k+1}[25] = 1$ a $D_{k+1}[25] = 0$ (viz tabulka Tab. MD5-4 – v tomto případě B_{k+1} odpovídá Q_{-1} a D_{k+1} odpovídá Q_0). Hodnota H_{k+1} závisí na hodnotě $H_k = (A,B,C,D)$ (počáteční hodnota útoku) a také na hodnotách registrů $Q_{61},...,Q_{64}$ při zpracovávání bloku M_k zprávy M následujícím způsobem:

$$H_{k+1} = (A_{k+1}, B_{k+1}, C_{k+1}, D_{k+1}) = (A + Q_{61}, B + Q_{64}, C + Q_{63}, D + Q_{62}).$$

Dále máme postačující podmínky $Q_{62}[25] = 0$ a $Q_{63}[25] = 0$ (viz článek [15] – tyto podmínky jsou nutné, aby nedocházelo k propagaci bitového rozdílu $+2^{25}$ na ostatní bity).

Abychom zároveň splnili podmínky $C_{k+1}[25] = 1$ a $Q_{63}[25] = 0$ máme dvě možnosti:

- $C[25] = 1$ a zároveň v součtu $C + Q_{63}$ nebude žáden přenos od bitů 0,...,24 k bitu 25;
- $C[25] = 0$ a zároveň v součtu $C + Q_{63}$ bude přenos od bitů 0,...,24 k bitu 25.

Abychom zároveň splnili podmínky $D_{k+1}[25] = 0$ a $Q_{62}[25] = 0$ máme dvě možnosti:

- $D[25] = 0$ a zároveň v součtu $D + Q_{62}$ nebude žáden přenos od bitů 0,...,24 k bitu 25;
- $D[25] = 1$ a zároveň v součtu $D + Q_{62}$ bude přenos od bitů 0,...,24 k bitu 25.

Může nastat situace, že splnění těchto podmínek současně nebude možné. Pokud máme například $C[25,...,0] = 0$ nebo $D[25] = 1 \wedge D[24,...,0] = 0$, pak k potřebnému přenosu nemůže dojít. Ovšem to, že máme jinou počáteční hodnotu útoku, než jsme požadovali ještě neznamená, že útok nelze provést vůbec. Tato požadovaná počáteční hodnota

umožňuje postupovat pomocí nejpravděpodobnější diferenční cesty, nicméně existují ještě další diferenční cesty (méně pravděpodobné), které jsou rovněž platné. Pokud nelze dosáhnout této typické nejpravděpodobnější diferenční cesty, pak průměrná složitost útoku závisí na pravděpodobnosti jiné použitelné diferenční cesty.

4.2.4. Další diferenční cesty

Tato část byla převzata z práce M. Stevense a kol. [38].

Diferenční cesta pro výpočet MD5 hashe je podrobný popis šíření jednotlivých rozdílů přes všechny kroky $t = 0,...,63$ způsobených hodnotami rozdílů mezihodnot hashe a počátečními rozdíly vstupních bloků zprávy. Diferenční cesta zahrnuje popis následujících hodnot:

$$\delta F_t = f_t(Q'_t, Q'_{t-1}, Q'_{t-2}) - f_t(Q_t, Q_{t-1}, Q_{t-2});$$
$$\delta T_t = \delta F_t + \delta Q_{t-3} + \delta W_t;$$
$$\delta R_t = ROTL(T'_t, RC_t) - ROTL(T_t, RC_t);$$
$$\delta Q_{t+1} = \delta Q_t + \delta R_t.$$

Bitové podmínky

K popisu diferenčních cest používáme bitové podmínky na hodnoty registrů Q_t a Q'_t, kde jednotlivá bitová podmínka udává přímo či nepřímo hodnoty bitů $Q_t[i]$ a $Q'_t[i]$, $i = 0,...,31$. Tedy diferenční cesta je vyjádřena maticí bitových podmínek se 68 řádky (pro $t = -3,-2,...,64$) a 32 sloupci (pro $i = 0,...,31$). Přímá bitová podmínka na $Q_t[i]$ a $Q'_t[i]$ neovlivní jiné bity, zatímco nepřímá bitová podmínka může ovlivnit některý z bitů $Q_{t-2}[i]$, $Q_{t-1}[i]$, $Q_{t+1}[i]$ nebo $Q_{t+2}[i]$. Ze znalosti bitových podmínek na Q_t a Q'_t můžeme určit hodnoty δQ_t, δF_t, δT_t a δR_t (z výše uvedených vztahů).

Jednu bitovou podmínku na $Q_t[i]$ a $Q'_t[i]$ označíme $q_t[i]$, 32 bitových podmínek $(q_t[i])_{i=0}^{31}$ (jeden řádek matice) označíme q_t. Rozeznáváme dva typy bitových podmínek, diferenční bitové podmínky a bitové podmínky dle logické funkce (*boolean function*) f_t. Diferenční bitové podmínky jsou přímé a stanovují hodnoty $k_i = Q'_t[i] - Q_t[i]$, (viz tabulka Tab. 6) jež určují $\delta Q_t = \sum_{i=0}^{31} 2^i k_i$ (pomocí BSDR zápisu).

Tab. 6: Diferenční bitové podmínky

$q_t[i]$	Podmínka na $Q_t[i]$ a $Q'_t[i]$	k_i
·	$Q_t[i] = Q'_t[i]$	0
+	$Q_t[i] = 0, \quad Q'_t[i] = 1$	+1
-	$Q_t[i] = 1, \quad Q'_t[i] = 0$	-1

Bitové podmínky dle logické funkce (viz tabulka Tab. 7) se používají pro rozhodování případných nejednoznačností v

$$\Delta F_t[i] = f_t(Q'_t[i], Q'_{t-1}[i], Q'_{t-2}[i]) - f_t(Q_t[i], Q_{t-1}[i], Q_{t-2}[i]) \in \{-1,0,+1\}.$$

Případná nejednoznačnost je způsobena různými možnostmi hodnot $Q_j[i]$, $Q'_j[i]$ pro dané bitové podmínky. Například pro $t = 0$ a podmínky $(q_t[i], q_{t-1}[i], q_{t-2}[i]) = (\cdot, +, -)$ máme nejednoznačnosti pro podmínku $q_t[i] = \cdot$

$$\text{pokud } Q_t[i] = Q_t[i] = 0 \text{ pak } \Delta F_t[i] = f_t(0,1,0) - f_t(0,0,1) = -1,$$

$$\text{pokud } Q_t[i] = Q_t[i] = 1 \text{ pak } \Delta F_t[i] = f_t(1,1,0) - f_t(1,0,1) = +1.$$

Abychom vyřešili tuto nejednoznačnost, můžeme bitové podmínky ($\cdot$, + , -) nahradit bitovými podmínkami (0 , + , -) nebo (1 , + , -).

Tab. 7: Bitové podmínky dle logické funkce

$q_t[i]$	Podmínka na $Q_t[i]$ a $Q'_t[i]$	přímá/nepřímá	směr
0	$Q_t[i] = Q'_t[i] = 0$	přímá	
1	$Q_t[i] = Q'_t[i] = 1$	přímá	
^	$Q_t[i] = Q'_t[i] = Q_{t-1}[i]$	nepřímá	dozadu
v	$Q_t[i] = Q'_t[i] = Q_{t+1}[i]$	nepřímá	dopředu
!	$Q_t[i] = Q'_t[i] = \overline{Q_{t-1}[i]}$	nepřímá	dozadu
y	$Q_t[i] = Q'_t[i] = \overline{Q_{t+1}[i]}$	nepřímá	dopředu
m	$Q_t[i] = Q'_t[i] = Q_{t-2}[i]$	nepřímá	dozadu
w	$Q_t[i] = Q'_t[i] = Q_{t+2}[i]$	nepřímá	dopředu
#	$Q_t[i] = Q'_t[i] = \overline{Q_{t-2}[i]}$	nepřímá	dozadu
h	$Q_t[i] = Q'_t[i] = \overline{Q_{t+2}[i]}$	nepřímá	dopředu
?	$Q_t[i] = Q'_t[i] \wedge (Q_t[i] = 1 \vee Q_{t-2}[i] = 0)$	nepřímá	dozadu
q	$Q_t[i] = Q'_t[i] \wedge (Q_{t+2}[i] = 1 \vee Q_t[i] = 0)$	nepřímá	dopředu

Všechny bitové podmínky dle logické funkce zahrnují podmínky, kdy $Q_t[i] = Q'_t[i]$, a tedy neovlivňují hodnotu δQ_t. Navíc nepřímé bitové podmínky dle logické funkce nikdy neovlivní bit s podmínkou + nebo -, a tedy je můžeme nahradit některou z přímých bitových podmínek $\cdot$, 0 nebo 1. Také je poměrně snadné zaměnit všechny bitové podmínky ve směru dozadu za bitové podmínky ve směru dopředu (a naopak) v platné diferenční cestě.

Postup sestrojení diferenční cesty

Základní představa konstrukce diferenční cesty je sestrojit částečnou dolní diferenční cestu pro kroky $t = 0,1,...,11$ a částečnou horní diferenční cestu pro kroky $t = 16,17,...,63$. Tedy hodnoty Q_t použité v jednotlivých částečných diferenčních cestách se setkají, ale nepřekryjí. V kroku $t = 11$ se použijí hodnoty Q_8, Q_9, Q_{10}, Q_{11} a Q_{12}, v kroku $t = 16$ se použijí hodnoty Q_{13}, Q_{14}, Q_{15}, Q_{16} a Q_{17}. Pak se pokusíme spojit tyto dvě částečné diferenční cesty v jednu plnou diferenční cestu pomocí zbývajících čtyř kroků. Sestrojení částečné dolní diferenční cesty začneme s bitovými podmínkami q_{-3}, q_{-2}, q_{-1} a q_0 (tyto získáme jednoduše z mezihodnot hashí obou bloků) a pak je rozšíříme krok po kroku.

Podobně získáme částečnou horní diferenční cestu rozšířením podmínek q_{63}, q_{62} a q_{61} zpět ke kroku $t = 16$.

Při konstrukci diferenční cesty je nutné zafixovat rozdíly jednotlivých bloků vstupních zpráv $\delta m_0, ..., \delta m_{15}$. Předpokládejme, že máme částečnou diferenční cestu sestávající alespoň z bitových podmínek q_{t-1} a q_{t-2}, navíc jsou známy hodnoty δQ_t a δQ_{t-3}. Chceme rozšířit tuto částečnou diferenční cestu dopředu pomocí kroku t vedoucí k hodnotě δQ_{t+1}. Předpokládáme, že všechny nepřímé bitové podmínky mají směr dopředu a neovlivní bity v Q_t. Pokud známe navíc i hodnotu q_t, pak můžeme přeskočit část *Šíření přenosu*.

Šíření přenosu

Nejprve potřebujeme pro hodnotu δQ_t získat bitové podmínky q_t. Můžeme zvolit nějaký vhodný BSDR zápis slova δQ_t, ze kterého přímo získáme bitové podmínky q_t. Protože chceme sestrojit diferenční cestu s nejmenším možným počtem bitových podmínek, jako dobrá volba BSDR zápisu se jeví takový, který má nízkou váhu (nejlépe NAF zápis s nejnižší váhou).

Logická funkce

Pro libovolné $i \in \{0,...,31\}$ definujme $(a,b,c) = (q_t[i], q_{t-1}[i], q_{t-2}[i])$ trojici bitových podmnínek takových, že všechny nepřímé bitové podmínky ovlivňují pouze hodnoty $Q_t[i]$, $Q_{t-1}[i]$ nebo $Q_{t-2}[i]$. Trojice (a,b,c) je asociována s množinou U_{abc} všech hodnot $(x,x',y,y',z,z') = (Q_t[i], Q'_t[i], Q_{t-1}[i], Q'_{t-1}[i], Q_{t-2}[i], Q'_{t-2}[i])$, které splňují podmínky (a,b,c).

Pokud $U_{abc} = \emptyset$, pak bitové podmínky (a,b,c) si odporují a nemohou být součástí žádné platné diferenční cesty. Definujeme množinu $\mathcal{F}_t$ všech trojic (a,b,c) takových, že všechny nepřímé bitové podmínky ovlivňují pouze hodnoty $Q_t[i]$, $Q_{t-1}[i]$ nebo $Q_{t-2}[i]$ a zároveň platí $U_{abc} \neq \emptyset$.

Definujeme množinu V_{abc} všech možných logických funkcí $f_t(x',y',z') - f_t(x,y,z)$ pro dané bitové podmínky $(a,b,c) \in \mathcal{F}_t$:

$$V_{abc} = \{f_t(x',y',z') - f_t(x,y,z); (x,x',y,y',z,z') \in U_{abc}\} \subset \{-1,0,+1\}.$$

Pokud $|V_{abc}| = 1$, pak pro bitové podmínky (a,b,c) nevzniká žádná nejednoznačnost a trojici (a,b,c) nazveme *řešením*, množinu všech řešení označíme S_t. Pokud je $|V_{abc}| > 1$, tj. máme více logických funkcí pro bitové podmínky $(a,b,c) \in \mathcal{F}_t$, pak pro každou logickou funkci $g \in V_{abc}$ definujeme množinu $W_{abc,g}$ všech řešení $(d,e,f) \in S_t$, které jsou kompatibilní s (a,b,c) a rozdíl logické funkce pro tato řešení je $g \in V_{abc}$:

$$W_{abc,g} = \{(d,e,f) \in S_t; U_{def} \subset U_{abc} \wedge V_{def} = \{g\}\}.$$

Poznamenejme, že pro každé $g \in V_{abc}$ vždy existuje takové řešení $(d,e,f) \in W_{abc,g}$, že obsahuje pouze přímé bitové podmínky `01+-`, tedy $W_{abc,g} \neq \emptyset$. Můžeme volit takové

přímé a dopředné bitové podmínky dle logické funkce, že pro všechna t, i a $(a,b,c) \in \mathcal{F}_t$ a pro všechna $g \in V_{abc}$ existuje trojice $(d,e,f) \in W_{abc,g}$ obsahující přímé a dopředné bitové podmínky takové, že

$$U_{def} \text{ je rovno } \{(x,x',y,y',z,z') \in U_{abc};\ f_t(x',y',z') - f_t(x,y,z) = g\}.$$

Jinými slovy, vždy můžeme zvolit takové bitové podmínky (d,e,f) (přímé a dopředné), které řeší případné nejednoznačnosti vzniklé podmínkami (a,b,c) optimálním způsobem. Pokud trojice (d,e,f) není jednoznačná, tj. můžeme najít více různých trojic (d,e,f), které splňují výše uvedenou podmínku, pak kvůli snadnějším výpočtům upřednostňujeme přímé nad nepřímými podmínkami a krátké nepřímé podmínky (`vy^!`) nad dlouhými nepřímými podmínkami (`whqm#?`).

Pro všechna $i = 0,1,...,31$ máme předpoklad platných bitových podmínek $(a,b,c) = (q_t[i], q_{t-1}[i], q_{t-2}[i])$, kde pouze podmínka c je nepřímá a ovlivňuje tedy pouze hodnotu $Q_{t-1}[i]$, a tedy $(a,b,c) \in \mathcal{F}_t$.

Pokud $|V_{abc}| = 1$, pak nevznikají žádné nejednoznačnosti a obdržíme přímo $\{g_i\} = V_{abc}$. V opačném případě, pokud $|V_{abc}| > 1$ pak zvolíme libovolné $g_i = V_{abc}$ a vyřešíme nejednoznačnost nahrazením podmínek (a,b,c) podmínkami (d,e,f) (přímé a dopředné), pro které je rozdíl logické funkce právě $g_i = V_{abc}$. Takto získáme všechna g_i, pro $i = 0,...,31$ a můžeme dopočítat hodnotu $\delta F_t = \sum_{i=0}^{31} 2^i g_i$ a také $\delta T_t = \delta F_t + \delta Q_{t-3} + \delta W_t$.

Bitová rotace

Z hodnoty δT_t nedokážeme jednoznačně určit hodnotu $\delta R_t = ROTL(T_t', n) - ROTL(T_t, n)$, kde $n = RC_t$. Přesto můžeme jednoduše spočítat $\delta R_t = ROTL(\delta T_t, n)$, kde použijeme NAF zápis pro výraz δT_t (NAF zápis hodnoty δT_t s velkou pravděpodobností vede ke správným výsledkům v takto jednoduché aproximaci – podrobnosti viz [38]). Pak můžeme již jednoduše zjistit hodnotu $\delta Q_{t+1} = \delta Q_t + \delta R_t$ a rozšířit naši částečnou diferenční cestu pro krok t.

Rozšiřování dolní diferenční cesty

Rozšiřování dolní částečné diferenční cesty probíhá obdobným způsobem jako rozšiřování horní částečné diferenciální cesty. Předpokládejme, že máme částečnou diferenční cestu obsahující alespoň bitové podmínky q_t a q_{t-1} a jsou známy hodnoty δQ_{t+1} a δQ_{t-2}. Chceme rozšířit tuto částečnou diferenční cestu pozpátku pomocí kroku t, jež vede k hodnotě δQ_{t-3} a bitovým podmínkám q_t, q_{t-1} a q_{t-2}. Předpokládáme, že všechny nepřímé podmínky mají směr dozadu (viz tabulka Tab. 7) a neovlivňují bity slova Q_{t-2}.

Zvolíme BSDR zápis slova δQ_{t-2} s váhou nejvýše o 1 nebo 2 větší než je minimální váha tohoto slova (můžeme zvolit NAF zápis). Z tohoto zápisu slova δQ_{t-2} přímo získáme hodnoty bitových podmínek q_{t-2}.

Pro všechna $i = 0,...,31$ máme dle předpokladu platné bitové podmínky $(a,b,c) = (q_t[i], q_{t-1}[i], q_{t-2}[i])$, kde pouze podmínka b je nepřímá a ovlivňuje tedy pouze hodnotu $Q_{t-2}[i]$, a tedy $(a,b,c) \in \mathcal{F}_t$. Pokud $|V_{abc}| = 1$, pak nevznikají žádné nejednoznačnosti a obdržíme přímo $\{g_i\} = V_{abc}$. V opačném případě, pokud $|V_{abc}| > 1$ pak zvolíme libovolné $g_i = V_{abc}$ a vyřešíme nejednoznačnost nahrazením podmínek (a,b,c) podmínkami (d,e,f) (přímé a se směrem dozadu), pro které je rozdíl logické funkce právě $g_i = V_{abc}$. Takto získáme všechna g_i, pro $i = 0,...,31$ a můžeme dopočítat hodnotu $\delta F_t = \sum_{i=0}^{31} 2^i g_i$.

Použitím rotace $\delta R_t = \delta Q_{t+1} - \delta Q_t$ pro $n = 32 - RC_t$ jednoduše určíme hodnotu $\delta T_t = ROTL(\delta R_t, n)$, kde použijeme NAF zápis pro výraz δR_t (NAF zápis hodnoty δR_t s velkou pravděpodobností vede ke správným výsledkům v takto jednoduché aproximaci). Nakonec určíme hodnotu $\delta Q_{t-3} = \delta T_t - \delta F_t - \delta W_t$ a můžeme rozšířit naši částečnou diferenční cestu zpět pro krok t.

Sestrojení plné diferenční cesty

Popišme nyní postup pro sestrojení plné diferenční cesty spojením obou částečných diferenčních cest. Zvolíme δQ_{-3} a bitové podmínky q_{-2}, q_{-1} a q_0 a rozšíříme tuto částečnou diferenční cestu až po krok $t = 11$. Dále zvolíme δQ_{64} a bitové podmínky q_{63}, q_{62} a q_{61} a rozšíříme tuto diferenční cestu zpětně až po krok $t = 16$. Toto vede k získání bitových podmínek $q_{-2}, q_{-1}, ..., q_{11}, q_{14}, q_{15}, ..., q_{63}$ a hodnot rozdílů $\delta Q_{-3}, ..., \delta Q_{64}$. Zbývá dokončit kroky $t = 12,13,14,15$. Pro tyto kroky jsme schopni spočítat δR_t, δT_t a $\delta F_t = \delta T_t - \delta W_t - \delta Q_{t-3}$ (stejně jako při zpětném rozšiřování horní částečné diferenční cesty).

Cílem je najít nové bitové podmínky $q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}$, které budou kompatibilní s původními a povedou na požadované hodnoty δQ_{12}, δQ_{13}, δF_{12}, δF_{13}, δF_{14}, δF_{15}, čímž se dokončí plná diferenční cesta. Nejprve musíme zjistit, zda je vůbec možné nalézt tyto nové bitové podmínky.

Pro $i = 0,1,...,32$ definujeme množinu $\mathcal{U}_i$ všech n-tic $(g_1, g_2, f_1, f_2, f_3, f_4)$ složených z 32-bitových slov a navíc platí $g_j \equiv f_k \equiv 0 \bmod 2^i$ pro $j = 1,2$ a pro $k = 1,2,3,4$. Chceme sestrojit všechny množiny $\mathcal{U}_i$ tak, že pro každou n-tici $(g_1, g_2, f_1, f_2, f_3, f_4) \in \mathcal{U}_i$ existují bitové podmínky $q_{10}[l], q_{11}[l], q_{12}[l], q_{13}[l], q_{14}[l], q_{15}[l]$ určující $\Delta Q_{11+j}[l]$ a $\Delta F_{11+k}[l]$ přes všechny bity $l = 0, ..., i-1$ tak, že

$$\delta Q_{11+j} = g_j + \sum_{l=0}^{i-1} 2^l \Delta Q_{11+j}[l], \qquad j = 1,2,$$

$$\delta F_{11+k} = f_k + \sum_{l=0}^{i-1} 2^l \Delta F_{11+k}[l], \qquad k = 1,2,3,4.$$

Z toho vyplývá pro $i = 0$ platnost $\mathcal{U}_0 = \{(\delta Q_{12}, \delta Q_{13}, \delta F_{12}, \delta F_{13}, \delta F_{14}, \delta F_{15})\}$. Další množiny $\mathcal{U}_i$ jsou sestrojeny induktivně pomocí algoritmu uvedeného níže. Velikost každé

množiny $\mathcal{U}_i$ je nejvýše 2^6, neboť pro každé g_j, f_k existují nejvýše 2 možné hodnoty, které splňují výše uvedené vztahy.

Pokud nalezneme $\mathcal{U}_{32} \neq \emptyset$, pak existuje cesta $u_0, u_1, ..., u_{32}$, kde $u_i \in \mathcal{U}_i$ a každé u_{i+1} se získalo z hodnoty u_i použitím algoritmu uvedeného níže. Tedy požadované nové bitové podmínky $(q_{15}[i], q_{14}[i], ..., q_{10}[i])$ jsou uloženy v proměnných $(a', b'', c''', d''', e'', f')$ po proběhnutí 13. kroku algoritmu uvedeného níže, kde iterace začíná hodnotou u_i a končí hodnotou u_{i+1}.

Nechť $\mathcal{U}_{i+1} = \emptyset$ a $(a, b, e, f) = (q_{15}[i], q_{14}[i], q_{11}[i], q_{10}[i])$. Pro každou n-tici hodnot $(g_1, g_2, f_1, f_2, f_3, f_4) \in \mathcal{U}_i$ postupujeme následovně:

1. Pro každou podmínku $d = q_{12}[i] \in \begin{cases} \{\cdot\} & \text{pokud } q_1[i] = 0 \\ \{+,-\} & \text{pokud } q_1[i] = 1 \end{cases}$ proveď
2. Nechť $g_1' = 0, -1, +1$ pro $d = \cdot, -, +$
3. Pro každé různé $f_1' \in \{-f_1[i], +f_1[i]\} \cap V_{def}$ proveď
4. Nechť $(d', e', f') = FC(12, def, f_1')$
5. Pro každou podmínku $c = q_{13}[i] \in \begin{cases} \{\cdot\} & \text{pokud } q_2[i] = 0 \\ \{+,-\} & \text{pokud } q_2[i] = 1 \end{cases}$ proveď
6. Nechť $g_2' = 0, -1, +1$ pro $c = \cdot, -, +$
7. Pro každé různé $f_2' \in \{-f_2[i], +f_2[i]\} \cap V_{cd'e'}$ proveď
8. Nechť $(c', d'', e'') = FC(13, cd'e', f_2')$
9. Pro každé různé $f_3' \in \{-f_3[i], +f_3[i]\} \cap V_{bc'd''}$ proveď
10. Nechť $(b', c'', d''') = FC(14, bc'e', f_3')$
11. Pro každé různé $f_4' \in \{-f_4[i], +f_4[i]\} \cap V_{ab'c''}$ proveď
12. Nechť $(a', b'', c''') = FC(15, ab'c'', f_4')$
13. Přidej $(g_1 - 2^i g_1', g_2 - 2^i g_2', f_1 - 2^i f_1', f_2 - 2^i f_2', f_3 - 2^i f_3', f_4 - 2^i f_4')$ do množiny $\mathcal{U}_{i+1}$.

Kde $FC(t, abc, f)$ znamená vyjádření podmínek (a, b, c) v kroku t jako přímé a dopředné, pro něž je f rozdíl logické funce. Do každé množiny $\mathcal{U}_{i+1}$ přidáváme každou nalezenou n-tici právě jednou.

Jedna iterace tohoto algoritmu sestrojí množinu $\mathcal{U}_{i+1}$ s požadovanými vlastnostmi. Můžeme si všimnout, že v posledním 13. kroku se právě nulují bity od $(i-1)$ po nultý v každém slově právě přidávané n-tice. Po proběhnutí celé iterace algoritmu získáme hodnoty proměnných $(a', b'', c''', d''', e'', f')$ jež jsou hledané nové podmínky $(q_{15}[i], q_{14}[i], ..., q_{10}[i])$. Všechny podmínky jsou přímé a dopředné neboť v průběhu algoritmu se postupně všechny původní podmínky převádějí na přímé a dopředné (podobně

jako v postupu rozšiřování dolní částečné diferenční cesty). Pokud se podíváme podrobněji na hodnoty proměnných $(a', b'', c''', d''', e'', f')$ zjistíme, že a' odpovídá naposled měněné hodnotě $q_{15}[i]$ (v každé iteraci se mění jen jednou), b'' odpovídá naposled měněné hodnotě $q_{14}[i]$ (v každé iteraci se mění dvakrát), atd. Tedy pokud je možné nalézt nové bitové podmínky pro $q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}$ (tj. pokud algoritmus dospěje až k hodnotě $\mathcal{U}_{32} \neq \emptyset$), pak tyto nalezené nové podmínky mají všechny předpokládané vlastnosti.

Poznamenejme nakonec, že konstrukce dalších diferenčních cest je založena na stejném rozdílu vstupních zpráv a mezihodnot hashí jako původní první a druhá diferenční cesta týmu dr. Wangové. Také počínaje krokem $t = 26$ jsou hodnoty diferenčních cest a postačujících podmínek shodné s údajemi v práci dr. Wangové. Proto i zde uvádíme hodnoty dalších diferenčních cest včetně příslušných postačujících podmínek pouze do kroku $t = 25$. Nová diferenční cesta pro první blok zprávy je uvedena v tabulce Tab. MD5-5 a příslušné postačující podmínky v tabulce Tab. MD5-6. Další diferenční cesty pro druhý blok zprávy jsou uvedeny v tabulkách Tab. MD5-7, Tab. MD5-9, Tab. MD5-11 a Tab. MD5-13, příslušné postačující podmínky v tabulkách Tab. MD5-8, Tab. MD5-10, Tab. MD5-12 a Tab. MD5-14.

4.2.5. Tunelování

Tato část vychází z práce V. Klímy [17].

Autorem metody tunelování je český kryptoanalytik Vlastimil Klíma. Tato metoda vychází z metody mnohonásobné modifikace zpráv, kterou rozšiřuje a dalo by se říct nahrazuje. Použitím metody tunelování se exponenciálně snižuje rychlost hledání kolizí, jak uvádí autor až pětsetkrát.

Metoda mnohonásobné modifikace zpráv spočítvá v tom, že se zvolí náhodně zpráva M a její modifikací se postupně splňují postačující podmínky $Q_1, Q_2, ..., Q_{64}$. Takto je možné postačující podmínky splnit maximálně pro Q_{24} (podrobný popis, jak toho dosahnout lze nalézt v práci [18]). Pro zbývající postačující podmínky již nelze zprávy jednoduše měnit (bez vlivu na již splněné postačující podmínky), pouze se musí ověřit, zda tyto podmínky platí. Pokud neplatí, zvolí se jiná vstupní zpráva a začne se od začátku.

Bod Q_{24} nazýváme *bodem verifikace* POV (*point of verification*), neboť následující podmínky již jen ověřujeme zda platí. Pro funkci MD5 je těchto zbývajících postačujících podmínek 29. Složitost metody mnohonásobné modifikace zpráv závisí na nalezení 2^{29} bodů POV, jeden z nich pak bude náhodně splňovat zbývající postačující podmínky a ten pak bude zvolen pro realizaci kolize.

Metoda tunelování začíná v bodě POV, několika tunely se z něj geometrickou řadou postupně vytvoří dostatečné množství dalších POV tak, aby nedošlo k narušení již splněných postačujících podmínek před body POV. V ideálním případě je potřeba pouze jeden bod POV. Tunelem o síle n z něj vytvoříme 2^n bodů POV. Tunely lze různě kombinovat, takže z jednoho bodu POV pomocí tunelu o síle r sestrojíme 2^r bodů POV a z každého z nich můžeme pomocí tunelu o síle s sestrojit 2^s bodů POV. Celkem tedy použitím obou tunelů získáme 2^{r+s} bodů POV.

Sestrojení a fungování tunelu si ukážeme na konkrétním případě tunelu Q_9. Popíšeme zde pouze tento jeden tunel názorně, ostatní jsou svou konstrukcí podobné a lze je podrobně popsané nalézt v [14], resp. [17].

Poznámka: Tunely se označují Q_m podle registru, jehož počáteční podmínky tunel mění.

Tunel Q_9

Pokud vyjdeme ze vzorců popisujcích hodnoty registru v jednotlivých krocích (viz popis v kapitole 4.1) dostaneme kompletní rovnici:

$$Q_{t+1} = Q_t + ROTL(f_t(Q_t, Q_{t-1}, Q_{t-2}) + Q_{t-3} + W_t + K_t, R_t), \quad t = 0,...,63.$$

Podívejme se speciálně na rovnice pro konkrétní hodnoty Q_{11} a Q_{12}:

$$Q_{11} = Q_{10} + ROTL(f_{10}(Q_{10}, Q_9, Q_8) + Q_7 + W_{10} + K_{10}, R_{10}),$$

$$Q_{12} = Q_{11} + ROTL(f_{11}(Q_{11}, Q_{10}, Q_9) + Q_8 + W_{11} + K_{11}, R_{11}).$$

Můžeme si povšimnout, že pokud i-tý bit Q_{10} bude 0 a i-tý bit Q_{11} bude 1, pak rovnice pro Q_{11} a Q_{12} nebudou záležet na hodnotě registru Q_9. Zkusme si tedy ověřit, jaký výsledek dají podmínky $Q_{10}[i] = 0$ a $Q_{11}[i] = 1$. Pro hodnoty nelineární funkce f_k pro $k = 10,11$ platí (viz popis funkce MD5 v kapitole 4.1):

$$f_k(X, Y, Z) = F(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z).$$

Po dosazení:

$$f_{10}(Q_{10}, Q_9, Q_8) = (Q_{10} \wedge Q_9) \oplus (\overline{Q_{10}} \wedge Q_8),$$

$$f_{10}(0, Q_9, Q_8) = (0 \wedge Q_9) \oplus (1 \wedge Q_8), \text{ pro } Q_{10}[i] = 0, \ \forall i,$$

$$f_{10}(0, Q_9, Q_8) = (0) \oplus (Q_8) = Q_8, \text{ pro } Q_{10}[i] = 0, \ \forall i,$$

$$f_{11}(Q_{11}, Q_{10}, Q_9) = (Q_{11} \wedge Q_{10}) \oplus (\overline{Q_{11}} \wedge Q_9),$$

$$f_{11}(1, Q_{10}, Q_9) = (1 \wedge Q_{10}) \oplus (0 \wedge Q_9), \text{ pro } Q_{11}[i] = 1, \ \forall i,$$

$$f_{11}(1, Q_{10}, Q_9) = (Q_{10}) \oplus (0) = Q_{10}, \text{ pro } Q_{11}[i] = 1, \ \forall i.$$

Výsledek dosadíme do rovnic pro Q_{11} a Q_{12}:

$$Q_{11} = Q_{10} + ROTL(Q_8 + Q_7 + W_{10} + K_{10}, R_{10}),$$

$$Q_{12} = Q_{11} + ROTL(Q_{10} + Q_8 + W_{11} + K_{11}, R_{11}).$$

Vidíme, že skutečně tyto rovnice nezávisí na hodnotě registru Q_9 a tedy i případná změna tohoto registu je nijak neovlivní. Změna v hodnotách registru Q_9 však zřejmě mají vliv na hodnoty registrů Q_9, Q_{10} a Q_{13} jak plyne z následujících rovnic:

$$\boldsymbol{Q_9} = Q_8 + ROTL(f_8(Q_8, Q_7, Q_6) + Q_5 + W_8 + K_8, R_8),$$

$$Q_{10} = \boldsymbol{Q_9} + ROTL(f_9(\boldsymbol{Q_9}, Q_8, Q_7) + Q_6 + W_9 + K_9, R_9),$$

$$Q_{13} = Q_{12} + ROTL(f_{12}(Q_{12}, Q_{11}, Q_{10}) + \boldsymbol{Q_9} + W_{12} + K_{12}, R_{12}).$$

Hodnoty těchto registrů (Q_9, Q_{10} a Q_{13}) však nemůžeme změnit, neboť jsou zatíženy mnoha postačujícími podmínkami, které již máme splněny (splněny jsou všechny postačující podmínky až do bodu POV, což představuje hodnota Q_{24}). Místo toho tyto změny kompenzujeme změnami příslušných slov vstupní zprávy, podle následujících pravidel. $ROTR(X,Y)$ značí pravou rotaci slova X o Y pozic.

$$m_8 = W_8 = ROTR(Q_9 - Q_8, R_8) - f_8(Q_8, Q_7, Q_6) - Q_5 - K_8,$$

$$m_9 = W_9 = ROTR(Q_{10} - Q_9, R_9) - f_9(Q_9, Q_8, Q_7) - Q_6 - K_9,$$

$$m_{12} = W_{12} = ROTR(Q_{13} - Q_{12}, R_{12}) - f_{12}(Q_{12}, Q_{11}, Q_{10}) - Q_9 - K_{12}.$$

Podívejme se nyní, jak se projeví změny ve vstupních slovech m_8, m_9 a m_{12}. Jednoduše můžeme ověřit (např. z tabulky Tab. 3), že změny v těchto slovech poprvé ovlivní hodnotu registru Q_{25} a následně každého poměrně složitým způsobem. Tedy tyto změny neovlivní hodnoty před bodem POV (včetně) a tedy také neovlivní příslušné postačující podmínky. Takže z jednoho bodu POV jsme získali sedm dalších POV. Našli jsme tedy tunel Q_9.

