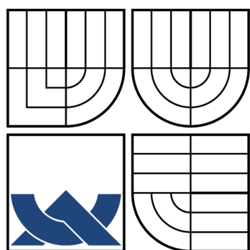
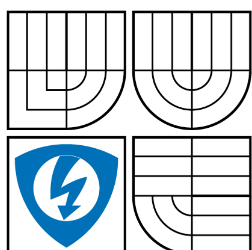


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

HASHOVACÍ FUNKCE A JEJICH VYUŽITÍ PŘI AUTENTIZACI

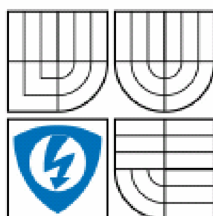
HASH FUNCTION AND THEIR USAGE IN USER AUTHENTICATION

DIPLOMOVÁ PRÁCE
MASTER´S THESIS

AUTOR PRÁCE Bc. IGOR PILLER
AUTHOR

VEDOUCÍ PRÁCE Ing. JAN HAJNÝ
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. Igor Piller *ID:* 83355
Ročník: 2 *Akademický rok:* 2008/2009

NÁZEV TÉMATU:

Hashovací funkce a jejich využití při autentizaci

POKYNY PRO VYPRACOVÁNÍ:

Uveďte základní konstrukční prvky hashovacích funkcí a jejich výhody/slabiny. Vyhledejte a popište současné hashovací algoritmy. Zaměřte se především na LMHash, MD4, MD5 a rodinu SHA. Jednotlivé funkce srovnejte především z bezpečnostního hlediska a vyhledejte vhodné nástroje a metody pro jejich zlomení. Popište souvislost mezi hashovacími funkcemi a autentizací uživatelů. Prakticky realizujte obecný autentizační rámec a zvolené metody v něm otestujte. Dbejte na flexibilitu řešení a budoucí možnost využití jiných metod autentizace.

DOPORUČENÁ LITERATURA:

[1] SMITH, Richard E. Authentication: From Passwords to Public Keys. [s.l.] : [s.n.], 2001. 576 s.
[2] STALLINGS, William. Cryptography and Network Security. [s.l.] : [s.n.], 2005. 592 s.

Termín zadání: 9.2.2009 *Termín odevzdání:* 26.5.2009

Vedoucí práce: Ing. Jan Hajný

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

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práve třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

Abstrakt

Práce se zabývá hashovacími funkcemi a jejich využitím při autentizaci. Obsahuje základní teorii o hashovacích funkcích a popis jejich základních konstrukčních prvků.

Konkrétně se práce zaměřuje na hashovací funkce LMHash, MD4, MD5 a funkce z rodiny SHA, které porovnává z hlediska bezpečnosti. Práce obecně popisuje nejpoužívanější útoky na hashovací funkce, poukazuje na slabiny současné konstrukce a nabízí výhled do budoucnosti hashovacích funkcí.

Dále práce nastiňuje problematiku autentizace a popisuje použití hashovacích funkcí v této oblasti.

V praktické části je realizován obecný autentizační rámec v programovacím jazyce C#. Výsledkem realizace jsou klientská a serverová aplikace, na kterých byly úspěšně vyzkoušeny dvě vybrané autentizační metody. Při realizaci bylo dbáno na flexibilitu řešení a možné budoucí využití jiných metod autentizace.

Klíčová slova:
hash, hashovací funkce, autentizace, LMHash, MD4, MD5, SHA

Abstract

This thesis concerns with hash functions and their usage in authentication. It presents basics of hash functions theory and construction elements.

In particular the thesis focuses on LMHash, MD4, MD5 and SHA family hash functions, which are compared from the security point of view. The thesis describes in general the most frequently used hash function attacks, points out the weaknesses of current construction and mentions the future perspective of hash functions.

Furthermore the thesis outlines the area authentication and describes usage of hash functions in the area.

Practical part of the thesis contains an implements of a general authentication framework implemented in programming language C#. The result is client and server applications, in which two selected authentication methods were successfully tested. The result implementation is flexible with respect to the possible future use of other authentication methods.

Keywords:
Hash, hash function, authentication, LMHash, MD4, MD5, SHA

Citace práce

PILLER, I. *Hashovací funkce a jejich využití při autentizaci.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 68s. Vedoucí diplomové práce Ing. Jan Hajný.

PROHLÁŠENÍ:

Prohlašuji, že svoji diplomovou práci na téma „hashovací funkce a jejich využití při autentizaci" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne..................... ....

podpis autora

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Janu Hajnému, za velmi užitečnou metodickou pomoc a cenné rady při zpracování diplomové práce.

V Brně dne ……………………… ……………………………………………

(podpis autora)

Seznam zkratek

AES – Advanced Encryption Standard

ASCII – American Standard Code for Information Interchange

DES – Data Encryption Standard

HDN – Hash Double Net

HMAC – Hash Message Authentication Code

LANMAN – LAN Manager

LMHash – LAN Manager hash

MD – Message-Digest

MDC – Modification detection codes

MIT – Massachusetts Institute of Technology

NIST – National Institute of Standards and Technology

NTLM – NT LAN Manager

OS – Operační systém

PIN – Personal identification number

POV – Point of verification

RIPEMD – RACE Integrity Primitives Evaluation Message Digest

SHA – Secure Hash Algorithm

SNMAC – Special Nested Message Authentication Code

XOR – Exkluzivní logický součet

OBSAH

1 ÚVOD 10
2 ZÁKLADNÍ KONSTRUKČNÍ PRVKY HASHOVACÍCH FUNKCÍ 11
2.1 DEFINICE HASHOVACÍ FUNKCE 11
2.2 ROSTOUCÍ POŽADAVKY KLADENÉ NA HASHOVACÍ FUNKCE 11
2.3 KONSTRUKČNÍ PRVKY 13
2.4 POUŽITÍ HASHOVACÍCH FUNKCÍ 17
3 NÁSTROJE A METODY PRO ZLOMENÍ 19
3.1 ÚTOKY METODOU HRUBÉ SÍLY 19
3.2 SLOVNÍKOVÝ ÚTOK A RAINBOW TABLES 19
3.3 KRYPTOANALÝZA 21
4 SOUČASNÉ HASHOVACÍ FUNKCE 22
4.1 LMHASH 22
4.2 MD4 23
4.3 MD5 25
4.3.1 Konstrukce MD5, bezpečnost 25
4.3.2 Tunelování 27
4.4 SHA-0 A SHA-1 27
4.5 RODINA SHA-2 29
5 BUDOUCNOST HASHOVACÍCH FUNKCÍ 31
5.1 PROBLÉMY KONSTRUKCE DNEŠNÍCH HASHOVACÍCH FUNKCÍ 31
5.1.1 Generické problémy iterativních hashovacích funkcí 31
5.1.2 R-násobná kolize s nižší složitostí 31
5.1.3 Kaskádovitá konstrukce 32
5.1.4 Nalezení druhého vzoru u dlouhých zpráv snadněji než se složitostí 2^n *33*
5.2 NIST, SOUTĚŽ SHA-3 33
5.3 HASHOVACÍ FUNKCE HDN TYPU SNMAC 34
5.3.1 Nové kryptografické primitivum 35
5.3.2 Hashovací funkce nové generace SNMAC 36
5.3.3 Hashovací funkce HDN 36
6 HASHOVACÍ FUNKCE A AUTENTIZACE 37
6.1 AUTENTIZACE 37
6.2 HASHOVACÍ FUNKCE A UKLÁDÁNÍ HESEL 37
6.2.1 Ukládání hesel v OS Windows 38
6.2.2 Ukládání hesel v OS Linux 39
6.3 AUTENTIZAČNÍ PROTOKOLY 39
6.3.1 Přímé ověření 39
6.3.2 Nepřímé ověření 41
7 REALIZACE OBECNÉHO AUTENTIZAČNÍHO RÁMCE 42
7.1 ROZBOR APLIKACE 42
7.1.1 Výběr programovacího jazyka 42

7.1.2 Zadání, hlavní myšlenka realizace 42
7.1.3 Schémata jednotlivých tříd, provázání tříd důležitých pro autentizaci......43
7.1.4 Logování, načítání dat z textového souboru..45
7.2 SERVER, UŽIVATELSKÉ ROZHRANÍ...47
7.3 CLIENT, UŽIVATELSKÉ ROZHRANÍ 48
7.4 TŘÍDA AUTHENTICATIONPROCEDURE..49
7.5 TŘÍDA SIMPLEPROCEDURE, OBYČEJNÁ METODA AUTENTIZACE 50
7.6 TŘÍDA ADVANCEDPROCEDURE 51
7.7 TŘÍDY HASHFUNCTION, IDENTITYFUNCTION, MD5, SERVER, CLIENT, MAINFORM .53
7.8 PŘIDÁNÍ AUTENTIZAČNÍ METODY A HASHOVACÍ FUNKCE..54
7.9 UKÁZKA KOMUNIKACE...59
8 ZÁVĚR 61
POUŽITÁ LITERATURA...63
SEZNAM OBRÁZKŮ 66
OBSAH CD 68

1 Úvod

Hashovací funkce jsou nedílnou součástí moderní kryptologie už po mnoho let. S rozvojem Internetu se jejich důležitost ještě mnohokrát znásobila a jejich využití je opravdu široké, používajíc různé vlastnosti těchto funkcí.

V posledních několika letech je vidět zvýšený zájem o studium těchto funkcí. Za největší zviditelnění můžeme považovat úspěšné útoky na hashovací funkce MD5, SHA-0, SHA-1. Díky těmto útokům se o hashovací funkce začalo zajímat mnohem více lidí a panuje všeobecný názor, že tyto funkce nebyly dostatečně studovány, přestože jim přikládáme takovou důležitost. I neodborná veřejnost se pak na základě vysvětlujících populárních článků začala zajímat a povědomí o těchto funkcích tak začíná stoupat.

V první části této diplomové práce se podíváme co vlastně hashovací funkce je, jaké má vlastnosti, jaké chceme, aby měla vlastnosti, z čeho je složena a k čemu se používá.

V další části jsou uvedeny obecně některé možné útoky na tyto funkce.

Ve třetí části jsou rozebrány nejpoužívanější dnešní hashovací funkce, kterými jsou LMHash, MD4, MD5, SHA-0, SHA-1 a funkce z rodiny SHA-2. U funkce MD5 je pak krátce rozebrán nový způsob kryptoanalýzy hashovacích funkcí, tzv. „tunelování“, které se ukázalo být velice efektivním útokem.

Čtvrtá část se zabývá problémy dnešních hashovacích funkcí, popisuje dosud známé odhalené slabiny konstrukce dnešních iterativních hashovacích funkcí. V této části jsou ale také popsány návrhy budoucích hashovacích funkcí, které by tyto problémy mohly vyřešit.

V páté části jsou zmíněny souvislosti autentizace a použití hashovacích funkcí.

Nakonec poslední část obsahuje dokumentaci k přiloženému praktickému řešení obecného autentizačního rámce.

2 Základní konstrukční prvky hashovacích funkcí

2.1 Definice hashovací funkce

Jako hashovací se dříve označovaly takové funkce, které pro libovolně velký vstup přiřadily krátký hashový kód s pevně definovanou délkou. Dnes se termínem hashovací funkce označují kryptografické hashovací funkce, u nichž je navíc požadováno, aby byly jednosměrné a bezkolizní.

Hashovací funkce vychází hlavně z pojmu „jednosměrná funkce“. To je funkce, kterou lze snadno vyčíslit, ale je výpočetně nemožné z výsledku funkce odvodit její vstup. Tedy funkce $f\colon X \rightarrow Y$, pro kterou je snadné z jakékoliv hodnoty $x \in X$ vypočítat $y=f(x)$, ale při znalosti $y \in f(X)$ nelze najít její vzor $x \in X$ tak, aby $y=f(x)$, tedy vypočítat inverzní funkci je výpočetně nemožné v krátkém čase. Vzor přitom existuje, nebo je jich dokonce velmi mnoho. Kromě hashovací funkce je dalším příkladem jednosměrné funkce součin dvou velkých prvočísel, který získáme velmi snadno, kdežto zpětně jej faktorizovat je pro nás již výpočetně nemožné.

2.2 Rostoucí požadavky kladené na hashovací funkce

Mimo schopnosti přiřadit libovolně velkému vstupu hashový kód o pevně definované délce se během času začaly na hashovací funkce klást větší a větší požadavky. Nejdůležitější zde uvedu. (*Pozn.: V definici se uvádí „libovolně velký vstup“, ve skutečnosti se však používá většinou vstup maximální délky* $D = 2^{64}-1$ *bitů, abychom takové číslo mohli vyjádřit pomocí 64 bitů.*)

Jednosměrnost

Též se uvádí jako pojem „odolnost argumentu“. Pro danou zprávu M je velmi jednoduché spočítat $h=H(M)$, ale pro dané h je výpočetně nemožné vypočítat M. Tato vlastnost je velmi žádoucí a využívá se například při ukládání hesel. Neukládáme a neschraňujeme samotné heslo, ale ukládáme pouze hashový kód. Při autentizaci se pak místo přímého porovnávaní zadaného a uloženého hesla porovnávají jejich hashové kódy.