Ještě se podívejme které bity registru Q_9 můžeme měnit. Měnit můžeme pouze ty bity Q_9 na které nejsou kladeny žádné postačující podmínky. Zároveň musí být možné dosáhnout podmínek $Q_{10}[i] = 0$ a $Q_{11}[i] = 1$. Podíváme-li se na příslušné postačující podmínky (viz tabulka Tab. 8) zjistíme, že toto je možné na místech $Q_9[21]$, $Q_9[22]$ a $Q_9[23]$. Získali jsme tedy tunel o síle 3.

Tab. 8: Postačující podmínky – extra podmínky jsou vyznačeny žlutě

t	Podmínky pro $Q_t = b_{31}...b_0$				
9	11111011	...	10000	0.1^1111	00111101
10	0111....	000	11111	1v01...0	01....00
11	0010.0v0	111	.0001	1^00.0.0	11....10

Poznámka: Autor metody tunelování vychází z postačujících podmínek sestavených Liang a Lai [12]. Pro úplnost zde uvádíme tyto postačující podmínky včetně extra podmínek pro tunel Q_9 a dalších tunelů, jejichž podrobný popis lze najít v práci V. Klímy [14], resp. [17].

Poznámka: Postačující podmínky podle Liang a Lai Tab. 9 jsou uvedeny jen do bodu POV Q_{24}, dále se shodují s postačujícími podmínkami podle Wangové a kol.

Poznámka: Vysvětlivky k postačujícím podmínkám:
^ *znamená bit stejný jako bit nad ním,*
A *znamená bit rovný negaci bitu nad ním,*
v *znamená bit stejný jako bit pod ním,*
V *znamená bit rovný negaci bitu pod ním.*

Poznámka: V tabulce Tab. 9 jsou také zobrazeny podmínky pro tunel Q_9 a další tunely. Tunel Q_4 zobrazen zeleně, tunel Q_9 zobrazen žlutě, tunel Q_{10} zobrazen modře a tunel Q_{14} zobrazen červeně.

Tab. 9: Postačující podmínky podle Liang a Lai a zobrazení tunelů.

t	Podmínky pro $Q_t = b_{31}...b_0$			
1	....	....	....	....
2	....	....	....	....
3	....	.vvv0vvv	vvvv0vvv	v0......
4	1.....1.	0^^^1^^^	^^^^1^^^	^011....
5	1000100v	01000000	00000000	0010v1v1
6	0000001^	01111111	10111100	0100^0^1
7	00000011	11111110	11111000	00100000
8	00000001	1..10001	0.0v0101	01000000
9	11111011	...10000	0.1^1111	00111101
10	0111....	00011111	1v01...0	01....00
11	0010.0v0	111.0001	1^00.0.0	11....10
12	000...^^	1000	0001...1	0.......
13	01....01	1111	111....0	0...1...
14	000...00	1011	111....1	1...1...
15	v1100001	..V.....	10......	.0000000
16	^01000..	..A.....	v.......	.000v000
17	^1v.....	......0.	^.......	....^...
18	^.^.....	......1.	....	....
19	^.......	......0.	....	....
20	^.......	......v.	....	....
21	^.......	......^.	....	....
22	^.......	....	....	....
23	0.......	....	....	....
24	1.......	....	....	....

4.3. Praktické použití útoků na funkci MD5

Představili jsme si útoky na funkci MD5 a postup jak nalézt její kolize. Zajímavější však bude ukázat použití těchto teoretických útoků v praxi. Praktická ukázka kolidujících certifikátů veřejného klíče jistě přesvědčí, že podané teoretické výsledky nelze brát na lehkou váhu. Uvedeme zde pouze stručný popis sestrojení dvou kolidujících certifikátů, podrobnější informace může čtenář nalézt v práci [20] resp. [21].

4.3.1. Kolize certifikátů X.509

Nejprve si stručně shrneme, co to certifikát je a k čemu slouží. Certifikát je datová struktura (řetězec bitů), která umožňuje ověření příslušnosti veřejného klíče danému subjektu. Certifikát je podepsán (elektronicky ověřen) certifikační autoritou (dále CA). Z certifikátu lze získat veřejný klíč uživatele, který se používá k šifrování zpráv, či ověření totožnosti uživatele. Certifikát je struktura popisovaná v jazyce ASN.1 a přenáší se v DER (Distinguished Encoding Rules - rozlišitelná kódovací pravidla) kódování.

Poznámka: Pro komunikaci mezi počítači je běžnější kódování Base64. Základy struktury jazyka ASN.1 a kódování BER (varianta kódování DER) lze nalézt např. v [19].

Popis standardu X.509

První standard formátu certifikátu byl definován v roce 1988, standard X.509 verze 1. V roce 1993 vznikla verze 2 a od roku 1996 se již běžně používá standard X.509 verze 3. Všechny verze jsou do značné míry podobné, liší se pouze přidáním několika rozšíření. Kompletní popis standardu X.509 verze 3 lze nalézt v RFC 3280 [22], zde si popíšeme jen ty části, které nás budou zajímat.

Samotný certifikát sestává ze tří částí:

- data, která mají být podepsána (tbsCertificate),
- specifikace algoritmu, kterým jsou tato data podepsána (signatureAlgorithm),
- hodnota vlastního elektronického podpisu (signatureValue).

Zápis v jazyce ASN.1:

```
Certificate  ::=  SEQUENCE  {
      tbsCertificate       TBSCertificate,
      signatureAlgorithm   AlgorithmIdentifier,
      signatureValue       BIT STRING}
```

Proberme si podrobněji jednotlivé části certifikátu.

Data, která mají být podepsána obsahují informace o subjektu certifikátu (komu je certifikát vydán) a také o certifikační autoritě, která certifikát vydala. Zápis části TBSCertificate v jazyce ASN.1:

```
TBSCertificate  ::=  SEQUENCE  {
      Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  },
      SerialNumber  ::=  INTEGER,
      Signature          AlgorithmIdentifier,
      Issuer             Name,
      Validity  ::=  SEQUENCE  {
            notBefore       UTCTime,
            notAfter        UTCTime},
```

```
Subject              Name,
SubjectPublicKeyInfo  ::=  SEQUENCE  {
      algorithm          AlgorithmIdentifier,
      subjectPublicKey   BIT STRING},
IssuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                     (rozšíření pouze pro verze 2 a 3)
SubjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                     (rozšíření pouze pro verze 2 a 3)
Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension
                     (rozšíření pouze pro verze 2 a 3)
}
```

Popišme si stručně význam jednotlivých položek části TBSCertificate.

- verze (Version) – celé číslo určující verzi certifikátu (0 = veze 1, 1 = verze 2, 2 = verze 3),
- sériové číslo (SerialNumber) – jedinečné sériové číslo v rámci dané certifikační autority,
- podpis (Signature) – obsahuje identifikátor algoritmu, kterým CA podepsala certifikát (musí být stejný jako v druhé části – specifikace algoritmu),
- vydavatel (Issuer) – obsahuje informace o vydavateli certifikátu (především jméno, tzv. distinguish name DN),
  příklad:
  `Issuer: C=CZ, O=Česká pošta, s. p., CN=PostSignum Qualified CA`
- platnost (Validity) – interval platnosti certifikátu (od-do = notBefore-notAfter),
- subjekt (Subject) – identifikace uživatele, pro něhož je certifikát vydán, tedy držitele soukromého a veřejného klíče,
  příklad:
  `Subject: C=CZ, L=Třinec Dukelská 997, OU=0400148099, CN=Ilona Galaczová`
- informace o veřejném klíči (SubjectPublicKeyInfo) – popis použitého algoritmu (identifikátor algoritmu včetně parametrů) a samotný veřejný klíč (modulus a exponent),
- rozšíření certifikátu (Extensions) – jsou definována pouze pro certifikáty X.509 verze 2 nebo 3.

Zbývající dvě části certifikátu představují podpis certifikační autority. Část algoritmus podpisu (signatureAlgorithm) obsahuje identifikátor algoritmu použitého CA k podpisu certifikátu včetně parametrů algoritmu. Poslední část tvoří vlastní hodnota podpisu CA.

Při vytváření podpisu certifikátu certifikační autoritou se nejprve spočte MD5 hash (může být použit i jiný hashovací algoritmus, např. SHA1 nebo SHA2) podpisované části a tu pak CA podepíše svým soukromým klíčem a připojí na konec certifikátu.

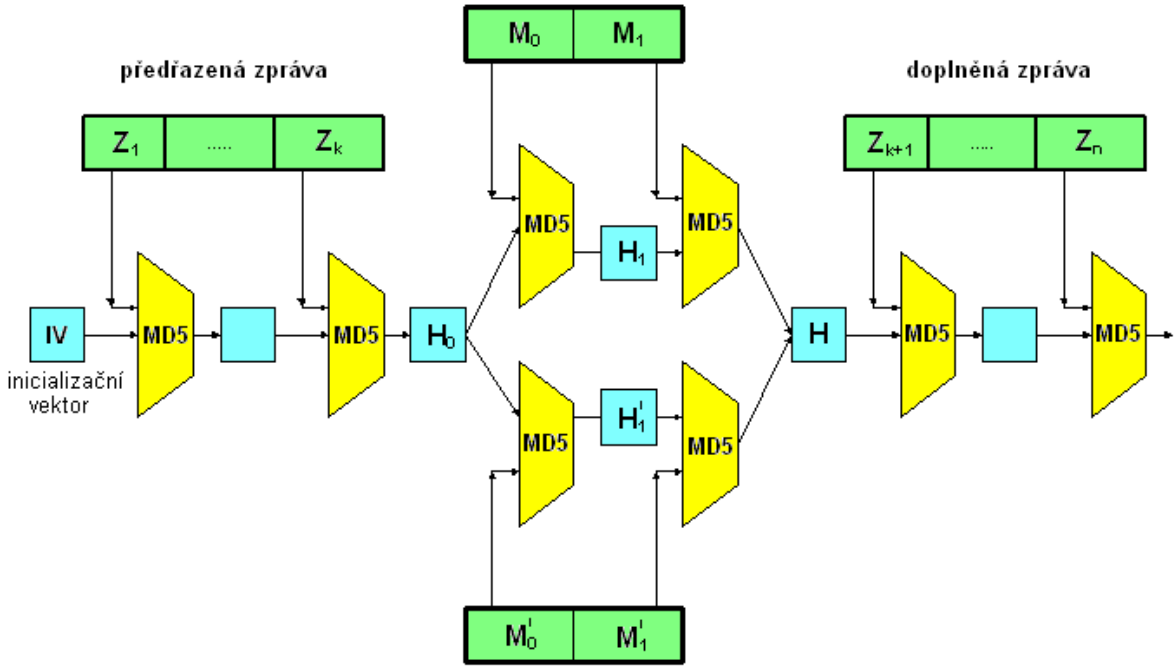
Nyní již máme přehled, jak takový certifikát veřejného klíče vypadá a můžeme se pustit do sestrojení dvou platných kolikujících X.509 certifikátů. Konkrétně se certifikáty budou lišit pouze v modulu veřejného klíče, ostatní položky budou totožné.

Konstrukce kolidujících X.509 certifikátů

Tato část je převzata z práce A. Lenstry a B. de Wegera [20].

Tým dr. Wangové ukázal postup, kterým lze nalézt dvě různé zprávy (délky dvou 512-bitových bloků) se stejnou MD5 hashí. Dokázal také, že při hledání kolizí nezáleží na počáteční hodnotě inicializačního vektoru. Díky tomu můžeme kolidující bloky vložit doprostřed libovolné zprávy s tím, že předřazená část zprávy musí mít délku násobku 512 bitů. Obrázek Obr. 5 názorně vystihuje princip využití kolizí MD5.

Obr. 5: Princip využití kolizí funkce MD5.



Popišme si detailně postup, kterým lze nalézt dva platné kolidující certifikáty.

1. Nejprve si připravíme šablonu pro certifikát, kde vyplníme všechny položky kromě RSA modulu veřejného klíče. V tomto kroku musíme splnit tři jednoduché požadavky:
   - struktura dat certifikátu musí vyhovovat X.509 standardu (viz. RFC 3280 [22]) a také kódování DER jazyka ASN.1 (viz. např. [19] a také RFC 3280);
   - délka RSA modulu a veřejného exponentu musí být pevná, délka RSA modulu bude 2048 bitů (k hledané kolizi dojde v prvních dvou 512-bitových blocích), jako veřejný exponent zvolíme běžně používanou hodnotu $e = 65537$ (tento musí být stejný pro oba certifikáty);
   - pozice, kde začíná RSA modul musí být násobek 512-ti bitů od začátku části, jež bude podepsána certifikační autoritou (což je právě předřazená zpráva – viz obrázek Obr. 5). Toho můžeme jednoduše dosáhnout přidáním „výplně“ do dat certifikátu, nejlépe do části pro popis subjektu certifikátu.
2. Pustíme MD5 algoritmus na část podpisovaných dat po bit, kterým začíná RSA modul. Tento vstupní řetězec dat bude mí délku násobku 512-bitů (dle třetího požadavku předcházejícího kroku). Výstup algoritmu MD5 v tomto kroku použijeme jako inicializační vektor pro další krok.

3. V tomto kroku využijeme útoku týmu dr. Wangové (popřípadě kteréhokoliv vylepšení algoritmu). Sestrojíme dva rozdílné bitové řetězce b_1 a b_2 každý délky 1024 bitů, jejichž výsledné MD5 hashe jsou stejné.
4. Nyní potřebujeme ze dvou bitových řetězců získat dva RSA moduly. Toho dosáhneme tím, že ke každému z řetězců b_1 a b_2 přidáme stejný bitový řetězec b, také délky 1024 bitů. Postupujeme následovně:
   - vygenerujeme náhodně dvě prvočísla p_1 a p_2 velikosti přibližně 512 bitů taková, že veřejný exponent e bude nesoudělný s čísly $p_1 - 1$ a $p_2 - 1$;
   - vypočteme číslo b_0, $0 < b_0 < p_1 p_2$ tak, že $p_1 \mid (b_1 2^{1024} + b_0)$ a $p_2 \mid (b_2 2^{1024} + b_0)$ z Čínské věty o zbytcích (pěkný popis čínské věty o zbytcích včetně jejího použití v šifrování RSA můžeme najít v [23]);
   - postupně volíme k rovno 0, 1, 2, ... a pro každé k spočteme $b = b_0 + k p_1 p_2$, ověříme, zda obě čísla $q_1 = (b_1 2^{1024} + b) / p_1$ a $q_2 = (b_2 2^{1024} + b) / p_2$ jsou prvočísla a zda e je nesoudělný s čísly $q_1 - 1$ a $q_2 - 1$;
   - pokud číslo k je příliš velké, že $b \geq 2^{1024}$, zvolíme náhodně nová prvočísla p_1 a p_2 a celý proces opakujeme;
   - pokud jsme našli příslušná prvočísla q_1 a q_2, našli jsme také dva RSA moduly $n_1 = b_1 2^{1024} + b = p_1 q_1$ a $n_2 = b_2 2^{1024} + b = p_2 q_2$.
     Pokud jsou prvočísla p_1 a p_2 velikosti okolo 500 bitů, pak tento postup vrátí výsledek obvykle do několika minut.
5. Vložíme RSA modul n_1 do certifikátu a tím máme kompletní část dat, kterou bude CA podepisovat. Můžeme nyní spočítat MD5 hash celé části k podpisu (včetně paddingu a standardní hodnoty inicializačního vektoru).
6. Nyní můžeme připojit podpis certifikační autority a tím máme kompletní první certifikát.
7. Druhý certifikát bude totožný, jen místo RSA modulu n_1 vložíme do certifikátu RSA modul n_2. Podpisy CA budou stejné neboť MD5 hashe obou certifikátů jsou stejné.

Poznámka: RSA moduly, které jsme takto vytvořili splňují všechny požadavky na bezpečnost RSA modulů (velikost modulu 2048 bitů, velikost prvočísel kolem 500 bitů a standardní použití veřejného exponentu).

Na závěr uvádíme příklad kolidujících certifikátů v kódování DER včetně čitelné podoby v jazyce ASN.1.

tag	**délka**	**data**	**poznámka**
30	820335		ASN.1 hlavička
30	82021D		zde začínají podpisovaná data
A0	03		
02	01	02	
02	04	03507449	
30	0D		
06	09	2A864886F70D010104	
05	00		
30	3D		v této části jsou informace o
31	1A		vydavateli certifikátu (CA)
30	18		

```
06   03     550403
13   11     4861736820436F6C6C697369
            6F6E204341
31   12
30   10
06   03     550407
13   09     45696E64686F76656E
31   0B
30   09
06   03     550406
13   02     4E4C
```

30	1E		část udávající platnost
17	0D	303530323031303030303030315A	certifikátu
17	0D	303730323031303030303030315A	

30	60		v této části jsou informace o
31	17		uživateli, pro kterého je
30	15		vydán – zde je možné přidat
06	03	550403	„výplň“ pro dosažení násobku
13	0E	4861736820436F6C6C69	512-ti bitů
		73696F6E	
31	24		
30	22		
06	03	55040A	
13	1B	7765207573656420612063 6F6C6C6973696F6E20666F72204D4435	
31	12		
30	10		
06	03	550407	
13	09	45696E64686F76656E	
31	0B		
30	09		
06	03	550406	
13	02	4E4C	
30	820122		

30	0D		
06	09	2A864886F70D010101	
05	00		
03	82010F	00	moduly veřejného klíče
30	82010A		jeden modře, druhý červeně
02	820101	00	odlišnosti jsou vyznačeny žlutě
		CAB9E742C4B626871AB9A524846B05C1	CAB9E742C4B626871AB9A524846B05C1
		8895FB9365E9A69F480392FF2C3B3F79	8895FB1365E9A69F480392FF2C3B3F79
		41AD3406FFADB4034BDF847A4D37014F	41AD3406FFADB4034BDF847A4DB7014F
		DB3283CB19D46FA8A765C6B3F016BF30	DB3283CB19D46FA8A765C633F016BF30
		6AFF7C2E5773689B3319B81564ABE7F5	6AFF7C2E5773689B3319B81564ABE7F5
		B9CF66C5E4FE790CEE047D36CC77B0AE	B9CF6645E4FE790CEE047D36CC77B0AE
		5D087F30B560EB8872B34D4067786 62D	5D087F30B560EB8872B34D4067F8652D
		D88464677DBD9B80989EF24FB82E0EA3	D88464677DBD9B80989EF2CFB82E0EA3
		2B5864AF33B8FE8659B094464699F477	2B5864AF33B8FE8659B094464699F477
		A6BFCA348C23CF681EC0A846A8B27A29	A6BFCA348C23CF681EC0A846A8B27A29
		071B563A1316B05F3827B82FB1F9DE1F	071B563A1316B05F3827B82FB1F9DE1F
		238F3D12AD0DDAA97DDBCFCEEAD10939	238F3D12AD0DDAA97DDBCFCEEAD10939
		5E46E018AE237CE59355AC931872284C	5E46E018AE237CE59355AC931872284C
		3A293FE9117941A1AD528364A0687AFF	3A293FE9117941A1AD528364A0687AFF
		6083B14B009DD952C866CA43A0F41A7D	6083B14B009DD952C866CA43A0F41A7D
		CE5876C16CB346E9A718091CEC3D57D9	CE5876C16CB346E9A718091CEC3D57D9
02	03	010001	

A3	1A		rozšíření certifikátu
30	18		
30	09		
06	03	551D13	

```
04   02     3000
30   0B
06   03     551D0F
04   04
03   02     05E0
-------------------------------------------------------------------
30   0D                                  identifikátor algoritmu podpisu
06   09     2A864886F70D010104
05   00
-------------------------------------------------------------------
03   820101 00                           podpis
     1319E6FF66EF8621AEAE0CFBD2C067B99C3834C00BE88E0A97E60205BC5ECD85
     646B6698BD2E91324826C8B10E2167EFF264C5E45A234FDE5723A751EA2B7913
     06221B54B4C20E4CD16562D698ADE4D633F053D653F8BE9C4D402EC9F92D3630
     98DD560596F7BF095AF3C9FED7EE2B49218018003F5C65F0511D454E6E522913
     2D0494B7B65EF9585AA9D433094FDB4F9C994610AFE0F23FB26E5D246539AEFF
     B6E0B0DF35B4D9AE3CF768C5AABC93558DF87BF421288E79E9ADCBB8DA236452
     8E74F81348FFB9F5FAC43E974F3D79CCA222FD675BFD3B808A3F66104232C806
     A25309A187D103D750893436D4A32909 FE5C76B45495F52F29CF66A9E3DD473F
```

Stejné kolidující ceritifikáty v jazyce ASN.1:

```
Certificate:
   Data:
      Version: 3 (0x2)
      Serial number: 55604297 (0x03507449)
      Signature Algorithm: md5withRSAEncryption
      Issuer: CN=Hash Collision CA, L=Eindhoven, C=NL
      Validity
         Not Before: Feb 1 00:00:01 2005 GMT
         Not After : Feb 1 00:00:01 2007 GMT
      Subject: CN=Hash Collision, OU=we used a collision for MD5,
L=Eindhoven, C=NL
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
            Modulus:
CA:B9:E7:42:C4:B6:26:87:1A:B9:A5:24 CA:B9:E7:42:C4:B6:26:87:1A:B9:A5:24
84:6B:05:C1:88:95:FB:93:65:E9:A6:9F 84:6B:05:C1:88:95:FB:13:65:E9:A6:9F
48:03:92:FF:2C:3B:3F:79:41:AD:34:06 48:03:92:FF:2C:3B:3F:79:41:AD:34:06
FF:AD:B4:03:4B:DF:84:7A:4D:37:01:4F FF:AD:B4:03:4B:DF:84:7A:4D:B7:01:4F
DB:32:83:CB:19:D4:6F:A8:A7:65:C6:B3 DB:32:83:CB:19:D4:6F:A8:A7:65:C6:33
F0:16:BF:30:6A:FF:7C:2E:57:73:68:9B F0:16:BF:30:6A:FF:7C:2E:57:73:68:9B
33:19:B8:15:64:AB:E7:F5:B9:CF:66:C5 33:19:B8:15:64:AB:E7:F5:B9:CF:66:45
E4:FE:79:0C:EE:04:7D:36:CC:77:B0:AE E4:FE:79:0C:EE:04:7D:36:CC:77:B0:AE
5D:08:7F:30:B5:60:EB:88:72:B3:4D:40 5D:08:7F:30:B5:60:EB:88:72:B3:4D:40
67:78:66:2D:D8:84:64:67:7D:BD:9B:80 67:F8:65:2D:D8:84:64:67:7D:BD:9B:80
98:9E:F2:4F:B8:2E:0E:A3:2B:58:64:AF 98:9E:F2:CF:B8:2E:0E:A3:2B:58:64:AF
33:B8:FE:86:59:B0:94:46:46:99:F4:77 33:B8:FE:86:59:B0:94:46:46:99:F4:77
A6:BF:CA:34:8C:23:CF:68:1E:C0:A8:46 A6:BF:CA:34:8C:23:CF:68:1E:C0:A8:46
A8:B2:7A:29:07:1B:56:3A:13:16:B0:5F A8:B2:7A:29:07:1B:56:3A:13:16:B0:5F
38:27:B8:2F:B1:F9:DE:1F:23:8F:3D:12 38:27:B8:2F:B1:F9:DE:1F:23:8F:3D:12
AD:0D:DA:A9:7D:DB:CF:CE:EA:D1:09:39 AD:0D:DA:A9:7D:DB:CF:CE:EA:D1:09:39
5E:46:E0:18:AE:23:7C:E5:93:55:AC:93 5E:46:E0:18:AE:23:7C:E5:93:55:AC:93
18:72:28:4C:3A:29:3F:E9:11:79:41:A1 18:72:28:4C:3A:29:3F:E9:11:79:41:A1
AD:52:83:64:A0:68:7A:FF:60:83:B1:4B AD:52:83:64:A0:68:7A:FF:60:83:B1:4B
00:9D:D9:52:C8:66:CA:43:A0:F4:1A:7D 00:9D:D9:52:C8:66:CA:43:A0:F4:1A:7D
CE:58:76:C1:6C:B3:46:E9:A7:18:09:1C CE:58:76:C1:6C:B3:46:E9:A7:18:09:1C
EC:3D:57:D9                         EC:3D:57:D9
            Exponent: 65537 (0x10001)
      X509v3 extensions:
         X509v3 Basic constraints:
            CA:False
         X509v3 Key Usage: critical
            Digital Signature, Key Encipherment
```

```
Signature Algorithm: md5withRSAEncryption
   13:19:E6:FF:66:EF:86:21:AE:AE:0C:FB:D2:C0:67:B9:9C:38:34:C0:0B
   E8:8E:0A:97:E6:02:05:BC:5E:CD:85:64:6B:66:98:BD:2E:91:32:48:26
   C8:B1:0E:21:67:EF:F2:64:C5:E4:5A:23:4F:DE:57:23:A7:51:EA:2B:79
   13:06:22:1B:54:B4:C2:0E:4C:D1:65:62:D6:98:AD:E4:D6:33:F0:53:D6
   53:F8:BE:9C:4D:40:2E:C9:F9:2D:36:30:98:DD:56:05:96:F7:BF:09:5A
   F3:C9:FE:D7:EE:2B:49:21:80:18:00:3F:5C:65:F0:51:1D:45:4E:6E:52
   29:13:2D:04:94:B7:B6:5E:F9:58:5A:A9:D4:33:09:4F:DB:4F:9C:99:46
   10:AF:E0:F2:3F:B2:6E:5D:24:65:39:AE:FF:B6:E0:B0:DF:35:B4:D9:AE
   3C:F7:68:C5:AA:BC:93:55:8D:F8:7B:F4:21:28:8E:79:E9:AD:CB:B8:DA
   23:64:52:8E:74:F8:13:48:FF:B9:F5:FA:C4:3E:97:4F:3D:79:CC:A2:22
   FD:67:5B:FD:3B:80:8A:3F:66:10:42:32:C8:06:A2:53:09:A1:87:D1:03
   D7:50:89:34:36:D4:A3:29:09:FE:5C:76:B4:54:95:F5:2F:29:CF:66:A9
   E3:DD:47:3F
```

Možnost vytvořit dva stejné certifikáty lišící se pouze ve veřejném klíči zcela ničí smyl a účel certifikátů. Hlavní funkce certifikátu je zaručení příslušnosti daného veřejného klíče subjektu certifikátu. Tedy nemohou existovat dva certifikáty se stejným sériovým číslem (v rámci jedné CA) a různými veřejnými klíči, my jsme však ukázali, že takové dva certifikáty je možné vyrobit.

Pro úplnost uveďme ještě hlavní myšlenku hledání kolidujících certifikátů popsáno Stevensem [7], jež se nechal inspirovat výše uvedeným postupem. Jeho hlavní odlišnou představou je vytvořit kolidující certifikáty jež budou mít různé subjekty certifikátu (tj. osoby, pro které je certifikát vystaven).

Kolize X.509 certifikátů s volitelným prefixem

Tato část je převzata z práce M. M. J. Stevense [7].

Kolizí s volitelným prefixem rozumíme dvojici zpráv M a M', jež sestávají z libovolně volitelných prefixů P a P' (ne nutně stejné délky) a dopočítaných sufixů S a S' tak, aby platilo $M = P \| S$ a $M' = P' \| S'$ a zároveň $MD5(M) = MD5(M')$. Dále požadujeme, aby přidání libovolné koncovky S'' opět vedlo ke kolizi $MD5(M \| S'') = MD5(M' \| S'')$ - což je již zřejmé, např. z obrázku Obr. 5.

Oba sufixy S a S' se skládají ze tří částí. Výplňové řetězce (padding bitstrings) S_p a S'_p mohou být libovolného obsahu, ale musí mít takový počet bitů, aby zajistily, že délka řetězců $P \| S_p$ a $P' \| S'_p$ bude v obou případech rovna $512n - 96$ bitů. Následují řetězce narozeninového útoku (birthday bitstrings) S_b a S'_b, jež mají oba délku 96 bitů a tedy doplňují *n*-tý blok zprávy. Tyto řetězce jsou za pomoci narozeninového útoku vybírány tak, aby rozdíl mezihodnot hashe MD5 pro řetězce $P \| S_p \| S_b$ a $P \| S'_p \| S'_b$ byl vhodně připraven pro aplikaci několika předkolizních bloků (near-collisions blocks) v řetězcích S_c a S'_c jež vede k vynulování tohoto rozdílu mezihodnot hashe. Tedy chceme vynulovat rozdíly mezihodnot hashe pomocí několika po sobě jdoucích skoro-kolizí, jež dohromady tvoří řetězce S_c a S'_c. Každému rozdílu mezihodnot hashe je přiřazena jeho váha. Pro každou skoro-kolizi sestavíme diferenční cestu tak že váha následujícího rozdílu bude nižší než váha rozdílu předchozího. Takto postupujeme dokud nezískáme nulovou hodnotu rozdílu mezihodnot hashí.

Přibližme si nyní, jak Stevens v práci [7] tento postup použil při sestavování kolidujících X.509 certifikátů. Výplňové části sufixů S_p a S'_p jsou uplatněny ještě před výpisem RSA

modulu, tj. před prvním bitem RSA moduju je přesně $512n-96$ bitů, které splňují podmínky standardu X.509. Výplňové bity lze nejlépe vložit opět do popisu subjektu certifikátu. Další části sufixů S_b a S_b', S_c a S_c' jsou pak vhodně skryty v první části výpisu RSA modulu.

Při dodržování pravidel pro diferenční cesty zjistíme, že k rozdílu mezihodnot hashe můžeme přidávat pouze řetězce tvaru $(0, 2^i, 2^i, 2^i)$. Dokážeme tedy vynulovat jen takové rozdíly mezihodnot hashe, které jsou tvatu $(0, \partial w, \partial w, \partial w)$, pro nějaké slovo ∂w. Narozeninový útok hledá kolize mezihodnot hashe $IHV = (a, b, c, d)$ a $IHV' = (a', b', c', d')$ tak, že $(a, b-c, b-d) = (a', b'-c', b'-d')$, kde použijeme podmínky $\partial a = 0$ a $\partial b = \partial c = \partial d = \partial w$. Výsledkem hledání jsou tři slova po 32 bitech, tedy celkem 96 bitů.

Část certifikátu po řetězec narozeninového útoku S_b resp. S_b' (včetně) bude mít délku, jež je násobkem 512-ti bitů. Zároveň rozdíl mezihodnot hashe bude tvaru $(0, \partial w, \partial w, \partial w)$, pro nějaké slovo ∂w. Nyní můžeme postupně sestavit řetězce S_c a S_c', které zajistí nulový rozdíl mezihodnot hashe. Stevens našel takové slovo ∂w, že řetězec S_c, resp. S_c' se skládal z 8 bloků, tedy po použití 8 bloků a jejich diferenčních cest byl nakonec rozdíl mezihodnot hashe nulový.

Řetězec narozeninového útoku S_b, resp. S_b' a 8 předkolizních bloků řetězce S_c, resp. S_c' představují celkem $96 + 8 \times 512 = 4192$ počátečních bitů vyjádření RSA modulu. Použitím metody popsané v předchozí sekci najdeme bitový řetězec S_m takový, že řetězce $S_b \parallel S_c \parallel S_m$ a $S_b' \parallel S_c' \parallel S_m$ splňují podmínky pro RSA modul, tj. jsou to výsledky násobení dvou velkých prvočísel. Takto vytvořené certifikáty mají stejný podpis certifikační autority.

4.3.2. Další praktické využití MD5 kolizí

Předložené útoky na funkci MD5 lze použít i v jiných oblastech než při konstrukci kolidujících X.509 certifikátů. Útoky na MD5 jsou prakticky použitelné všude kde se používá hashovací funkce MD5. Vždy se postupuje stejným způsobem jako na obrázku Obr. 5; kolidující bloky vložíme doprostřed zprávy tak, aby předřazená zpráva měla délku násobku 512-ti bitů. Už pouze dvě zveřejněné kolize MD5 (bez postupu jak je získat) je možné využít k praktickému útoku jak ukázal Ondrej Mikle v práci [25] a o pár dnů později Dan Kaminsky v práci [24].

Prakticky všude, kde se používá hashovací funkce MD5 je možné využít popsané teoretické útoky a způsobit tak větší či menší škodu. Je možné sestrojit různé dokumenty (v libovolném formátu – PDF, EXE, PS...), které mají stejnou MD5 hash. Pak můžete jako zaměstnanec předložit svému šéfovi k podpisu potvrzení o dovolené, aniž by tušil, že zároveň podepisuje vaše zvýšení platu či vyplacení velké prémie.

5. Hashovací funkce SHA-1

Hashovací funkce SHA-1 patří do rodiny funkcí SHA (Secure Hash Alghoritm). Autorem SHA algoritmu je NSA, národní bezpečnostní agentura v USA a vydal jej NIST, národní institut pro standardy v USA, jako americký standard FIPS. Původní specifikace byla vydána v roce 1993 jako standard FIPS 180 (verze dnes známá jako SHA-0). Ta byla však za krátko stažena a roku 1995 byl vydán nový standard FIPS 180-1, který specifikoval algoritmus hashovací funkce SHA-1 (opravená verze funkce SHA-0, jež obsahovala značné bezpečnostní slabiny).

5.1. Popis algoritmu SHA-1

SHA-1 vytváří hash dlouhou 160 bitů. Velikost vstupní zprávy je omezena na $2^{64}-1$ bitů. Algoritmus hashovací funkce SHA-1 je založen na stejném principu jako hashovací funkce třídy MD (MD5, resp. MD4). Tedy SHA-1 je bloková hashovací funkce (vsupní zpráva se rozdělí do stějně dlouhých bloků), která je založena na Merkle-Damgardově konstrukci (viz. obrázek Obr. 2: Merkle-Damgardova konstrukce). Popišme si nyní algoritmus funkce SHA-1, jehož úplné znění lze nalézt v doporučení RFC 3174 [26] případně také ve standardu FIPS 180-1.

Hashovací funkce SHA-1 zpracová vstupní zprávu po jednotlivých blocích velikosti 512 bitů. Aby bylo zajištěno, že délka vstupní zprávy bude násobkem 512 bitů, používá se výplň (*padding*) zprávy. Vstup je doplněn zprava podle následujícího postupu. Délku původní vstupní zprávy označíme l.

1. Přidá se bit „1“, délka zprávy je nyní $l+1$.
2. Přidá se tolik bitů „0“, aby délka vstupu byla rovna $512k+448$, tedy 64 bitů zůstane nevyplněno.
3. Do zbývajících 64 bitů se vyplní délka původní vstupní zprávy v bitech. Z čehož je také zřejmé omezení velikosti vstupní zprávy na $2^{64}-1$ bitů.

Takto upravený vstupní řetězec má nyní délku násobku 512 bitů. Vstup M se následně rozdělí na N stejných bloků $M_1, M_2, ..., M_N$, každý délky 512 bitů. Funkce SHA-1 zpracovává postupně jednotlivé bloky, každý blok v jedné rundě. Každá runda se skládá ze čtyř cyklů a v každém cyklu proběhne 20 operací. Tedy v každé rundě proběhne celkem 80 kroků. Očíslujeme tyto kroky $t = 0,...,79$.

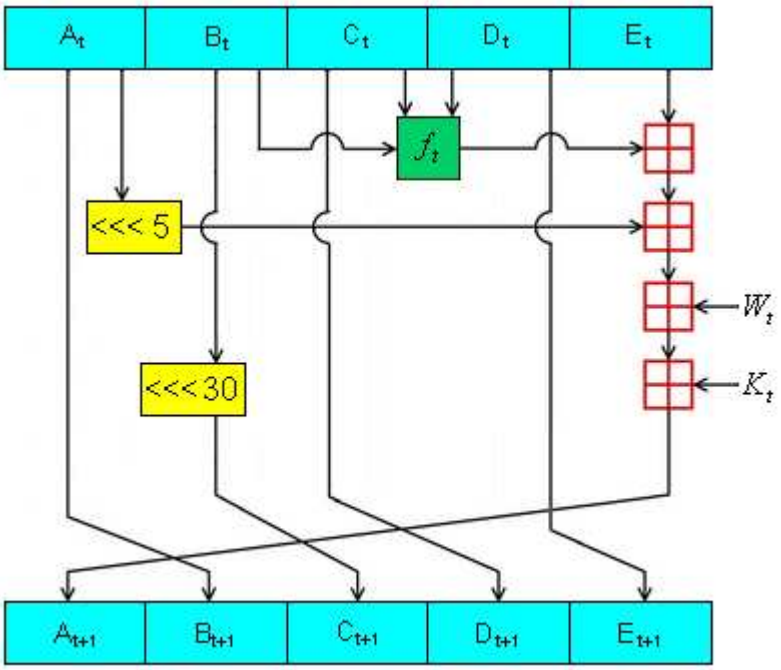
Délka výstupní hashe funkce SHA-1 je 160 bitů, tedy kompresní funkce SHA-1 pracuje s kontextem o délce 160 bitů. Kontext tvoří 5 slov A, B, C, D, E a každé slovo má délku 32 bitů. Na začátku výpočtu jsou tato slova naplněna inicializačními hodnotami. Tyto hodnoty jsou uvedeny v tabulce Tab. 10 v little endian uspořádání (nejméně významný byte je první z leva). Uvedené počáteční nastavení pěti registrů se použije pro první rundu při počítání SHA-1 hashe. Po dokončení celého výpočtu lze nalézt výslednou SHA-1 hash právě v těchto pěti registrech.

Tab. 10: Inicializační hodnoty slov funkce SHA-1

Slovo	Hexadecimální hodnota
A_0	67452301
B_0	efcdab89
C_0	98badcfe
D_0	10325476
E_0	c3d2e1f0

Popišme si nyní detailně operace, které probíhají v kompresní funkci SHA-1. Obrázek Obr. 6 představuje průběh jednoho kroku t $(t = 0,...,79)$ kompresní funkce algoritmu. Všechny operace kompresní funkce probíhají v rámci slov (tj. 32-bitových bloků), přičemž symbol $\boxplus$ znamená sčítání slov (tj. sčítání modulo 2^{32}) a symboly $<<<5$ a $<<<30$ znamenají rotaci bitů vlevo o 5 bitových pozic pro slovo A_t a o 30 bitových pozic pro slovo B_t. Oproti funkci MD5 má kompresní funkce SHA-1 o jeden regisr více, což se ve výsledku projeví právě rozdílnou délkou hashe (u MD5 je to 128 bitů – 4 registry, u SHA-1 je to 160 bitů – 5 registrů).

Obr. 6: Detail kompresní funkce algoritmu SHA-1



Popišme si průběh jedné rundy, při které dochází ke zpracování právě jednoho bloku vstupní zprávy. V závislosti na prováděném kroku t $(t = 0,...,79)$ se použije příslušná aditivní konstanta K_t, nelineární funkce f_t a příslušné slovo W_t právě zpracovávaného bloku vstupní zprávy. Použití hodnoty aditivní konstanty K_t se řídí následujícími pravidly (hodnoty jsou uvedeny v hexadecimálním vyjádření):

$$K_t = \begin{cases} \texttt{5a827999} & \text{pro } 0 \le t \le 19, \\ \texttt{6ed9eba1} & \text{pro } 20 \le t \le 39, \\ \texttt{8f1bbcdc} & \text{pro } 40 \le t \le 59, \\ \texttt{ca62c1d6} & \text{pro } 60 \le t \le 79. \end{cases}$$

Nelineární funkce f_t, kde proměnné X, Y, Z, reprezentují postupně vstupní slova B_t, C_t, D_t (jak je vidět na obrázku Obr. 6), se pro jednotlivé kroky $t = 0,...,79$ použije podle následujícího předpisu:

$$f_t(X,Y,Z) = \begin{cases} F(X,Y,Z) = (X \wedge Y) \vee (\overline{X} \wedge Z) & \text{pro } 0 \le t \le 19, \\ G(X,Y,Z) = X \oplus Y \oplus Z & \text{pro } 20 \le t \le 39, \\ H(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) & \text{pro } 40 \le t \le 59, \\ I(X,Y,Z) = X \oplus Y \oplus Z & \text{pro } 60 \le t \le 79. \end{cases}$$

Všechny operace se provádějí na bitové úrovni, přičemž $\overline{X}$ je bitový doplněk (operace NOT) a $\oplus$ značí bitový součet (operace XOR).

Vstupní 512-ti bitový blok se rozdělí na šestnáct 32-bitových slov $m_0, m_1, ..., m_{15}$. Těchto 16 slov se expanduje do 80 slov $W_0,...,W_{79}$ následujícím způsobem.

$$W_t = \begin{cases} m_t & \text{pro } 0 \le t \le 15, \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1 & \text{pro } 16 \le t \le 79. \end{cases}$$

Tento krok expandování vstupního bloku je značně složitější než u funkce MD5, kde jsme si vždy vystačili s prostým výběrem některého ze slov zptacovávaného vstupního bloku. Pouze v tomto místě se SHA-1 liší od SHA-0, tedy funkce SHA-0 neobsahuje uvedený bitový posun při výpočtu expandovaných vstupních slov.

Celkový postup výpočtu hashe funkce SHA-1 je názorně vidět na obrázku Obr. 6. Při výpočtu postupujeme po jednotlivých krocích, $t = 0,...,79$. Znaménko + zde značí sčítání slov, tedy sčítání modulo 2^{32}.

$$\begin{aligned} A_{t+1} &= (A_t <<< 5) + f_t(B_t, C_t, D_t) + E_t + W_t + K_t \\ B_{t+1} &= A_t \\ C_{t+1} &= B_t <<< 30 \\ D_{t+1} &= C_t \\ E_{t+1} &= D_t \end{aligned}$$

Po zpracování celého jednoho vstupního bloku zprávy, tedy po dokončení jedné rundy, k výsledným hodnotám v jednotlivých registrech přičteme ještě iniciační hodnoty těchto registrů a tím získáme hodnotu mezihashe, jehož jednotlivá slova použijeme jako iniciační hodnoty pro další rundu.

$$H = (A_0 + A_{80}, B_0 + B_{80}, C_0 + C_{80}, D_0 + D_{80}, E_0 + E_{80})$$

Po zpracování celé vstupní zprávy lze stejným způsobem, tedy přičtením iniciačních hodnot registrů pro poslední rundu k hodnotám registru po doběhnutí této rundy, získat výslednou hodnotu SHA-1 hashe.

5.2. Vývoj útoků na funkci SHA-1

Výstupem hashovací funkce SHA-1 je hash délky 160 bitů. Obecný útok hrubou silou na SHA-1 má složitost 2^{80}, což by by tímto způsobem na běžném PC trvalo stovky miliónů let. Tedy základní vlastnost rezistence vůči kolizím tato funkce zaručeně splňuje. Pro překonání této složitosti je tedy nutné nalézt vhodný algoritmus, jež umožní nalézt kolizi SHA-1 v reálném čase.

První útok na funkci SHA-1 s nižší složitostí než 2^{80} publikovali V. Rijmen a E. Oswaldová [27]. Zvolili však zjednodušenou funkci SHA-1, místo 80 rund použili pouze 53.

Prvním významným útokem byl algoritmus zveřejněný čínskými kryptoanalytiky na začátku roku 2005. Tento útok dokáže nalézt kolizi plné funkce SHA-1 se složitostí 2^{69}, je založen na základním diferenčním útoku na funkci SHA-0 a také na technice modifikace zpráv, kterou použili při útoku na funkci MD5. Později téhož roku čínští matematici svůj útok vylepšili a dosáhli tak složitosti 2^{63}.

Dalším významným útočníkem na SHA-1 byl ústav IAIK Univerzity v Grazu, který snížil složitost útoku na 2^{61} a v srpnu roku 2007 zavedl projekt hledání aktuálních kolizí SHA-1 hrubou silou pomocí distribuovaného rozložení výpočtů na počítačích účastníků. Tento projekt jistě vyděsil mnoho lidí a zřejmě i kvůli tomu byl nakonec v květnu roku 2009 zastaven.

V první polovině roku 2009 tým autorů C. McDonald, P. Hawkes a J. Pieprzyk oznámili, že nalezli kryptoanalytický způsob jak nalézt kolizi plné funkce SHA-1 se složitostí pouhých 2^{52}. Příslušný článek „Differential Path for SHA-1 with complexity $O(2^{52})$“ byl zveřejněn v červnu 2009 a již v srpnu téhož roku požádali autoři o jeho stažení, neboť zjistili, že matematický předpoklad, na kterém stavěli nebyl korektní a tedy ani předložená složitost neplatí.

Přestože dosud nebyla nalezena konkrétní kolize pro plnou funkci SHA-1, je na místě s největší rozvážností plánovat další používání tého hashovací funkce. Přinejmenším v oblastech, jež pracují s velmi citlivými daty je vhodné zaměnit funkci SHA-1 za bezpečnější variantu SHA-2.

V ČR bylo vyhláškou Ministerstva Vnitra upraveno používání hashovací funkce SHA-1 pro účely zaručeného elektronického podpisu a všechny certifikační autority u nás od 1. 1. 2010 vydávají certifiáty zaručeného elektronického podpisu s použitím bezpečnější hashovací funkce SHA-2.

5.2.1. Hledání kolizí podle čínských matematiků

Tato část je převzata z práce dr. Wangové a kol. [28].

Na začátku roku 2005 čínští kryptoanalytici zveřejnili postup [28], pomocí něhož je možné nalézt kolizi prvního řádu funkce SHA-1 se složitostí významně nižší než je předpokládaná složitost útoku hrubou silou. Uváděná složitost 2^{69} je sice stále příliš vysoká pro získání reálné kolize plné funkce SHA-1, je to však zřejmě ten správný směr, kterým se vydat pro úplnou kompromitaci této hashovací funkce.

Autoři při sestavování algoritmu vycházeli především z vlastních výzkumů, úplné reálné kolize funkce SHA-0 [29] se složitostí pod 2^{40} a metody modifikace zpráv při hledání kolizí funkce MD5 [11].

Shrňme zde poznatky a pozorování pro hledání kolizí funkce SHA-0, jež jsou podstatné pro hledání částečné i plné kolize funkce SHA-1. Podrobné informace o hledání kolizí funkce SHA-0 lze nalézt v [29].

Jednoduše řečeno, lokální kolize jsou kolize pro několik kroků hashovací funce. Prosté avšak velmi důležité pozorování učiněné v [29] říká, že SHA-0 má 6-ti krokovou lokální kolizi, jež může začínat jakýmkoliv libovolným krokem.

Předpokládejme, že rozdíl vstupních zpráv v bitě j se poprvé objeví v kroku i (tj. $\delta m_{i,j} = m'_{i.j} - m_{i,j} = 1$, kde $i = 0,...,79$ a $j = 0,...,31$). Rozdíl vstupních slov je pak $\delta m_i = 2^j$. Tato změna ovlivní hodnoty registrů A, B, C, D, E postupně v následujících pěti krocích, tedy dostaneme nenulové hodnoty rozdilů registrů A_i, B_{i+1}, C_{i+2}, D_{i+3}, E_{i+4}. Pro kompenzaci těchto rozdílů a dosažení částečné kolize je nutné učinit více změn v dalších slovech zprávy. V tabulce Tab. 11 je uvedena diferenční cesta této 6-ti krokové lokální kolize (právě v šestém kroku nalézáme kolizi – v tomto kroku jsou opět všechny rozdíly registrů nulové).

Tab. 11: 6-ti kroková lokální kolize SHA-0 začínající krokem *i*. Měřítkem rozdílů je XOR. Sčítání v exponentech je modulo 32. Označení „nc“ znamená bez přenosu, δf je rozdíl výsledku použité nelineární funkce.

Krok	δm	δA	δB	δC	δD	δE	Podmínky
i	2^j	2^j					nc
$i + 1$	2^{j+5}		2^j				
$i + 2$	2^j			2^{j+30}			nc, $\delta f = 2^j$
$i + 3$	2^{j+30}				2^{j+30}		nc, $\delta f = 2^{j+30}$
$i + 4$	2^{j+30}					2^{j+30}	nc, $\delta f = 2^{j+30}$
$i + 5$	2^{j+30}						nc

Pravděpodobnost získání uvedené lokální kolize závisí na nelineární funkci f, bitové pozici j a některých podmínkách na bity zprávy. Pro získání lokální kolize SHA-0 se zvolí $j = 1$ a tedy bit $j + 30 = 31$ se stane nejdůležitějším bitem pro odstranění efektu přenosů v posledních třech krocích. Navíc následující podmínka

$$m_{i,1} = \neg m_{i+1,6}$$

pomůže úplně kompenzovat proměnlivý rozdíl v druhém kroku lokální kolize, kde $\neg x$ znamená bitovou negaci slova x.

Podmínka na vstupní zprávu pro kroky $40 \le t \le 59$ (tedy při použití nelineární funkce $H(X,Y.Z)$)

$$m_{i,1} = \neg m_{i+2,1}$$

pomůže eliminovat rozdíl způsobený nelineární funkcí f ve třetím kroku lokální kolize. Proto lokální kolize funkce SHA-0 nezávisí na expanzi vstupních slov zprávy, což můžeme

také aplikovat na funkci SHA-1 (neboť obě hashovací funkce mají stejný postup expandování vstupních slov zprávy).

Diferenční cesta pro funkci SHA-0 použitá v [29] je v podstatě posloupnost lokálních kolizí spojených dohromady. Pro sestavení takovéto diferenční cesty potřebujeme nalézt soubor příslušných počátečních kroků pro každou lokální kolizi. Tyto počáteční kroky mohou být určeny 80-ti bitovým vektorem $x = (x_0,...,x_{79})$ nazývaným *poruchový vektor (disturbance vector),* $x_i = 1$ znamená, že lokální kolize začíná krokem i.

Pro 80 proměnných x_i je 16 po sobě jdoucích určujících pro všechny ostatní (neboť poruchový vektor splňuje rekurzivní rovnici pro expanzi vstupních slov). Můžeme libovolně volit 16 proměnných x_i pro celkový počet 2^{16} možných poruchových vektorů. Pak tedy „dobrý" poruchový vektor splňující určité podmínky může být nalezen se složitostí 2^{16}.

Metoda konstrukce diferenčních cest pro funkci SHA-0 je přirozeně rozšířena na funci SHA- 1, viz [30] a [31]. Pro případ funkce SHA-1 každý člen x_i poruchového vektoru je 32-bitové slovo místo jednotlivého bitu. Pro přehlednost označme těchto 80 vektorů *poruchovou maticí,* přičemž každé 32-bitové slovo v této matici nazveme *poruchovým vektorem*. Tyto poruchové vektory splňují rekurzivní rovnici pro expanzi slov vstupní zprávy funkce SHA-1. Tedy pro $i = 16,...,79$ platí:

$$x_i = (x_{i-3} \oplus x_{i-8} \oplus x_{i-14} \oplus x_{i-16}) <<< 1$$

Aby poruchová matice vedla k možné kolizi, musí splňovat několik podmínek, detaily viz [29]. Tyto podmínky mohou být zřejmým způsobem rozšířeny také na funkci SHA-1, jsou shrnuty v tabulce Tab. 12.

Tab. 12: Podmínky na poruchovou matici SHA-1 s *t* kroky.

Podmínka	Účel
$x_i = 0$ pro $i = t-5,..., t-1$	pro zajištění kolize v posledním kroku *t*
$x_i = 0$ pro $i = -5,...,-1$	předejít zkráceným lokálním kolizím v několika prvních krocích
žádné po sobě jsoucí jedničky na stejných bitových pozicích v prvních 16-ti proměnných	předejít nepravděpodobné kolizní cestě kvůli vlastnosti funkce *F(X,Y,Z)*

Poznamenejme, že Hammingova váha (počet nenulových bitů) poruchového vektoru je úzce spojena se složitostí útoku. Pro minimalizaci složitosti by měla být Hammingova váha tak malá, jak je to jen možné (jsou zde však ještě další drobné podmínky). Pro funkci SHA-0 byly nalezeny 3 poruchové vektory ze všech možných 2^{16}, přičemž pouze dva z nich splňovaly všechny tři podmínky. Pro funkci SHA-1 je mnohem komplikovanější najít dobrý poruchový vektor s nízkou Hammingovou váhou v tak velkém prostoru možností. Neboť Hammingova váha platného poruchového vektoru roste rychleji s každým krokem algoritmu, zdá se, že najít kolizi plné 80-ti krokové funkce SHA-1 pod hranicí složitosti 2^{80} je nemožné.

Nalezení poruchových vektorů s nízkou Hammingovou váhou je nezbytné pro sestrojení platné diferenční cesty vedoucí ke kolizi funkce SHA-1. Na druhé straně, tři podmínky

kladené na poruchové vektory (viz tabulka Tab. 12) se zdají být velkou překážkou. Vznikla tedy snaha odstranit některé z podmínek.

Klíčová myšlenka čínského útoku spočívá v eliminování všech podmínek na poruchové vektory. Tedy na poruchové vektory nejsou kladeny žádné podmínky kromě splnění podmínky rekurzivní rovnice pro expanzi slov vstupní zprávy. Toto umožňuje nalézt poruchové vektory s mnohem menší Hammingovou váhou, než postupy používané v předchozích útocích na funkci SHA-1.

Uvedeme zde několik postupů pro získání platné diferenční cesty dané uvedenými poruchovými vektory. Výsledná diferenční cesta je značně složitá, především pro první rundu kvůli postupným poruchám a také kvůli zkráceným lokálním kolizím v krocích -5 až -1 (první dvě podmínky z tabulky Tab. 12). Toto je nejsložitější, ale zásadní část celého algoritmu.

Po sestavení platné diferenční cesty aplikujeme metodu modifikace zpráv (představenou také v algoritmu útoku na funkci MD5) pro snížení složitosti dalšího hledání kolidujících zpráv. Toto rozšíření vyžaduje pečlivé odvozování počátečních podmínek na jednotlivá slova vstupní zprávy a hodnoty registrů.

Kombinací všech uvedených technik a jednoduchého triku „včasného zastavení“ při implementaci hledání kolizí, můžeme dosáhnout nalezení kolize funkce SHA-1 se složitostí menší než 2^{69}.

Hledání poruchových vektorů s nízkou Hammingovou váhou

První důležitý krok celého algoritmu je najít „dobré“ poruchové vektory. Pokud nevyžadujeme splnění podmínek z tabulky Tab. 12, je hledání poruchových vektorů poněkud jednodušší. Protože můžeme libovolně volit 16 členů poruchové matice a každý z nich má 32 bitů, prostor hledání poruchových vektorů pak může mít velikost až 2^{512}. Místo hledání poruchových vektorů s minimální Hammingovou váhou v celém tomto prostoru použijeme heuristiky, které omezí hledání na podprostor nejpravděpodobnějších výskytů „dobrých“ poruchových vektorů.

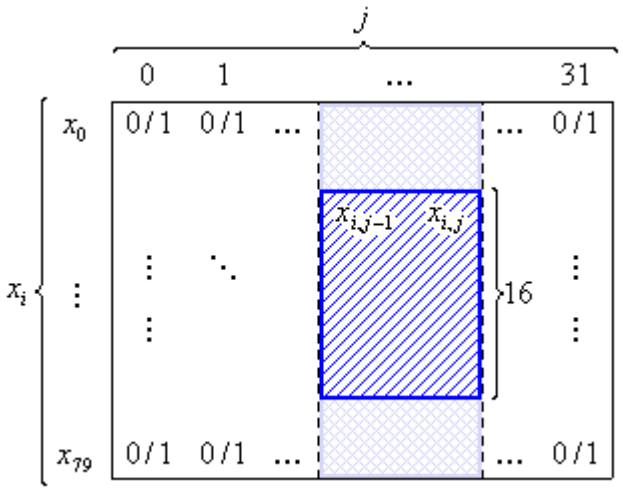
Poruchová matice je matice typu (80, 32), kde každý prvek matice je jednotlivý bit (0 nebo 1). Řádky poruchové matice představují jednotlivé poruchové vektory. Jednoduché pozorování: pro matice s nízkou Hammingovou váhou se budou nenulové prvky pravděpodobně koncentrovat do několika po sobě jdoucích sloupců matice (viz [28]). Proto můžeme zvolit dva prvky matice $x_{i,j-1}$ a $x_{i,j}$ a dva 16-ti bitové sloupce začínající na těchto pozicích, tzv. *informační okénko* (znázorněno na obrázku Obr. 7).

Velikost informačního okénka je dána podmínkou splnění rekurzivní rovnice pro expanzi vstupních slov zprávy, pak je možné 16 vektorů volit libovolně a ostaní lze jednoznačně dopočítat. Počet všech možností, jež může informační okénko nabývat je 2^{32}. Máme tedy 64 možností pro volbu pozice i ($i = 0,...,63$) a 31 možností pro volbu pozice j ($j = 1,...,31$). Ve skutečnosti různé volby pozice j pro stejné i produkují poruchové vektory, které jsou vzájemnou rotací se stejnou Hammingovou váhou. Volbou $j = 1$ můžeme minimalizovat efekt přenosu (viz výše) a tedy velikost prostoru hledání poruchových vektorů bude $64 \times 2^{32} = 2^{38}$.

Pomocí uvedeného postupu nalezneme nejprve nejvhodnější poruchové vektory pro jednoblokovou kolizi (tj. kolizi pro jeden blok vstupní zprávy). Pro úplnou funkci SHA-1 dostaneme nejlepších výsledků volbou $x_{64,2} = 1$ a $x_{i,2} = 0$ pro $i = 65,...79$. Výsledné

poruchové vektory jsou uvedeny v tabulce Tab. SHA1-1. Nejvhodnější poruchové vektory pro funkci SHA-1 zkrácenou na t kroků jsou stejné jako pro úplnou funkci SHA-1 s vynecháním prvních $80-t$ vektorů. Pro varianty SHA-1 do 75 kroků je Hammingova váha poruchové matice stále dostatečně malá na to, aby bylo možné provést útok se složitostí menší než 2^{80}, přehled těchto variant je uveden v tabulce Tab. SHA1-2.

Obr. 7: Poruchová matice - modře znázorněno informační okénko a jeho možný rozsah pro konkrétní volbu pozice *j*.



Ve výpočtu „dobrých“ poruchových vektorů pro blízké kolize a pro kolize dvou bloků vstupní zprávy pokračujeme i po kroku 79. Přitom zajišťujeme splnění podmínky rekurzivní rovnice pro expanzi vstupních slov zprávy přirozeně rozšířenou na další kroky. Tyto vektory jsou rovněž uvedeny v tabulce Tab. SHA1-1 až do kroku 95.

Potom hledáme všechny možné 80-ti vektorové intervaly $[x_i,...,x_{i+79}]$. Pro konstrukci blízké kolize lze použít kteroukoliv sadu 80-ti vektorů s dostatečně nízkou Hammingovou váhou. Čínští matematici nalezli 12 vhodných sad vektorů (viz [28]). To nám dává určitou možnost volby vhodné sady vektorů pro dosažení nejnižší složitosti hledání kolizí, pokud vezmeme v úvahu ještě další kritéria mimo Hammingovy váhy.

Techniky sestavení diferenčních cest

Jak již bylo řečeno, poruchové vektory nesplňují žádnou z podmínek z tabulky Tab. 12. Proto také je nalezení korektní diferenční cesty, která vede ke kolizi, mnohem komplikovanější. Toto je nejsložitěšní a zároven nejdůležitější část útoku na hashovací funkci SHA-1, protože bez konkrétní diferenční cety není možné nalézt reálnou kolizi.

Uvedeme zde pouze stručně hlavní myšlenky hledání vhodné diferenční cesty. Podrobnější popis celého postupu lze nalézt v [29], případně také v [28].

- Použití rozdílu (odčítání) místo bitového rozdílu (operace XOR) jako měřítka rozdílu, aby bylo možné provést přesnější analýzu.
- Využití speciálních diferenciálních vlastností nelineární funkce $F(X,Y,Z)$. Pokud je bitový rozdíl v jednom ze tří vstupů, pak na výstupu dostaneme 1 s pravděpodobností ¼, -1 s pravděpodobností ¼ a 0 s pravděpodobností ½. (Toto lze jednoduše ukázat výčem všech možností). Tato funkce tedy může vstupní rozdíl *zachovat*, *obrátit* nebo *vstřebat*, což nám dává značnou flexibilitu při tvorbě diferenční cesty.

- Využití efektu artimetického přenosu (carry). Neboť platí:

  $$2^j = -2^j - 2^{j+1} - ... - 2^{j+k-1} + 2^{j+k}$$, pro libovolné *k*,

  jeden bitový rozdíl *j* může být rozšířen do několika bitů. Tato vlastnost umožňuje zavést další speciální bitové rozdíly.

- Použití různých rozdílů zprávy pro 6-ti krokové lokální kolize. Např. soubor $(2^j, 2^{j+5}, 0, 0, 0, 2^{j+30})$ jsou rozdíly zprávy pro lokální kolizi v prvních krocích.

Odvozování podmínek

Pokud sestavíme platnou diferenční cestu pro funkci SHA-1, pak můžeme odvodit podmínky pro bity vstupní zprávy a hodnoty registrů. Detailní postup odvozování podmínek pro funkci SHA-0 je uveden v [29]. Stejný postup lze také použít pro odvozování podmínek pro SHA-1, proto zde uvedeme pouze odlišnosti.

Vzhledem k tomu, že funkce SHA-1 obsahuje navíc oproti funkci SHA-0 bitový posun při expanzi slov vstupní zprávy lze předpokládat, že porucha (je vyjádřena jedničkou v poruchovém vektoru) se může objevit také na jiných bitových pozicích vstupních slov než je bit $j = 1$ (jak můžeme zjistit z tabulky Tab. SHA1-1). Zatímco u funkce SHA-0 všechny poruchy začínají v bitě $j = 1$ (tj. ve druhém bitě) slov vstupní zprávy. Pokud se toto stane v krocích $t = 20,...,39$ nebo $t = 60,...,79$ kde se používají nelineární funkce $G(X,Y,Z)$ a $I(X,Y,Z)$ (XOR funkce), pak počet podmínek se může zvýšit ze dvou až na čtyři pro každou lokální kolizi. To se může výrazně projevit na celkovém počtu podmínek.

Na příkladu si ukážeme jak lze snížit počet podmínek. Předpokládejme, že máme dvě sady rozdílů zpráv odpovídající dvěma po sobě jdoucím poruchám v rámci stejného kroku i, které produkují jednu 6-ti krokovou lokální kolizi. Máme tedy poruchy v bitech 0 a 1 poruchového vektoru x_i. Můžeme nastavit znaménka u těchto dvou bitů v rozdílu zpráv Δm_i tak, aby byla opačná. Tímto způsobem můžeme tyto rozdíly zpráv považovat za rozdíl jednoho bitu na pozici 0, neboť $2^1 - 2^0 = 2^0$.

Metody modifikace zpráv

Metody modifikace zpráv pro funkci SHA-1 lze použít stejné jako pro funkci MD5 (viz [10], [11] nebo odstavec *Modifikace zpráv* v podkapitole 4.1 Popis algoritmu MD5). Použitím *jednoduché modifikace zpráv* můžeme jednoduše upravit vstupní zprávy tak, aby byly splněny postačující podmínky pro jednotlivé hodnoty registrů pro prvních 16 kroků algoritmu SHA-1. Pomocí metody *mnohonásobné modifikace zpráv* můžeme docílit spnění některých dalších postačujících podmínek.

Všechny metody modifikace zpráv spočívají v tom, že se náhodně zvolí vstupní zpráva M a její modifikací se postupně splňují postačující podmínky pro hodnoty registrů v jednotlivých krocích. Pro funkci MD5 je možné se dostat až ke splnění počátečních podmínek v kroku $t = 24$ (viz [18]). Čínští matematikové však uvádějí, že po větším úsilí se jim podařilo splnit počáteční podmínky až do kroku $t = 22$ pro funkci SHA-1.