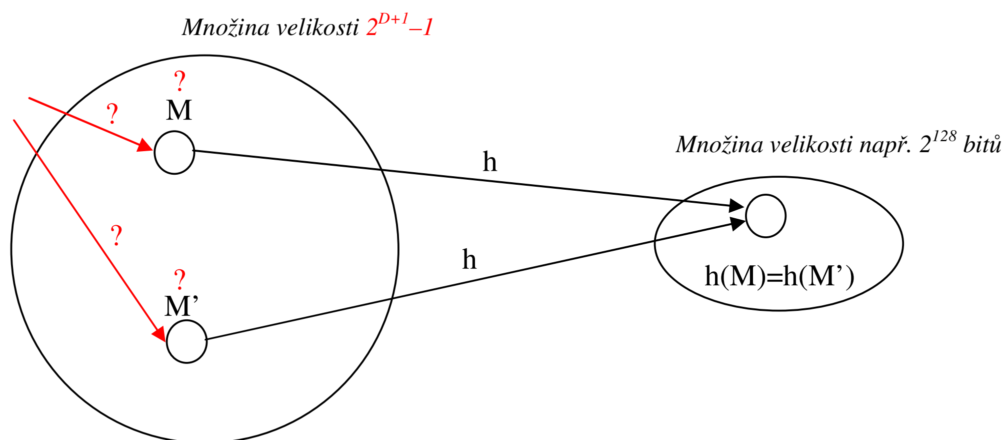
Odolnost proti kolizi prvního řádu

Dalším požadavkem kladeným na hashovací funkce začala být „bezkoliznost“, jinak v literatuře nazývaná i „slabá odolnost proti kolizi“, nebo „odolnost proti kolizi prvního řádu“. Požadujeme, aby bylo výpočetně nemožné najít různé M a M', tedy $M \neq M'$ (byť dvě naprosto nesmyslná M a M') tak, že $h(M)=h(M')$. Jinak řečeno, pokud najdeme dva různé vstupy, které mají stejný hashový kód, tak řekneme, že jsme našli kolizi a můžeme použitou hashovací funkci považovat za prolomenou.

Pokud totiž nejsme schopni takovou kolizi najít (vypočítat), můžeme libovolnou velkou zprávu z velmi velké množiny $2^{D+1}-1$ reprezentovat hashovým kódem malé velikosti, například 128, 256, 512, apod. bitů, tzn. množiny velikosti 2^{128}, 2^{256}, 2^{512} bitů, apod. Tedy je to pro nás jednoznačné zobrazení, i když matematicky samozřejmě není. Kolizí existuje obrovské množství, ale díky výpočetní složitosti není snadné takové kolize najít, takže můžeme hashový kód považovat za jednoznačné zobrazení. Jinak řečeno, libovolně dlouho zprávu můžeme reprezentovat malým hashovým kódem.

Tato vlastnost se využívá například u digitálního podpisu, kdy nepodepisujeme přímo celou zprávu (protože ta může být i velmi dlouhá), ale podepisujeme právě jen její hashový kód, protože víme, že najít jakoukoliv jinou (i nesmyslnou) zprávu se stejným hashovým kódem je výpočetně nemožné.

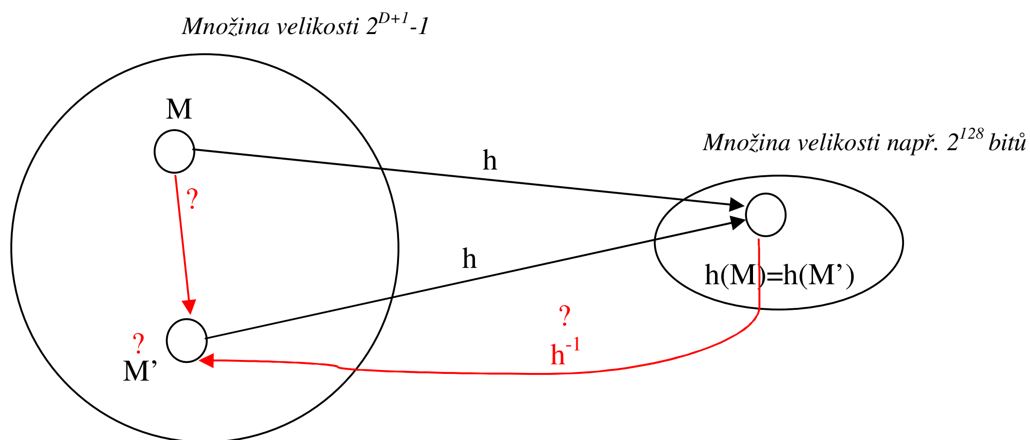
Obecně je nalezení kolize prvního řádu lehčí, než nalezení kolize druhého řádu, protože útočník hledá jakékoliv *M* a *M'*, které by měly mít stejný hashový kód, a při tomto hledání existuje tzv. „narozeninový paradox“. Tento název byl inspirován otázkou, jak velkou skupinu lidí potřebujeme, abychom měli alespoň 50% pravděpodobnost, že v ní najdeme dva lidi se stejným dnem narozenin. Výpočtem dostaneme, že $P(365,23) = 0{,}507$. Tedy stačí nám jen 23 lidí. Pokud máme skupinu 30 lidí, pravděpodobnost stoupne již na $P(365,30) = 0{,}706$, tedy 71%. Kdežto kdybychom chtěli k jednomu konkrétnímu člověku najít dalšího se stejným datem narození s alespoň 50% pravděpodobností, potřebovali bychom polovinu z množiny všech možností, tedy $356/2 = 178$ lidí. Analogie s hashovacími funkcemi je zřejmá, díky tomuto paradoxu je nalezení kolize prvního řádu řádově snazší než nalezení kolize druhého řádu. Bez zajímavosti jistě není, že většina uživatelů si tyto odolnosti plete a pokud je nějaká hashovací funkce prolomena, lidé si myslí, že funkce není odolná právě proti kolizi druhého řádu, přičemž první útoky na hashovací funkce bývají na odolnost prvního řádu.



Obr. 1: Odolnost proti kolizi prvního řádu

Odolnost proti kolizi druhého řádu

Jinak také zvaná „odolnost proti nalezení druhého vzoru“, „bezkoliznost druhého řádu“ nebo „silná odolnost proti kolizi“. Hashovací funkce je odolná proti kolizi druhého řádu, pokud pro daný náhodný vzor M je výpočetně nemožné nalézt druhý vzor $M' \neq M$ tak, že platí $h(M)=h(M')$. Jinak řečeno, je pro nás výpočetně nemožné nalézt k jedné dané zprávě jinou (byť i nesmyslnou) se stejným hashovým kódem. Pokud hashovací funkce není odolná proti kolizi druhého řádu, tak představuje velká bezpečnostní rizika. Taková funkce není doporučena k používání.



Obr. 2: Odolnost proti kolizi druhého řádu

Náhodné orákulum

Na hashovací funkce se začal klást požadavek, aby výstup byl co nejvíce náhodný. Tedy aby se co nejvíce podobaly tzv. „náhodnému orákulu“. Náhodné orákulum, jinak též „stroj podivuhodných vlastností“, je orákulum, které má dvě vlastnosti:

- Na tentýž vstup vždy odpovídá stejným výstupem, tedy pamatuje si jen ty vstupy, na které už odpovědělo a odpovídá na ně stejně.
- Na vstupy, na které ještě neodpovídalo, vybírá z množiny výsledku výstup zcela náhodným výběrem, tedy je to dokonalá náhodná funkce.

2.3 Konstrukční prvky

Vzhledem k tomu, že vstupní data mohou mít délku až 2^{64}–1 bitů, je zřejmé, že hashovací funkce musí velké zprávy zpracovávat po částech. Musíme tedy zprávy rozdělit do bloků, ale navíc ještě musíme zarovnat vstupní zprávy na celistvý počet bloků [1].

Zarovnání

Hashovací funkce zpracovává vstupní data sekvenčně po blocích určité délky. Je tedy nutné doplnit vstupní zprávu o takové množství bitů, aby délka zprávy byla dělitelná velikostí bloku. Zarovnání ale musí umožňovat jednoznačné odejmutí. Protože pokud bychom například doplňovali jen nulovými bity, nebylo by poznat, kde končí zpráva a kde začíná zarovnání. To by vedlo k mnohonásobným kolizím, protože například již doplněná zpráva ...1010000000 by mohla vzniknout ze zpráv končících …101, …1010, …10100, atd. a všechny by měly stejný hashový kód. Zarovnání se tedy u moderních hashovacích funkcí řeší přidáním bitu 1 a poté potřebným počtem nulových bitů. To umožňuje jednoznačné odejmutí přebývajících bitů.

Damgård-Merklova konstrukce

Tento princip používají drtivá většina dnes používaných hashovacích funkcí. Algoritmus je založený na opakovaném použití kompresní funkce *f*, která je tak vlastně jádrem hashovacích funkcí.

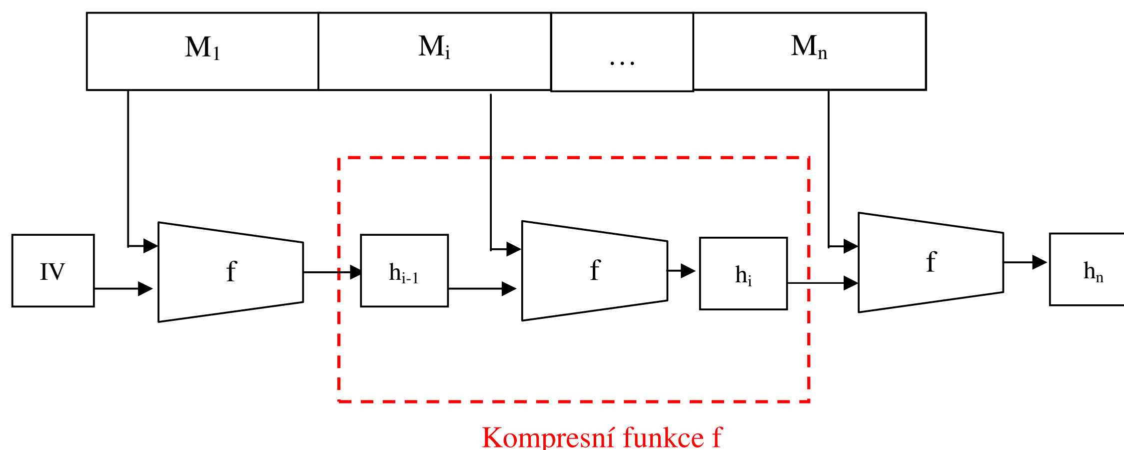
Proces iteračního výpočtu hashového kódu *h* je možné vyjádřit v následujícím tvaru [3]:

$$IV = h_0$$
$$h_i = f(h_{i-1}, M_i) \quad pro \quad 1 \le i \le n$$
$$h = h_n$$

Samotnou kompresní funkci pak: $f : \{0,1\}^h \mathrm{X} \{0,1\}^m \rightarrow \{0,1\}^h$

To tedy znamená, že kompresní funkce má dva vstupy a jeden výstup. Zpracovává aktuální blok zprávy M_i a hodnotu h_{i-1}. Výstupem je určitá hodnota h_i označovaná jako kontext. Tento kontext je pak použit jako vstup do kompresní funkce v dalším kroku. Počáteční hodnota kontextu h_0 je nazývána inicializačním vektorem *IV*. Ten určuje blok bitů vstupující do první kompresní funkce a zahajuje tak vlastní hashování.

Kompresní funkce má přiléhavý název, protože je vidět, že zpracovává širší vstup na mnohem kratší h_i. Blok zprávy M_i se funkčně promítne do kontextu h_i, jehož šířka ale zůstává stejná, čímž dochází ke ztrátě informace. Kontextem pak bývá většinou několik 16bitových nebo 32bitových slov. Příkladem jsou čtyři slova u hashovací funkce MD5. Dohromady tvoří 128 bitů, což je šířka kontextu. Výsledný hashový kód pak je buď část, nebo většinou celý poslední kontext h_N.