Výběr nejlepšího poruchového vektoru

Pokud máme odvozeny všechny podmínky a aplikovali jsme některou (nebo všechny) z metod modifikací zpráv, můžeme spočítat zbývající podmínky pro kroky $t = 20,...,79$.

Pravidla pro počítání podmínek závisí na použité nelineární funkci f_t a na pozicích výskytu poruch v jednotlivých krocích. Detaily jsou uvedeny v tabulce Tab. SHA1-3.

Podívejme se na poruchové vektory uvedené v tabulce Tab. SHA1-1, vidíme, že pro 80-ti krokovou blízkou kolizi je minimální Hammingova váha rovna 25 pro vektory 14 – 93 (indexované podle Kroku *i*). Naproti tomu minimální počet podmínek je 71 pro 80 vektorů s indexy 16 – 95. Podrobný výpočet počtu podmínek pro tyto vektory je uveden v tabulce Tab. SHA1-4.

Pokud použijeme minimální počet podmínek jako kritérium výběru, pak zvolíme vektory indexované 16 – 95 (viz tabulka Tab. SHA1-1) jako poruchové vektory pro sestrojení 80-ti krokové blízké kolize.

Hledání kolizí s využitím blízké kolize

Použitím myšlenky kolize více bloků (viz např. [11], [30] a také obrázek Obr. 5) můžeme sestrojit kolizi dvou bloků zprávy s využitím blízkých kolizí. Pro funkci MD5 je složitost hledání prvního bloku vyšší než složitost hledání druhého bloku. Pokud ponecháme bitové rozdíly na posledních dvou krocích jako volné proměnné, můžeme dosáhnout toho, že složitost hledání obou bloků kolizní zprávy bude stejná.

Nechť M_0 a M_0' jsou dva bloky (první bloky kolidujících zpráv) a $\Delta h_1 = h_1' - h_1$ je výsledek rozdílu jejich hashí po 80-ti krocích. Pokud se podíváme blíže na poruchové vektory jež jsme zvolili, zjistíme, že 4 poruchy v posledních pěti krocích se promítnou do rozdílu hashí Δh_1. Tato hodnota je pak použita jako vstupní rozdíl iniciačních vektorů pro druhý blok zprávy.

Pro sestrojení diferenční cesty pro druhé bloky zpráv M_1 a M_1' můžeme použít dvě techniky. Nejprve aplikujeme techniku popsanou v části *Techniky sestavení diferenčních cest*, takže Δh_1 může být „*vstřebáno*“ v prvních 16-ti krocích diferenčních cest. Poté nastavíme podmínky na M_1 tak, aby výsledný rozdíl Δh_2 měl opačná znaménka pro každý z rozdílů Δh_1. Jinými slovy, určíme podmínky tak, aby $\Delta h_2 + \Delta h_1 = 0$, čímž získáme kolizi po druhém bloku zprávy. Tyto podmínky nezvýší počet celkový počet podmínek pro výslednou diferenční cestu a tedy neovlivní složitost. Blízká kolize druhého bloku zprávy může být nalezena se stejnou složitostí jako blízká kolize prvního bloku zprávy.

Analýza složitosti

Použitím modifikačních technik popsaných výše můžeme opravit podmínky v krocích 16 – 21. Nejprve si můžeme předvypočítat a zafixovat sadu zpráv v prvních 10-ti krocích a zbytek ponechat jako volné proměnné. Pro kroky 22 – 76 máme 70 podmínek. Pro tři podmínky v krocích 22 – 23 použijeme techniku „*včasného zastavení*“, tedy provedeme výpočet po krok 23 a pak ověříme, zda jsou podmínky pro kroky 22 – 23 splněny. K tomu je potřeba 12 operací včetně modifikace zprávy pro opravení podmínek v krocích 16 – 21. Toto je ekvivaletní zhruba dvěma SHA-1 operacím. Celková složitost hledání kolize plné SHA-1 funkce je přibližně 2^{69} výpočtů.

5.3. Budoucnost funkce SHA-1

Všechny dosud provedené útoky na funkci SHA-1 jsou prvního řádu, kdy se hledají dvě různé vstupní zprávy, které vedou na stejnou hodnotu SHA-1 hashe. Efektivní algoritmus na hledání kolizí druhého řádu (pro danou zprávu a její hash se hledá druhá zpráva se stejnou hashí) pro funci SHA-1 zatím nebyl nalezen, tedy není snadné kompromitovat již podepsané certifikáty.

Uvedený útok čínských matematiků je pouze teoretický (dosud nebyly zveřejněny žádné konkrétní kolize pro plnou funkci SHA-1), avšak může již za několik málo let dospět k praktickému použití. Funkce SHA-1 je tedy považována za teoreticky prolomenou a již ne zcela bezpečnou. Pro použití v oblastech přísného utajení a striktní bezpečnosti je tato funkce již nevhodná.

V roce 2005 byl navržen hardwarový SHA-1 cracker (podobně jako u DESu) jež se skládá ze tříset výkonných počítačů, na každém počítači je 16 desek a na každé desce je 32 jader. Takto navržený cracker je schopen nalézt kolizi SHA-1 v rozumném čase a je možné si jej pořídit zhruba za milión dolarů.

V ČR bylo vyhláškou Ministerstva vnitra upraveno používání hashovací funkce SHA-1 pro účely zaručeného elektronického podpisu. Všichi kvalifikovaní poskytovatelé certifikačních služeb u nás již od 1.1.2010 vydávají certifikáty zaručeného elektronického podpisu s přednostním použitím hashovací funkce SHA-256 (je možné také použít další hashovací funkce z rodiny SHA-2, např. SHA-512) a šifrovacího algoritmu RSA s klíčem délky 2048 bitů.

Certifikáty zaručeného elektronického podpisu se vydávají na dobu jednoho roku, tedy pokud poslední certifikát používající hash SHA-1 byl vydán 31. 12. 2009, jeho platnost vypršela dne 31. 12. 2010 a nyní by již neměl existovat žádný platný certifikát využívající hashovací funci SHA-1. Nicméně i po vypršení platnosti certifikátu se nadále používají pro zpětné ověření (tj. pro ověření potřebujeme certifikát, který byl platný v době podpisu) platnosti elektronického podpisu a to několik let (zpravidla 5 – 10 let). Vzhledem k této skutečnosti, bychom se měli zabývat také otázkou, zda budou certifikáty používající hash SHA-1 bezpečné pro zpětné ověřování. Zajímá nás tedy možnost nalezení druhého vzoru (tj. pro danou vstupní zprávu a její SHA-1 hash hledáme jinou vztupní zprávu, která bude mít tentýž SHA-1 hash) pro daný certifikát v budoucnosti.

Prozatím nebyl zveřejněn žádný algoritmus, který by urychlil hledání druhého vzoru, tedy můžeme použít útok hrubou silou se složitostí 2^n, kde n je velikost výstupní hashe, což pro SHA-1 je $n = 160$. Ve skutečnosti je tato složitost závislá na délce vstupní zprávy (viz dokument vydaný NIST – National Institut of Standards and Technology [39]).

Skutečná složitost nalezení druhého vzoru dané hashovací funkce je $2^{(L-M)}$, kde L je velikost výstupní hashe v bitech a M je dána následujícím vztahem

$$M = \log_2\left(\frac{\text{velikost vstupní zprávy v bitech}}{\text{velikost zpracovávaného bloku v bitech}}\right).$$

Pro funkci SHA-1 dostaneme konkrétní hodnoty: délka vstupní zprávy může mít nejvýše 2^{64} bitů, velikost zpracovávaného bloku je 512 bitů ($= 2^9$) a $L = 160$. Pro největší možnou délku vstupní zprávy 2^{64} bitů získáme

$$M = \log_2\left(\frac{2^{64}}{2^9}\right) = (64-9) = 55\,.$$

A tedy složitost hledání druhého vzoru pro délku vstupní zprávy 2^{64} je $2^{160-55} = 2^{105}$. Pro krátké vstupní zprávy se složitost blíží hodnotě 2^{160}.

Uvažujme nejmenší možnou složitost nalezení druhého vzoru 2^{105}, tedy abychom našli druhý vzor musíme projít všechny možnosti 105-bitových řetězců a spočítat jejich SHA-1 hash. Algoritmus SHA-1 má 80 kroků a v každém kroku proběhne zhruba 25 operací (sčítání, odčítání, posuny a logické operace). Tedy počet potřebných operací k jednomu výpočtu SHA-1 hashe je roven zhruba 2000. Tedy celkem zhruba 2^{116} operací pro nalezení druhého vzoru.

Dneštní průměrný osobní počítač zvládne vypočítat 2 miliardy operací za sekundu, tj. zhruba 2^{31} operací za sekundu, což znamená, že hledání druhého vzoru mu bude trvat kolem $2^{116-31} = 2^{85}$ sekund, tj. 10^{20} let. Za posledních pět let se výkon počítačů zhruba zdvojnásobil, takže i kdyby za 10 let byl výkon počítačů desetinásobný, bude pořád efektivně nemožné nalézt druhý vzor SHA-1 hashe. Tento odhad je pouze zevrubný, záleží samozřejmě také na kryptoanalytickém vývoji algoritmů pro hledání druhého vzoru, případně na technologickém vývoji nových výpočetních strojů.

Uvedený odhad lze dobře aplikovat na nestrukturované dokumenty (zprávy, e-maily) jež mohou mít v podstatě libovolnou délku, přičemž podepisované certifikáty mají obvykle velikost zhruba několika málo kilobytů a tedy pro tyto hodnoty je složitost nalezení druhého vzoru rovna 2^{160} a jejich hledání ještě o hodně delší.

Můžeme tedy usuzovat, že pokud nebude nalezen algoritmus na výpočet druhého vzoru hashovacích funkcí, jež by toto hledání výrazně uspíšil, budou po následující desetiletí všechny podepsané zprávy (nestrukturované dokumenty) a také podepsané certifikáty s SHA-1 hashí bezpečné a nenapadnutelné.

6. Hashovací funkce SHA-2

Označení SHA-2 zahrnuje celou rodinu hashovacích funkcí. Nejznámější a v současné době nejrozšířenější je funkce SHA-256. Díky délce výsledné hashe a tedy i odolnosti vůči útokům hrubou silou je také funkce SHA-512 perspektivní pro blízkou budoucnost. Méně známé funkce patřící do rodiny SHA-2 jsou hashovací funkce SHA-224 a SHA-384, jež se od svých známějších variant liší v podstatě jen délkou výsledné hashe.

Funkce rodiny SHA-2 byly poprvé zveřejněny v roce 2001 v návrhu standardu FIPS PUB 180-2 a uznány jako standardy v roce 2002. Funkce SHA-224 byla definována o něco později, v roce 2004. Hashovací funkce rodiny SHA-2 začínají masivně nahrazovat funkci SHA-1, neboť SHA-1 je považována za teoreticky prolomenou.

Popíšeme zde pouze algoritmy SHA-256 a SHA-512. Funkce SHA-224 má totožný výpočet jako funkce SHA-256, liší se pouze v použití jiných iniciačních vektorů a výsledná hash se zkrátí na 224 bitů. Podobně funkce SHA-384 má stejný výpočet jako funkce SHA-512 s odlišnými iniciačními vektory a délkou hashe zkrácenou na 384 bitů.

Iniciační vektory používané pro funkce SHA-224 a SHA-384 jsou uvedeny v příloze v tabulkách Tab. SHA2-1 a Tab. SHA2-2.

6.1. Popis algoritmu SHA-256

Hashovací funkce SHA-256 vytváří hash dlouhou 256 bitů. Funkce zpracovává vstupní zprávu po jednotlivých blocích délky 512 bitů. Před samotným výpočtem hashe je nutné doplnit vstupní zprávu tak, aby její délka byla násobkem 512 bitů.

Předpokládejme, že máme zprávu M, jejíž délka je l bitů. Použitím následujícího popisu doplníme zprávu M:

1. Připojíme bit „1“ na konec zprávy, nyní je délka zprávy $l+1$ bitů.
2. Připojíme tolik bitů „0“, aby výsledná délka byla rovna $512k+448$, tedy 64 bitů zůstane nevyplněno.
3. Do zbývajících 64 bitů se zapíše délka původní zprávy, což omezuje délku vstupní zprávy na $2^{64}-1$ bitů.

Takto upravený vstupní řetězec má nyní délku násobku 512 bitů. Vstupní zpráva M se následně rozdělí na N stejných bloků $M_1, M_2, ..., M_N$, každý délky 512 bitů. Hashovací funkce SHA-256 zpracovává postupně jednotlivé bloky, každý blok v jedné rundě. Každá runda obsahuje 64 jednotlivých kroků, označme tyto kroky $t = 0,...,63$.

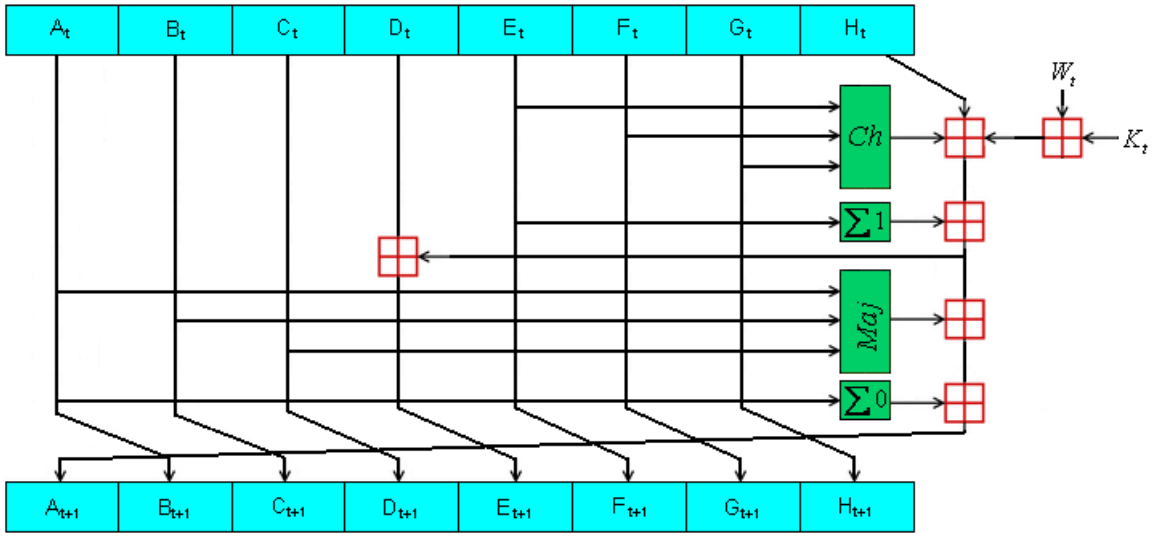
Délka výstupní hashe funkce SHA-256 je 256 bitů, tedy kompresní funkce SHA-256 pracuje s kontextem o délce 256 bitů. Kontext je tvořen osmi slovy, jež označíme A, B, C, D, E, F, G a H, přičemž každé slovo má délku 32 bitů. Na začátku výpočtu jsou tato slova naplněna inicializačními hodnotami. Tyto hodnoty jsou uvedeny v tabulce Tab. 13 v little endian uspořádání (nejméně významný byte je první z leva). Uvedené počáteční nastavení osmi registrů se použije pro první rundu při počítání SHA-256 hashe. Po dokončení celého výpočtu lze nalézt výslednou SHA-256 hash právě v těchto osmi registrech.

Tab. 13: Inicializační hodnoty slov funkce SHA-256.

Slovo	Hexadecimální hodnota
A_0	6a09e667
B_0	bb67ae85
C_0	3c6ef372
D_0	a54ff53a
E_0	510e527f
F_0	9b05688c
G_0	1f83d9ab
H_0	5be0cd19

Popišme nyní podrobně operace, které probíhají v kompresní funkci SHA-256. Obrázek Obr. 8 představuje průběh jednoho kroku t kompresní funkce algoritmu. Všechny operace kompresní funkce probíhají v rámci slov (tj. 32-bitových bloků), přičemž symbol ⊞ znamená sčítání slov (tj. sčítání modulo 2^{32}). Poznamenejme, že kompresní funkce, má stejný průběh pro všechny algoritmy rodiny SHA-2, liší se pouze ve struktuře použitých nelineárních funkcí, rozdílných konstantách a inicializačních vektorech.

Obr. 8: Detail kompresní funkce algoritmu SHA-2



V průběhu jedné rundy dochází ke zpracování právě jednoho bloku vstupní zprávy. Každá runda má 64 kroků ($t = 0,...,63$). Algoritmus SHA-256 používá celkem 6 nelineárních funkcí, kde každá zpracovává 32-bitová slova a výsledkem každé funkce je nové 32-bitové slovo. Dále je použita sekvence 64 konstant, 32-bitových slov K_t, $t = 0,...,63$, jež reprezentují prvních 32 bitů desetinné části třetí odmocniny prvních 64 prvočísel. Konstanty K_t jsou uvedeny v následující tabulce Tab. 14 v hexadecimálním vyjádření.

Tab. 14: Soubor aditivních konstant pro funkci SHA-256

t	K_t	t	K_t	t	K_t	t	K_t
0	$428a2f98_{16}$	16	$e49b69c1_{16}$	32	$27b70a85_{16}$	48	$19a4c116_{16}$
1	71374491_{16}	17	$efbe4786_{16}$	33	$2e1b2138_{16}$	49	$1e376c08_{16}$
2	$b5c0fbcf_{16}$	18	$0fc19dc6_{16}$	34	$4d2c6dfc_{16}$	50	$2748774c_{16}$
3	$e9b5dba5_{16}$	19	$240ca1cc_{16}$	35	$53380d13_{16}$	51	$34b0bcb5_{16}$
4	$3956c25b_{16}$	20	$2de92c6f_{16}$	36	$650a7354_{16}$	52	$391c0cb3_{16}$
5	$59f111f1_{16}$	21	$4a7484aa_{16}$	37	$766a0abb_{16}$	53	$4ed8aa4a_{16}$
6	$923f82a4_{16}$	22	$5cb0a9dc_{16}$	38	$81c2c92e_{16}$	54	$5b9cca4f_{16}$
7	$ab1c5ed5_{16}$	23	$76f988da_{16}$	39	$92722c85_{16}$	55	$682e6ff3_{16}$
8	$d807aa98_{16}$	24	$983e5152_{16}$	40	$a2bfe8a1_{16}$	56	$748f82ee_{16}$
9	$12835b01_{16}$	25	$a831c66d_{16}$	41	$a81a664b_{16}$	57	$78a5636f_{16}$
10	$243185be_{16}$	26	$b00327c8_{16}$	42	$c24b8b70_{16}$	58	$84c87814_{16}$
11	$550c7dc3_{16}$	27	$bf597fc7_{16}$	43	$c76c51a3_{16}$	59	$8cc70208_{16}$
12	$72be5d74_{16}$	28	$c6e00bf3_{16}$	44	$d192e819_{16}$	60	$90befffa_{16}$
13	$80deb1fe_{16}$	29	$d5a79147_{16}$	45	$d6990624_{16}$	61	$a4506ceb_{16}$
14	$9bdc06a7_{16}$	30	$06ca6351_{16}$	46	$f40e3585_{16}$	62	$bef9a3f7_{16}$
15	$c19bf174_{16}$	31	14292967_{16}	47	$106aa070_{16}$	63	$c67178f2_{16}$

Nelineární funkce používané v algoritmu SHA-256 jsou definovány následujícím způsobem:

$$Ch(X,Y,Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$
$$Maj(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

kde proměnné X, Y, Z reprezentují 32-bitové hodnoty příslušných registrů. Všechny operace se provádějí na bitové úrovni, přičemž $\overline{X}$ je bitový doplněk (operace NOT) a $\oplus$ značí bitový součet (operace XOR).

$$\sum 0(X) = ROTR(X,2) \oplus ROTR(X,13) \oplus ROTR(X,22)$$
$$\sum 1(X) = ROTR(X,6) \oplus ROTR(X,11) \oplus ROTR(X,25)$$
$$\sigma 0(X) = ROTR(X,7) \oplus ROTR(X,18) \oplus SHR(X,3)$$
$$\sigma 1(X) = ROTR(X,17) \oplus ROTR(X,19) \oplus SHR(X,10)$$

kde proměnná X reprezentuje 32-bitové slovo, operace $ROTR(X,A)$ značí pravou bitovou rotaci slova X o A pozic a operace $SHR(X,A)$ značí bitový posun slova X o A pozic. Poslední dvě uvedené funkce $\sigma 0$ a $\sigma 1$ se použijí pro expanzi 16-ti slov $(m_0,...,m_{15})$ 512-ti bitového bloku vstupní zprávy.

$$W_t = \begin{cases} m_t & \text{pro} \quad 0 \le t \le 15, \\ \sigma 1(W_{t-2}) + W_{t-7} + \sigma 0(W_{t-15}) + W_{t-16} & \text{pro} \quad 16 \le t \le 63 \end{cases}$$

Znaménko „+“ v tomto případě značí sčítání slov modulo 2^{32}. Oproti funkci SHA-1 je způsob expandování slov vstupní zprávy výrazně složitější.

Jednotlivé kroky výpočtu jsou názorně vidět na obrázku Obr. 8, uveďme ještě pro úplnost matematické vyjádření jednotlivých kroků ($t = 0,...,63$). V tomto případě znaménko „+“ je také sčítání modulo 2^{32}.

$$A_{t+1} = H_t + \sum 1(E_t) + Ch(E_t, F_t, G_t) + W_t + K_t + \sum 0(A_t) + Maj(A_t, B_t, C_t)$$
$$B_{t+1} = A_t$$
$$C_{t+1} = B_t$$
$$D_{t+1} = C_t$$
$$E_{t+1} = D_t + H_t + \sum 1(E_t) + Ch(E_t, F_t, G_t) + W_t + K_t$$
$$F_{t+1} = E_t$$
$$G_{t+1} = F_t$$
$$H_{t+1} = G_t$$

Po zpracování celého jednoho vstupního bloku zprávy, tedy po dokončení jedné rundy, k výsledným hodnotám v jednotlivých registrech přičteme ještě iniciační hodnoty těchto registrů a tím získáme hodnotu mezihashe, jehož jednotlivá slova použijeme jako iniciační hodnoty pro další rundu.

$$H = \left(A_0 + A_{63}, B_0 + B_{63}, C_0 + C_{63}, D_0 + D_{63}, E_0 + E_{63}, F_0 + F_{63}, G_0 + G_{63}, H_0 + H_{63}\right)$$

Po zpracování celé vstupní zprávy lze stejným způsobem, tedy přičtením iniciačních hodnot registrů pro poslední rundu k hodnotám registru po doběhnutí této rundy, získat výslednou hodnotu SHA-256 hashe.

6.2. Popis algoritmu SHA-512

Hashovací funkce SHA-512 je v podstatě variantou funkce SHA-256, tedy i postup výpočtu hashe je velice podobný. Funkce SHA-512 zpracovává vstupní zprávu po jednotlivých blocích délky 1024 bitů a vytváří hash dlouhou 512 bitů. Před samotným výpočtem hashe je pořeba upravit vstupní zprávu tak, aby její délka byla násobkem 1024 bitů.

Předpokládejme, že máme zprávu M, jejíž délka je l bitů. Použitím následujícího popisu doplníme zprávu M:

1. Připojíme bit „1“ na konec zprávy, nyní je délka zprávy $l+1$ bitů.
2. Připojíme tolik bitů „0“, aby výsledná délka byla rovna $1024k + 896$, tedy 128 bitů zůstane nevyplněno.
3. Do zbývajících 128 bitů se zapíše délka původní zprávy, což omezuje délku vstupní zprávy na $2^{128} - 1$ bitů.

Takto upravený vstupní řetězec má nyní délku násobku 1024 bitů. Vstupní zpráva M se následně rozdělí na N stejných bloků $M_1, M_2, ..., M_N$, každý délky 1024 bitů. Hashovací funkce SHA-512 zpracovává postupně jednotlivé bloky, každý blok v jedné rundě. Každá runda obsahuje 80 jednotlivých kroků, označme tyto kroky $t = 0,...,79$.

Délka výstupní hashe funkce SHA-512 je 512 bitů, tedy kompresní funkce SHA-512 pracuje s kontextem o délce 512 bitů. Kontext je tvořen osmi slovy, jež označíme A, B, C, D, E, F, G a H, přičemž každé slovo má délku 64 bitů. Na začátku výpočtu jsou

tato slova naplněna inicializačními hodnotami. Tyto hodnoty jsou uvedeny v tabulce Tab. 15 v little endian uspořádání (nejméně významný byte je první z leva). Uvedené počáteční nastavení osmi registrů se použije pro první rundu při počítání SHA-512 hashe. Po dokončení celého výpočtu lze nalézt výslednou SHA-512 hash právě v těchto osmi registrech.

Tab. 15: Inicializační hodnoty slov funkce SHA-512.

Slovo	Hexadecimální hodnota
A_0	6a09e667f3bcc908
B_0	bb67ae8584caa73b
C_0	3c6ef372fe94f82b
D_0	a54ff53a5f1d36f1
E_0	510e527fade682d1
F_0	9b05688c2b3e6c1f
G_0	1f83d9abfb41bd6b
H_0	5be0cd19137e2179

Algoritmus SHA-512 je téměř stejný jako algoritmus SHA-256, liší se pouze počtem kroků v jednotlivé rundě (80 místo 64), používá odlišné konstanty K_t a trochu odlišné nelineární funkce. Průběh jednoho kroku t kompresní funkce algoritmu SHA-512 je totožný s průběhem jednoho kroku kompresní funkce algoritmu SHA-256 (viz. obrázek Obr. 8). Konstanty K_t pro funkci SHA-512 jsou uvedeny v příloze v tabulce Tab. SHA2-3 v hexadecimálním vyjádření.

Nelineární funkce $Ch(X,Y,Z)$ a $Maj(X,Y,Z)$ jsou stejné jako v algoritmu SHA-256. Expanze 16-ti 64-bitových slov každého bloku vstupní zprávy také probíhá stejně jako v algoritmu SHA-256, ve výpočtu se pokračuje až do kroku $t = 79$. Liší se pouze následující čtyři nelineární funkce.

$$\sum 0(X) = ROTR(X,28) \oplus ROTR(X,34) \oplus ROTR(X,39)$$

$$\sum 1(X) = ROTR(X,14) \oplus ROTR(X,18) \oplus ROTR(X,41)$$

$$\sigma 0(X) = ROTR(X,1) \oplus ROTR(X,8) \oplus SHR(X,7)$$

$$\sigma 1(X) = ROTR(X,19) \oplus ROTR(X,61) \oplus SHR(X,6)$$

Opět stejně jako v algoritmu SHA-256, po zpracování celého jednoho vstupního bloku zprávy, tedy po dokončení jedné rundy, k výsledným hodnotám v jednotlivých registrech přičteme ještě iniciační hodnoty těchto registrů a tím získáme hodnotu mezihashe, jehož jednotlivá slova použijeme jako iniciační hodnoty pro další rundu.

Po zpracování celé vstupní zprávy lze stejným způsobem, tedy přičtením iniciačních hodnot registrů pro poslední rundu k hodnotám registru po doběhnutí této rundy, získat výslednou hodnotu SHA-512 hashe.

6.3. Vývoj útoků na funkci SHA-2

V posledních letech byla funkce SHA-256 v hledáčku mnoha kryptoanalytiků. Po praktickém předvedení nalezení kolize funkce MD5 a po zveřejnění teoretického postupu prolomení funkce SHA-1, z nejpoužívajnějších hashovacích funkcí zbyla pouze rodina funkcí SHA-2 jež je prozatím považována za bezpečnou. Z vývoje útoků na tuto funkci je možné se domnívat, že je otázka krátkého času, kdy bude potřeba i tuto hashovací funkci vyměnit za novou a opravdu bezpečnou.

Bezpečností funkce SHA-256 se jako první zabývali Takanori Isobe a Kyoji Shibutani v [33], kde zveřejnili svůj teoretický postup pro nalezení kolize funkce SHA-256 redukovanou na 24 kroků. Tento útok byl později vylepšen Aoki [34] na 43 kroků. Oba útoky jsou jen o málo rychlejší než generický útok, který má složitost kolem 2^{256}.

Mendel a kol. [35] zkoumali bezpečnost funkce SHA-2 s přihlédnutím k dosavadním útokům a prezentovali nový postup hledání kolizí pro SHA-256 redukovanou na 19 kroků. Tento postup byl poté několikrát vylepšen mnoha vědci. Především Nikolic a Biryukov [36] vylepšili tuto techniku pro SHA-256 redukovanou na 23 kroků. Tyto útoky mají již prakticky použitelnou složitost a navíc byly prokázány nenáhodné vlastnosti SHA-256 až do kroku 31. Prozatím nejlepšího výsledku dosáhli Lamberger a Mendel [37], jejichž postup je teoreticky schopen nalézt kolizi SHA-256 redukovanou na 46 kroků s prakticky použitelnou složitostí.

Z hlediska certifikátů zaručeného elektronického podpisu má také hashovací funkce SHA-2 svou omezenou životnost. Zveřejňování vhodných kryptografických algoritmů a jejich parametrů zajišťuje Ministerstvo vnitra, jež se řídí dokumenty vydávanými Evropským ústavem pro telekomunikační normy (ETSI). Jedním z nejdůležitějších dokumentů je ETSI TS 102 176-1 V2.0.0 (viz. [40]) tzv. “ALGO Paper”, který mimo jiné určuje dobu po kterou jsou doporučené algoritmy bezpečné pro používání v oblasti kvalifikovaných certifikátů.

Tab. 16: Předpokládaná doba použitelnosti hashovacích funkcí, kde popisek jednotlivých sloupců udává použitelnost do konce uvedeného roku.

Hashovací funkce	2007	2009	2012	2016 (výhledově)
SHA-1	použitelné	neznámé	nepoužitelné	nepoužitelné
SHA-224	použitelné	použitelné	použitelné	neznámé
SHA-256	použitelné	použitelné	použitelné	neznámé
SHA-384	použitelné	použitelné	použitelné	použitelné
SHA-512	použitelné	použitelné	použitelné	použitelné

Poznámka: Jednotlivé položky v tabulce Tab. 16 značí
použitelné = algoritmus může být považován za bezpečný v dané době,
neznámé = bezpečnost algoritmu v dané době je neznámá, může být bezpečný za určitých okolností,
nepoužitelné = algoritmus nemůže být v dané době považován za bezpečný pro používání ve spojitosti s elektronickým podpisem.

Poznámka: Předpokládá se, že uvedené hashovací funkce jsou rezistentní vůči nalezení vzoru a vůči nalezení druhého vzoru po delší časové období (v řádu desítek let).

Z tabulky Tab. 16 můžeme vidět, že hashovací funkce SHA-256 je považována za bezpečnou do konce roku 2012 a funkce SHA-512 výhledově do konce roku 2016. Vzhledem k technickému pokroku a vývoji útoků na hashovací funkce můžeme usoudit, že praktické používání funkce SHA-256 pro vydávání certifikátů zaručeného elektronického podpisu bude ukončeno nejpozději během roku 2013 a nahrazeno bezpečnější hashovací funkcí SHA-512. Ukončení používání hashovací funkce SHA-512 se dá předpokládat kolem roku 2015, kdy bude nutné mít již připravenou novou bezpečnější variantu hashovací funkce, očekávanou SHA-3.

7. Praktická implementace hledání kolizí MD5

Jako součást diplomové práce přikládám CD obsahující vlastní program pro výpočet MD5 a pro hledání kolizí MD5 založený na postupu V. Klímy (viz [17]).

Program je napsán v programovacím jazyce C#. Oproti zveřejněnému programu Vlastimila Klímy v jazyce C se snažím o použití obecnějších funkcí a zpřehlednění zdrojového kódu.

Výpočet kolizí pomocí mého programu je oproti programu V. Klímy přibližně třináctkrát pomalejší (viz kapitola 7.3 Porovnání rychlosti). To je především proto, že oproti programu V. Klímy, který při výpočtu upravuje celá 32-bitová slova najednou pomocí předpřipravených konstant, používám ve svém programu z důvodu jednoduchosti a přehlednosti zdrojového kódu obecnější funkce, které upravují postupně jednotlivé bity podle zadaných podmínek.

Výhodou mého programu je možnost velmi snadných úprav v postačujících podmínkách a tunelech (lze snadno vyměnit sadu postačujících podmínek nebo doplnit další tunely bez většího zásahu do zdrojového kódu).

7.1. Uživatelská dokumentace

Uživatelská dokumentace popisuje jednotlivé funkce programu GeneratorKoliziMD5. Program se spouští spustitelným souborem GeneratorKoliziMD5.exe a vyžaduje nainstalovaný .NET verze alespoň 2.0.

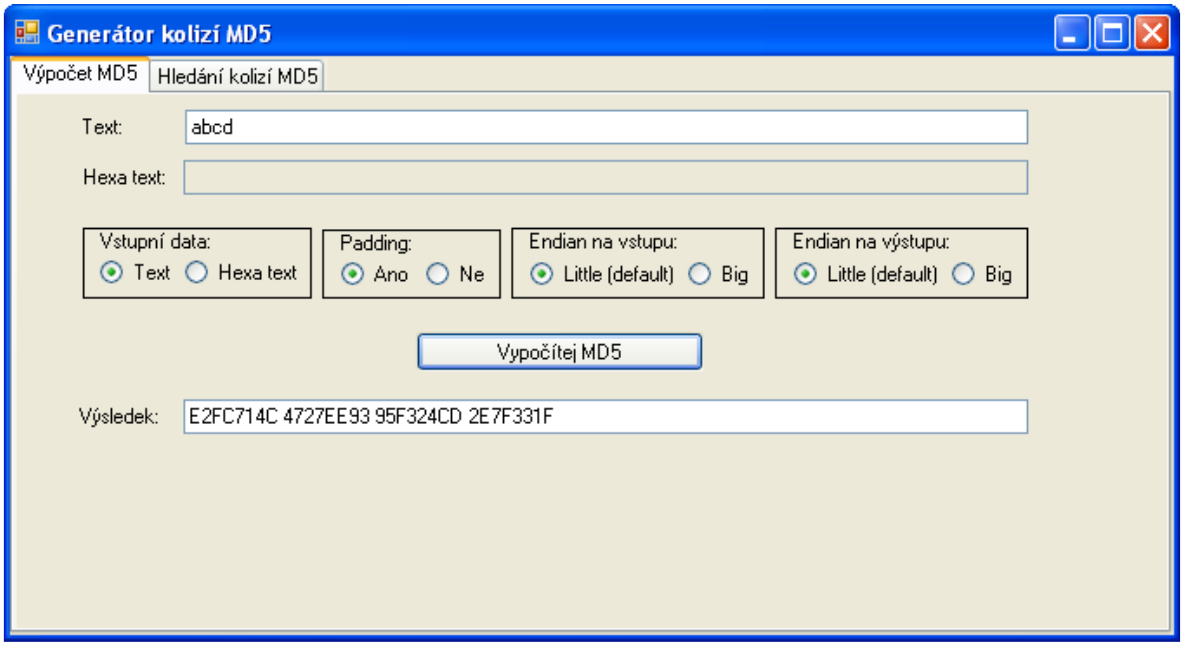
7.1.1. Výpočet MD5

Program umožňuje výpočet hashe MD5 dle vstupu v podobě textu nebo hexadecimálního čísla. Po zadání vstupu stiskněte tlačítko „Vypočítej MD5“, výsledná hash se zapíše do políčka „Výsledek“ v podobě hexadecimálního zápisu čtyř 32-bitových slov.

Program umožňuje nastavení následujících parametrů:

- **Vstupní data** – umožňuje vybrat zadání vstupu buď jako text nebo jako hexadecimální číslo.
- **Padding** – umožňuje zapnutí nebo vypnutí paddingu (doplnění délky vstupu na násobek 512 bitů). Při vypnutém paddingu program kontroluje, zda délka vstupu odpovídá násobku 512 bitů.
- **Endian na vstupu** – udává, jakým způsobem má program při čtení vstupu převádět posloupnost bytů na 32-bitová slova. Možnosti jsou *little endian* (little end first) nebo *big endian* (big end first).
- **Endian na výstupu** – udává, jakým způsobem má program při zápisu výstupu převádět 32-bitová slova na posloupnost bytů. Možnosti jsou opět *little endian* nebo *big endian*.

Obr. 9: Formulář pro výpočet MD5 hashe.



Příklad nastavení parametrů pro výpočet MD5 dle RFC 1321 [5]:

- Vstupní data = Text;
- Padding = Ano;
- Endian na vstupu = Little (default);
- Endian na výstupu = Little (default);

(*Příklad*: MD5 hash prázdného řetězce je je roven hodnotě `d41d8cd9 8f00b204 e9800998 ecf8427e`).

Příklad nastavení parametrů pro výpočet hodnoty *H* dle tabulky Tab. 5:

- Vstupní data = Hexa text;
- Padding = Ne;
- Endian na vstupu = Big;
- Endian na výstupu = Big;

(výsledný MD5 hash *H* dle tabulky Tab. 5 pro zprávy M_0+M_1 a M_0'+M_1' je stejný a je roven hodnotě `9603161f a30f9dbf 9f65ffbc f41fc7ef`).

Příklad nastavení parametrů pro výpočet hodnoty *H** dle tabulky Tab. 5:

- Vstupní data = Hexa text;
- Padding = Ano;
- Endian na vstupu = Big;
- Endian na výstupu = Little;

(výsledný MD5 hash *H** dle tabulky Tab. 5 pro zprávy M_0+M_1 a M_0'+M_1' je stejný a je roven hodnotě `a4c0d35c 95a63a80 5915367d cfe6b751`).

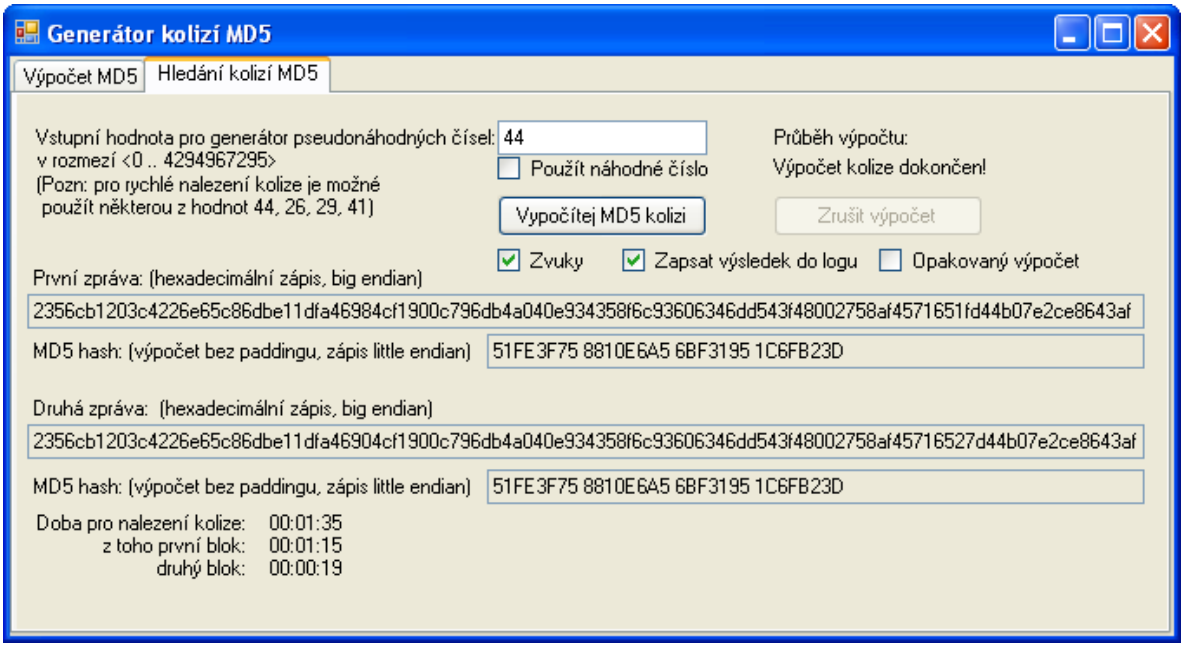
7.1.2. Hledání kolizí MD5

Program umožňuje hledání kolizí hashovací funkce MD5. Po stisknutí tlačítka „Vypočítej MD5 kolizi“ se spustí výpočet hledání kolize. Výpočet je možné přerušit stisknutím tlačítka „Zrušit výpočet“. Po dokončení výpočtu se zapíšou výsledné zprávy do políček „První zpráva“ a „Druhá zpráva“, také se vypočtou jejich MD5 hashe. Program také zobrazí dobu, jak dlouho kolizi hledal.

Program umožňuje nastavení následujících parametrů:

- Vstupní hodnota pro generátor pseudonáhodných čísel – inicializuje generátor pseudonáhodných čísel. Protože použitý generátor pseudonáhodných čísel má deterministické chování, bude při opakovaném použití stejného čísla proveden stejný výpočet trvající přibližně stejnou dobu.
- Použít náhodné číslo – inicializuje generátor pseudonáhodných čísel náhodnou hodnotou v rozmezí <0..4294967295>, což je rozsah možných hodnot 32-bitového slova.
- Zvuky – po nalezení prvního bloku a po dokončení kolize program pípne.
- Zapsat výsledek do logu – vytvoří soubor s názvem md5_kolize_X=0x[číslo].log, kde [číslo] je iniciační hodnota pseudonáhodného generátoru v hexadecimálním zápisu, a zapíše do něj průběh výpočtu a výsledek.
- Opakovaný výpočet – pokud je toto políčko zaškrtnuto, provede program po nalezení kolize zvýšení vstupní hodnoty o 1 (resp. v případě, že vstupní hodnota již je 4294967295, nastaví vstupní hodnotu na 0) a začne výpočet další kolize.

Obr. 10: Formulář pro hledání kolizí funkce MD5.



Příklad nastavení parametrů: pro rychlé nalezení kolize je možné použít některou z hodnot 44, 26, 29 nebo 41. Tyto hodnoty byly zjištěny jako nejrychlejší při spuštění programu na opakovaný výpočet pro hodnoty od 1 do 50 (viz kapitola 7.3 Porovnání rychlosti).

7.2. Programátorská dokumentace

7.2.1. Výpočet MD5

Výpočet MD5 je zahájen stisknutím ovládacího prvku *vypocetMD5Button*. Ve funkci *vypocetMD5Button_Click* postupně provádíme následující kroky:

- Uložíme vstupní data do proměnné *mujVstup* typu string. Zdroj dat určíme dle vybraného typu vstupu (text/hexa text).
- V případě paddingu zavoláme funkci *md5padding* a uložíme data do proměnné *mujVstupPadded* typu string.
- V případě výpočtu bez paddingu uložíme do proměnné *mujVstupPadded* přímo hodnotu *mujVstup*. Při výpočtu bez paddingu také kontrolujeme, zda délka zprávy bez paddingu je násobkem 64 bytů.
- Zavoláme funkci *MD5* s parametry *mujVstupPadded*, *endian na vstupu* a *endian na výstupu*. Tato funkce vrátí typ string, který zapíšeme do výstupního políčka *vystup*.

Funkce pro výpočet MD5 hashe je napsána v samostatné třídě MD5Calc, kterou zde nyní stručně popíšeme.

Popis třídy MD5Calc

Třída MD5Calc obsahuje následující proměnné:

- *a0, b0, c0, d0* typu uint (uint=32-bitové slovo) – iniciační hodnoty algoritmu MD5;
- *a, b, c, d* typu uint – průběžné hodnoty registrů odpovídající popisu dle kapitoly 4.1;
- *R, K a W* – pole konstant pro výpočet MD5 dle popisu v kapitole 4.1;
- *blok* – pole 16 proměnných typu uint (tj. celková délka 512 bitů) – slouží pro uložení jednoho bloku vstupních dat.

Třída MD5Calc obsahuje následující pomocné funkce:

- *MD5padding* – jako parametr dostane string a vrátí upravený string obsahující padding.
- *MD5blokInit* – jako parametr dostane string a číslo udávající pozici v něm. Projde hodnoty na pozicích *pozice..pozice+63* ve stringu a uloží jejich číselnou hodnotu do proměnné *blok*. Interpretace vstupních dat záleží na parametru *endian*. Tato funkce slouží k načtení vstupních dat do proměnné *blok*.
- *f* – tato funkce vypočte hodnoty funkcí F, G, H, I tak, jak je popsáno v kapitole 4.1. To, která funkce F, G, H nebo I se má použít, se řídí parametrem *step* udávajícím aktuální krok výpočtu.
- *MD5init* – tato funkce inicializuje registry *a, b, c, d* hodnotami *a0, b0, c0, d0*.
- *vratRegistry* – vrátí string obsahující hexadecimální zápis aktuální hodnoty registrů *a, b, c, d*. Interpretace dat záleží na parametru *endian*, pro otáčení pořadí bytů ve slově používá pomocnou funkci *prevratByty*.

Pro výpočet algoritmu MD5 poté používáme následující funkce:

- *MD5step* – provede jeden krok výpočtu MD5 přesně dle obrázku Obr. 4:
  - k hodnotě *a* přičte hodnotu funkce *f(b,c,d)*;
  - dále přičte příslušnou hodnotu vstupu: *blok[W[step]]*;
  - dále přičte příslušnou hodnotu konstanty *K[step]*;
  - provede bitovou rotaci doleva o *R[step]* bitů;
  - prohodí hodnoty registrů a,b,c,d.

- *MD5round* – tato funkce provede zpracování jednoho bloku vstupní zprávy. Pomocí funkce *MD5Init* nastaví hodnotu proměnné *blok*, poté provede 64 kroků *MD5step* a nakonec přičte k registrům *a*, *b*, *c*, *d* jejich původní hodnoty.
- *MD5* – tato funkce spočítá MD5 hash pro zadaný vstup. Nejprve pomocí funkce *MD5Init* inicializuje počáteční hodnoty *a*, *b*, *c*, *d* a poté dle délky zprávy volá potřebný počet rund *MD5round*. Na konci vrátí hodnotu registrů pomocí funkce *vratRegistry*.

Tyto funkce jsem se snažila psát pokud možno co nejpřehlednější, aby odpovídaly popisu uvedeném v kapitole 4.1. Jako ukázku uvádím zdrojový kód těchto tří funkcí přímo zde:

```
public void MD5step(int step)
//provede jeden krok MD5, step nabyva hodnot od 0 do 63
{
    uint x;
    a = a + f(b, c, d, step) + blok[W[step]] + K[step];
    a = (a << R[step]) | (a >> (32 - R[step]));
    a = a + b;
    x = a;
    a = d;
    d = c;
    c = b;
    b = x;
    return;
}

public void MD5round(string s, int cislobloku, bool endian)
//provede jednu rundu MD5. Parametr cislobloku je cislovan od nuly.
{
    //zapamatujeme pocatecni hodnoty registru
    uint ainit = this.a;
    uint binit = this.b;
    uint cinit = this.c;
    uint dinit = this.d;

    //naplnime promennou blok spravnou casti vstupu s
    //tj. pro 1.blok pouzijeme byty 0..63, pro 2.blok 64..127, atd.
    MD5blokInit(s, cislobloku * 64, endian);

    //provedeme 64 kroku
    for (int i = 0; i < 64; i++)
    {
        MD5step(i);
    }

    //pricteme pocatecni hodnoty registru
    a = a + ainit;
    b = b + binit;
    c = c + cinit;
    d = d + dinit;
    return;
}
```

```
public string MD5(string input, bool inputEndian, bool outputEndian)
{
    //inicializujeme a,b,c,d hodnotami a0,b0,c0,d0
    MD5init();

    //spocitame pocet rund dle delky vstupu. 1 runda = 64 bytu (512 bytu)
    int pocetrund = (input.Length / 64);

    //provedeme potrebny pocet rund
    for (int i = 0; i < pocetrund; i++)
    {
        MD5round(input, i, inputEndian);
    }

    //vratime hodnoty registru a,b,c,d udavajici vyslednou MD5 hash
    return vratRegistry(outputEndian);
}
```

7.2.2. Hledání kolizí MD5

Algoritmus hledání kolizí MD5 je založen na postupu V. Klímy [17]. Stejně jako V. Klíma používáme metodu modifikace zpráv s použitím postačujících podmínek dle [12] s použitím tunelů Q4, Q9, Q10, Q13, Q14 a Q20.

Oproti programu V. Klímy jsem se snažila o definici obecných funkcí pro jednotlivé kroky, což mi umožnilo přehlednější zápis hlavního algoritmu. Bohužel, vzhledem k tomu, že můj postup je založen na tom, že kontroluje a modifikuje jednotlivé bity zvlášť dle parametrizovatelných podmínek, je oproti programu V. Klímy, který efektivně zpracovává celá 32-bitová slova najednou pomocí předpřipravených konstant, přibližně 13x pomalejší. Přínos mého programu tedy lze spatřovat především v didaktické rovině – dle mého názoru je můj kód snadněji pochopitelný a umožňuje tedy snadnější pochopení metody tunelování.

Nyní zde postupně popíšeme použité pomocné třídy a funkce a nakonec pomocí nich sestavíme hlavní algoritmus.

Základní principy

Cílem je nalezení dvou 512-bitových bloků splňujících požadované podmínky. Pomocí metody modifikace zpráv (viz kapitola 4.2) hledáme postupně pro oba bloky hodnoty registrů Q1..Q64, které budou tyto podmínky splňovat.

Pomocné třídy – složky Utils a Variables

Ve složce Utils jsou umístěny následující třídy:

- třída *MyUint* – toto je vlastní třída určená pro uložení hodnot typu uint (32-bitové slovo). Obsahuje jedinou proměnnou, která se jmenuje *value* a je typu *uint*. Důvodem pro existenci této třídy jsou její následující funkce umožňující kontrolu nebo nastavení konkrétního bitu ve slově:
    - funkce *getBit(i)* – vrátí hodnotu i-tého bitu (počítáno odleva od nuly);
    - funkce *setBit(i,bitValue)* – nastaví hodnotu i-tého bitu (počítáno odleva od nuly) na hodnotu *bitValue*. Tato funkce využívá pomocných funkcí *setBitAsZero* a *setBitAsOne*;

- nastavování a získávání hodnot typu *uint* je možné pomocí funkcí *getValue* a *setValue*. Třída *MyUint* také obsahuje implicitní operátory umožňující automatickou konverzi typu *MyUint* na typ *uint* a zpět.

- třída *Rng* – tato třída je určena pro generování pseudonáhodných čísel. Obsahuje funkci *rng()*, která je převzata z programu V. Klímy.
- třída *Logger* je jednoduchá pomocná třída určená pro zápis textových informací do souboru.

Ve složce Variables jsou umístěny následující třídy:

- třída *MyVariable* obsahující dvě hodnoty – klíč (key) typu char a hodnotu (value) typu boolean. Tato třída umožňuje uložení hodnoty jedné proměnné. Do klíče se uloží její název a do hodnoty její hodnota.
- třída *MyVariables* obsahuje seznam *listOfVariables* typu *MyVariable*. Tato třída obsahuje následující funkce:
  - *variableExists(x)* – pokud je v seznamu *listOfVariables* obsažen prvek s klíčem *x*, vrátí true, jinak vrátí false.
  - *returnVariable(x)* – najde v seznamu *listOfVariables* prvek s klíčem *x* a vrátí jeho hodnotu (true/false).
  - *setVariable(x,value)* – najde v seznamu *listOfVariables* prvek s klíčem *x* a nastaví jeho hodnotu dle parametru *value*. Pokud prvek s tímto klíčem v seznamu neexistuje, vytvoří nový záznam a přidá ho do seznamu.

Třídu *MyVariables* budeme využívat pro nastavování a kontrolu hodnot „I, J, A“ při splňování a kontrole podmínek.

Kontext

Ve složce Context jsou umístěny následující třídy:

- třída *RegistersOneBlock* – jednoduchá pomocná třída umožňující přehledné uložení hodnot *Q[-3]..Q[68]*, tj registrů pro jeden blok výpočtu.
  - Hodnoty Q[-3]..Q[0] jsou určeny pro počáteční hodnoty registru.
  - Hodnoty Q[1]..Q[64] slouží k uložení průběžných hodnot registru.
  - Hodnoty Q[65]..Q[68] používáme k uložení mezihodnoty hashe (IHV).
- třída *OneBlockContext* – obsahuje veškeré potřebné hodnoty a funkce vztahující se k jednomu bloku MD5. Tato třída obsahuje následující proměnné:
  - *R, K, W* sloužící jako konstanty pro výpočet MD5 stejně jako v třídě *MD5Calc*.
  - *Q* třídy *RegistersOneBlock*, ve které jsou uloženy hodnoty Q[-3]..Q[68].
  - *x* typu uint[16] udávající hodnotu jednoho bloku vstupní zprávy.
  - *variables* typu *MyVariables* obsahující aktuální hodnoty proměnných „I, J, A“.
  - *conditions* typu *ListOfConditionsForBlock* udávající seznam podmínek kladených na jednotlivé kroky *Q[-3]..Q[68]*. Popis třídy *ListOfConditionsForBlock* je uveden níže.

- *tunnels* typu *Tunnels* obsahující aktuální hodnotu tunelů pro tento blok. Popis třídy *Tunnels* je uveden níže.

Třída OneBlockContext obsahuje následující funkce:

- *setRegisterAccordingConditions(step)* – nastaví náhodnou hodnotu registru *Q[step]* a zavolá funkci *conditions.setOneStepConditions(step)*, která upraví hodnotu *Q[step]* tak, aby splňovala požadované podmínky pro tento krok. Zdrojový kód funkce je intuitivní:

```
public void setRegisterAccordingConditions(int step)
{
    Q.get(step).setValue(Rng.rng());
    conditions.setOneStepConditions(step, this);
}
```

- *checkRegisterAgainstConditions(step)* – zavolá funkci *conditions.checkOneStepConditions(step)*, která ověří, zda hodnota registru *Q[step]* splňuje požadované podmínky pro tento krok. Pokud ano, vrátí *true*, jinak vrátí *false*. Zdrojový kód této funkce je opět velmi jednoduchý:

```
public bool checkRegisterAgainstConditions(int step)
{
    return conditions.checkOneStepConditions(step, this);
    //pokud byly vsechny podminky splneny, vratime true,
    //jinak vratime false

  }
```

- *md5step(step)* – tato funkce provede jeden krok MD5 – na základě hodnot *Q[step-1]*, *Q[step-2]*, *Q[step-3]*, *Q[step-4]* a *x[W[step-1]* spočítá s pomocí konstant K, W a R standardní krok hashovací funkce MD5. Hodnota *step* může nabývat hodnot 1..64. Zápis funkce je následující:

```
public void md5step(int step)
{
  Q.get(step).setValue(Q.get(step-1) + ROTL(f(Q.get(step-1),
                       Q.get(step-2), Q.get(step-3),step-1)+
                       Q.get(step-4) + K[step-1] +
                       x[W[step-1]], R[step-1]));
}
```

- *inverseMd5step(step)* – tato funkce na základě rovnice pro krok číslo *step* spočítá hodnotu *x[W[step-1]]* pomocí aktuálního obsahu hodnot *Q*. Hodnota *step* může nabývat hodnot 1..64. Zápis funkce je následující:

```
public void inverseMd5step(int stepProQ)
{
  int step = stepProQ - 1;
  x[W[step]] = ROTR(Q.get(step+1) - Q.get(step), R[step]) –
               f(Q.get(step), Q.get(step-1), Q.get(step-2),
               step) - Q.get(step-3) - K[step];
}
```

- *initRegisters(a0, b0, c0, d0)* – tato funkce nastaví počáteční hodnoty registrů *Q[-3], Q[-2], Q[-1] a Q[0]* dle popisu v kapitole 4.1.

```
public void initRegisters(MyUint a0, MyUint b0, MyUint c0,
                          MyUint d0)
{
    Q.get(-3).setValue(a0);
    Q.get(-2).setValue(d0);
    Q.get(-1).setValue(c0);
    Q.get(0).setValue(b0);

}
```

- *md5calculateIhv* – tato funkce na základě hodnot registrů *Q[-3], Q[-2], Q[-1], Q[0], Q[61], Q[62], Q[63] a Q[64]* spočítá mezihodnotu hashe (IHV) a uloží ji do registrů *Q[65], Q[66], Q[67] a Q[68]*. To proto, abychom podmínky pro IHV mohli zkontrolovat pomocí stejných funkcí jako podmínky pro registry *Q[1]..Q[64]*.

```
public void md5calculateIhv()

{

    Q.get(65).setValue(Q.get(-3) + Q.get(61));
    Q.get(66).setValue(Q.get(0) + Q.get(64));
    Q.get(67).setValue(Q.get(-1) + Q.get(63));
    Q.get(68).setValue(Q.get(-2) + Q.get(62));
}
```

Podmínky

Ve složce Conditions jsou uloženy třídy, které jsou určeny pro kontrolu, zda hodnoty *Q* splňují požadované podmínky. Seznamy podmínek jsou uloženy v následujících třídách:

- Třída *ListOfConditionsForBlock* – tato třída obsahuje seznam 72 proměnných typu *ListOfConditionsForStep* (jedná se o podmínky pro *Q[-3]* až *Q[68]*) a slouží k tomu, abychom se mohli snadno odkazovat na podmínky pro jednotlivé kroky. Třída obsahuje následující funkce:
  - *initConditions* – umožňuje hromadné nastavení podmínek na základě parametru obsahující pole 72 stringů. Ve for cyklu postupně vytváří podmínky pro jednotlivé kroky (*ListOfConditionsForStep*) pro kroky -3..68. Zdrojový kód:

```
public void initConditions(String[] stringConditions)
{
    for (int i = 0; i <= 71; i++)
    {//nastavuje podminky pro Q[-3] az Q[68]
    conditionsOneStep[i]=
        new ListOfConditionsForStep(i-3,stringConditions[i]);
    }
}
```

  - *addCondition(step, condition)* – přidá jednu podmínku pro krok *step*.
  - *checkOneStepConditions(step)* slouží k ověření, zda platí podmínka pro příslušný krok. Parametr step je v rozmezí -3..68, vzhledem k tomu, že podmínky máme uloženy v poli v rozmezí 0..71, musíme při získávání dat z pole přičíst trojku.

```
public bool checkOneStepConditions(int step,
                                   OneBlockContext context)
{
    return conditionsOneStep[step + 3].check(context);
}
```

- *setOneStepConditions(step)* slouží k upravení *Q* tak, aby *Q* splnilo podmínku pro příslušný krok. Parametr step je opět v rozmezí -3..68.

```
public void setOneStepConditions(int step,
                                 OneBlockContext context)
{
    conditionsOneStep[step + 3].set(context);
}
```

- Třída *ListOfConditionsForStep* je seznam podmínek pro jeden krok. Obsahuje seznam proměnných typu *Condition*. Součástí třídy jsou následující funkce:
  - konstruktor *ListOfConditionsForStep(step, initString).* Slouží k hromadnému nastavení podmínek pro *Q[step]* na základě *initStringu*. Délka *initStringu* musí být přesně 32 znaků, jednotlivé hodnoty udávají podmínky pro jednotlivé bity *Q[step]*. Hodnoty v initStringu mohou nabývat následujících hodnot (podobně jako v tabulce Tab. 9):
    - „0“ – tento bitu musí být nulový,
    - „1“ – tento bit musí být jednička,
    - „^“ – tento bit musí mít stejnou hodnotu jako v *Q[step-1]*,
    - „~“ – tento bit musí mít opačnou hodnotu než v *Q[step-1]*,
    - „I“, „J“, „A“ – tento bit musí odpovídat hodnotě proměnné s příslušným názvem,
    - „K“ – tento bit musí odpovídat opačné hodnotě proměnné „I“,
    - „B“ – tento bit musí odpovídat opačné hodnotě proměnné „A“,
    - pro ostatní hodnoty (např. tečka, „v“ a „x“ nenastavujeme žádnou podmínku.

    Na základě hodnot v *initStringu* vytvoříme příslušnou *Condition* a přidáme ji do seznamu podmínek pro tento krok.
  - *check* – tato funkce projde všechny podmínky pro tento krok a zkontrouluje, zda jsou pro *Q[step]* splněny. Jakmile alespoň jedna podmínka není splněna, vrátí false. Pokud jsou všechny podmínky splněny, vrátí true.
  - *set* – tato funkce projde všechny podmínky pro tento krok a modifikuje *Q[step]* tak, aby byly splněny.

- Třída *Condition* je abstraktní třídou pro podmínku na jeden bit. Od třídy *Condition* jsou odvozeny třídy pro konkrétní podmínky. Společnými vlastnostmi všech podmínek je, že si pamatují, pro který *step* a který *bit* platí a umí samy sebe zkontrolovat zda pro *Q[step]* platí (funkce *check*) a také umí modifikovat *Q[step]*, aby samy sebe splnily (funkce *set*).

- Jako příklad konkrétní podmínky odvozené od abstraktní třídy *Condition* uvádíme implementaci funkcí *check* a *set* podmínky *ConditionZero*, což je podmínka na nulový bit:

```
override public bool check(OneBlockContext context)
{
    if (context.Q.get(this.step).getBit(this.bit) == false)
    //pokud je tam nula(tj. getBit==false),je podminka splnena
    {
        return true;
    }
```

```
        else  //jinak podminka neni splnena
        {
            return false;
        }
    }

    override public void set(OneBlockContext context)
    {  //nastavime prislusny bit na nulu
        context.Q.get(this.step).setBit(this.bit, false);
    }
```

- Podmínky *ConditionOne*, *ConditionPrevious* a *ConditionNegativePrevious* jsou koncipovány podobně, odpovídají hodnotám „1“, „^“ a „~“ ve vstupním *initStringu*.

- Podmínky *ConditionVariable* a *ConditionNegativeVariable* využívají pomocnou třídu Variables. Funkce *check* funguje podle následujícího postupu:
  - pokud proměnná ještě není definována, zapamatuje si ji a nastaví její hodnotu dle hodnoty bitu,
  - pokud proměnná je již definována, zkontroluje, zda hodnota bitu odpovídá hodnotě proměnné.

  Funkce *set* zde funguje následovně:
  - pokud proměnná ještě není definována, zapamatuje si ji a nastaví její hodnotu dle hodnoty bitu (stejně jako ve funkci check),
  - pokud proměnná je již definována, nastaví hodnotu příslušného bitu v *Q[step]* na hodnotu odpovídající hodnotě proměnné.

- Trochu složitější jsou podmínky typu *Zavorka...* které při své kontrole vypočtou hodnotu *zavorka_Q* (stejně jako v programu V. Klímy) a ověřují její bity, stále se ale vždy jedná o jednoduchou podmínku, která prostě je nebo není splněna.

Tunely

Ve složce Tunnels jsou definovány třídy umožňující modifikaci hodnot *Q* pomocí tunelů dle metody V. Klímy:

- Třída *Tunnel* je abstraktní třídou, ze které jsou odvozeny konkrétní tunely. Každý tunel má následující vlastnosti:
  - zná počet svých bitů (uloženo v proměnné *bitCount*),
  - zná svou aktuální hodnotu (uloženo v proměnné *tunnelValue*. Například tunel Q9 postupně nabývá hodnot 000, 100, 010, 110, 001, 101, 011, 111),
  - umí nastavit svou další hodnotu pomocí funkce *nextValue* (např. z 000 na 100, z 100 na 010, atd.),
  - umí se resetovat pomocí funkce *reset* (nastaví se na hodnotu 000, resp. u pravděpodobnostního tunelu na první platnou hodnotu),
  - umí se použít pomocí funkce *apply*. Tato funkce je v třídě Tunnel definována jako abstraktní, její konkrétní implementace jsou napsány v konkrétních tunelech. Při implementaci funkcí apply dodržujeme ve všech tunelech následující pravidla:

- funkce *apply* nastaví následující hodnotu *tunnelValue* pomocí funkce *nextTunnelValue*. Pokud už další hodnota není, vrátí false (=tunel už vyčerpal své možné hodnoty),
- dle nové hodnoty *tunnelValue* změní hodnotu tunelovaných bitů v některém *Q*,
- provede potřebné korekce hodnot *x* a *Q*,
- pokud se jedná o pravděpodobnostní tunel, zkontroluje platnost podmínek, které potenciálně mohou být porušeny. Pokud některá z nich je porušena, tunel zkouší další hodnotu *nextTunnelValue*.
- pokud byl tunel úspěšně aplikován, vrátí funkce *apply* hodnotu true.

Na ukázku zde uvádíme zdrojový kód funkce apply v tunelu Q9:

```
override public bool apply(OneBlockContext context)
{
    if (nextTunnelValue(0) == false)
    {
        return false;
    }

    //tunnel Q9
    //(poradi bitu je pocitano zleva od nuly)
    context.Q.get(9).setBit(8, tunnelValue[0]);
    context.Q.get(9).setBit(9, tunnelValue[1]);
    context.Q.get(9).setBit(10, tunnelValue[2]);

    //korekce tunnelu Q9
    context.inverseMd5step(9);    //spocitam x[8] dle Q9
    context.inverseMd5step(10);    //spocitam x[9] dle Q10
    context.inverseMd5step(13);    //spocitam x[12] dle Q13

    return true;

}
```

- Třída *Tunnels* obsahuje proměnnou *listOfTunnels* obsahující seznam použitých tunelů. V konstruktoru této třídy naplníme tento seznam tunely Q9, Q4, Q14, Q10, Q13 a Q20 pro první blok, a tunely Q9 a Q4 pro druhý blok. Třída *Tunnels* obsahuje následující funkce:
  - *reset* – resetuje hodnotu všech tunelů. Fakticky to znamená, že pro každý tunel zavolá jeho funkci *reset*, čímž nastaví tunelované bity na nulu (resp. v případě pravděpodobnostních tunelů na první hodnotu, která neporušuje podmínky).
  - *apply* – zavolá funkci *apply* prvního tunelu v pořadí, čímž nastaví jeho nejbližší platnou hodnotu.
    - Pokud funkce *apply* tohoto tunelu vrátí true, znamená to, že tunel byl úspěšně použit. Funkce *apply* třídy *Tunnels* v tomto případě vrátí true (=došlo k úspěšné aplikaci některého z tunelů).
    - Pokud funkce *apply* použitého tunelu vrátí false, znamená to, že tento tunel vyčerpal své možnosti. V tomto případě resetujeme tento tunel a voláme rekurzivně *apply* pro další tunel v seznamu (Ten se v případě vyčerpání také resetuje a volá rekurzivně další tunel v seznamu, atd.). Tím zkoušíme všechny možné hodnoty tunelů.