Obr. 3: Damgård-Merklova konstrukce s vyznačenou kompresní funkcí

Damgård-Merklovo zesílení

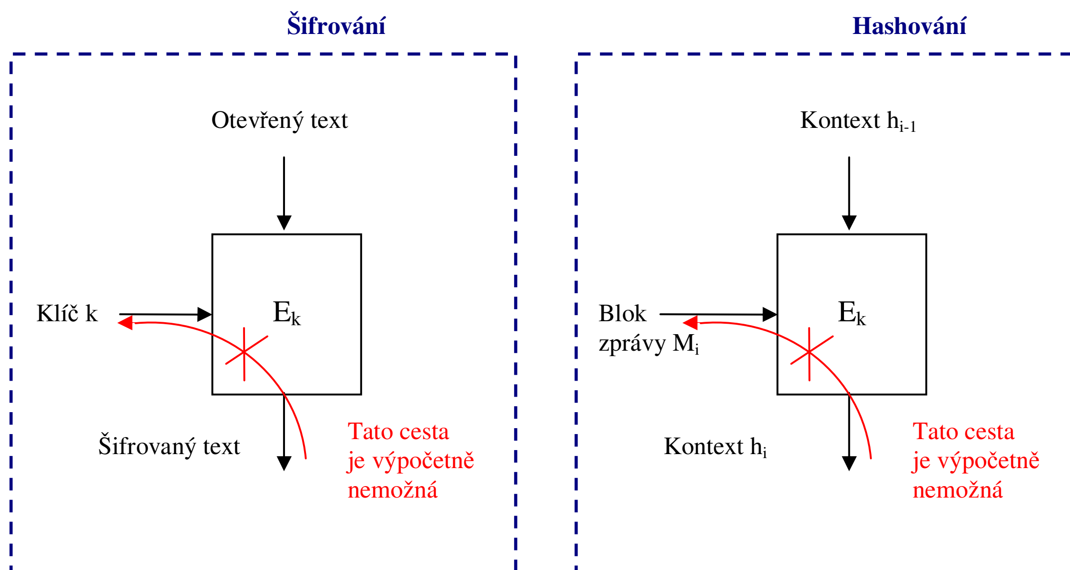
Damgård-Merklovo zesílení bylo nezávisle oběma autory navrženo na konferenci Crypto 1989. Na závěr zprávy *M* se mimo zarovnání ještě rezervuje 64 bitů, které vyjadřují počet bitů původní zprávy *M*. 64bitové vyjádření je proto, abychom mohli hashovat zprávy až do délky $D=2^{64} -1$ *bitů*. Do hashového kódu se tedy funkčně promítne i délka zprávy. Toto zesílení je z bezpečnostního hlediska velmi důležité, protože složitost nalezení kolize bez tohoto opatření je řádově snazší.

Konstrukce kompresní funkce

Pokud máme zprávu o délce jen jednoho bloku, vidíme, že již kompresní funkce musí být bezkolizní, aby byla bezkolizní samotná hashovací funkce. Bezkoliznost kompresní funkce tedy implikuje bezkoliznost hashovací funkce. Damgård a Merkle dokázali, že pokud je použita kompresní funkce, která je bezkolizní, tak i celá jejich konstrukce je bezkolizní.

Jak ale takovou funkci vlastně vytvořit? Taková funkce musí být jednocestná a těch moc známo není. Kryptologové si tedy pomohli znalostmi z oblasti blokových šifer. Kvalitní bloková šifra *Ek(x)* se totiž při stejném klíči *k* chová jako náhodné orákulum. I když známe spoustu vzorů a obrazů, tak nejsme schopni určit klíč a nejsme schopni zjistit zbytek zobrazení. Na známé vstupy odpovídá šifra stejnými výstupy, ale pro další vstupy nám to nijak nepomůže. Známe-li jakoukoliv množinu vstupů a výstupů, stejně nemůžeme pomocí nich, kvůli výpočetní složitosti, určit klíč *k*. Bloková šifra se tedy chová jako náhodné orákulum a vzhledem ke klíči *k* je jednocestná. Z této úvahy již konstrukce hashovací funkce

vyplývá. Místo klíče *k* použijeme blok zprávy a šifrujeme pomocí něj kontext. Jinak zapsáno takto: $h_i = E_{mi}(h_{i-1})$, kde *E* je kvalitní bloková šifra.

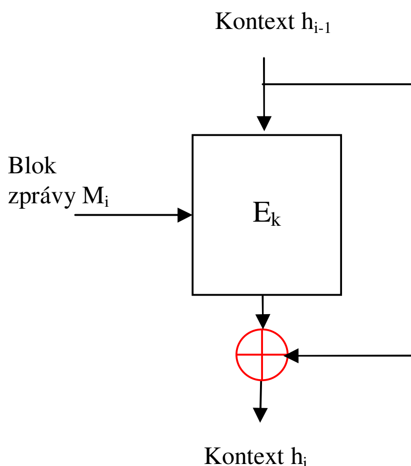


Obr. 4: Porovnání použití blokové šifry u šifrování a hashování

Davies-Meyerova konstrukce kompresní funkce

Tato konstrukce zesiluje vlastnost jednocestnosti kompresní funkce, a to přičtením vzoru před výstupem. Formálně zapsáno[3]: $H_i = f(H_{i-1}, m_i) = E_{mi}(H_{i-1})\ xor\ H_{i-1}$.

Výstup je tedy ještě více zamaskován, protože jej ovlivňuje i přičtení samotného vzoru.



Obr. 5: Davies-Meyerova konstrukce kompresní funkce

2.4 Použití hashovacích funkcí

Hashovací funkce mají v praxi řadu různorodých využití, například:

Kontrola integrity (MDC – Modification Detection Codes) – Pomocí hashovacích funkcí se kontroluje shodnost velkých souborů dat (zejména na záznamových médiích) a v telekomunikaci se používají pro získání kontrolních součtů přenášených dat.

Ukládání a kontrola přihlašovacích hesel – Velmi často se využívají hashovací funkce k uložení hesel. Tzn. že se neukládají přímo hesla, ale jen jejich hashové kódy. Toho se využívá v přihlašovacích a autentizačních systémech.

Jednoznačná identifikace dat – Využívá se hlavně vlastnosti bezkoliznosti. Jakémukoliv bloku dat přiřadí jednoznačný identifikátor. Danému využití se říká různě, například jednoznačná reprezentace vzoru, digitální otisk dat nebo jednoznačný identifikátor dat. Využívá se u digitálních podpisů.

Porovnávání obsahu databází – lze vypočítat hashový kód obou databází a jen pomocí něj porovnávat shodnost databází. Přenášíme tak jen malý hashový kód a nemusíme pro porovnání posílat celou databázi.

Prokazování znalosti – Příkladem je konstrukce HMAC, která zpracovává pomocí hashovací funkce nejen zprávu M, ale i nějaký tajný klíč k. Toho lze využít k autentizaci, tedy prokázání znalosti tajného klíče k tak, že dotazovatel odešle výzvu, tzv. „challenge“, a od prokazovatele obdrží odpověď, tzv. „response“: $response = HMAC(challenge, k)$

Tímto postupem prokazovatel prokázal, že zná hodnotu tajného klíče k, přičemž útočníkovi odposlouchávajícímu komunikační kanál hodnota response nijak nepomůže. Mimochodem tuto konstrukci považujeme za nedotčenou současnými útoky na odolnost proti kolizím, neboť není známo žádné oslabení funkce autentizačního kódu.

Autentizace původu dat – Konstrukce HMAC umí detekovat chybu při přenosu zprávy, čímž zabraňuje útočníkovi zprávu změnit, a tak zaručuje autentizaci původu dat.

Pseudonáhodné funkce při tvorbě klíčů z hesel – jinak také nazývané derivace klíčů. Hashovací funkce se v této konstrukci využívá k tvorbě pseudonáhodného šifrovacího klíče z hesla. Princip je v podstatě jednoduchý, získáme hashový kód původního hesla, a ten ještě mnohonásobně zhashujeme. Doporučuje se minimálně tisíckrát. Pak z výsledného hashového kódů využijeme tolik bitů, kolik zrovna potřebujeme. Tuto konstrukci považujeme také za nedotčenou současnými útoky na odolnost proti kolizím.

Pseudonáhodné generátory – Používají se hlavně v případech, kdy máme k dispozici nějaký řetězec dat (tzv. „seed“) s dostatečnou entropií a potřebujeme z tohoto vzorku získat

posloupnost o větší délce, třeba i delší, v řádech gigabajtů apod. Jako seed pak lze použít například předchozí příklad, u internetového bankovnictví se využívá například záznam pohybu myši apod. Rozdíl oproti předchozímu použití není velký. Jen místo vstupu nepoužijeme heslo, ale právě náhodný, námi zvolený zdroj a výsledek se nehashuje stále dokola, ale vzniká dlouhá posloupnost jako výstup. Ani tuto konstrukci nepovažujeme za dotčenou současnými útoky na odolnost proti kolizím.

3 Nástroje a metody pro zlomení

3.1 Útoky metodou hrubé síly

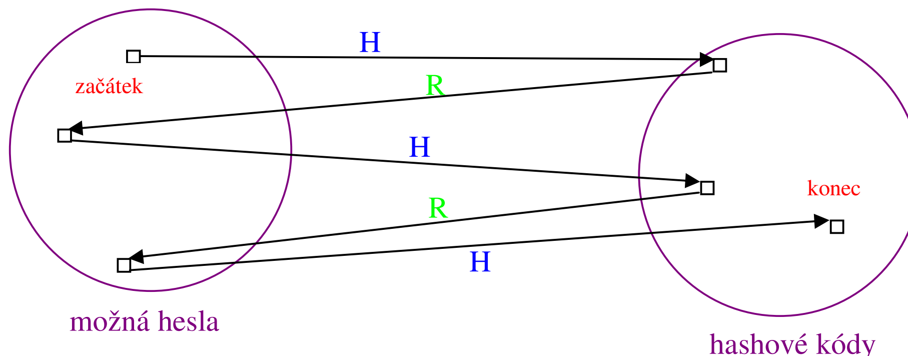
Jinak také nazývané útoky metodou totálních zkoušek, jsou zaměřené hlavně na odolnost hashovacích funkcí proti kolizím. Pokud totiž není hashovací funkce prolomena pomocí kryptoanalýzy, závisí bezpečnost hashovací funkce hlavně na délce hashového kódu. Nechť je výstup z hashovací funkce, délka hashového kódu, n bitů. Poté potřebujeme vyzkoušet 2^n kombinací pro nalezení kolize druhého řádu. Pro hashovací funkci s délkou kódu 160 bitů (například SHA-1) je to tedy 2^{160} kombinací. Nalezení kolize druhého řádu je díky narozeninovému paradoxu snazší, stačí nám $2^{n/2}$ kombinací. Pro uvedenou hashovací funkci tedy 2^{80} kombinací. Příkladem takového útoku byl například projekt MD5crack, který se snažil o nalezení kolize pomocí hrubé síly u hashovací funkce MD5. Cílem bylo přesvědčit bezpečnostní architekty, aby od používání MD5 upustili. V roce 2004 však byl projekt zastaven, protože čínský tým vedený profesorkou Wangovou publikoval zprávu [4], kde dokázala, že pomocí kryptoanalýzy lze nalézt kolize u MD5 mnohem efektivněji.

Útok hrubou silou lze ale aplikovat i jiným způsobem. Pokud je totiž někde uložené heslo ve formě hashového kódu a není pozměněno tzv. solí, tak můžeme z tohoto hashového kódu získat heslo také pomocí tohoto útoku. Většina uživatelů totiž volí jednoduchá hesla bez kombinace číslic a speciálních znaků, a tak je výpočetně možné vyzkoušet všechny kombinace například velkých a malých písmen, pomocí hashovací funkce získat jejich hashový kód, a ten porovnat s uloženým. Doba potřebná k odhalení hesla je pak závislá na délce hesla, jakou uživatel zvolil, a na tom, jestli v heslu použil například i číslice či speciální znaky.

3.2 Slovníkový útok a Rainbow tables

Vzhledem k tomu, že uživatelé často používají jako hesla běžně používaná slova, je možný i slovníkový útok. Připravíme nebo stáhneme již připravený slovník, který běžně obsahuje desetitisíce nejpoužívanějších slov v jednom nebo více jazycích a poté z těchto slov vytvoříme hashové kódy, které při útoku porovnáme s hashovými kódy, ze kterých chceme zjistit hesla..

Jakousi kombinací dvou předchozích metod útoků je útok pomocí Rainbow tables. Tento útok využívá předem připravených tabulek, ve kterých jsou uložena hesla a hashové kódy. Tyto hashové kódy ale nejsou přímým výsledkem hashovaní hesel, ale mezi heslem a uloženým hashovým kódem dojde několikrát k takzvané redukci a opětovnému hashování. Princip ukládání a vyhledávání v takové tabulce se pokusím vysvětlit pomocí Obr. 6.



Obr. 6: Princip ukládání tabulky u útoku Rainbow Tables

Hlavním trikem u Rainbow Tables je nadefinování a používání tzv. redukční funkce *R*, která převádí hashový kód zpět na jiné možné heslo. Jedno z hesel z množiny možných se vybere a vypočítá se jeho hashový kód pomocí hashovací funkce *H*, tento se převede pomocí redukční funkce *R* zpátky na jiné možné heslo, opět se vypočítá jeho hashový kód a toto se opakuje tolikrát, kolikrát si určíme. Konečný hashový kód poté uložíme. Pro další řádek tabulky vybereme další heslo z množiny možných a celý postup opakujeme.

Při vyhledávání hesla pomocí takové tabulky je nejběžnějším postupem porovnat hashový kód, z kterého chceme získat heslo, s konečnými hashovými kódy v tabulce. Pokud tabulka námi vyhledávaný hashový kód neobsahuje, pomocí redukční funkce *R* vypočítáme další možné heslo a z něj pomocí H hashový kód. Ten opět porovnáváme s hashovým kódy v tabulce, tento postup opakujeme. Pokud tabulka hashový kód obsahuje, vezmeme heslo ze stejného řádku a vypočítáváme z něj hashové kódy pomocí *H* a možná hesla pomocí *R* a zpět hashové kódy tak dlouho, dokud nenarazíme na námi vyhledávaný hashový kód. Heslo, z kterého jsme tento hashový kód vypočítali je pak heslem, které hledáme. Útok pomocí Rainbow Tables je tedy jakýmsi kompromisem, mezi útokem s předem připravenou tabulkou všech možných hesel, což je velice paměťově náročné a útokem pomocí hrubé síly, což je velice výpočetně náročné. Tabulka u metody Rainbow Tables je několikrát menší, než předem připravená tabulka všech možných hesel a jejich hashových kódů a zároveň je mnohem rychlejší, než útok hrubou silou.

Metody útoku pomocí Rainbow tables využívají například programy Ophcrack[1], RainbowCrack[2], Cain[3] a jiné.

[1] WWW: http://ophcrack.sourceforge.net/
[2] WWW: http://project-rainbowcrack.com/
[3] WWW: http://www.oxid.it/cain.html

Navíc je možno takovouto předem připravenou tabulku k útoku na konkrétní hashovací funkci získat z Internetu. Dokonce existují stránky, kde lze přímo do formuláře na www stránkách vložit hashový kód, který se na serveru porovná s uloženou databází hashových kódů a ta téměř okamžitě vrátí výsledek – tomuto útoku se také někdy říká „on-line cracking". Vzhledem k tomu, že jsou využívané počítače, které jsou vyhrazené pouze pro počítání dalších a dalších kombinací, vznikají velice rozsáhlé databáze hashových kódů a jim odpovídajících hesel.

Tento útok je tedy velice efektivní a je hlavním důvodem, proč by se měly hashové kódy uživatelských hesel ukládat pouze s tzv. solí. Viz. kapitola 6.2 Hashovací funkce a ukládání hesel.

3.3 Kryptoanalýza

Kryptoanalýza hashovacích funkcí je zaměřena na vnitřní strukturu komprimační funkce *f*. Cílem kryptoanalýzy je pak najít kolizi alespoň u jedné realizace kompresní funkce *f*. Její vnitřní struktura je totiž známá, a navíc je založena na známých blokových šifrách. Diferenciální kryptoanalýza pak je relativně efektivní nástroj sloužící nejen k útokům na hashovací funkce, ale i na různé blokové šifry. Princip diferenciální kryptoanalýzy by se dal jednoduše shrnout jako hledání rozdílů mezi vstupem do kompresní funkce a výstupem z ní. Výstupy a hlavně nalezené anomálie se pak statisticky vyhodnocují. Příkladem kryptoanalýzy je například takzvaná metoda tunelování, popsána v kapitole 4.3 MD5.

4 Současné hashovací funkce.

4.1 LMHash

Název je zkratkou LAN Manager hash. Tato hashovací funkce byla používána k ukládání uživatelských hesel kratších než 15 znaků především v operačních systémech Windows až do verze Windows XP. V novějších Windows Vista je ve výchozím nastavení tato funkce z bezpečnostních důvodů vypnuta, ale pro zpětnou kompatibilitu je stále dostupná.

Hashový kód hashovací funkce LM hash [2] se počítá následovně:

1. Uživatelské heslo je konvertováno na velká písmena.
2. Heslo, pokud je potřeba, je doplněno nulami do velikosti 14 bajtů.
3. Takto doplněné heslo je rozděleno na dvě 7bajtové hodnoty.
4. Tyto hodnoty jsou použity pro vytvoření dvou DES klíčů. Zašifrováním konstant pomocí těchto klíčů pak vzniknou dva kryptogramy.
5. Hashový kód vznikne spojením těchto dvou kryptogramů.

Přestože je tato hashovací funkce založena na velmi dobře známé blokové šifře DES, lze velice jednoduše provést útok díky dvěma zásadním slabinám. První slabinou je skutečnost, že hashovací funkce heslo delší než 7 znaků rozdělí na dvě části, a každou část hashuje separátně. Druhá slabina pak je důsledkem větší uživatelské přívětivosti. Pro případ, že by uživatel měl při vytváření hesla například aktivovanou klávesu Caps Lock, jsou před samotným hashováním všechna malá písmena převedena na písmena velká.

Pokud by tyto dvě slabiny tato funkce neměla, tak by možných hesel teoreticky bylo 95^{14}, tedy 95 znaků ASCII a délka hesla 14 znaků. Vzhledem k rozdělení na půlky však můžeme útočit na každou část separátně, a stačí nám tedy vyzkoušet 95^{7} kombinací. Díky převedení znaků na velká písmena však musíme počet znaků redukovat už jen na *69*. Stačí nám tedy již vyzkoušet jen 69^{7} kombinací, což je přibližně 2^{43} kombinací, a to moderní počítač zvládne pomocí útoku hrubou silou během několika hodin. Vzhledem k tomu, že tato hashovací funkce nepodporuje tzv. „zasolení“, můžeme si všechny možné kombinace předem spočítat, nebo použít běžně dostupné programy jako RainbowCrack, L0phtCrack, CAIN a jiné. Pak je odhalení hesla z hashového kódu záležitostí řádově sekund. Z bezpečnostního hlediska je tedy používání této funkce vyloženě nebezpečné.

Ukázka 14 místného hashového kódu:

- Vstup: *UkazkaHashKodu*
- Výstup: 34E3C1DB803D174A535F9A93DD9B8922

Dalším příklad demonstruje druhou popsanou slabinu, tedy že velikost písmen nemá na výsledný hashový kód vliv:

- Vstup: *UkAzKaHaShKoDu*
- Výstup: 34E3C1DB803D174A535F9A93DD9B8922

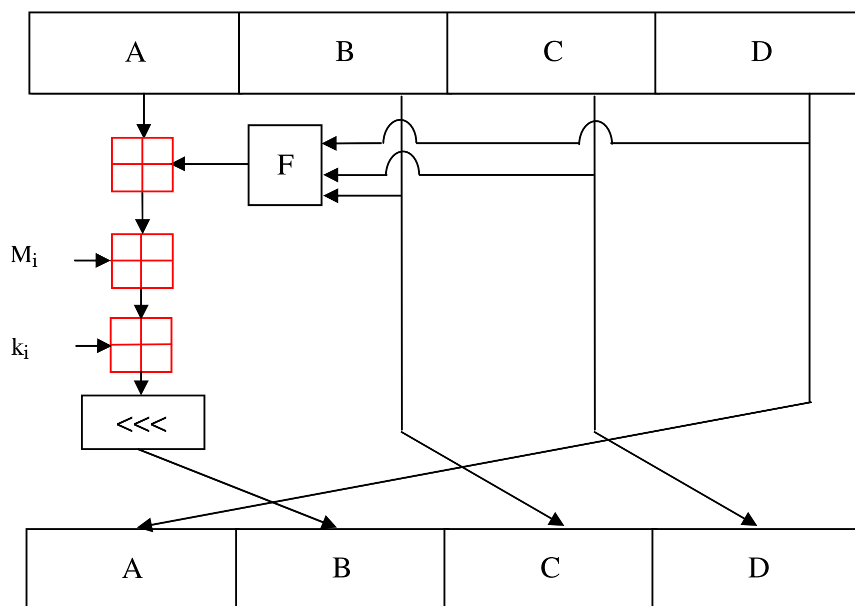
První slabinu lze také velice jednoduše demonstrovat. Díky doplnění vstupu nulami do velikosti 14 bajtů a rozdělení na půlky je druhá část hesla složeného ze 7 písmen vlastně jen zhashovaných sedm nul, takže druhá část kódu je u obou vstupů stejná:

- Vstup1: *prvnipr*
- Výstup1: E75881017480121EAAD3B435B51404EE
- Vstup2: *druhypr*
- Výstup2: 710A766B0F43C931AAD3B435B51404EE

4.2 MD4

Hashovací funkce MD4 [5]byla navržena již v roce 1990 profesorem Ronaldem Rivestem z MIT (Massachusetts Institute of Technology). Na svojí dobu měla pokročilou konstrukci, která se stala inspirací i pro další hashovací funkce jako MD5, funkce z rodiny SHA nebo RIPEMD.

Vstupní data jsou rozdělena na bloky dat o velikosti 512 bitů, zarovnána potřebným počtem bitů a doplněna v posledních 64 bitech posledního bloku o informaci o délce zprávy. MD4 pak pracuje s délkou kontextu 128 bitů rozděleným na čtyři 32 bitová slova A, B, C a D. Každý blok výpočtu se skládá ze 3 rund, z nichž každá obsahuje 16 operací s nelineárními funkcemi F, bitovým xor a levou bitovou rotací. Na Obr. 7 je ukázka jedné takové operace. Výsledný hashový kód je stejně dlouhý jako kontext, tedy 128 bitů.



Obr. 7: Ukázka jedné operace hashovací funkce MD4

Ukázka hashového kódu MD4:

- Vstup: *UkazkaHashKodu*
- Výstup: A1FB04EA608E56E4BF2E0D25FA8D9A32

Bezpečnost MD4

První slabiny MD4 byly publikovány již rok po uvedení, tedy v roce 1991. První kolize pak byla nalezena roku 1996 [6]. Čínský tým soustředěný kolem profesorky Wangové našel velice efektivní způsob, jak generovat u této funkce kolize [4]. Postup byl dále zefektivněn a na dnešních počítačích je možné generovat kolize již v řádu sekund i méně. Z uvedeného je patrné, že dnes již je tato funkce z bezpečnostního hlediska nepoužitelná.

Svět v roce 1996 oblétl velice výmluvný příklad [6] kolize:

****

CONTRACT

At the price of **$176,495** Alf Blowfish sells his house to Ann Bonidea. ....

****

CONTRACT

At the price of **$276,495** Alf Blowfish sells his house to Ann Bonidea. ......

Obr. 8: Kolize u MD4

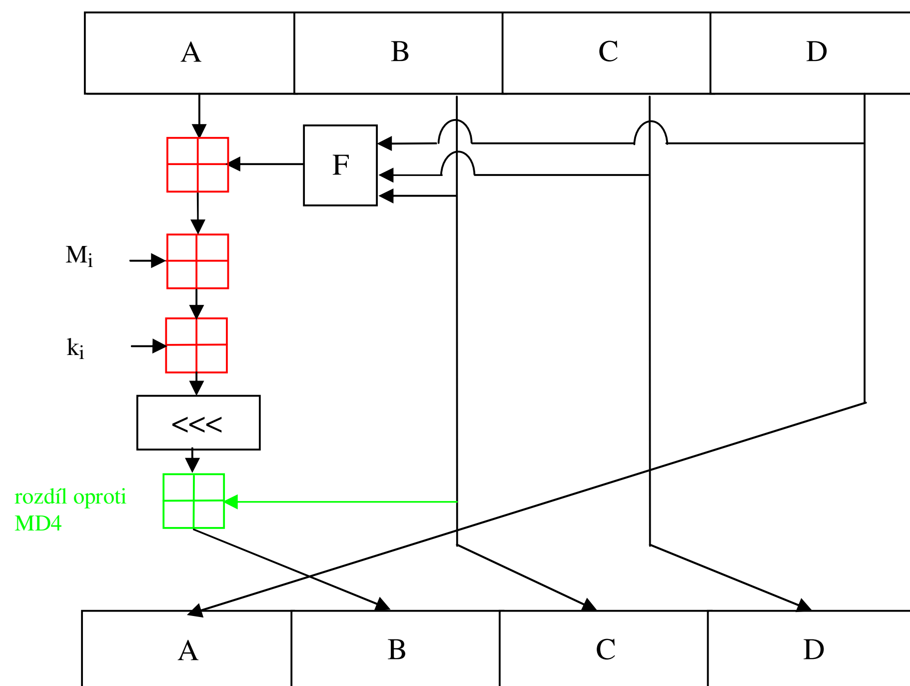
Jak je vidět, tyto zprávy se liší velice málo, ale přesto mají stejný hashový kód.