- V případě, že jsme došli v seznamu tunelů až na konec, znamená to, že byly vyčerpány všechny hodnoty všech tunelů, což znamená, že bude muset být přepočítán nový POV od začátku. V tom případě vrátí funkce *apply* třídy *Tunnels* hodnotu false.

Inicializace podmínek

Jak jsme již popsali, výhodou programu je možnost snadné definice podmínek pro výpočet kolizí. Na ukázku zde uvádíme zdrojový kód pro definici postačujících podmínek pro Q[-3]..Q[64] a IHV0..IHV4 pro první blok.

```
String[] podminkyPrvniBlok = {
        "............................",  //podminka pro Q[-3]
        "............................",  //podminka pro Q[-2]
        "............................",  //podminka pro Q[-1]
        "............................",  //podminka pro Q[0]
        "............................",  //podminka pro Q[1]
        "............................",  //podminka pro Q[2]
        ".........vvv0vvvvvvv0vvvv0......",  //podminka pro Q[3]
        "1.....x.0^^^1^^^^^^^1^^^^011....",  //podminka pro Q[4]
        "1000100v01000000000000000100101",  //podminka pro Q[5]
        "0000001^01111111101111000100^0^1",  //podminka pro Q[6]
        "00000011111111101111100000100000",  //podminka pro Q[7]
        "000000011..100010.0v010101000000",  //podminka pro Q[8]
        "11111011xxx100000.1^111100111101",  //podminka pro Q[9]
        "0111.x.x000111111v01.x.001....00",  //podminka pro Q[10]
        "0010.0v0111.00011^00.0.011....10",  //podminka pro Q[11]
        "000...^^....10000001...10.......",  //podminka pro Q[12]
        "01.xx.01.xxx1111111.xxx00x.x1...",  //podminka pro Q[13]
        "000xxx00....1011111....11xxx1xxx",  //podminka pro Q[14]
        "v1100001..V.....10.......0000000",  //podminka pro Q[15]
        "^01000....~.....v........000v000",  //podminka pro Q[16]
        "^1v...........0.^...........^...",  //podminka pro Q[17]
        "^.^...........1.................",  //podminka pro Q[18]
        "^.............0.................",  //podminka pro Q[19]
        "^.......x.x...v..x....x.......xx",  //podminka pro Q[20]
        "^.............^.................",  //podminka pro Q[21]
        "^............................",  //podminka pro Q[22]
        "0............................",  //podminka pro Q[23]
        "1............................",  //podminka pro Q[24]
        "............................",  //podminka pro Q[25]
        "............................",  //podminka pro Q[26]
        "............................",  //podminka pro Q[27]
        "............................",  //podminka pro Q[28]
        "............................",  //podminka pro Q[29]
        "............................",  //podminka pro Q[30]
        "............................",  //podminka pro Q[31]
        "............................",  //podminka pro Q[32]
        "............................",  //podminka pro Q[33]
        "............................",  //podminka pro Q[34]
        "............................",  //podminka pro Q[35]
        "............................",  //podminka pro Q[36]
        "............................",  //podminka pro Q[37]
        "............................",  //podminka pro Q[38]
        "............................",  //podminka pro Q[39]
        "............................",  //podminka pro Q[40]
        "............................",  //podminka pro Q[41]
        "............................",  //podminka pro Q[42]
        "............................",  //podminka pro Q[43]
        "............................",  //podminka pro Q[44]
```

```
            "............................",  //podminka pro Q[45]
            "I...........................",  //podminka pro Q[46]
            "J...........................",  //podminka pro Q[47]
            "I...........................",  //podminka pro Q[48]
            "J...........................",  //podminka pro Q[49]
            "K...........................",  //podminka pro Q[50]
            "J...........................",  //podminka pro Q[51]
            "K...........................",  //podminka pro Q[52]
            "J...........................",  //podminka pro Q[53]
            "K...........................",  //podminka pro Q[54]
            "J...........................",  //podminka pro Q[55]
            "K...........................",  //podminka pro Q[56]
            "J...........................",  //podminka pro Q[57]
            "K...........................",  //podminka pro Q[58]
            "J...........................",  //podminka pro Q[59]
            "I.....0.....................",  //podminka pro Q[60]
            "J.....1.....................",  //podminka pro Q[61]
            "I.....0.....................",  //podminka pro Q[62]
            "J...........................",  //podminka pro Q[63]
            "............................",  //podminka pro Q[64]
            "............................",  //podminka pro Q[65] (=IHV0)
            "A....00................0....",  //podminka pro Q[66] (=IHV1)
            "A....01.....................",  //podminka pro Q[67] (=IHV2)
            "A.....0....................."}; //podminka pro Q[68] (=IHV3)

//inicializuju zakladni podminky pro prvni blok
contextFirstBlock.conditions.initConditions(podminkyPrvniBlok);
//nyni inicializuji podminky pro zavorky:
//step 19 - mala podminka NotAllOnes pro bity 14..28 (cisl.od nuly zleva)
contextFirstBlock.conditions.addCondition(19,
                            new ZavorkaConditionNotAllOnes(19, 14, 28));
//step 20 - mala podminku NotAllZeros pro bity 0..2
contextFirstBlock.conditions.addCondition(20,
                            new ZavorkaConditionNotAllZeros(20, 0, 2));
//step 23 - podminka Zero pro bit 14 zavorka_23
contextFirstBlock.conditions.addCondition(23,
                            new ZavorkaConditionZero(23, 14));
//step 35 - podminka Zero pro bit 16 zavorka_35
contextFirstBlock.conditions.addCondition(35,
                            new ZavorkaConditionZero(35, 16));
//step 62 - mala podminku NotAllOnes pro bity 10..16
contextFirstBlock.conditions.addCondition(62,
                            new ZavorkaConditionNotAllOnes(62, 10, 16));
```

Algoritmus hledání kolize

Postup hledání kolize se skládá ze čtyřech základních kroků:

```
inicializujPodminky();
najdiPrvniBlok();
najdiDruhyBlok();
sestrojKolidujiciZpravy();
```

Vzhledem k tomu, že jsme ve zdrojovém kódu použili intuitivní názvy funkcí, uvádíme zde jako ukázku přímo zdrojový kód pro výpočet prvního bloku.

```
//nastavim konstanty pro zacatek vypoctu MD5 pro prvni blok
const uint a0 = 0x67452301;
const uint b0 = 0xefcdab89;
const uint c0 = 0x98badcfe;
const uint d0 = 0x10325476;
contextFirstBlock.initRegisters(a0, b0, c0, d0);
```

```
Repeat: //sem se vracim, pokud zacinam pocitat novy POV od zacatku

    //smazu ulozene hodnoty promennych I,J,K,A,B...
    contextFirstBlock.variables.deleteVariables();

    //nastavim hodnoty registru Q[1]..Q[17] nahodne
    //a upravim je tak, aby splnovaly pozadovane podminky
    for (int i = 1; i <= 17; i++)
    {
        contextFirstBlock.setRegisterAccordingConditions(i);
    }

    contextFirstBlock.inverseMd5step(17);    //spocitam x[1] dle Q17
    contextFirstBlock.md5step(2);   //prepocitam Q2 kvuli zmene x[1]

    //spocitame x[0] .. x[15] dle Q1..Q16
    for (int i = 1; i <= 16; i++)
    {
        contextFirstBlock.inverseMd5step(i);
    }

    //spocitame Q18..Q24
    //pokud neplati podminka pro Q(i), vratim se zpet na zacatek
    for (int i = 18; i <= 24; i++)
    {
        contextFirstBlock.md5step(i);
        if (!contextFirstBlock.checkRegisterAgainstConditions(i))
           goto Repeat;
    }
    //---------------------- POV ----------------------
    //resetuju hodnoty vsech tunelu
    contextFirstBlock.tunnels.reset(contextFirstBlock);

RepeatTunnel:
    contextFirstBlock.variables.deleteVariables(); //mazu promenne I,J,A

    //funkce tunnels.apply nastavi novou hodnotu tunelu–vytvori novy POV
    if (contextFirstBlock.tunnels.apply(contextFirstBlock) == false)
    {
        //pokud vsechny tunely vycerpaly sve moznosti, vracim se
        //na zacatek – budu pocitat novy POV od zacatku
        goto Repeat;
    }

    //spocitame Q25..Q34. Nejsou zadne podminky
    for (int i = 25; i <= 34; i++)
    {
        contextFirstBlock.md5step(i);
    }

    contextFirstBlock.md5step(35);   //spocitame Q35
    //pokud neplati podminka pro Q(35), vratime se zpet na tunely
    if (!contextFirstBlock.checkRegisterAgainstConditions(35))
        goto RepeatTunnel;

    //spocitame Q36..Q45. Nejsou zadne podminky
    for (int i = 36; i <= 45; i++)
    {
        contextFirstBlock.md5step(i);
    }
```

```
    //spocitame Q46..Q64
    //pokud neplati podminka pro Q(i), vratim se zpet na tunely
    for (int i = 46; i <= 64; i++)
    {
        contextFirstBlock.md5step(i);
        if (!contextFirstBlock.checkRegisterAgainstConditions(i))
           goto RepeatTunnel;
     }

    //nyni spocitam mezihodnotu hashe (IHV). Ulozi se do Q65..Q68
    contextFirstBlock.md5calculateIhv();

    //nyni overim podminky pro IHV
    //pokud neplati nektera podminka pro IHV, vratim se zpet na tunely
    for (int i = 65; i <= 68; i++)
    {
        if (!contextFirstBlock.checkRegisterAgainstConditions(i))
           goto RepeatTunnel;
    }

    //nyni urcim x[0] .. x[15] pro kolizni zpravu
    for (int i = 0; i <= 15; i++)
    {
        contextFirstBlockB.x[i] = contextFirstBlock.x[i];
    }
    //nastavim rozdily: 2^31 v x[4], 2^15 v x[11] a 2^31 v x[14]
    contextFirstBlockB.x[4] = contextFirstBlock.x[4] + 0x80000000;
    contextFirstBlockB.x[11] = contextFirstBlock.x[11] + 0x00008000;
    contextFirstBlockB.x[14] = contextFirstBlock.x[14] + 0x80000000;

    //nyni spocitam MD5 pro kolizni zpravu
    contextFirstBlockB.initRegisters(a0, b0, c0, d0);
    for (int i = 1; i <= 64; i++)
    {
        contextFirstBlockB.md5step(i);   //i-ty krok MD5
    }
    contextFirstBlockB.md5calculateIhv(); //spocitam IHV (Q65..Q68)

    //overim podminky pro rozdil IHV pro puvodni a kolizni zpravu
    //pokud nektera neplati, vratim se zpet na tunely
     if (contextFirstBlockB.Q.get(65) - contextFirstBlock.Q.get(65) –
         0x80000000 != 0) {goto RepeatTunnel;} //podminka na rozdil IHV0
     if (contextFirstBlockB.Q.get(66) - contextFirstBlock.Q.get(66) –
         0x82000000 != 0) {goto RepeatTunnel;} //podminka na rozdil IHV1
     if (contextFirstBlockB.Q.get(67) - contextFirstBlock.Q.get(67) –
         0x82000000 != 0) {goto RepeatTunnel;} //podminka na rozdil IHV2
     if (contextFirstBlockB.Q.get(68) - contextFirstBlock.Q.get(68) –
         0x82000000 != 0) {goto RepeatTunnel;} //podminka na rozdil IHV3

//tim jsme splnili vsechny podminky pro prvni blok
```

7.3. Porovnání rychlosti

V této kapitole uvádíme tabulku s porovnáním průměrné doby pro nalezení kolize v našem programu s programem V.Klímy. V našem programu i v programu V.Klímy bylo provedeno 50 iterací hledání kolizí a vypočtena průměrná doba pro nalezení kolize. Výpočet byl prováděn na notebooku Fujitsu, procesor Intel Core i5 2,4 GHz, Windows 7 (64-bit). Zápis časů je ve formátu "hodiny:minuty:sekundy" a je zaokrouhlen na celé sekundy nahoru.

Tab. 17: Porovnání rychlostí programů pro hledání kolizí funkce MD5

Výsledky našeho programu			
Postup: spuštěno hledání kolize s inicializováním generátoru pseudonáhodných čísel hodnotou "Iterace"			
Iterace	Čas pro nalezení prvního bloku	Čas pro nalezení druhého bloku	Celkový čas pro nalezení kolize
1	0:05:42	0:01:20	0:07:03
2	0:16:21	0:01:52	0:18:14
3	0:06:23	0:01:54	0:08:18
4	0:00:13	0:03:23	0:03:37
5	0:03:03	0:03:42	0:06:45
6	0:05:35	0:03:31	0:09:06
7	0:03:13	0:00:39	0:03:52
8	0:02:27	0:02:42	0:05:09
9	0:04:06	0:00:32	0:04:38
10	0:00:58	0:03:40	0:04:38
11	0:03:22	0:24:41	0:28:04
12	0:00:04	0:42:39	0:42:44
13	0:05:34	0:00:38	0:06:12
14	0:01:57	0:01:26	0:03:24
15	0:01:52	0:12:09	0:14:02
16	0:00:08	0:02:55	0:03:03
17	0:06:14	0:00:44	0:06:59
18	0:04:13	0:00:42	0:04:55
19	0:00:36	0:03:19	0:03:55
20	0:13:58	0:01:15	0:15:13
21	0:09:17	0:03:29	0:12:46
22	0:00:17	0:04:16	0:04:33
23	0:03:57	0:04:43	0:08:40
24	0:02:23	0:01:51	0:04:15
25	0:06:24	0:01:21	0:07:46
26	0:00:32	0:01:04	0:01:36
27	0:08:20	0:04:53	0:13:13
28	0:01:45	0:01:25	0:03:11
29	0:01:11	0:00:45	0:01:57
30	0:07:47	0:00:21	0:08:08
31	0:05:47	0:02:26	0:08:13
32	0:04:40	0:08:32	0:13:12
33	0:15:17	0:00:46	0:16:03
34	0:12:24	0:27:47	0:40:11
35	0:07:53	0:14:58	0:22:52
36	0:05:22	0:01:40	0:07:02
37	0:09:35	0:00:00	0:09:35
38	0:02:12	0:01:32	0:03:44
39	0:17:05	0:04:00	0:21:06
40	0:03:17	0:06:08	0:09:25
41	0:00:37	0:01:53	0:02:31
42	0:03:29	0:01:17	0:04:47
43	0:01:36	0:01:25	0:03:02
44	0:00:34	0:00:08	0:00:43
45	0:02:35	0:04:47	0:07:23
46	0:04:35	0:00:20	0:04:56
47	0:03:40	0:02:25	0:06:06
48	0:05:27	0:19:35	0:25:02
49	0:13:35	0:01:58	0:15:33
50	0:01:27	0:06:19	0:07:47
	Průměrný čas pro nalezení prvního bloku	**Průměrný čas pro nalezení druhého bloku**	**Průměrný čas pro nalezení kolize**
	0:04:59	**0:04:55**	**0:09:54**

Výsledky programu V.Klímy (verze 0)			
Postup: spuštěn program V.Klímy verze 0 Zápis na příkazové řádce: "md5tunnel.exe 1"			
Iterace	Čas pro nalezení prvního bloku	Čas pro nalezení druhého bloku	Celkový čas pro nalezení kolize
1	0:01:09	0:00:01	0:01:10
2	0:00:04	0:00:04	0:00:08
3	0:00:13	0:00:14	0:00:27
4	0:00:52	0:00:22	0:01:14
5	0:00:12	0:00:24	0:00:36
6	0:00:27	0:00:01	0:00:28
7	0:01:11	0:00:01	0:01:12
8	0:00:20	0:00:01	0:00:21
9	0:00:13	0:00:01	0:00:14
10	0:00:01	0:00:08	0:00:09
11	0:01:26	0:00:01	0:01:27
12	0:00:02	0:00:02	0:00:04
13	0:01:08	0:00:28	0:01:36
14	0:00:26	0:00:01	0:00:27
15	0:01:05	0:00:01	0:01:06
16	0:00:32	0:00:03	0:00:35
17	0:00:41	0:00:01	0:00:42
18	0:00:30	0:00:08	0:00:38
19	0:01:04	0:00:01	0:01:05
20	0:01:16	0:00:03	0:01:19
21	0:00:06	0:00:01	0:00:07
22	0:00:48	0:00:01	0:00:49
23	0:01:12	0:00:21	0:01:33
24	0:00:42	0:00:18	0:01:00
25	0:00:05	0:00:05	0:00:10
26	0:00:26	0:00:01	0:00:27
27	0:00:08	0:00:05	0:00:13
28	0:00:26	0:00:39	0:01:05
29	0:00:11	0:00:07	0:00:18
30	0:00:34	0:00:01	0:00:35
31	0:01:33	0:00:57	0:02:30
32	0:00:14	0:00:14	0:00:28
33	0:00:08	0:00:11	0:00:19
34	0:00:43	0:00:01	0:00:44
35	0:00:09	0:00:01	0:00:10
36	0:00:01	0:00:01	0:00:02
37	0:00:07	0:00:01	0:00:08
38	0:01:19	0:01:47	0:03:06
39	0:00:05	0:00:01	0:00:06
40	0:00:02	0:00:01	0:00:03
41	0:00:29	0:00:23	0:00:52
42	0:00:16	0:00:01	0:00:17
43	0:00:29	0:00:01	0:00:30
44	0:01:38	0:01:38	0:03:16
45	0:00:05	0:00:02	0:00:07
46	0:00:48	0:00:01	0:00:49
47	0:01:52	0:00:39	0:02:31
48	0:00:02	0:00:01	0:00:03
49	0:00:07	0:00:01	0:00:08
50	0:00:11	0:00:01	0:00:12
	Průměrný čas pro nalezení prvního bloku	**Průměrný čas pro nalezení druhého bloku**	**Průměrný čas pro nalezení kolize**
	0:00:33	**0:00:12**	**0:00:45**

8. Závěr

Ukázali jsme, jak fungují jedny z nejznámějších hashovacích funkcí MD5, SHA-1 a SHA-1. Podrobně jsme rozebrali útoky na tyto funkce provedené čínskými matematiky. Přidali jsme také různá zefektivnění útoků, včetně aplikace tunelů – českého kryptoanalytika V. Klímy [14] jež umožnuje najít kolizi do minuty na běžném PC. Ukázali jsme také jak lze využít uvedené útoky v praxi (hledání dvou kolidujících certifikátů X.509). Jaká budoucnost tedy čeká funkci MD5 a jí podobné hashovací funkce?

Hashovací funkce MD5 byla prakticky prolomena a na konkrétních příkladech je možné ukázat nalezení kolize MD5 do minuty. Algoritmus SHA-1 byl zatím prolomen teoreticky, avšak na praktické předvedení kolizí plné SHA-1 funkce si budeme muset ještě počkat. Pro použití v oblasti elektronického podpisu a bankovnictví bezpečná není. Aktuálně používaná a dosud bezpečná je funkce SHA-2. Všechny certifikační autority v současnosti používají hashovací funkci SHA-256, do tří let se předpokládá úplný přechod na funci SHA-512.

Certifikační autorita má vystaven vlastní certifikát (certifikát certifikační autority) podepsaný jí nadřazenou certifikační autoritou. Tato nadřazená certifikační autorita má také certifikát podepsaný jí nadřazenou certifikační autoritou, atd. Takto popsaný řetězec je řetězec důvěry a na jejím vrcholu je kořenová certifikační autorita, jež má vlastní kořenový certifikát a ten si podepíše sama.

Při ověřování důvěryhodnosti podepsaného dokumentu uživatel postupuje od konce řetězce důvěry od jedné certifikační autority k druhé až narazí na takovou certifikační autoritu jíž důvěřuje (může dojít až ke kořenové certifikační autoritě – ta je považována za důvěryhodnou). Tedy aby nedošlo k narušení takto sestaveného řetězce důvěry, je potřeba, aby nedošlo ke kompromitaci žádné ze zúčastněných certifikačních autorit.

Je tedy nutné, aby certifikační autoriy dbaly na svou důvěryhodnost – nechaly si vystavit svůj certifikát od důvěryhodné nadřazené certifikační autority a nevystavovaly certifikáty, které by mohly být snadno napadnutelné. Pokud tedy všechny certifikační autority zainteresované v řetězci důvěry používají hashovací funkci SHA-256 nebo SHA-512 (či jinou bezpečnou hashovací funkci), nedojde k narušení řetězce důvěry.

Hashovací funkce SHA-512 je považována za bezpečnou zhruba do konce roku 2016, v tomto roce bude nutný přechod na novou bezpečnou hashovací funkci. V současné době však neexistuje žádná nová bezpečná a schválená hashovací funkce, která by se dala za pár let použít.

8.1. Scénáře útoků

Tato část vychází z práce M. Stevense a kol. [32].

Rozeberme reálné použití kolizí hashovacích funkcí a uvedeme možné scénáře útoků. Všechny scénáře je možné provést s libovolnou hashovací funkcí za předpokladu, že dokážeme získat kolizi prvního řádu této funkce.

Pro kolize dvou certifikátů, které se liší jen v několika málo bitech RSA modulu (viz podkapitola 4.3.1 Kolize certifikátů X.509) můžeme sestrojit jen jednoduché scénáře. Všechny útoky jsou však při podrobnějším zkoumání snadno odhalitelné.

Jeden z možných scénářů je, že Alice si vytvoří dva kolidující certifikáty. Jeden si nechá podepsat certifikační autoritou (CA) a na druhý si tento podpis zkopíruje (neboť podpisy

budou stejné). CA neví, že existuje kolidující certifikát a předložený certifikát podepíše. Tím dochází k porušení jednoho z principů infrastruktury správy veřejných klíčů PKI (Public Key Infrastructure) a to, že CA zaručuje jednoznačnou identifikaci mezi veřejným klíčem a subjektem certifikátu.

Alice může rozeslat své dva certifikáty různým skupinám lidí. Pak může Alice používat své kolidující veřejné klíče tak, jak se jí to hodí. Např. když Bob pošle Alici zašifrovanou zprávu, pak Alice může použít „nesprávný“ (kolidující) soukromý klíč aby ukázala, že zprávu nedokáže přečíst. Vzhledem k tomu, že hledání druhého vzoru (kolize druhého řádu) je efektivně neuskutečnitelné, je Bobovi jasné, že Alice lže a vlastní dva kolidující certifikáty.

Další možný scénář je, že si Alice nevygenerovala klíč sama, ale obdržela jej od generátoru klíčů (může být přes webovou aplikaci, nebo off-line software). Generátor klíčů může záměrně vygenerovat dva kolidující klíče, jeden odevzdá Alici a druhý si schová, aniž by o tom Alice věděla. Generátor klíčů pak může Bobovi podstrčit kolidující certifikát se svým klíčem a identifikačními údajemi Alice. Bob pak v domění, že posílá zprávy Alici, používá k šifrování podstrčený veřejný klíč, a takové zprávy může dešifrovat poze generátor klíčů. Tento útok je však také snadno odhalitelný. Stačí aby Alice poslala Bobovi zašifrovanou zprávu svým soukromým klíčem a Bob nebude schopen takovou zprávu dešifrovat. Nyní již Alici i Bobovi je zřejmé, že existuje útočník s kolidujícím certifikátem.

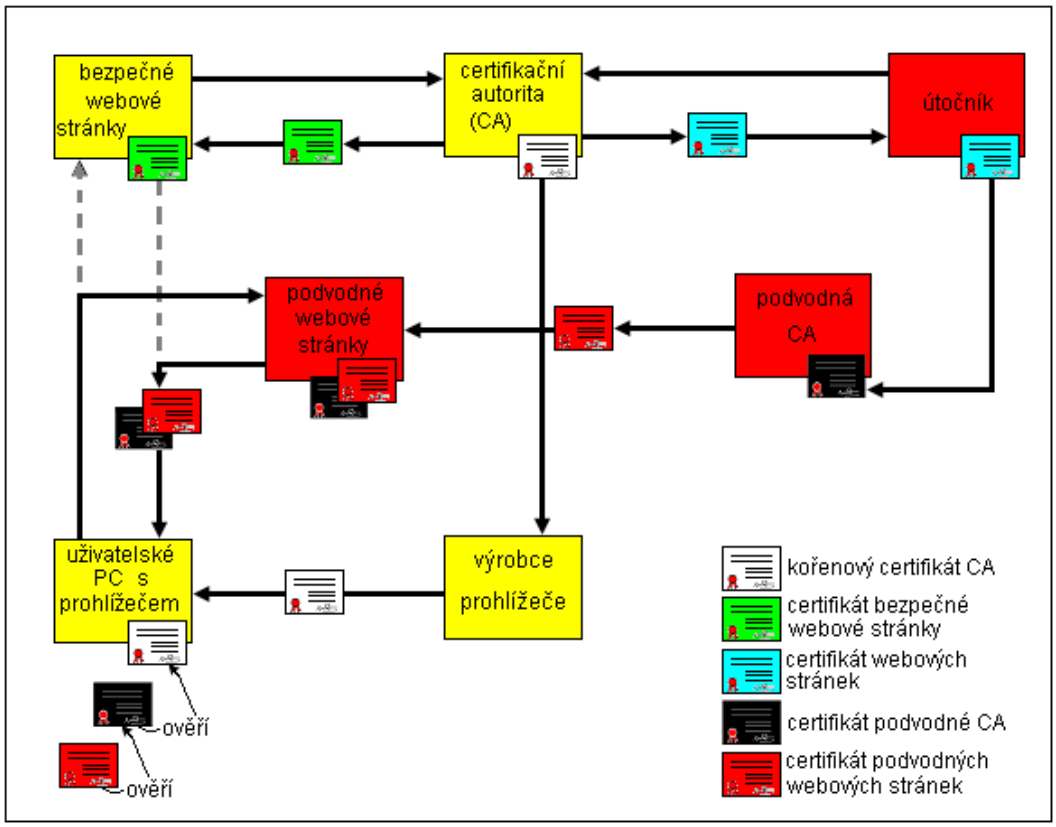
Mnohem zajímavější a sofistikovanější scénář útoku můžeme najít pokud budeme uvažovat kolize certifikátů s volitelným prefixem (viz [32]). Zde je potřeba upozornit, že se stále jedná o kolize prvního řádu, kdy je nutné znát a upravovat oba certifikáty (původní i kolidující).

Tento scénář útoku je založen na vytvoření dvou kolidujících certifikátů, jeden pro webové stránky a druhý pro podvodnou certifikační autoritu. Důvěryhodná kořenová certifikační autorita podepíše certifikát pro webové stránky, přičemž nemá tušení, že existuje kolidující certifikát.

Nejprve popíšme, jak fungují webové certifikáty. Důvěryhodná certifikační autorita vytvoří svůj kořenový certifikát a prostřednictvím výrobce prohlížeče je distribuován spolu s prohlížečem. Tento kořenový certifikát se následně uloží do uživatelského PC do seznamu důvěryhodných certifikátů. Tedy i všechny certifikáty podepsané touto autoritou budou automaticky uznány za důvěryhodné.

Pokud chce firma provozovat bezpečné webové stránky, požádá certifikační autoritu o vydání certifikátu pro své webové stránky. Pak tento certifikát rozesílají uživatelům, kteří jej vyžadují pro ověření bezpečnosti těchto webových stránek. Následná komunikace mezi uživatelem a bezpečnými webovými stránkami je již zabezpečena šifrováním.

Obr. 11: Použití certifikátu podvodné CA



Cílem útočníka je, aby s ním koncový uživatel komunikoval v domění, že je připojen na bezpečné webové stránky. Útočník si tedy vytvoří dva kolidující certifikáty: pro libovolné webové stránky a pro podvodnou CA. Certifikát pro webové stránky si nechá podepsat důvěryhodnou CA a tento podpis zkopíruje na svůj certifikát pro podvodnou CA (kolidující certifikáty mají stejný podpis CA).

Pak útočník vytvoří podvodné webové stránky, které budou vypadat stejně jako bezpečné webové stránky a pomocí podvodné CA si vydá certifikát, jež bude mít stejné identifikační údaje jako bezpečné webové stránky jen jiný veřejný klíč.

Pokud bude chtít uživatel přistupovat k bezpečným webovým stránkám, útočník může jeho komunikaci odchytit a přesměrovat jej na své podvodné stránky. Uživatel bude samozřejmě požadovat certifikát webových stránek. Útočník tedy pošle certifikát stránek spolu s certifikátem podvodné CA. Certifikát podvodné CA je podepsán důvěryhodnou certifikační autoritou a tedy uživatel bude důvěřovat této podvodné CA a také podvodným webovým stránkám. Nyní již útočník dokáže snadno vylákat důvěrné údaje uživatele (např. přihlašovací údaje k internetovému bankovnictví).

8.2. Nové hashovací funkce

Jak jsme již zmínili, bezpečná použitelnost hashovacích funkcí rodiny SHA-2 se předpokládá zhruba do konce roku 2015. Je tedy na místě, se aktivně zajímat o to, která hashovací funkce nahradí SHA-2 po roce 2015. Tento problém neunikl pozornosti ani Národnímu institutu pro standardy a technologie (NIST) a vyhlásil soutěž o nový standard SHA-3. Podobnou situaci řešil NIST již jednou veřejnou soutěží, a to v roce 1997, kdy hledal novou blokovou šifru, která nahradila DES – vznikl tak nový standard AES.

Veřejná soutěž o funkci SHA-3 byla vyhlášena dne 2. listopadu 2007 a konečný termín pro odevzdání návrhů byl stanoven na 31. 10. 2008. K tomuto datu NIST obdržela 64 kandidátů, přičemž pro první výběrové kolo bylo přijato 51 z nich. NIST pak ve dnech 25. – 28. 2. 2009 uspořádala první konferenci kandidátů SHA-3. Následně 24. 7. 2009 postoupilo pouze 14 kandidátů do druhého kola. Autoři těchto 14-ti kandidátů dostali možnost upravit některé nedostatky a 9. prosince 2010 bylo zvoleno 5 kandidátů postupujících do třetího výběrového kola. Jsou to funkce s názvy BLAKE, Grøstl, JH, Keccak a Skein. Nyní procházejí tyto funkce obdobím veřejného připomínkování a na jaře roku 2012 je naplánována konference a diskuze s autory uvedených hashovacích funkcí, přičemž nejpozději v zimě roku 2012 chce NIST vybrat vítěze. Vítězná funkce pak bude označena jako SHA-3 a specifikována ve standardu FIPS 180-3.

Soutěže se také zúčastnili čeští kryptoanalytici (v týmu byl také již mnohokrát zmiňovaný Vlastimil Klíma) se svými hashovacími funkcemi Blue Midnight Wish a EDON-R. Ani jedna z funkcí však neprošla do nejužšího výběru. U funkce Blue Midnight Wish bylo zjištěno, že nalezení blízké kolize je jednodušší, než by matematicky mělo být a funkce EDON-R neprošla kvůli používání kvazigrup,jež je dosud neprobádanou oblastí a mnozí účastníci soutěže požadovali diskvalifikaci tohoto návrhu.

V roce 2007 uvedl V. Klíma třídu nových hashovacích funkcí (viz [41]), jež nevyužívají základního Merkle-Damgardova modelu, ale jsou založeny na známém modelu NMAC a HMAC a na návrhu nové speciální blokové šifře. Klíma zde uvádí také důkazy bezpečnosti těchto funkcí, což žádná dosavadní hashovací funkce nemá. U nynějších používaných hashovacích funkcí se prostě věří, že jsou bezpečné.

Konkrétní návrh hashovací funkce z této nové třídy, funkce HDN se zdá být reálně použitelná. Má délku výstupní hashe 512-bitů a její výpočetní složitost je srovnatelná s funkcí SHA-512. Navíc u HDN lze dokázat její teoretické vlastnosti a odolnost vůči kolizím a hledání druhého vzoru. Zdá se, že hashovací funkce založené na takto navržené konstrukci se mohou stát třídou hashovacích funkcí nové generace a tato koncepce se může stát vzorem pro všechny budoucí hashovací funkce.

Máme tedy návrhy nové hashovací funkce SHA-3 a velice silný návrh hashovacích funkcí nové generace V. Klímy. Čas ukáže, který z těchto kandidátů úspěšně nahradí dosavadní hashovací funkce a bude dobře plnit svou malou, ale důležitou roli ve službách kryptografie.