4.3 MD5

4.3.1 Konstrukce MD5, bezpečnost

Poté co se ukázaly slabiny MD4, navrhl Ronald Rivest již v roce 1991 [7] funkci MD5. MD5 je vlastně jen optimalizovaná funkce MD4, která se snaží odstranit kritické chyby. Používá navíc jedinečnou aditivní konstantu v každém cyklu výpočtu kontextu, takže oproti třem konstantám v MD4 jednu přidává.

Konstrukčně jsou MD4 a MD5 velice podobné. Změna je, že blok výpočtů se neskládá ze tří rund jako MD4, ale navíc jednu rundu přidává. Každá runda také obsahuje 16 operací. Celkově je tedy u MD5 na jeden blok potřeba 64 operací, oproti 48 operacím u MD4. Ukázka jedné takové operace a vyznačený rozdíl oproti MD4 – viz Obr. 9.



Obr. 9: Ukázka jedné operace funkce MD5 s vyznačeným rozdílem od MD4

Ukázka hashového kódu MD5.

- Vstup: UkazkaHashKodu
- Výstup: F98A108F4CCFC291EBDEE7B0BDDDFA8C

Bezpečnost

Již v roce 1994 navrhli pánové P. van Oorschot a M. Wiener paralelně pracující stroj na vyhledávání kolizí, ten by však měl tehdy cenu 10 miliónů dolarů. Tvrdili, že kolizi by uměl vyhledat přibližně za 24 dnů. V roce 1996 na konferenci Eurocrypt prezentoval Dobbertin [8] kolizi kompresní funkce MD5. Demonstroval nalezení *h*, *X* a *X'* tak, že *f(h, X)* = *f(H, X')*. Přesto však mohla být funkce dále používána.

V roce 2004 ale čínský tým vedený profesorkou Wangovou prezentoval na konferenci Crypto 2004 kolize funkcí MD4, MD5, HAVAL-128 a RIPEMD [4]. Nalezli metodu, pomocí níž lze nalézt kolizi dvou 1024 bitových zpráv v řádu několika hodin. Postup spočívá v tom, že se naleznou dva různé 512bitové bloky M_1 a M_2. To trvalo na 32procesorovém počítači zhruba hodinu. Poté je potřeba nalézt k nim další dva bloky N_1 a N_2 tak, že po složení mají zprávy (M_1, N_1) a (M_2, N_2) stejný hashový kód.

Na začátku března 2005 pak Vlastimil Klíma publikoval zprávu [9], ve které popisuje způsob jak nalézt kolizi dokonce na běžně dostupném notebooku, pro libovolnou inicializační

hodnotu, a to ještě efektivněji, než se to povedlo čínskému týmu. Průměrná doba nalezení kolize na notebooku s procesorem Pentium 1,6 GHz pak činila 8 hodin. Poté Vlastimil Klíma přišel s myšlenkou tzv. tunelování [10]. Tunelování nahradilo metodu mnohonásobné modifikace zpráv a zkrátilo čas potřebný pro generování kolizí jen na přibližně 17 sekund na běžném počítači (Pentium 4 – 3,2 GHz).

4.3.2 Tunelování

Tuto metodu bych tu chtěl krátce zmínit, protože je velice revoluční, otevřela novou možnost pro kryptoanalýzu hashovacích funkcí a představuje již dnes, ale hlavně v dohledné budoucnosti další bezpečnostní problém dnešních hashovacích funkcí. Autor Vlastimil Klíma je přesvědčen, že se naleznou způsoby, jak konstruovat diferenční schémata (pro hashovací funkce jako jsou MD5, SHA-0, SHA-1, SHA-2, ale i jiné), která již budou zaměřena přímo na existenci využitelných tunelů v nich obsažených.

První kolize u hashovací funkce MD5 byly nalezeny pomocí metody mnohonásobné modifikace zpráv [4]. Tato metoda vede k naplňování množiny postačujících podmínek od začátku zprávy až do dosažení tzv. bodu verifikace (POV), za kterým již nejsme schopni nic ovlivnit. Vzhledem k tomu, že za tímto bodem již nejsme schopni ovlivnit výpočty, je pro nalezení kolize potřeba najít velké množství takových bodů, aby se náhodně splnily i všechny dostačující podmínky. U hashovací funkce MD5 je to konkrétně 2^{29} bodů POV. Jeden z těchto bodů POV bude splňovat i zbývající postačující podmínky, a ten je pak použit pro kolizi.

Metoda tunelování naproti tomu začíná až v bodě POV. Několika tzv. tunely z takovéhoto bodu vytvoří geometrickou řadou dostatečné množství dalších POV, aniž přitom narušíme počáteční podmínky před bodem POV. Těchto tunelů je například u MD5 známo více, tyto mají tzv. „sílu“ n a vytvoříme pomocí nich 2^n bodů POV. Tunely se dají navíc kombinovat, takže z jednoho původního bodu POV lze vytvořit jedním tunelem 2^{n_1} bodů POV, a z každého z nich pak získat dalším tunelem 2^{n_2} dalších bodů POV. U MD5 je například známa kombinace tunelů, která dohromady dává sílu $n = 24$. Z každého jednoho bodu POV tak jsme schopni vytvořit 2^{24} dalších bodů POV. Stačí tedy vygenerovat jen $2^5 = 32$ nestejných bodů POV pro nalezení kolize. To je velice malé číslo oproti 2^{29} potřebných bodů POV u metody mnohonásobné modifikace zpráv.

4.4 SHA-0 a SHA-1

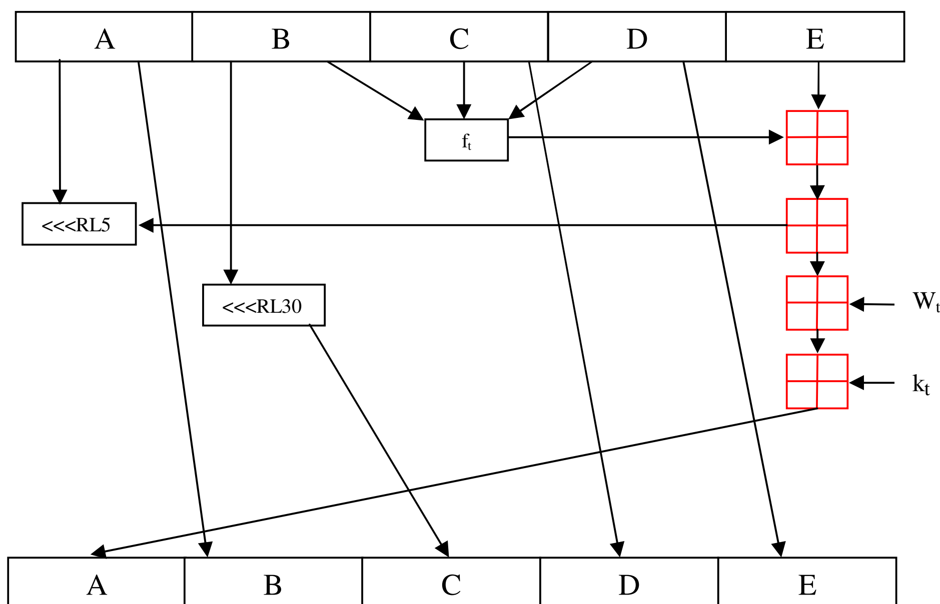
Jedná se o hashovací funkce přímo vycházející z konstrukce funkcí MD4 a MD5. SHA-0 byla publikována úřadem NIST již v roce 1993 [11]. Tato verze však byla kvůli blíže nespecifikované chybě stažena a jako standard byla schválena v roce 1995 pokročilejší verze

SHA-1 [12]. SHA-1 se liší od původní SHA-0 přidáním jedné bitové operace do expanzní funkce vstupních dat, jež je součástí kompresní funkce.

Délka vstupu může být až $2^{64} - 1$. Vstup je zarovnán a rozdělen na bloky o délce 512 bitů. Výstupní hashový kód má délku 160 bitů. Pracuje tedy s kontextem o velikosti 160 bitů, který je rozdělen na pět 32bitových slov: A, B, C, D a E. Výpočet se pak skládá z 4 rund, z nichž každá obsahuje 20 operací založených na nelinearních funkcích *F*, bitovém xor a levé bitové operaci. Viz Obr. 10.

Ukázka hashového kódu SHA-1.

- Vstup: *UkazkaHashKodu*
- Výstup: FB8F0B19C9D24AA7033D7638701119F8E48D8714



Obr. 10: Kompresní funkce SHA-1

Bezpečnost

SHA-0 je již považována za prolomenou. Již v roce 1998 byl publikován způsob [13], jak lze generovat kolizi se složitostí 2^{61}. Čínský tým pod vedením profesorky Wangové pak vydal v roce 2005 zprávu[13], v níž ukazují plnou kolizi SHA-0. To vše se složitostí jen 2^{39} operací.

SHA-1 je v současné době stále velmi používanou hashovací funkcí. Soustředí se na ní tedy velká pozornost. NIST doporučil tuto funkci přestat používat do roku 2010. V únoru

2005 tým profesorky Wangové publikoval zprávu [14], kde dokazují, že lze nalézt kolizi se složitostí 2^{69}, kdežto teoretická složitost pro nalezení prvního vzoru by měla být $2^{n/2}$, tedy 2^{80}. V říjnu téhož roku pak publikovali další zprávu [15], kde metodu ještě vylepšili – složitost nalezení prvního vzoru snížili na 2^{63}. To je pořád velmi vysoké číslo a žádná kolize ještě nebyla veřejně demonstrována, vzhledem k narůstajícím rychlostem počítačů se však soudí, že při použití distribuovaného výpočtu je útok dosažitelný již dnes. Z bezpečnostního hlediska tedy nelze považovat tuto funkci za bezpečnou. Vývojáři ji však pravděpodobně budou používat nadále, minimálně do doby, než bude veřejně představena alespoň jedna kolize této funkce.

4.5 Rodina SHA-2

Tato rodina hashovacích funkcí [16] obsahuje několik variant – SHA-224, SHA-256, SHA-384 a SHA-512. SHA-2 je jejich souhrnné označení. Číslo za názvem algoritmu značí délku výstupu v bitech. Složitost nalezení prvního vzoru je tedy pro tyto funkce 2^{112}, 2^{128}, 2^{192} a 2^{256}.

Tyto funkce lze rozdělit na dvě podskupiny, kdy SHA-224 je vlastně jen varianta SHA-256 s kratším výstupním hashovým kódem a jinými inicializačními vektory. Podobně i funkce SHA-384 je odvozena od SHA-512. Výstup odvozených funkcí vznikne jen zkrácením výstupu funkcí, ze kterých jsou odvozeny. Protože mají jiné inicializační vektory, mají rozdílný hashový kód, takže například výstupní hashový kód funkce SHA-224 není pouze kusem hashového kódu funkce SHA-256.

Algoritmus u funkcí **SHA-224 a SHA-256** pracuje s kontextem 256 bitů. Ten se rozdělí na osm 32 bitových slov A, B, C, D, E, F, G a H. V kompresní funkci pak zpracovává bloky dat o velikosti 512 bitů, pomocí kterých modifikuje kontext. Každý blok výpočtu se pak skládá z 64 operací založených na operacích +*, and, or, xor, shr a rotr*. Velikost vstupních dat může mít délku až $2^{64}-1$ bitů.

SHA-384, SHA-512 se od předchozích dvou liší samozřejmě velikostí výstupního hashového kódu a tím i velikostí kontextu, který u těchto funkcí má velikost 512 bitů. Ten se rozdělí na osm 64 bitových slov A, B, C, D, E, F, G a H. V kompresní funkci pak zpracovává bloky dat o velikosti 1024 bitů, pomocí kterého modifikuje kontext. Každý blok výpočtu se pak skládá z 80 operací založených opět na operacích +*, and, or, xor, shr a rotr*. Velikost vstupních dat pak může mít délku dokonce až $2^{128}-1$ bitů.

Bezpečnost

Co se týče konstrukce, jsou tyto hashovací funkce velice podobné funkci SHA-1 a starším. Pracují však se složitějšími funkcemi a širšími vstupy, poskytují tak větší odolnost proti kolizi a jsou považovány za bezpečné. Začínají se objevovat první útoky na tyto funkce. Například

nalezené kolize pro až 23 a 24 kroků funkce SHA-256 se složitostí přibližně 2^{50} a 2^{53} [17]. Tyto útoky však zatím nepředstavují reálné riziko.