9. Přílohy

9.1. Tabulky pro funkci MD5

Tab. MD5-1: První diferenční cesta dle Wangové a kol.

t	δW_t	R_t	δQ_t	δF_t	δT_t
0 - 3	—	·	—	—	—
4	2^{31}	7	—	—	2^{31}
5	—	12	$+ 2^{6} \ldots + 2^{21}, - 2^{22}$	$+ 2^{11} + 2^{19}$	$+ 2^{11} + 2^{19}$
6	—	17	$- 2^{6} + 2^{23} + 2^{31}$	$- 2^{10} - 2^{14}$	$- 2^{10} - 2^{14}$
7	—	22	$+ 2^{0} \ldots + 2^{4}, - 2^{5}, + 2^{6} \ldots + 2^{10} - 2^{11}, - 2^{23} \ldots - 2^{25}, + 2^{26} \ldots + 2^{31}$	$- 2^{2} + 2^{5} + 2^{10} + 2^{16} - 2^{25} - 2^{27}$	$- 2^{2} + 2^{5} + 2^{10} + 2^{16} - 2^{25} - 2^{27}$
8	—	7	$+ 2^{0} + 2^{15} - 2^{16} + 2^{17} + 2^{18} + 2^{19} - 2^{20} - 2^{23}$	$+ 2^{6} + 2^{8} + 2^{10} + 2^{16} - 2^{24} + 2^{31}$	$+ 2^{8} + 2^{10} + 2^{16} - 2^{24} + 2^{31}$
9	—	12	$- 2^{0} + 2^{1} + 2^{6} + 2^{7} - 2^{8} - 2^{31}$	$+ 2^{0} + 2^{6} - 2^{20} - 2^{23} + 2^{26} + 2^{31}$	$+ 2^{0} - 2^{20} + 2^{26}$
10	—	17	$- 2^{12} + 2^{13} + 2^{31}$	$+ 2^{0} + 2^{6} + 2^{13} - 2^{23}$	$+ 2^{13} - 2^{27}$
11	2^{15}	22	$+ 2^{30} + 2^{31}$	$- 2^{0} - 2^{8}$	$- 2^{8} - 2^{17} - 2^{23}$
12	—	7	$+ 2^{7} - 2^{8}, + 2^{13} \ldots + 2^{18}, - 2^{19} + 2^{31}$	$+ 2^{7} + 2^{17} + 2^{31}$	$+ 2^{0} + 2^{6} + 2^{17}$
13	—	12	$- 2^{24} + 2^{25} + 2^{31}$	$- 2^{13} + 2^{31}$	$- 2^{12}$
14	2^{31}	17	$+ 2^{31}$	$+ 2^{18} + 2^{31}$	$+ 2^{18} - 2^{30}$
15	—	22	$+ 2^{3} - 2^{15} + 2^{31}$	$+ 2^{25} + 2^{31}$	$- 2^{7} - 2^{13} + 2^{25}$
16	—	5	$- 2^{29} + 2^{31}$	$+ 2^{31}$	$+ 2^{24}$
17	—	9	$+ 2^{31}$	$+ 2^{31}$	—
18	2^{15}	14	$+ 2^{31}$	$+ 2^{31}$	$+ 2^{3}$
19	—	20	$+ 2^{17} + 2^{31}$	$+ 2^{31}$	$- 2^{29}$
20	—	5	$+ 2^{31}$	$+ 2^{31}$	—
21	—	9	$+ 2^{31}$	$+ 2^{31}$	—
22	—	14	$+ 2^{31}$	$+ 2^{31}$	$+ 2^{17}$
23	2^{31}	20	—	—	—
24	—	5	—	$+ 2^{31}$	—
25	2^{31}	9	—	—	—
26 - 33	—	·	—	—	—
34	2^{15}	16	—	—	$+ 2^{15}$
35	2^{31}	23	$\delta Q_{35} = 2^{31}$	$+ 2^{31}$	—
36	—	4	$\delta Q_{36} = 2^{31}$	—	—
37	2^{31}	11	$\delta Q_{37} = 2^{31}$	$+ 2^{31}$	—
38 - 49	—	·	$\delta Q_t = 2^{31}$	$+ 2^{31}$	—
50	2^{31}	15	$\delta Q_{50} = 2^{31}$	—	—
51 - 59	—	·	$\delta Q_t = 2^{31}$	$+ 2^{31}$	—
60	2^{31}	6	$\delta Q_{60} = 2^{31}$	—	—
61	2^{15}	10	$\delta Q_{61} = 2^{31}$	$+ 2^{31}$	$+ 2^{15}$
62	—	15	$\delta Q_{62} = 2^{31} + 2^{25}$	$+ 2^{31}$	—
63	—	21	$\delta Q_{63} = 2^{31} + 2^{25}$	$+ 2^{31}$	—
64	x	x	$\delta Q_{64} = 2^{31} + 2^{25}$	x	x

Tab. MD5-2: Druhá diferenční cesta dle Wangové a kol.

t	δW_t	R_t	δQ_t	δF_t	δT_t
-3	x	x	$+ 2^{31}$	x	x
-2	x	x	$+ 2^{25} + 2^{31}$	x	x
-1	x	x	$- 2^{25} + 2^{26} + 2^{31}$	x	x
0	—	7	$+ 2^{25} + 2^{31}$	$+ 2^{31}$	—
1	—	12	$+ 2^{25} + 2^{31}$	$+ 2^{31}$	$+ 2^{25}$
2	—	17	$+ 2^{5} + 2^{25} + 2^{31}$	$+ 2^{25}$	$+ 2^{26} + 2^{31}$
3	—	22	$- 2^{5} - 2^{6} + 2^{7} - 2^{11} + 2^{12} - 2^{16} \ldots - 2^{20}$, $+ 2^{21} - 2^{25} \ldots - 2^{29}$, $+ 2^{30} + 2^{31}$	$- 2^{11} - 2^{21} + 2^{25} - 2^{27} + 2^{31}$	$- 2^{11} - 2^{21} - 2^{26}$
4	$+ 2^{31}$	7	$+ 2^{1} + 2^{2} + 2^{3} - 2^{4} + 2^{5} - 2^{25} + 2^{26} + 2^{31}$	$+ 2^{1} - 2^{3} - 2^{18} + 2^{26} + 2^{30}$	$+ 2^{1} + 2^{2} - 2^{18} + 2^{25} + 2^{26} + 2^{30}$
5	—	12	$+ 2^{0} - 2^{6} + 2^{7} + 2^{8} - 2^{9} - 2^{10} - 2^{11} + 2^{12} + 2^{31}$	$- 2^{4} - 2^{5} - 2^{8} - 2^{20} - 2^{25} - 2^{26} + 2^{28} + 2^{30}$	$- 2^{4} - 2^{8} - 2^{20} - 2^{26} + 2^{28} - 2^{30}$
6	—	17	$+ 2^{16} - 2^{17} + 2^{20} - 2^{21} + 2^{31}$	$+ 2^{3} - 2^{5} - 2^{10} - 2^{11} - 2^{16} - 2^{21} - 2^{25}$	$+ 2^{3} - 2^{10} - 2^{21} - 2^{31}$
7	—	22	$+ 2^{6} + 2^{7} + 2^{8} - 2^{9} + 2^{27} - 2^{28} + 2^{31}$	$+ 2^{16} - 2^{27} + 2^{31}$	$- 2^{1} + 2^{5} + 2^{16} + 2^{25} - 2^{27}$
8	—	7	$- 2^{15} + 2^{16} - 2^{17} + 2^{23} + 2^{24} + 2^{25} - 2^{26} + 2^{31}$	$- 2^{6} + 2^{16} + 2^{25}$	$+ 2^{0} + 2^{8} + 2^{9} + 2^{16} + 2^{25} - 2^{31}$
9	—	12	$- 2^{0} + 2^{1}$, $- 2^{6} \ldots - 2^{8}$, $+ 2^{9} + 2^{31}$	$+ 2^{0} + 2^{16} - 2^{26} + 2^{31}$	$+ 2^{0} - 2^{20} - 2^{26}$
10	—	17	$+ 2^{12} + 2^{31}$	$+ 2^{6} + 2^{31}$	$- 2^{27}$
11	$- 2^{15}$	22	$+ 2^{31}$	$+ 2^{31}$	$- 2^{17} - 2^{23}$
12	—	7	$- 2^{7}$, $+ 2^{13} \ldots + 2^{18} - 2^{19} + 2^{31}$	$+ 2^{17} + 2^{31}$	$+ 2^{0} + 2^{6} + 2^{17}$
13	—	12	$- 2^{24} \ldots - 2^{29}$, $+ 2^{30} + 2^{31}$	$- 2^{13} + 2^{31}$	$- 2^{12}$
14	$+ 2^{31}$	17	$+ 2^{31}$	$+ 2^{18} + 2^{30}$	$+ 2^{18} + 2^{30}$
15	—	22	$+ 2^{3} + 2^{15} + 2^{31}$	$- 2^{25} + 2^{31}$	$- 2^{7} - 2^{13} - 2^{25}$
16	—	5	$- 2^{29} + 2^{31}$	$+ 2^{31}$	$+ 2^{24}$
17	—	9	$+ 2^{31}$	$+ 2^{31}$	—
18	$- 2^{15}$	14	$+ 2^{31}$	$+ 2^{31}$	$+ 2^{3}$
19	—	20	$+ 2^{17} + 2^{31}$	$+ 2^{31}$	$- 2^{29}$
20	—	5	$+ 2^{31}$	$+ 2^{31}$	—
21	—	9	$+ 2^{31}$	$+ 2^{31}$	—
22	—	14	$+ 2^{31}$	$+ 2^{31}$	$+ 2^{17}$
23	$+ 2^{31}$	20	—	—	—
24	—	5	—	$+ 2^{31}$	—
25	$+ 2^{31}$	9	—	—	—
26 - 33	—	·	—	—	—
34	$- 2^{15}$	16	—	—	$- 2^{15}$
35	$+ 2^{31}$	23	$\delta Q_{35} = 2^{31}$	$+ 2^{31}$	—
36	—	4	$\delta Q_{36} = 2^{31}$	—	—
37	$+ 2^{31}$	11	$\delta Q_{37} = 2^{31}$	$+ 2^{31}$	—
38 - 49	—	·	$\delta Q_t = 2^{31}$	$+ 2^{31}$	—
50	$+ 2^{31}$	15	$\delta Q_{50} = 2^{31}$	—	—
51 - 59	—	·	$\delta Q_t = 2^{31}$	$+ 2^{31}$	—
60	$+ 2^{31}$	6	$\delta Q_{60} = 2^{31}$	—	—
61	$- 2^{15}$	10	$\delta Q_{61} = 2^{31}$	$+ 2^{31}$	$- 2^{15}$
62	—	15	$\delta Q_{62} = 2^{31} - 2^{25}$	$+ 2^{31}$	—
63	—	21	$\delta Q_{63} = 2^{31} - 2^{25}$	$+ 2^{31}$	—
64	x	x	$\delta Q_{64} = 2^{31} - 2^{25}$	x	x

Tab. MD5-3: Postačující podmínky pro jednotlivé bity prvního bloku zprávy (Wangová a kol. + rozšíření)

t	Podmínky pro $Q_t = b_{31}...b_0$
3	....0... ....0... .0......
4	1....... 0^^^1^^^ ^^^^1^^^ ^011....
5	1000100. 01..0000 00000000 0010.1.1
6	0000001^ 01111111 10111100 0100^0^1
7	00000011 11111110 11111000 00100000
8	00000001 1..10001 0.0.0101 01000000
9	11111011 ...10000 0.1^1111 00111101
10	0111.... 0..11111 1101...0 01....00
11	0010.... ....0001 1100...0 11....10
12	000...^^ ....1000 0001...1 0.......
13	01....01 ....1111 111....0 0...1...
14	0.0...00 ....1011 111....1 1...1...
15	0.1...01 ....0 1....... ....0...
16	0!1..... ......!. ....
17	0!...... ......0. ^....... ....^...
18	0.^..... ......1. ....
19	0....... ......0. ....
20	0....... .....!.. ....
21	0....... ......^. ....
22	0....... ....
23	0....... ....
24	1....... ....
25 - 45	....
46	I....... ....
47	J....... ....
48	I....... ....
49	J....... ....
50	K....... ....
51	J....... ....
52	K....... ....
53	J....... ....
54	K....... ....
55	J....... ....
56	K....... ....
57	J....... ....
58	K....... ....
59	J....... ....
60	I....... ....
61	J....... ....
62	I....... ....
63	J....... ....
64	....

Poznámka: $I, J, K \in \{0,1\}$ *a zároveň platí* $K = \bar{I}$.

Tab. MD5-4: Postačující podmínky pro jednotlivé bity druhého bloku zprávy (Wangová a kol. + rozšíření

t	Podmínky pro $Q_t = b_{31}...b_0$
-2	A.....0. ....
-1	A....01. ....
0	A....00. ....0.....
1	B...010. ..1....0 ....0... .10.....
2	B^^^110. ..0^^^^0 1..^1... ^10..00.
3	B011111. ..011111 0..01..1 011^^11.
4	B011101. ..000100 ...00^^0 0001000^
5	A10010.. ..101111 ...01110 01010000
6	A..0010. 1.10..10 11.01100 01010110
7	B..1011^ 1.00..01 10.11110 00.....1
8	B..00100 0.11..10 1.....11 111...^0
9	B..11100 0.....01 0..^..01 110...01
10	B....111 1....011 11001.11 11....00
11	B....... ....^101 11000.11 11....11
12	B^^^^^^^ ....1000 0001.... 1.......
13	A0111111 0...1111 111..... 0...1...
14	A1000000 1...1011 111..... 1...1...
15	01111101 .... 00...... ....0...
16	0.10.... ......!. ....
17	0!...... ......0. ^....... ....^...
18	0.^..... ......1. ....
19	0....... ......0. ....
20	0....... .....!.. ....
21	0....... ......^. ....
22	0....... ....
23	0....... ....
24	1....... ....
25 - 45	....
46	I....... ....
47	J....... ....
48	I....... ....
49	J....... ....
50	K....... ....
51	J....... ....
52	K....... ....
53	J....... ....
54	K....... ....
55	J....... ....
56	K....... ....
57	J....... ....
58	K....... ....
59	J....... ....
60	I....... ....
61	J....... ....
62	I....... ....
63	J....... ....
64	....

Poznámka: $A, B, I, J, K \in \{0,1\}$ *a zároveň platí* $B = \overline{A}$ *a* $K = \overline{I}$.

Tab. MD5-5: Nová diferenční cesta pro první blok zprávy.

t	δW_t	R_t	δQ_t	δF_t	δT_t
0 - 3	—	·	—	—	—
4	2^{31}	7	—	—	2^{31}
5	—	12	$- 2^{6} \ldots - 2^{24} + 2^{25}$	$- 2^{8} + 2^{14} - 2^{19} - 2^{23} + 2^{25}$	$- 2^{8} + 2^{14} - 2^{19} - 2^{23} + 2^{25}$
6	—	17	$+ 2^{0} - 2^{1} + 2^{3} - 2^{4} + 2^{5} - 2^{6} - 2^{7} + 2^{8} + 2^{20} + 2^{21} - 2^{22} + 2^{26} - 2^{31}$	$+ 2^{3} - 2^{9} + 2^{15} + 2^{18} - 2^{20} - 2^{22}$	$+ 2^{3} - 2^{9} + 2^{15} + 2^{18} - 2^{20} - 2^{22}$
7	—	22	$- 2^{6} + 2^{31}$	$- 2^{0} + 2^{6} - 2^{10} + 2^{13} - 2^{25}$	$- 2^{0} + 2^{6} - 2^{10} + 2^{13} - 2^{25}$
8	—	7	$- 2^{0} + 2^{3} - 2^{6} - 2^{15} - 2^{22} + 2^{28} + 2^{31}$	$- 2^{5} + 2^{8} + 2^{15} - 2^{21} + 2^{26} - 2^{28}$	$2^{5} + 2^{8} + 2^{15} - 2^{21} + 2^{26} - 2^{28}$
9	—	12	$+ 2^{0} - 2^{6} + 2^{12} + 2^{31}$	$- 2^{0} + 2^{3} - 2^{6} + 2^{31}$	$- 2^{1} + 2^{5} - 2^{20} + 2^{26}$
10	—	17	$- 2^{12} + 2^{17} + 2^{31}$	$2^{0} - 2^{6} + 2^{12} + 2^{31}$	$2^{0} - 2^{7} + 2^{12}$
11	2^{15}	22	$- 2^{12} + 2^{18} - 2^{24} + 2^{29} + 2^{31}$	$2^{0} - 2^{6} - 2^{17} - 2^{29} + 2^{31}$	$2^{3} - 2^{7} - 2^{17} - 2^{22} - 2^{28}$
12	—	7	$- 2^{7} - 2^{13} + 2^{24} + 2^{31}$	$+ 2^{7} - 2^{12} + 2^{31}$	$2^{0} + 2^{6}$
13	—	12	$+ 2^{24} + 2^{31}$	2^{31}	$- 2^{12} + 2^{17}$
14	2^{31}	17	$+ 2^{29} + 2^{31}$	$+ 2^{24} + 2^{29} + 2^{31}$	$- 2^{12} + 2^{18} - 2^{30}$
15	—	22	$+ 2^{3} - 2^{15} - 2^{31}$	$+ 2^{24} + 2^{31}$	$- 2^{7} - 2^{13} + 2^{25}$
16	—	5	$- 2^{29} - 2^{31}$	2^{31}	2^{24}
17	—	9	$- 2^{31}$	$- 2^{29} + 2^{31}$	—
18	2^{15}	14	$- 2^{31}$	2^{31}	2^{3}
19	—	20	$+ 2^{17} - 2^{31}$	2^{31}	$- 2^{29}$
20	—	5	$- 2^{31}$	2^{31}	—
21	—	9	$- 2^{31}$	2^{31}	—
22	—	14	$- 2^{31}$	2^{31}	2^{17}
23	2^{31}	20	—	—	—
24	—	5	—	2^{31}	—
25	2^{31}	9	—	—	—

Tab. MD5-6: Postačující podmínky pro novou diferenční cestu.

t	Podmínky pro $Q_t = b_{31}...b_0$
3	.......1 .1111... .1....01 11......
4	0.....^0 ^0000^^^ ^0^^^^10 11..0...
5	01...^01 11111111 11111111 11^^1.^^
6	10.1.000 01001011 10000010 11010.10
7	0..0.010 01000000 00011011 .1000.11
8	0!.0.0.. .101.... 1..1...0 11010.11
9	0!10...0 .0...1^. 0..0.... 011.1..0
10	0.01...0 .1...00. 1..1.... 1...1..1
11	0!0....1 .....01. ..^1.... 00.....0
12	0!0....0 ..!..01. ..1..... 1.......
13	0.1....0 .....1.. 1.01.... 0...1...
14	0!0..... .... 1.1..... 1...1...
15	1.0....0 ....! 1....... ....0...
16	1!1..... ......!. ....
17	1!...... ......0. ^....... ....^...
18	1.^..... ......1. ....
19	1....... ......0. ....
20	1....... .....!.. ....
21	1....... ......^. ....
22	1....... ....
23	0....... ....
24	1....... ....

Tab. MD5-7: Nová diferenční cesta pro druhý blok zprávy, číslo 1.

t	δW_t	R_t	δQ_t	δF_t	δT_t
-3	x	x	$+ 2^{31}$	x	x
-2	x	x	$+ 2^{25} + 2^{31}$	x	x
-1	x	x	$+ 2^{25} + 2^{31}$	x	x
0	—	7	$+ 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{25}
1	—	12	$+ 2^{0} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
2	—	17	$+ 2^{0} + 2^{6} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
3	—	22	$+ 2^{0} + 2^{6} + 2^{11} + 2^{25} + 2^{31}$	$2^{0} - 2^{11} + 2^{25} + 2^{31}$	$2^{0} - 2^{11} + 2^{26}$
4	$+ 2^{31}$	7	$+ 2^{0} - 2^{1} - 2^{6} - 2^{7} + 2^{8} + 2^{11} + 2^{16} + 2^{22} - 2^{25} - 2^{26} + 2^{27} + 2^{31}$	$2^{0} - 2^{6} + 2^{8} + 2^{11} - 2^{16} - 2^{22} + 2^{25} - 2^{27} + 2^{31}$	$2^{1} - 2^{6} + 2^{8} + 2^{11} - 2^{16} - 2^{22} - 2^{26} + 2^{31}$
5	—	12	$+ 2^{0} + 2^{2} + 2^{3} + 2^{4} - 2^{5} + 2^{8} + 2^{11} - 2^{13} - 2^{15} - 2^{17} + 2^{19} + 2^{22} + 2^{23} + 2^{24} + 2^{29} + 2^{30}$	$2^{0} + 2^{3} - 2^{5} + 2^{7} + 2^{11} + 2^{16} - 2^{22} + 2^{25} - 2^{27} + 2^{31}$	$2^{1} + 2^{3} + 2^{5} + 2^{7} + 2^{11} + 2^{16} + 2^{19} + 2^{22} - 2^{29} + 2^{31}$
6	—	17	$+ 2^{1} + 2^{8} + ... + 2^{17} - 2^{18} - 2^{20} + 2^{21} - 2^{22} - 2^{23} - 2^{24} + 2^{26} + 2^{28} + 2^{29} + 2^{30} + 2^{31}$	$- 2^{0} - 2^{3} + 2^{5} + 2^{7} + 2^{9} + 2^{11} - 2^{13} + 2^{15} - 2^{17} - 2^{21} + 2^{23} - 2^{25} + 2^{28} + 2^{31}$	$- 2^{3} - 2^{5} - 2^{8} + 2^{10} - 2^{12} + 2^{15} - 2^{17} - 2^{21} + 2^{23} + 2^{28}$
7	—	22	$- 2^{0} - 2^{6} + 2^{13} - 2^{27} - 2^{29}$	$- 2^{0} + 2^{3} + 2^{5} + 2^{7} - 2^{10} - 2^{13} + 2^{18} + 2^{22} - 2^{24} + 2^{27} - 2^{29} + 2^{31}$	$- 2^{1} + 2^{3} - 2^{5} + 2^{8} + 2^{10} - 2^{13} + 2^{16} + 2^{18} - 2^{23} + 2^{25} + 2^{27} - 2^{29}$
8	—	7	$- 2^{3} + 2^{8} + 2^{15} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$	$2^{0} + 2^{9} + 2^{13} - 2^{15} - 2^{19} + 2^{21} + 2^{26} - 2^{28} + 2^{30}$	$- 2^{1} - 2^{8} - 2^{10} + 2^{12} + 2^{16} - 2^{18} - 2^{21} - 2^{25} - 2^{27} + 2^{29} + 2^{31}$
9	—	12	$- 2^{0} + 2^{2} - 2^{6}$	$- 2^{0} + 2^{8} - 2^{23} + 2^{25} + 2^{28}$	$2^{0} + 2^{20} - 2^{22} + 2^{26}$
10	—	17	$+ 2^{12}$	$- 2^{0} - 2^{6} + 2^{8}$	$- 2^{1} + 2^{7} + 2^{13} - 2^{27} - 2^{29}$
11	$- 2^{15}$	22	$- 2^{14} - 2^{18} + 2^{24} + 2^{30}$	—	$- 2^{3} + 2^{8} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$
12	—	7	$+ 2^{7} - 2^{9} + 2^{13} - 2^{24} - 2^{31}$	—	$- 2^{0} + 2^{2} - 2^{6}$
13	—	12	$- 2^{24} - 2^{31}$	—	2^{12}
14	$+ 2^{31}$	17	$- 2^{31}$	$- 2^{24} + 2^{31}$	$- 2^{14} - 2^{18} + 2^{30}$
15	—	22	$- 2^{3} + 2^{15}$	$- 2^{24} + 2^{31}$	$2^{7} - 2^{9} + 2^{13} - 2^{25}$
16	—	5	$+ 2^{29} - 2^{31}$	2^{31}	$- 2^{24}$
17	—	9	$- 2^{31}$	2^{31}	—
18	$- 2^{15}$	14	$+ 2^{31}$	—	$- 2^{3}$
19	—	20	$- 2^{17} + 2^{31}$	2^{31}	2^{29}
20	—	5	$+ 2^{31}$	2^{31}	—
21	—	9	$+ 2^{31}$	2^{31}	—
22	—	14	$+ 2^{31}$	2^{31}	$- 2^{17}$
23	$+ 2^{31}$	20	—	—	—
24	—	5	—	2^{31}	—
25	$+ 2^{31}$	9	—	—	—

Tab. MD5-8: Postačující podmínky pro novou diferenční cestu, číslo 1.

t	Podmínky pro $Q_t = b_{31}...b_0$
-2	......0. ....
-1	^.....0. 1
0	^.....0. .... .1.....1
1	^.....0. ....1... .1!....0
2	^...1.0. .1.....1 ....0..0 00.....0
3	^00.0^00 00..0.10 1.1.0..1 101.0.^0
4	01100110 000.1.10 1.1.0.00 110^1^10
5	.0010100 001^0^11 1^1^0^10 00100000
6	^0001001 11010100 00000000 01011000
7	0.111001 01001011 1101.100 .1011011
8	10100001 11011000 01.11100 .00.1001
9	.1111.10 1...0.0. 0.11...1 .11.00.1
10	1111..10 1...1^1. 1^.0...0 .1..10.1
11	100....0 .....1.! .1^0..^. ^1...1.1
12	.01....1 ..!..0.. .001..1. 0.......
13	^1.....1 .....1.. 110...0. 0...1...
14	100..... .... 1.1...1. 1...1...
15	001....0 ....! 0....... ....1...
16	1!0..... ......!. ....
17	1!..... ......0. ^....... ....^...
18	0.^..... ......1. ....
19	0....... ......1. ....
20	0....... .....!.. ....
21	0....... ......^. ....
22	0....... ....
23	0....... ....
24	1....... ....

Tab. MD5-9: Nová diferenční cesta pro druhý blok zprávy, číslo 2.

t	δW_t	R_t	δQ_t	δF_t	δT_t
-3	x	x	$+ 2^{31}$	x	x
-2	x	x	$+ 2^{25} + 2^{31}$	x	x
-1	x	x	$+ 2^{25} + 2^{31}$	x	x
0	—	7	$+ 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{25}
1	—	12	$+ 2^{0} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
2	—	17	$+ 2^{0} + 2^{6} + 2^{25} + 2^{31}$	$2^{0} + 2^{25} + 2^{31}$	$2^{0} + 2^{26}$
3	—	22	$+ 2^{0} + 2^{6} + 2^{11} + 2^{17} + 2^{25} + 2^{31}$	$2^{0} + 2^{25} + 2^{31}$	$2^{0} + 2^{26}$
4	$+ 2^{31}$	7	$- 2^{0} - 2^{1} + 2^{2} - 2^{6} - 2^{7} + 2^{8} - 2^{11} - 2^{12} + 2^{13} - 2^{16} - 2^{18} + 2^{19} + 2^{22} - 2^{25} + 2^{26} + 2^{31}$	$2^{0} + 2^{2} + 2^{6} + 2^{8} - 2^{11} + 2^{16} - 2^{18} - 2^{22} - 2^{25} + 2^{27} + 2^{31}$	$- 2^{1} + 2^{3} + 2^{6} + 2^{8} - 2^{11} + 2^{16} - 2^{18} - 2^{22} + 2^{27} + 2^{31}$
5	—	12	$+ 2^{0} - 2^{2} - 2^{3} - 2^{4} + 2^{5} - 2^{8} - 2^{10} - 2^{12} + 2^{14} - 2^{15} + 2^{22} - 2^{23} + 2^{24} + 2^{29} + 2^{30}$	$2^{2} - 2^{6} + 2^{8} - 2^{11} - 2^{14} + 2^{16} + 2^{19} - 2^{22} - 2^{25} - 2^{30}$	$2^{0} + 2^{2} + 2^{8} - 2^{11} - 2^{14} + 2^{16} + 2^{19} - 2^{22} + 2^{30}$
6	—	17	$- 2^{1} - 2^{2} + 2^{3}, + 2^{8} + ... + 2^{15} - 2^{16} - 2^{20} + 2^{21} + 2^{22} + 2^{26} + 2^{27} + 2^{29} + 2^{30} - 2^{31}$	$- 2^{0} - 2^{3} + 2^{5} - 2^{7} - 2^{9} + 2^{11} - 2^{13} + 2^{15} + 2^{17} - 2^{19} - 2^{22} - 2^{24} + 2^{31}$	$- 2^{3} - 2^{5} - 2^{9} - 2^{12} + 2^{15} - 2^{18} - 2^{22} + 2^{24}$
7	—	22	$- 2^{0} - 2^{2} + 2^{7} + 2^{27} - 2^{30}$	$2^{0} + 2^{2} - 2^{5} + 2^{8} - 2^{10} - 2^{13} - 2^{16} - 2^{22} - 2^{24} + 2^{27} - 2^{29} + 2^{31}$	$- 2^{1} + 2^{3} + 2^{5} + 2^{8} + 2^{10} - 2^{13} + 2^{17} + 2^{25} + 2^{27} - 2^{29}$
8	—	7	$+ 2^{2} - 2^{4} + 2^{8} + 2^{15} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$	$2^{0} - 2^{3} - 2^{13} - 2^{15} - 2^{17} + 2^{21} + 2^{23} - 2^{26} + 2^{30}$	$- 2^{1} - 2^{8} - 2^{10} + 2^{12} + 2^{16} - 2^{18} - 2^{21} - 2^{23} - 2^{25} + 2^{29} + 2^{31}$
9	—	12	$- 2^{0} + 2^{2} - 2^{6} - 2^{30}$	$- 2^{0} + 2^{8} - 2^{19} - 2^{23} - 2^{27} + 2^{29}$	$2^{0} + 2^{19} - 2^{22} + 2^{26}$
10	—	17	$+ 2^{12} + 2^{30}$	$- 2^{0} - 2^{2} + 2^{28}$	$- 2^{1} + 2^{7} - 2^{27} - 2^{29}$
11	$- 2^{15}$	22	$- 2^{14} - 2^{18} + 2^{24} + 2^{30}$	2^{2}	$- 2^{3} + 2^{8} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$
12	—	7	$+ 2^{7} - 2^{9} + 2^{13} - 2^{24} - 2^{31}$	2^{30}	$- 2^{0} + 2^{2} - 2^{6}$
13	—	12	$- 2^{24} - 2^{31}$	$- 2^{30}$	2^{12}
14	$+ 2^{31}$	17	$+ 2^{31}$	$- 2^{24} + 2^{31}$	$- 2^{14} - 2^{18} + 2^{30}$
15	—	22	$- 2^{3} + 2^{15}$	$- 2^{24} + 2^{31}$	$2^{7} - 2^{9} + 2^{13} - 2^{25}$
16	—	5	$+ 2^{29} - 2^{31}$	2^{31}	$- 2^{24}$
17	—	9	$+ 2^{31}$	2^{31}	—
18	$- 2^{15}$	14	$- 2^{31}$	—	$- 2^{3}$
19	—	20	$- 2^{17} - 2^{31}$	2^{31}	2^{29}
20	—	5	$- 2^{31}$	2^{31}	—
21	—	9	$- 2^{31}$	2^{31}	—
22	—	14	$- 2^{31}$	2^{31}	$- 2^{17}$
23	$+ 2^{31}$	20	—	—	—
24	—	5	—	2^{31}	—
25	$+ 2^{31}$	9	—	—	—

Tab. MD5-10: Postačující podmínky pro novou diferenční cestu, číslo 2.

t	Podmínky pro $Q_t = b_{31}...b_0$
-2	......0. ....
-1	^.....0. 0
0	^.....0. .... .1.....0
1	^.....0. ......0. ....1..1 .1.....0
2	^....00. .1..1100 ..111..0 .0...0.0
3	^01..10. 00..0001 000000.1 ^01.11^0
4	011.001^ 10..0111 11011010 11^^1011
5	000.0010 10^^1001 10010101 01011100
6	10010001 10011011 00000000 01100110
7	01.00001 0.001.01 0.011110 01100101
8	1010.00. 11011.00 00011110 01111001
9	01.10.00 1...1.0. 01.10..1 11.0.0.1
10	00.0..10 1...1^1. 11.01..1 .0.1...1
11	00!....0 .....1.. .1^00.^. ^1...0.1
12	10.....1 ....!0.. .001..1. 0.......
13	10.....1 .....1.. 110...0. 0...1...
14	0.0..... .... 1.1...1. 1...1...
15	111....0 ....! 0....... ....1...
16	1.0..... ......!. ....
17	0....... ......0. ^....... ....^...
18	1.^..... ......1. ....
19	1....... ......1. ....
20	1....... .....!.. ....
21	1....... ......^. ....
22	1....... ....
23	0....... ....
24	1....... ....

Tab. MD5-11: Nová diferenční cesta pro druhý blok zprávy, číslo 3.

t	δW_t	R_t	δQ_t	δF_t	δT_t
-3	x	x	$+ 2^{31}$	x	x
-2	x	x	$+ 2^{25} + 2^{31}$	x	x
-1	x	x	$+ 2^{25} + 2^{31}$	x	x
0	—	7	$+ 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{25}
1	—	12	$+ 2^{0} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
2	—	17	$+ 2^{0} + 2^{6} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
3	—	22	$+ 2^{0} + 2^{6} + 2^{11} + 2^{25} + 2^{31}$	$2^{0} + 2^{6} + 2^{25} + 2^{31}$	$2^{0} + 2^{6} + 2^{26}$
4	$+ 2^{31}$	7	$- 2^{0} - 2^{1}, - 2^{6} \ldots - 2^{10} + 2^{12} - 2^{16} - 2^{17} - 2^{18} + 2^{19} - 2^{22} + 2^{23} + 2^{25} - 2^{28} + 2^{29} - 2^{31}$	$2^{0} - 2^{6} + 2^{9} - 2^{11} + 2^{13} + 2^{17} + 2^{19} + 2^{22} + 2^{25} - 2^{29} + 2^{31}$	$2^{1} - 2^{6} + 2^{9} - 2^{11} + 2^{13} + 2^{17} + 2^{19} + 2^{22} + 2^{26} - 2^{29} + 2^{31}$
5	—	12	$- 2^{0} + 2^{2} + 2^{4} - 2^{5} + 2^{8} + 2^{11} + 2^{13} + 2^{14} + 2^{15} - 2^{16} + 2^{17} + 2^{18} - 2^{19} - 2^{20} + 2^{21} + 2^{22} - 2^{24} + 2^{27} - 2^{28} + 2^{30} + 2^{31}$	$2^{1} - 2^{5} + 2^{11} - 2^{14} + 2^{18} - 2^{21} + 2^{24} + 2^{30}$	$- 2^{0} + 2^{2} + 2^{5} + 2^{11} - 2^{14} + 2^{18} - 2^{21} - 2^{24} + 2^{26} - 2^{30}$
6	—	17	$- 2^{0} \ldots - 2^{3} + 2^{4}, - 2^{5} - 2^{6} + 2^{7} + 2^{8} + 2^{10}, - 2^{12} \ldots - 2^{18} + 2^{19} + 2^{20} - 2^{22} + 2^{26} - 2^{28}$	$- 2^{0} - 2^{3} - 2^{5} + 2^{9} + 2^{11} - 2^{13} - 2^{20} - 2^{23} + 2^{26} - 2^{28} + 2^{31}$	$- 2^{3} + 2^{5} + 2^{9} - 2^{12} - 2^{20} - 2^{23} - 2^{25} - 2^{27}$
7	—	22	$- 2^{0} - 2^{27} - 2^{29}$	$2^{0} + 2^{3} + 2^{5} + 2^{7} + 2^{10} + 2^{12} - 2^{14} + 2^{16} - 2^{19} - 2^{22} - 2^{24} - 2^{27} - 2^{29} + 2^{31}$	$2^{1} + 2^{3} - 2^{5} + 2^{8} - 2^{10} - 2^{13} + 2^{17} - 2^{19} + 2^{25} + 2^{27} - 2^{29}$
8	—	7	$- 2^{3} + 2^{7} - 2^{9} + 2^{15} + 2^{17} - 2^{19} + 2^{23} + 2^{25} + 2^{28}$	$- 2^{0} + 2^{4} + 2^{9} + 2^{13} + 2^{15} + 2^{17} + 2^{19} + 2^{23} + 2^{25} + 2^{28}$	$2^{1} - 2^{8} - 2^{10} + 2^{12} + 2^{15} - 2^{19} - 2^{21} - 2^{25} - 2^{27} + 2^{29} + 2^{31}$
9	—	12	$- 2^{0} + 2^{2} - 2^{6} - 2^{22} - 2^{24}$	$2^{7} - 2^{9} + 2^{28}$	$2^{0} + 2^{5} - 2^{7} + 2^{10} + 2^{12} + 2^{20} - 2^{22} + 2^{26}$
10	—	17	$+ 2^{12} + 2^{17} - 2^{19}$	$2^{7} - 2^{12}$	$2^{0} + 2^{7} - 2^{12} - 2^{27} - 2^{29}$
11	$- 2^{15}$	22	$- 2^{14} - 2^{18} + 2^{24} - 2^{29}$	$- 2^{24}$	$- 2^{3} + 2^{7} - 2^{9} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$
12	—	7	$+ 2^{7} - 2^{9} + 2^{13} - 2^{24} - 2^{31}$	—	$- 2^{0} + 2^{2} - 2^{6} - 2^{22} - 2^{24}$
13	—	12	$- 2^{24} - 2^{29}$	—	$2^{12} + 2^{17} - 2^{19}$
14	$+ 2^{31}$	17	$- 2^{31}$	$- 2^{24} - 2^{29}$	$- 2^{14} - 2^{18} + 2^{30}$
15	—	22	$- 2^{3} + 2^{15}$	$- 2^{24} + 2^{31}$	$2^{7} - 2^{9} + 2^{13} - 2^{25}$
16	—	5	$+ 2^{29} - 2^{31}$	2^{29}	$- 2^{24}$
17	—	9	$+ 2^{31}$	2^{31}	—
18	$- 2^{15}$	14	$- 2^{31}$	—	$- 2^{3}$
19	—	20	$- 2^{17} - 2^{31}$	2^{31}	2^{29}
20	—	5	$- 2^{31}$	2^{31}	—
21	—	9	$- 2^{31}$	2^{31}	—
22	—	14	$- 2^{31}$	2^{31}	$- 2^{17}$
23	$+ 2^{31}$	20	—	—	—
24	—	5	—	2^{31}	—
25	$+ 2^{31}$	9	—	—	—

Tab. MD5-12: Postačující podmínky pro novou diferenční cestu, číslo 3.

t	Podmínky pro $Q_t = b_{31}...b_0$
-2	......0. ....
-1	^.....0. 1
0	^.....0. .... .0.....1
1	^.....0. ....00.. .0!....0
2	^.11..0. .1..0110 ...00100 00.....0
3	^001..00 ^0111100 00110010 1011.0^0
4	!101^001 01010111 11101111 11010001
5	.00101.1 00011001 00000000 10101011
6	.0110011 01000111 11110000 01101111
7	01111001 1110101. 1001111. 01..0110
8	1.100101 01.01000 0001.011 01101101
9	..111.01 01..0.0. 0..0..1. 111100.1
10	1011..10 10..1^0. 1^.0..1. 00..10.0
11	111....0 .1..010! .1^0..1. 01...1.1
12	100....1 ....101. .001..1. 0.......
13	011....1 .....1.. 110...0. 0...1...
14	111..... .... 1.1...1. 1...1...
15	101....0 ....! 0....... ....1...
16	100..... ......!. ....
17	0....... ......0. ^....... ....^...
18	1.^..... ......1. ....
19	1....... ......1. ....
20	1....... .....!.. ....
21	1....... ......^. ....
22	1....... ....
23	0....... ....
24	1....... ....

Tab. MD5-13: Nová diferenční cesta pro druhý blok zprávy, číslo 4.

t	δW_t	R_t	δQ_t	δF_t	δT_t
-3	x	x	$+ 2^{31}$	x	x
-2	x	x	$+ 2^{25} + 2^{31}$	x	x
-1	x	x	$+ 2^{25} + 2^{31}$	x	x
0	—	7	$+ 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{25}
1	—	12	$+ 2^{0} + 2^{25} + 2^{31}$	$2^{25} + 2^{31}$	2^{26}
2	—	17	$+ 2^{0} + 2^{6} + 2^{25} + 2^{31}$	$2^{0} + 2^{25} + 2^{31}$	$2^{0} + 2^{26}$
3	—	22	$+ 2^{0} + 2^{6} + 2^{11} + 2^{17} + 2^{25} + 2^{31}$	$2^{0} + 2^{6} - 2^{11} + 2^{25} + 2^{31}$	$2^{0} + 2^{6} - 2^{11} + 2^{26}$
4	$+ 2^{31}$	7	$- 2^{0} - 2^{6} + 2^{7}$, $- 2^{11}$ ... $- 2^{14} + 2^{15} - 2^{16} - 2^{18} - 2^{19} + 2^{20} + 2^{22} - 2^{25} + 2^{26} - 2^{28} + 2^{29} - 2^{31}$	$2^{0} - 2^{6} - 2^{12} - 2^{16} + 2^{18} - 2^{20} + 2^{22} - 2^{25} - 2^{27} + 2^{31}$	$2^{1} - 2^{6} - 2^{12} - 2^{16} + 2^{18} - 2^{20} + 2^{22} - 2^{27} + 2^{31}$
5	—	12	$+ 2^{0} + 2^{1} + 2^{3} + 2^{4} + 2^{5} - 2^{6} - 2^{8} + 2^{9} - 2^{11} - 2^{12} + 2^{16} - 2^{17} + 2^{18}$ ... $+ 2^{21}$, $+ 2^{23} + 2^{24} + 2^{25} - 2^{27} - 2^{28} + 2^{30} - 2^{31}$	$2^{1} - 2^{4} + 2^{8} - 2^{10} + 2^{15} + 2^{18} - 2^{21} + 2^{23} - 2^{26} - 2^{28} + 2^{30}$	$- 2^{0} + 2^{2} - 2^{4} + 2^{6} + 2^{8} - 2^{10} + 2^{15} + 2^{18} - 2^{21} + 2^{23} - 2^{25} - 2^{28} - 2^{30}$
6	—	17	$- 2^{0} + 2^{1} - 2^{5} - 2^{10} - 2^{11} - 2^{13} - 2^{14} + 2^{15} - 2^{17} - 2^{20} + 2^{21} + 2^{23} + 2^{24} - 2^{25} + 2^{26} + 2^{28} - 2^{29}$	$- 2^{3} + 2^{9} + 2^{11} - 2^{13} - 2^{15} - 2^{17} + 2^{20} + 2^{26} - 2^{28} + 2^{31}$	$2^{0} - 2^{3} + 2^{6} + 2^{9} - 2^{12} - 2^{15} + 2^{20} - 2^{25} - 2^{27}$
7	—	22	$- 2^{27} - 2^{29}$	$- 2^{0} - 2^{2} - 2^{4} - 2^{6} + 2^{8} - 2^{11} - 2^{13} - 2^{14} - 2^{16} - 2^{19} - 2^{22} - 2^{27} - 2^{29} + 2^{31}$	$2^{1} + 2^{3} - 2^{5} + 2^{8} - 2^{13} + 2^{17} - 2^{19} + 2^{25} + 2^{27} - 2^{29}$
8	—	7	$- 2^{3} + 2^{7} - 2^{9} + 2^{15} + 2^{17} - 2^{19} + 2^{23} + 2^{25} + 2^{28}$	$- 2^{0} + 2^{3} + 2^{9} + 2^{13} - 2^{15} - 2^{17} + 2^{21} + 2^{25} + 2^{27} - 2^{29}$	$2^{1} - 2^{8} - 2^{10} + 2^{12} + 2^{15} - 2^{19} - 2^{21} - 2^{25} - 2^{27} + 2^{29} + 2^{31}$
9	—	12	$- 2^{0} + 2^{2} - 2^{6} - 2^{22} - 2^{24}$	$2^{17} + 2^{22} + 2^{28}$	$2^{0} - 2^{5} + 2^{10} + 2^{12} + 2^{20} - 2^{22} + 2^{26}$
10	—	17	$+ 2^{12} + 2^{17}$	$- 2^{0} + 2^{7} - 2^{12}$	$- 2^{0} + 2^{7} - 2^{12} - 2^{27} - 2^{29}$
11	$- 2^{15}$	22	$- 2^{14} - 2^{18} + 2^{24} - 2^{29}$	$- 2^{24}$	$- 2^{3} + 2^{7} - 2^{9} + 2^{17} - 2^{19} - 2^{23} + 2^{25} + 2^{28}$
12	—	7	$+ 2^{7} - 2^{9} + 2^{13} - 2^{24} - 2^{31}$	—	$- 2^{0} + 2^{2} - 2^{6} - 2^{22} - 2^{24}$
13	—	12	$- 2^{24} - 2^{29}$	$- 2^{18}$	$2^{12} + 2^{17} - 2^{19}$
14	$+ 2^{31}$	17	$+ 2^{31}$	$- 2^{24} - 2^{29}$	$- 2^{14} - 2^{18} + 2^{30}$
15	—	22	$- 2^{3} + 2^{15}$	$- 2^{24} + 2^{31}$	$2^{7} - 2^{9} + 2^{13} - 2^{25}$
16	—	5	$+ 2^{29} + 2^{31}$	2^{29}	$- 2^{24}$
17	—	9	$- 2^{31}$	2^{31}	—
18	$- 2^{15}$	14	$+ 2^{31}$	—	$- 2^{3}$
19	—	20	$- 2^{17} + 2^{31}$	2^{31}	2^{29}
20	—	5	$+ 2^{31}$	2^{31}	—
21	—	9	$+ 2^{31}$	2^{31}	—
22	—	14	$+ 2^{31}$	2^{31}	$- 2^{17}$
23	$+ 2^{31}$	20	—	—	—
24	—	5	—	2^{31}	—
25	$+ 2^{31}$	9	—	—	—

Tab. MD5-14: Postačující podmínky pro novou diferenční cestu, číslo 4.

t	Podmínky pro $Q_t = b_{31}...b_0$			
-2	......0.	....	....	....
-1	^.....0.	....	....	0
0	^.....0.	....	....	.0.....0
1	^.....0.	......1.	....1...	.0.....0
2	^.10.00.	.0.11111	...00...	10.....0
3	^0111101	11001100	^^^10.10	0011..00
4	!0010011	10101111	01111001	0110^.11
5	10111100	01000010	10011001	01000.00
6	00100010	01010111	011.1110	1.111.01
7	10111101	00.00100	0011.000	10110..0
8	0..01001	01011.0.	0001111.	001.1^10
9	1.111.01	11..0.1.	0..0..0.	010.00.1
10	1111..10	10..1^1.	1^.0..1.	00..10.1
11	101....0	.1...10!	.1^0..1.	01...1.1
12	010....1	..!..01.	.001..1.	0.......
13	0011...1	.....0..	110...0.	0...1...
14	0010....	....	1.1...1.	1...1...
15	1110...0	.......!	0.......	....1...
16	0101....	......!.	....	....
17	1.......	......0.	^.......	....^...
18	0.^.....	......1.	....	....
19	0.......	......1.	....	....
20	0.......	.....!..	....	....
21	0.......	......^.	....	....
22	0.......	....	....	....
23	0.......	....	....	....
24	1.......	....	....	....

9.2. Tabulky pro funkci SHA-1

Tab. SHA1-1: Poruchové vektory pro SHA-1. Poruchové vektory splňují pouze rovnici pro expanzi slov vstupní zprávy. Použité vektory jsou výběrem 80-ti poruchových vektorů, které jsou použity pro konstrukci nejlepší blízké kolize pro plnou funkci SHA-1.

Krok *i*	Použité vektory	Vektor x_i	Krok *i*	Použité vektory	Vektor x_i	Krok *i*	Použité vektory	Vektor x_i
0		e0000000	32	*16*	80000002	64	*48*	2
1		2	33	*17*	0	65	*49*	0
2		2	34	*18*	2	66	*50*	0
3		80000000	35	*19*	0	67	*51*	0
4		1	36	*20*	3	68	*52*	0
5		0	37	*21*	0	69	*53*	0
6		80000001	38	*22*	2	70	*54*	0
7		2	39	*23*	2	71	*55*	0
8		40000002	40	*24*	1	72	*56*	0
9		2	41	*25*	0	73	*57*	0
10		2	42	*26*	2	74	*58*	0
11		80000000	43	*27*	2	75	*59*	0
12		2	44	*28*	1	76	*60*	0
13		0	45	*29*	0	77	*61*	0
14		80000001	46	*30*	0	78	*62*	0
15		0	47	*31*	2	79	*63*	0
16	*0*	40000001	48	*32*	3	80	*64*	4
17	*1*	2	49	*33*	0	81	*65*	0
18	*2*	2	50	*34*	2	82	*66*	0
19	*3*	80000002	51	*35*	2	83	*67*	8
20	*4*	1	52	*36*	0	84	*68*	0
21	*5*	0	53	*37*	0	85	*69*	0
22	*6*	80000001	54	*38*	2	86	*70*	10
23	*7*	2	55	*39*	0	87	*71*	0
24	*8*	2	56	*40*	0	88	*72*	8
25	*9*	2	57	*41*	0	89	*73*	20
26	*10*	0	58	*42*	2	90	*74*	0
27	*11*	0	59	*43*	0	91	*75*	0
28	*12*	1	60	*44*	2	92	*76*	40
29	*13*	0	61	*45*	0	93	*77*	0
30	*14*	80000002	62	*46*	2	94	*78*	28
31	*15*	2	63	*47*	0	95	*79*	80

Tab. SHA1-2: Složitost hledání jednoblokové (1BC) kolize pro funkci SHA-1 redukovanou na *t* kroků. Použité poruchové vektory jsou výběrem z tabulky Tab. SHA1-1 indexované podle Kroku *i*. Hammingova váha poruchové matice se počítá pouze pro kroky 20 – 79 (vynecháním prvních dvaceti kroků je možné redukovat druhou podmínku na poruchové vektory).

SHA-1 redukovaná na *t* kroků	Použité poruchové vektory	H. váha pro kroky 20 - 79	Složitost 1BC
80	0 - 79	31	2^{93}
79	1 - 79	30	2^{92}
78	2 - 79	30	2^{87}
77	3 - 79	28	2^{85}
76	4 - 79	27	2^{80}
75	5 - 79	26	2^{78}
74	6 - 79	25	2^{76}
73	7 - 79	25	2^{74}
72	8 - 79	25	2^{74}
71	9 - 79	24	2^{71}
70	10 - 79	24	2^{68}
69	11 - 79	22	2^{66}
68	12 - 79	21	2^{60}
67	13 - 79	19	2^{56}
66	14 - 79	19	2^{53}
65	15 - 79	18	2^{49}
64	16 - 79	18	2^{46}
63	17 - 79	16	2^{46}
62	18 - 79	16	2^{43}
61	19 - 79	15	2^{39}
60	20 - 79	14	2^{37}
59	21 - 79	13	2^{36}
58	22 - 79	13	2^{33}
57	23 - 79	12	2^{29}
56	24 - 79	11	2^{26}
55	25 - 79	10	2^{24}
54	26 - 79	10	2^{22}
53	27 - 79	10	2^{19}
52	28 - 79	9	2^{15}
51	29 - 79	7	2^{14}
50	30 - 79	7	2^{12}

Tab. SHA1-3: Pravidla pro počítání počtu podmínek pro kroky *t* = 18, ..., 79

Krok	Poruchy v bitě 1	Poruchy v ostatních bitech
18	0	1
19	0	2
20	1	3
21 - 35	2	4
36	3	4
37 - 39	4	4
40 - 59	4	4
60 - 75	2	4
76	2	3
77	2	2
78	(1)	(1)
79	(1)	(1)

Pravidla pro počítání podmínek:

1. Pokud dvě poruchy začínají v bitě 1 a v bitě 0 ve stejném kroku, pak mohou být nahrazeny jedinou poruchou s výsledným počtem podmínek 4.
2. Pro kroky $t = 40,...59$, dvě po sobě jdoucí poruchy ve stejném bitě dávají celkem 8 podmínek (toto je díky vlastnostem použité nelineární funkce $f_t(X,Y,Z) = H(X,Y,Z)$).
3. Podmínky pro kroky 78 a 79 mohou být při analýze ignorovány.

Tab. SHA1-4: Výpočet počtu podmínek pro 80-ti krokovou blízkou kolizi. Hodnoty ve sloupci Index odpovídají hodnotám ve sloupci Použité vektory v tabulce Tab. SHA1-1.

Index	Počet podmínek
20	4 - 1 - 1 = 2
22, 23, 26, 27, 31, 34, 35	2 x 7 = 14
24, 28, 32, 38, 42, 44, 46, 48, 64, 67, 70, 72, 73	4 x 4 = 16 4 x 4 = 16 4 x 5 = 20
76	3
78	0
79	0
Celkem	71

9.3. Tabulky pro funkci SHA-2

Tab. SHA2-1: Inicializační hodnoty slov pro funkci SHA-224

Slovo	Hexadecimální hodnota
A_0	c1059ed8
B_0	367cd507
C_0	3070dd17
D_0	f70e5939
E_0	ffc00b31
F_0	68581511
G_0	64f98fa7
H_0	befa4fa4

Tab. SHA2-2: Inicializační hodnoty slov pro funkci SHA-384

Slovo	Hexadecimální hodnota
A_0	cbbb9d5dc1059ed8
B_0	629a292a367cd507
C_0	9159015a3070dd17
D_0	152fecd8f70e5939
E_0	67332667ffc00b31
F_0	8eb44a8768581511
G_0	db0c2e0d64f98fa7
H_0	47b5481dbefa4fa4

Tab. SHA2-3: Soubor aditivních konstant pro funkci SHA-512.

t	K_t	t	K_t	t	K_t
0	$428a2f98d728ae22_{16}$	27	$bf597fc7beef0ee4_{16}$	54	$5b9cca4f7763e373_{16}$
1	$7137449123ef65cd_{16}$	28	$c6e00bf33da88fc2_{16}$	55	$682e6ff3d6b2b8a3_{16}$
2	$b5c0fbcfec4d3b2f_{16}$	29	$d5a79147930aa725_{16}$	56	$748f82ee5defb2fc_{16}$
3	$e9b5dba58189dbbc_{16}$	30	$06ca6351e003826f_{16}$	57	$78a5636f43172f60_{16}$
4	$3956c25bf348b538_{16}$	31	$142929670a0e6e70_{16}$	58	$84c87814a1f0ab72_{16}$
5	$59f111f1b605d019_{16}$	32	$27b70a8546d22ffc_{16}$	59	$8cc702081a6439ec_{16}$
6	$923f82a4af194f9b_{16}$	33	$2e1b21385c26c926_{16}$	60	$90befffa23631e28_{16}$
7	$ab1c5ed5da6d8118_{16}$	34	$4d2c6dfc5ac42aed_{16}$	61	$a4506cebde82bde9_{16}$
8	$d807aa98a3030242_{16}$	35	$53380d139d95b3df_{16}$	62	$bef9a3f7b2c67915_{16}$
9	$12835b0145706fbe_{16}$	36	$650a73548baf63de_{16}$	63	$c67178f2e372532b_{16}$
10	$243185be4ee4b28c_{16}$	37	$766a0abb3c77b2a8_{16}$	64	$ca273eceea26619c_{16}$
11	$550c7dc3d5ffb4e2_{16}$	38	$81c2c92e47edaee6_{16}$	65	$d186b8c721c0c207_{16}$
12	$72be5d74f27b896f_{16}$	39	$92722c851482353b_{16}$	66	$eada7dd6cde0eb1e_{16}$
13	$80deb1fe3b1696b1_{16}$	40	$a2bfe8a14cf10364_{16}$	67	$f57d4f7fee6ed178_{16}$
14	$9bdc06a725c71235_{16}$	41	$a81a664bbc423001_{16}$	68	$06f067aa72176fba_{16}$
15	$c19bf174cf692694_{16}$	42	$c24b8b70d0f89791_{16}$	69	$0a637dc5a2c898a6_{16}$
16	$e49b69c19ef14ad2_{16}$	43	$c76c51a30654be30_{16}$	70	$113f9804bef90dae_{16}$
17	$efbe4786384f25e3_{16}$	44	$d192e819d6ef5218_{16}$	71	$1b710b35131c471b_{16}$
18	$0fc19dc68b8cd5b5_{16}$	45	$d69906245565a910_{16}$	72	$28db77f523047d84_{16}$
19	$240ca1cc77ac9c65_{16}$	46	$f40e35855771202a_{16}$	73	$32caab7b40c72493_{16}$
20	$2de92c6f592b0275_{16}$	47	$106aa07032bbd1b8_{16}$	74	$3c9ebe0a15c9bebc_{16}$
21	$4a7484aa6ea6e483_{16}$	48	$19a4c116b8d2d0c8_{16}$	75	$431d67c49c100d4c_{16}$
22	$5cb0a9dcbd41fbd4_{16}$	49	$1e376c085141ab53_{16}$	76	$4cc5d4becb3e42b6_{16}$
23	$76f988da831153b5_{16}$	50	$2748774cdf8eeb99_{16}$	77	$597f299cfc657e2a_{16}$
24	$983e5152ee66dfab_{16}$	51	$34b0bcb5e19b48a8_{16}$	78	$5fcb6fab3ad6faec_{16}$
25	$a831c66d2db43210_{16}$	52	$391c0cb3c5c95a63_{16}$	79	$6c44198c4a475817_{16}$
26	$b00327c898fb213f_{16}$	53	$4ed8aa4ae3418acb_{16}$		

Seznam použité literarury

[1] NIST (2001): Announcing the Advanced Encryption Standard (AES), FIPS PUB 197, http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] Barreto, P. (2000): The Hashing Function Lounge, http://www.larc.usp.br/~pbarreto/hflounge.html

[3] Suchan, M. (2007): Bakalářská práce: Porovnání současných a nových hašovacích funkcí, Univerzita Karlova v Praze, Matematicko-fyzikální fakulta.

[4] Pinkava, J. (2004): Hashovací funkce v roce 2004, PVT a.s., http://crypto-world.info/pinkava/clanky/hash_2004.pdf

[5] RFC 1321 – specifikace funkce MD5, http://www.ietf.org/rfc/rfc1321.txt

[6] Heys, H. M.: A Tutorial on Linear and Differencial Cryptanalysis, http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf

[7] Stevens, M. M. J. (2007): Master's thesis: On Collisions For MD5, Eindhoven University of Technology, Department of Mathematics and Computing Science, http://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf

[8] Bosselaers, A. and B. den Boer (1993): Collisions for the compression function of MD5, EUROCRYPT 1993, LNCS, Springer, svazek 765, http://homes.esat.kuleuven.be/~cosicart/pdf/AB-9300.pdf

[9] Dobbertin, H. (1996): The Status of MD5 after a recent attack, CryptoBytes 2:2, ftp://ftp.rsa.com/pub/cryptobytes/crypto2n2.pdf

[10] Wang, X., Feng, D., Lai, X., Yu, H. (2004): Collisions for hash function MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199, http://eprint.iacr.org/2004/199.pdf

[11] Wang, X., Yu, H. (2005): How to break MD5 and other hash functions, EUROCRYPT 2005, LNCS, Springer, svazek 3494, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.6718-1.pdf

[12] Liang, J., Lai, X. (2005): Improved collision attack on hash function MD5, Cryptology ePrint Archive, Report 2005/425, http://eprint.iacr.org/2005/425.pdf

[13] Klíma, V. (2005): Finding MD5 collisions on a notebook PC using multi-message modifications, Cryptology ePrint Archive, Report 2005/102, http://eprint.iacr.org/2005/102.pdf

[14] Klíma, V. (2006): Tunnels in hash functions: MD5 collisions within a minute, Cryptology ePrint Archive, Report 2006/105, http://eprint.iacr.org/2006/105.pdf

[15] Hawkes, P., Paddon, M., Rose, G. G. (2004): Musings on the Wang at al. MD5 collision, Cryptology ePrint Archive, Report 2004/264, http://eprint.iacr.org/2004/264.pdf

[16] Yajima, J., Shimoyama, T. (2005): Wang's sufficient conditions of MD5 are not sufficient, Cryptology ePrint Archive, Report 2005/263, http://eprint.iacr.org/2005/263.pdf

[17] Klíma, V. (2006): Tunely v hašovacích funkcích: kolize MD5 do minuty, http://cryptography.hyperlink.cz/2006/tunely.pdf

[18] Sasaki, Y., Naito, Y., Kunihiro, N., Ohta, K. (2005): Improved Collision Attack on MD5, Cryptology ePrint Archive: Report 2005/400, http://eprint.iacr.org/2005/400.pdf

[19] Kmet, V. (2005): ASN.1 – příklad X.509, základy a kódování, http://www.lupa.cz/clanky/asn-1-priklad-x-509-zaklady-a-kodovani/

[20] Lenstra, A., Benne de Weger (2005): On the possibility of constructing meaningful hash collisions for public keys, ACISP 2005, LNCS, vol. 3547, Springer, http://www.win.tue.nl/~bdeweger/CollidingCertificates/ddl-full.pdf

[21] Lenstra, A., Wnag, X., Benne de Weger (2005): Colliding X.509 Certificates, http://www.cecm.sfu.ca/~lisonek/cryptography/x509-collisions.pdf

[22] RFC 3280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, http://tools.ietf.org/html/rfc3280#section-3.1

[23] Demlová, M. (2008): Algebra pro VT, přednáška 2, http://math.feld.cvut.cz/demlova/teaching/avt/pred-a02.pdf

[24] Kaminsky, D. (2004): MD5 to be considered harmful someday, http://eprint.iacr.org/2004/357.pdf

[25] Mikle, O. (2004): Practical Attacks on Digital Signatures Using MD5 Message Digest, Cryptology ePrint Archive, Report 2004/356, http://eprint.iacr.org/2004/356.pdf

[26] RFC 3174 – specifikace funkce SHA-1, http://www.ietf.org/rfc/rfc3174.txt

[27] Oswaldová, E., Rijmen, V. (2005): Update on SHA-1, Cryptology ePrint Archive, Report 2005/010, http://eprint.iacr.org/2005/010.pdf

[28] Wang, X., Yu, H., Yin, L. Y. (2005): Finding Collisions in the Full SHA-1, Crypto 2005, http://www.infosec.sdu.edu.cn/uploadfile/papers/Finding%20Collisions%20in%20the%20Full%20SHA-1.pdf, Na Rump Session Crypto 2005 dne 17. 8. 2005 týmem Wang-Yao-Yao bylo oznámeno zlepšení prezentovaného útoku z 2^{69} na složitost 2^{63}

[29] Wang, X., Yu, H., Yin, L. Y. (2005): Efficient Collision Search Attacks on SHA-0, Crypto 2005, http://www.cs.cmu.edu/~dbrumley/srg/spring06/sha-0.pdf

[30] Biham, E. and Chen, R. (2004): New Results on SHA-0 and SHA-1, Crypto 2004, Rump Session, August 2004, http://www.cs.technion.ac.il/~biham/Reports/Slides/invited-talk-sac-2004.ps.gz

[31] Osward, E. and Rijmen, V. (2005): Update on SHA-1, RSA Crypto Track 2005, http://saluc.engr.uconn.edu/refs/algorithms/hashalg/rijmen05update.pdf

[32] Sotirov, A. et al. (2008): MD5 considered harmful today, Creating a rogue CA certificate, http://www.win.tue.nl/hashclash/rogue-ca/

[33] Isobe, T. and Shibutani, K. (2009): Preimage Attacks on Reduced Tiger and SHA-2. In Orr Dunkelman, editor, FSE, volume 5665 of LNCS, pages 139 – 155, Springer. http://www.springerlink.com/content/d36t7p53mm585629/

[34] Aoki, K. a kol. (2009): Preimages for Step-Reduced SHA-2. In Mitsuru Matsui, editor, ASIACRYPT, volume 5912 of LNCS, pages 578 – 597, Springer. http://www.springerlink.com/content/dwrt65440t416307/

[35] Menedel, F. a kol. (2006): Analysis of Step-Reduced SHA-256. In Matthew J. B. Robshaw, editor, FSE, volume 4047 of LNCS, pages 126 – 143, Springer. https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=14119

[36] Nicolic, I. and Biryukov, A. (2008): Collisions for Step-Reduced SHA-256. In Kaisa Nyberg, editor, FSE, volume 5086 of LNCS, pages 1 – 15, Springer. https://cryptolux.org/mediawiki/uploads/a/ac/SHA-2.pdf

[37] Lamberger, M. a Menedel, F. (2011): Higher-Order Differential Attack on Reduced SHA-256. Institute for Applied Information Processing and Communications Graz University of Technology, Austria. http://eprint.iacr.org/2011/037.pdf

[38] Stevens, M. a kol. (2007): Chosen-prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities, In: Naor, M. (ed.), EUROCRYPT 2007 Heidelberg, volume 1545 of LNCS, pages 1 – 22, Springer.

[39] NIST (2009): Recommendation for Applications Using Approved Hash Algorithms, NIST SP 800-107, http://csrc.nist.gov/publications/nistpubs/800-107/NIST-SP-800-107.pdf

[40] ETSI TS 102 176-1 V2.0.0 (2007), Technická specifikace, Algorithms and Parameters for Secure Electronic Signatures, http://www.etsi.org/deliver/etsi_ts/102100_102199/10217601/02.00.00_60/ts_10217601v020000p.pdf

[41] Klíma, V. (2006): Nový koncept hašovacích funkcí SNMAC s využitím speciální blokové šifry a konstrukcí NMAC/HMAC, http://cryptography.hyperlink.cz/SNMAC/SNMAC_CZ.pdf