Ukázka hashového kódu funkce SHA-512:

- Vstup: *UkazkaHashKodu*
- Výstup: F9DAE2F23CADC5D90CB279E5FC3F38FBA72E26F33350222DCBFC5C133C434860

5 Budoucnost hashovacích funkcí

5.1 Problémy konstrukce dnešních hashovacích funkcí

Čím dál více se ukazuje, že všechny současné hashovací funkce jsou už ze své podstaty slabé. Objevuje se čím dál více postupů, jak různé hashovací funkce oslabit. Proč tomu tak je? Vzhledem k tomu, že hashovací funkce musí plnit svoji funkci a zachovávat si všechny vlastnosti i při hashování jediného bloku, můžeme si vlastně zkoumání zjednodušit pouze na zkoumání kompresní funkce.

Všechny dnešní hashovací funkce používají v kompresních funkcích blokové šifry. Jenže předpokladem u konstrukce blokové šifry bylo, že obsahuje hodnotu, kterou útočník buď nezná, nebo nemůže modifikovat – šifrovací klíč. Pomocí této neznámé hodnoty pak utajuje způsob převodu otevřeného textu na zašifrovaný a naopak. Jenže u hashovacích funkcí může manipulovat se všemi vstupy. A to je základní bezpečnostní nedostatek dnešních hashovacích funkcí, žádná klasická bloková šifra není připravena na situaci, kdy útočník zná šifrovací klíč.

U hashovacích funkcí je však klíčem zpráva, kterou hashujeme. Klíč je tedy útočníkovi volně k dispozici a může jej dokonce libovolně modifikovat, na to nejsou blokové šifry vůbec připraveny. Jak uvádí Vlastimil Klíma [18], přestože by snad blokové šifry teoreticky pro bezpečnou kompresní funkci použít šly, nelze to udělat efektivně. Hashovací funkce ale musí být efektivní, a tak se tento problém se dostává do rozporu s bezpečností.

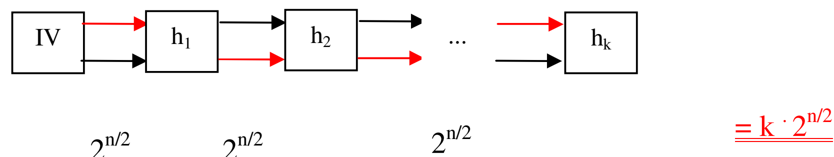
Bohužel podmínky a kritéria nově vznikajícího standardu SHA-3 jsou nastaveny velice konzervativně a při pohledu na zaslané příspěvky je patrné, že tento koncept zůstane zachován, což může do budoucna znamenat velké problémy.

5.1.1 Generické problémy iterativních hashovacích funkcí

Posledních několik let se ukazuje [19][20], že dokonce samotná iterativní konstrukce hashovacích funkcí implikuje odlišnost těchto funkcí od náhodného orákula, a že tato konstrukce má svoje slabiny. Sama o sobě generuje problémy, protože bez ohledu na to, co je uvnitř hashovací funkce, umožňuje generovat multikolize a kaskádovitá konstrukce u ní pozbývá smyslu.

5.1.2 R-násobná kolize s nižší složitostí

Jednu z prací na toto téma prezentoval Joux [19] na konferenci Crypto 2004. Uvádí se v ní, že generovat mnohonásobné kolize je mnohem jednodušší, než ve srovnání s náhodným orákulem. Nalezení mnohonásobné kolize by mělo mít matematicky složitost $2^{n(r-1)/r}$, ale Joux ukázal, že lze tyto kolize produkovat jednodušeji, a to bez ohledu na to, co je uvnitř hashovací funkce.



Obr. 11: Ukázka nalezení multikolize s nižší složitostí

Princip je až překvapivě jednoduchý. Vezmeme inicializační vektor *IV*, produkujeme kolizi, vedoucí na kontext se stejným hashovým kódem, což umíme se složitostí $2^{n/2}$. Poté produkuje druhou kolizi, vedoucí z jednoho kontextu na druhý, což umíme opět se složitostí $2^{n/2}$. Tento postup opakujeme k-krát až dostaneme kontext h_k.

Trik spočívá v tom, že si lze vybrat u kolizí vždy jednu z cest u nalezených kolizí jednotlivých kontextů. Dostaneme tedy 2^k multikolizí. Složitost by měla být zhruba 2^n, ale místo toho je $k.2^{n/2}$. U tohoto postupu je naprosto jedno, jaký je vnitřek hashovací funkce. Jediná záchrana tedy je dostatečně složitá hashovací funkce, která by zaručila, aby tento postup byl výpočetně nemožný.

5.1.3 Kaskádovitá konstrukce

Když se ukázalo, že i používané hashovací funkce jako MD5, SHA-0 a SHA-1 jsou prolomeny, předpokládalo se, že jejich nepoužitelnost oddálí využití kaskádovité konstrukce.

Myšlenkou je zřetězení dvou hashovacích funkcí jako například MD5 a SHA-1. Výstupní hashový kód této konstrukce má délku 128 + 160 bitů, tedy dostatečných 288 bitů. Některé certifikační autority tuto konstrukci ještě relativně nedávno používaly, některé možná dokonce doteď používají. Předpokladem je, že složitost $S(F \parallel G)$ nalezení kolize hashového kódu bude rovna součinu nalezení kolize dílčích hashových kódů $S(F \parallel G)=S(F)*S(G)$.

Joux ve své práci [19] ale dokázal, že tato konstrukce složitost navyšuje jen velmi málo. Postup vychází z postupu nalézání r-násobných multikolizí. Pokud je první hashovací funkce F iterativní s délkou hashového kódu $n_f < n_g$, můžeme vytvořit $n_g/2$ návazných kolizí funkce F se složitostí $n_g/2*S(F)$. Tím dostaneme $2^{ng/2}$-násobnou kolizi vzhledem k funkci F.

Poté nám stačí mezi těmito zprávami najít jednu kolizi vzhledem k funkci G, a dostaneme dvě zprávy, které mají stejný hashový kód vzhledem k F i G. Složitost tohoto výpočtu

je rovna $n_g/2*S(F)+2^{ng/2}$, což je přibližně $S(F) + S(G)$, tj. řádově mnohem méně než očekávaná složitost $S(F \,||\, G)$. Tato konstrukce tedy nemá smysl.

5.1.4 Nalezení druhého vzoru u dlouhých zpráv snadněji než se složitostí 2^n

Autoři Kelsey a Schneier publikovali zprávu [20], ve které dokazují, že lze pro dlouhé zprávy o délce přibližně 2^k blízké $2^{n/2}$, zkonstruovat druhý vzor u iterativních hashovacích funkcí s mnohem menší složitostí. A to díky Davies-Meyerově konstrukci, která umožňuje vytvoření tzv. pevného bodu $h_j = f(h_j,N_j)$ a vytvoření několika seznamů průběžných kontextů, mezi kterými pak hledáme kolizi. Mezi těmito seznamy tedy nalezneme kolizi, ale takové zprávy mají různou délku. Využijeme tedy pevného bodu, který může být vložen do funkce tolikrát, kolikrát je potřeba. Tím dostaneme potřebný počet bloků, přičemž kontext se nezmění. Za takto nalezené dvě zprávy, které vedou na kontext se stejným hashovým kódem, připojíme zbytek zprávy *M,* a tím dostaneme druhý vzor zprávy *M*. To vše se složitostí $2^{n/2} + 1 + 2^{n-k} + 1$, což je mnohem méně, než očekávaných 2^n. Například u funkce SHA-1 můžeme tímto postupem vygenerovat druhý vzor se složitostí 2^{106}, což dnes sice stále dostačuje, ale je to řádově mnohem méně než teoretických 2^{160}.

5.2 NIST, soutěž SHA-3

Hashovací funkce mají mnoho využití, například při ukládání hesel, bezpečném připojení do sítě/Internetu, pomáhají při skenování virů, a desítky dalších využití. V podstatě by bez nich Internet, tak jak jej známe, nemohl existovat. Na to, jak jsou v dnešním světě důležité, jsou ale prozkoumány relativně málo a jsou velmi málo studovány. Nejen tento problém chce vyřešit americký Národní Institut pro standardy a technologie, zkráceně NIST (The National Institute of Standards and Technology), soutěží o nový standard SHA-3, který má nahradit hashovací funkce ze stávající rodiny SHA. Algoritmy použité v této rodině jsou považovány za jednoduché a útoky na tyto funkce přibývají. Již dnes je jisté, že prolomení těchto funkcí je v podstatě jen otázka času. NIST tedy chce tyto funkce nahradit právě SHA-3.

NIST zvolil veřejnou soutěž. S tímto postupem už slavil úspěch, kdy v roce 1997 vyhlásil soutěž o blokovou šifru, která nahradila DES. Tehdy se přihlásilo 15 kandidátů. Vznikl úspěšný Advanced Encryption Standard – AES [31].

Příspěvky do soutěže o SHA-3 mohly být posílány do 31. října 2008. Bylo přijato 64 příspěvků. Většinu příspěvků do soutěže AES poslali profesoři. Co se týče zasílatelů do soutěže SHA-3, účast již je různorodější, zasílatelé jsou nejenom z akademické půdy, ale i z průmyslu, nebo od lidí, kteří mají kryptografii jen jako hobby. Zajímavostí je, že jednomu ze zasílatelů je dokonce jen 15 let. Na neoficiálních internetových stránkách The SHA-3 ZOO [21] jsou kandidáti vypsání, společně s výčtem hashovacích funkcí, u kterých již byli objeveny slabiny nebo na ně byl podniknut úspěšný útok. 56 přihlášených funkcí bylo

v době psaní této diplomové práce známo. Do prvního kola postoupilo 51 hashovacích funkcí. Odborná veřejnost a samozřejmě i zasílatelé mezi sebou, se snaží poukázat na slabosti hashovacích funkcí, nebo je dokonce prolomit. Nutno říci, že úspěšně. Situace se mění téměř každým dnem a neoficiálně [21] zůstává neprolomeno či bez větších slabin již jen 26 příspěvků. Ty obsahují hashovací funkce: ARIRANG, BLAKE, Blue Midnight Wish, CHI, CRUNCH, CubeHash, ECHO, Edon-R, ESSENCE, FSB, Fugue, Grøstl, Hamsi, JH, Keccak, Lane, Lesamnta, Luffa, MD6, SANDstorm, Shabal, SHAvite-3, SIMD, Skein, SWIFFTX, TIB3.

NIST chce co nejrychleji vybrat cca 15 nejslibnějších příspěvků a ty předložit veřejnosti minimálně rok ke kryptoanalýze. Poté bude následovat ještě užší výběr, až nakonec v roce 2011 by měl být vybrán vítěz. Finální standard je očekáván v roce 2012.

Jako zajímavost bych zde chtěl zmínit dva návrhy, u kterých je spoluautorem Vlastimil Klíma. Návrhy obsahují dva nejrychlejší (nejvýkonnější) algoritmy soutěže:

Blue Midnight Wish

Doprovodná dokumentace [30] k této funkci uvádí, že tato funkce má výstup dlouhý 224, 256, 384, nebo 512 bitů, tedy stejně jako SHA-2. Dále uvádí, že tato funkce je odolná proti útoku na délku zprávy a proti multikoliznímu útoku. Také je tato funkce mnohem efektivnější (výkonnější) než SHA-2 při zachování stejné, nebo lepší bezpečnosti. Søren S. Thomsen však publikoval dokument, kde uvádí, že nalezení tzv. blízké kolize (tj. nalezení dvou vstupů, které se liší jen velice málo, tedy o pár bitů, ale které vedou na stejný hashový kód) je u této funkce jednodušší, než by matematicky mělo být, a to u všech verzí této funkce.

EDON-R

Tento návrh je ze všech zaslaných návrhů hashovacích funkcí v soutěži SHA-3 nejrychlejší (nejvýkonnější). Základním prvkem jsou kvazigrupy. Kvazigrupa je vlastně grupoid, na kterém lze neomezeně provádět inverzní operaci. Grupoid je algebraická struktura s jednou operací. Kvazigrupové operace jsou v tomto návrhu definovány pomocí rychlých a dostupných operací implementovaných ve všech procesorech: sčítání, xor a bitové rotace. V použití kvazigrup je kouzlo rychlosti této navrhované hashovací funkce a zároveň nejvíce kritizovanou částí návrhu, protože přispěvatelé jiných návrhu namítají, že kvazigrupy jsou neprobádanou oblastí a snaží se o diskvalifikaci tohoto návrhu.

5.3 Hashovací funkce HDN typu SNMAC

Problémy dnešních hashovacích funkcí již byly v této práci rozebrány. Jak navrhnout nové hashovací funkce? Vlastimil Klíma a kolegové uvedli na kryptologické konferenci Eurocrypt 2007 velice zajímavý koncept [22],[24] tzv. hashovacích funkcí nové generace.

5.3.1 Nové kryptografické primitivum

Kompresní funkce obsažená v hashovacích funkcích by měla být hlavně a přednostně jednosměrná. Jak již bylo řečeno, použití blokové šifry je tedy nevhodné, protože pomocí ní lze dešifrovat. To je samozřejmě u hashovacích funkcí nepřípustné, a tak je v kompresních funkcích bloková šifra různě modifikována, aby se i přes znalost klíče stala jednocestnou funkcí. To je ale již jen nástavba, která může způsobovat další problémy. Měla by být tedy použita funkce, která dešifrovat neumí.

Nabízí se zajímavá analogie s kryptografií s veřejným klíčem. Zde jsou totiž použity jednosměrné funkce, bohužel ale s tzv. padacími vrátky, kdy útočník není schopen dešifrovat, ale majitel soukromého klíče ano. To je však pro hashovací funkci opět nevhodné. Pokud bychom tato tzv. padací vrátka odstranili, dostaneme ideální konstrukci.

Na základě této myšlenky bylo stvořeno nové kryptografické primitivum, a to speciální bloková šifra. Základní vlastností a myšlenkou je manipulovatelnost s klíčem a homogenita ve zpracování všech vstupních bitů. Takové vlastnosti nemá žádná současná klasická bloková šifra. Konstrukce speciální blokové šifry pak je: $f\colon \{0, 1\}^K \rightarrow \{0, 1\}^n : k \rightarrow Ek(Const_0)$. $Const_0$ je konstanta, *E* je vhodná (speciální) bloková šifra.

Její vlastnosti jsou [23]:

- Zpracovává klíč na stejné úrovni kvality jako datový vstup.
- Zpracovává všechny bity klíče stejně kvalitně (homogenně).
- Na rozdíl od klasických blokových šifer bude přirozené použít délku klíče obvykle mnohonásobně delší než délku bloku, například $K = 4096$, resp. *8192* a $n = 256$, resp. *512*.
- Je konstruována pomocí technologie blokových šifer.
- Není primárně určena k šifrování dat.
- Je použita v hashovací funkci s konstantním otevřeným textem, veškerá proměnná vstupuje do *E* prostřednictvím klíče.
- Když uvažujeme, že má také proměnný otevřený text, měla by to být kryptograficky silná klasická bloková šifra.
- Útočník může libovolně manipulovat s klíčem.

Definice speciální blokové šifry však stále ještě není úplně uzavřena a dále probíhá zkoumání.

5.3.2 Hashovací funkce nové generace SNMAC

SNMAC je odvozeno od konstrukce NMAC [24], proto i tento název, který je zkratkou Special Nested Message Authentication Code. Tato hashovací funkce vznikne, když právě do konstrukce NMAC zakomponujeme náhodné orákulum *f* pomocí speciální blokové šifry: $f\colon \{0, 1\}^K \rightarrow \{0, 1\}^n : k \rightarrow Ek(Const_0)$ a místo náhodného orákula g zakomponujeme speciální blokovou šifru $g\colon \{0, 1\}^K \rightarrow \{0, 1\}^n : k \rightarrow E_k(Const_0)$.

5.3.3 Hashovací funkce HDN

Konkrétní realizací konstrukce SNMAC je hashovací funkce HDN. Její hashový kód je dlouhý 512 bitů a podle autorů je odolná proti všem známým útokům na dnešní hashovací funkce, dokonce by měla být odolná i proti útokům, jako jsou útoky postranními kanály, útoky příbuznými klíči, pravoúhelníkové útoky a jiné. Její rychlost je podle testů ve srovnání s SHA-512 pouze asi tři až čtyřikrát nižší. V porovnání s dnešními hashovacími funkcemi je pak mnohem robustnější a má velkou bezpečnostní rezervu. Navíc pokud se doopravdy realizuje, bude mít veřejná návrhová kritéria a již dnes jsou dokázány její teoretické vlastnosti a existuje důkaz výpočetní odolnosti proti kolizi a nalezení vzoru. To jsou vlastnosti, které žádná současná hashovací funkce nemá, a ani SHA-3 je pravděpodobně mít nebude.

6 Hashovací funkce a autentizace

6.1 Autentizace

Autentizace patří mezi bezpečnostní opatření týkající se „řízení přístupu“. Právoplatným uživatelům potřebujeme umožnit přístup k aktivům (například ke službě) a naopak útočníkovi v takovém přístupu zabránit.

Autentizaci rozdělujeme:

- Autentizace znalostí – uživatel prokazuje svoji identitu znalostí například hesla, kódu PIN apod.
- Autentizace žadatelem – uživatel prokazuje svoji identitu vlastní charakteristikou, jakou je například hlas, otisk prstu apod.
- Autentizace předmětem – uživatel prokazuje svoji identitu předmětem, kterým může být například platební karta, flash karta, token apod.

Nás pak v souvislosti z hashovacími funkcemi bude zajímat autentizace znalostí.

6.2 Hashovací funkce a ukládání hesel

U autentizace znalostí je problém, jak heslo bezpečně uložit. Heslo je totiž třeba na straně kontroléru nějakým způsobem uložit. Pokud by se heslo ukládalo jako obyčejný text, vznikl by obrovský bezpečnostní nedostatek a útočník by přístupem na sdílenou paměť (tzn. třeba pevný disk počítače/serveru) získal všechna hesla na něm uložené. Pokud hesla zašifrujeme, vznikne vlastně stejný problém, protože musíme někde bezpečně uložit dešifrovací klíče.

Právě na ukládání hesel se díky vlastnostem bezkoliznosti a jednocestnosti výborně hodí hashovací funkce. Hesla jsou ukládány jen ve formě jejich hashových kódů. Na takto uložený hashový kód by však šly velice dobře použít útoky popsané v kapitolách „Útoky metodou hrubé síly“ a „Slovníkový útok a Rainbow tables“. V praxi se tedy ukládání pomocí hashovacích funkcí doplňuje o techniku mnohonásobné iterace a zasolení.

Mnohonásobná iterace znamená, že při ukládání neprojde heslo hashovací funkcí jen jednou, ale několikrát. Teprve výstup z x-tého opakování hashování je hashový kód, který použijeme pro uložení, případně při autentizaci pro kontrolu s kódem již uloženým.

Proces hashování a tím pádem i celé autentizace se sice x-krát zpomalí, ale pro jedno heslo je toto zdržení zanedbatelné, přičemž pro útočníka může jít o podstatné zpomalení, které zapříčiní, že heslo již nepůjde například při útoku metodou hrubé síly vypočítat. Navíc slovníkové útoky i útok pomocí rainbow tables je podstatně znesnadněn, protože všechna hesla musí přepočítat.

Zasolení je další z technik, zesilujících bezpečnost hesel uložených ve formě hashového kódu. K heslu přidáme nějaký námi zvolený nebo vygenerovaný další text, a teprve z této kombinace vytvoříme hashový kód.

Na rozdíl od šifrovacího klíče nevyžaduje tato sůl nijakou zvláštní ochranu nebo utajení. Slouží hlavně k zamezení použití slovníkového útoku nebo útoku pomocí Rainbow tables. Samozřejmě útočník může tuto sůl získat a v některých případech i velice jednoduše, ale pro útok by si musel všechna hesla přepočítat právě s touto solí.

V rámci jedné aplikace můžeme používat na všechna hesla stejnou sůl, ale to pak hrozí, že útočník může s nově vypočítaným slovníkem nebo Rainbow tables útočit na všechna uložená hesla. Doporučení tedy je, aby sůl byla generována pro každého uživatele zvlášť.

Tento požadavek lze v praxi uspokojit například tak, že se k heslu hashuje i uživatelské jméno, nebo tak, že se heslo před vlastním hashováním zdvojí, takže například z hesla „pavel“ vznikne vstup „pavelpavel“. Náročnost odhalení jednoho konkrétního hesla zůstává stejná, ale se slovníky či tabulkami připravenými pro jednoho konkrétního uživatele již nelze útočit na hesla dalších uživatelů.

6.2.1 Ukládání hesel v OS Windows

V operačních systémech Microsoft Windows se hesla ukládají ve formátech LMHash a NTHash. LMHash je popsán v kapitole „LMHash“ této práce, NTHash pak používá pro výpočet hashového kódu hashovací funkci MD4, která je popsána v kapitole 4.2 MD4.

Windows 95 a Windows 98 používaly k ukládání hesel pouze LMHash. Windows založené na jádře NT 4.0 a vyšší pak používají i NTHash. Z důvodů zpětné kompatibility v případech, že v síti mohou být i počítače s Windows 95/98, se hesla ukládají nejen pomocí NTHash, ale i pomocí LMHash, a to i v novějších systémech Windows.

Teprve ve verzi Windows XP jde využívání hashovací funkce LMHash zakázat, v základním nastavení je však povolena. Ukládání hesel pomocí LMHash je v základním nastavení vypnuto až v nejnovějším OS Windows Vista.

To představuje velké bezpečnostní riziko, protože jak bylo popsáno v kapitole „LMHash“, tato hashovací funkce je velmi jednoduše napadnutelná a i nejpomalejší útok pomocí hrubé síly odhalí heslo maximálně v řádu hodin. NTHash je mnohem bezpečnější. Bohužel stejně jako LMHash nepodporuje zasolení. Slovníkový útok a útok pomocí Rainbow tables tak může být velice efektivní. Příkladem je zcela legálně stažitelná tabulka, která má velikost přibližně 8GB a obsahuje hesla

- do délky 6 znaků, používajíc znaky

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$% &'()*+,-./:;<=>?@[\]^_`{|}~ (včetně mezery)

- délky 7 znaků, používajíc znaky

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

- délky 8 znaků, používajíc znaky

0123456789abcdefghijklmnopqrstuvwxyz

S tolika hesly je již velká pravděpodobnost úspěchu.

Z bezpečnostního hlediska je doporučeno volit dlouhá hesla, nejlépe i s použitím nestandardních znaků. Pro OS Windows založené na jádře NT je pak silně doporučováno vypnout ukládání hesel pomocí LMHash. Windows Vista již hesla pomocí LMHash standardně neukládají.

6.2.2 Ukládání hesel v OS Linux

Dříve se v operačních systémech Unix a Linux používalo ukládání hesel pomocí šifrování DES. V podstatě byla tato bloková šifra použita jako hashovací funkce, jejímž klíčem bylo upravené heslo se solí. Tato šifra však používala klíč délky jen 56 bitů a pro dnešní počítače již není problém takto uložené heslo odhalit pomocí útoku hrubou silou.

Začala se tedy využívat hashovací funkce MD5. Ta není použita přímo, nýbrž jen jako základní funkční prvek ve složitější konstrukci. Vzhledem k tomu, že se hashový kód pro každého uživatele mění díky „zasolení“, je slovníkový a rainbow tables útok velmi obtížný. Ani odhalené slabiny v odolnosti této funkce proti kolizi prvního řádu nemají na takto uložená hesla praktický vliv. Vzhledem k odhaleným slabinám se ale doporučuje přejít na ukládání hesel s využitím bezpečnějších hashovacích funkcí, v praxi nejčastěji SHA-512, nebo využít šifry BlowFish.

6.3 Autentizační protokoly

Jsou protokoly sloužící k samotné autentizaci, využívajíc přitom právě hashovací funkce popsané výše.

6.3.1 Přímé ověření

Protokoly sloužící k přímému ověření nejčastěji používají systém *challenge-response* (výzva-odpověď). Posílat totiž po komunikačním kanálu heslo, byť zašifrované, je z bezpečnostního hlediska nevhodné. Díky tomuto systému může probíhat autentizace mnohem bezpečněji. Je totiž založena na hashovacích funkcích.

Klient, který chce například získat přístup na server s požadovanou službou, tento server kontaktuje. K tomu se ale musí autentizovat, prokázat že zná tajnou informaci. Server pošle klientovi nazpět nějakou náhodně vygenerovanou hodnotu. Klient z této hodnoty a své tajné informace (např. hesla) získá hashový kód a tento pošle serveru. Server získá hashový kód stejným způsobem a porovná s kódem došlým od klienta. Pokud se shodují, je klientovi povolen přístup. Tento systém je velmi používaný, protože v žádné části autentizace se komunikačním kanálem nepřenáší heslo v žádné podobě.

LANMAN

Protokol, který nativně používají starší verze OS Windows jako Windows 95, Windows 98 a Windows ME. Bohužel kvůli zpětné kompatibilitě jej obsahují i novější Windows až do Windows verze XP. To představuje bezpečnostní riziko, protože tento protokol využívá LMHash, který obsahuje spoustu slabin, jak tu již bylo popsáno, a je považován za nebezpečný. Z bezpečnostního hlediska se nedoporučuje používat a je silně doporučeno i samotnou společností Microsoft podporu tohoto protokolu vypnout (pokud není například nutná síťová kompatibilita s počítači s OS Windows 95/98).

NTLMv1

Také využívá systém *challenge-response*. Využívá LMHash a MD4. Byly nalezeny slabiny a z bezpečnostního hlediska je doporučováno přejít na NTLMv2, který je například ve Windows XP implementován, ale standardně nevyužíván, nezapnut, opět kvůli zpětné kompatibilitě. NTLMv1 [26] využívá opět LMHash, což je jeho největší slabina. Konstrukce je ale složitější než u LANMAN, a navíc využívá pro výpočet i MD4, takže je tento protokol bezpečnější. Nese si však s sebou nevýhody hesel ukládaných pomocí LMHash, které mohou být jen 14 znaků dlouhé a nerozlišují velká a malá písmena.

NTLMv2

Vzhledem k odhaleným slabostem NTLMv1 byl navrhnut tento kryptograficky silnější autentizační protokol. Využívá ve své konstrukci velmi perspektivní HMAC.

HMAC je typ autentizačního kódu zprávy (MAC), který počítá s použitím hashovací funkce, ale i tajného šifrovacího klíče. Teoreticky je možno použít pro výpočet jakoukoliv hashovací funkci. Za název HMAC [27] se poté přidává i název použité funkce, máme tak například HMAC-MD5, HMAC-SHA-1 apod.

Tato konstrukce má obrovské možnosti využití a rozhodně na sebe strhává oproti MAC čím dál tím větší pozornost. Hlavně kvůli tomu, že oproti MAC, která je založená na symetrických blokových šifrách, má HMAC výhodu, že hashovací funkce jsou v softwarových implementacích všeobecně rychlejší než symetrické blokové šifry a navíc

na hashovací funkce nejsou aplikována různá omezení, nejvíce ze strany USA, jak k tomu dochází u symetrických šifer.

6.3.2 Nepřímé ověření

Používané hlavně ve větších sítích. Autentizace je centralizovaná a není tak potřeba schraňovat všechny hesla, či hashové kódy u každého účastníka zvlášť.

Kerberos

Kerberos [28] je klasický zástupce autentizačního protokolu využívající nepřímé ověření. Hashovací funkce jsou zde opět použity pro převod hesel na klíče použité při šifrování. Samozřejmě jsou použity při ukládání hesel a pro zaručení integrity dat. Již z podstaty nepřímého ověření potřebuje důvěryhodnou třetí stranu.

Přestože byl navržen již roku 1993, je stále tento protokol považován za bezpečný a je velmi používaný.

Radius

Protokol RADIUS [29] je využívaný hlavně k řízení přístupu z vnější sítě. Jde o takzvaný AAA protokol (authentication, authorization and accounting). Tento protokol je například často používán v bezdrátových sítích IEEE 802.1x. Hashovací funkce MD5 je zde využita jako pseudonáhodný generátor pro tvorbu klíčů a samozřejmě pro uložení hesel. V tomto protokolu se využívají hashovací funkce i pro zachování integrity dat.

7 Realizace obecného autentizačního rámce

7.1 Rozbor aplikace

7.1.1 Výběr programovacího jazyka

Pro realizaci obecného autentizačního rámce jsem vybral programovací jazyk C#. Hlavním důvodem je zkušenost s tímto programovacím jazykem při psaní bakalářské práce. [32] V této bakalářské práci je vysvětleno, že C# a celá architektura .NET má velkou budoucnost, je posunem v pohledu na programování a dalším evolučním skokem nejen pro programování ve Windows. Na síťové programování je tento jazyk velmi vhodný a na rozdíl od Javy bývá program napsaný v C# rychlejší. Nevýhodou je, že u aplikací napsaných v C# jsou menší možnosti přenositelnosti, nejsou tak multiplatformní jako aplikace napsané v programovacím jazyku Java. C++ má výhodu překladu přímo do strojového kódu, to zaručuje maximální výkon a rychlost aplikací, což však nebylo u této realizace prioritou. Práce v něm je pracnější, z tohoto výčtu jazyků nejstarší a trpí několika nedostatky či nelogičnostmi, které byly v novějších programovacích jazycích odstraněny.

V celé kapitole o praktické realizaci se budu držet názvosloví jazyka C#. Obecně celou praktickou realizaci nazývám **řešení** (solution). Řešení obsahuje všechny projekty, které tvoří určitou aplikaci. Aplikace tedy může, a v tomto případě i je, tvořena z více projektů. **Projekty** (project) obsahují jednotlivé soubory (třídy, schémata, apod.), které jsou poté přeloženy do výsledné **aplikace**. Jednotlivé třídy obsahují **metody** (methods). Metoda je vlastně funkce třídy, je jakousi černou skříňkou, u které okolní program ví, jaké má vstupy, jaké výstupy, ale nepotřebuje vědět, co je vevnitř, tj. jakým způsobem se provádí zpracování vstupu na výstup uvnitř metody. V této práci také používám pojem **autentizační metoda** (authentication procedure). Ta definuje komunikaci mezi klientem a serverem a zpracování dat (jejich hashování, spojení hodnot před odesláním, apod.) jednotlivých zúčastněných stran při této komunikaci.

7.1.2 Zadání, hlavní myšlenka realizace

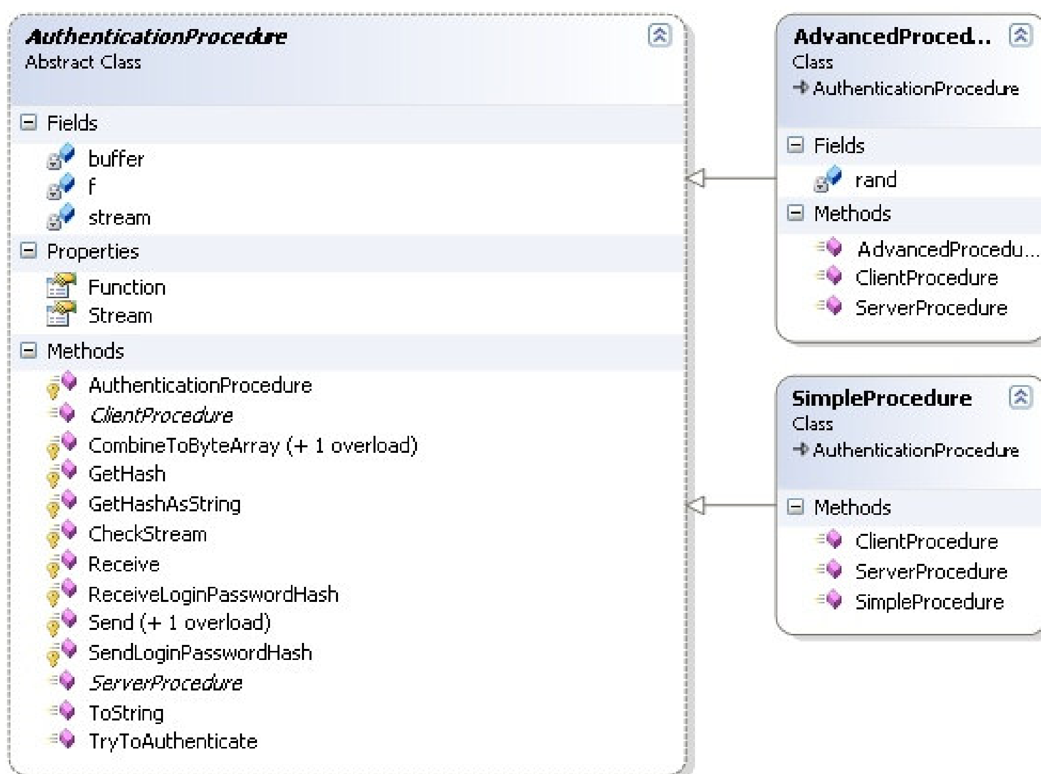
Zadání praktické realizace zní: „Prakticky realizujte obecný autentizační rámec a zvolené metody v něm otestujte. Dbejte na flexibilitu řešení a budoucí možnost využití jiných metod autentizace."

Rozhodl jsem se pro aplikaci typu server – klient, kdy klient se stará o získání dat potřebných pro autentizaci od uživatele a jejich odeslání vybranému serveru. Server naslouchá na určeném portu, zpracovává přijatá data a po vzájemné komunikaci se podle nich rozhoduje, zda klienta autentizuje, či ne. Svoje rozhodnutí poté zasílá klientovi, který ve svém formuláři ukáže uživateli výsledek.

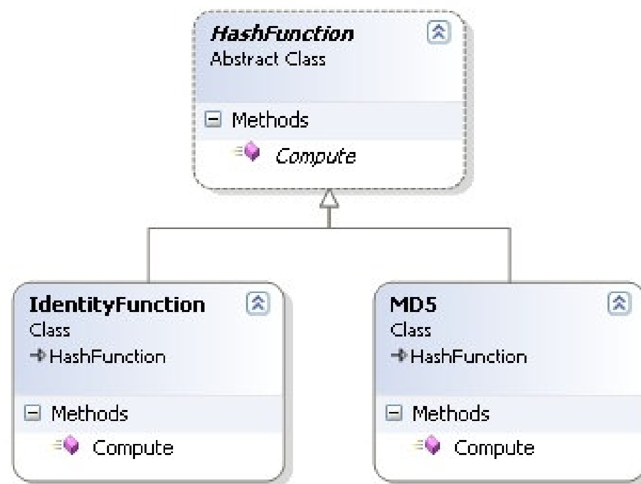
Flexibility a možnosti budoucího rozšíření jsem se snažil dosáhnout hlavně rozdělením programu na několik projektů a tříd. Pro vytvoření nové autentizační metody, či přidání nové hashovací funkce stačí jen přidat novou třídu do projektu (viz. kapitola 7.8) nebo pozměnit stávající. O samotné posílání dat, načítání jména a hesla z textového souboru, obsluhu uživatelského rozhraní, vytváření logu apod, se již postará zdrojový kód ostatních tříd.

7.1.3 Schémata jednotlivých tříd, provázání tříd důležitých pro autentizaci

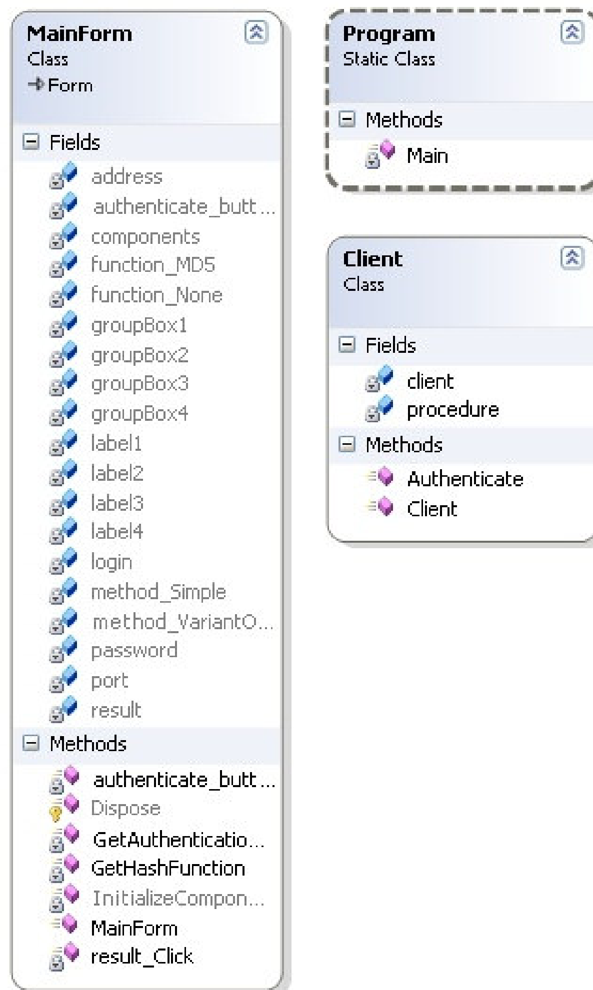
Následující schémata jsou vyexportována pomocí funkce View Class Diagram obsažené v uživatelském rozhraní programu Microsoft Visual Studio 2008. Tato schémata dávají jasnou představu o struktuře celého programu. Na Obr. 12 a Obr. 13 je vidět hierarchie třídy AuthenticationProcedure, respektive třídy HashFunction. Dále jsou v těchto schématech vypsány metody obsažené v jednotlivých třídách a vlastnosti těchto tříd. Schéma na Obr. 14 ukazuje obsažené metody a vlastnosti v třídách u aplikace Client, konkrétně třídy Client, MainForm a Program. Schéma na Obr. 15 ukazuje obsažené metody a vlastnosti v třídách u aplikace Server, konkrétně třídy Server, MainForm a Program.



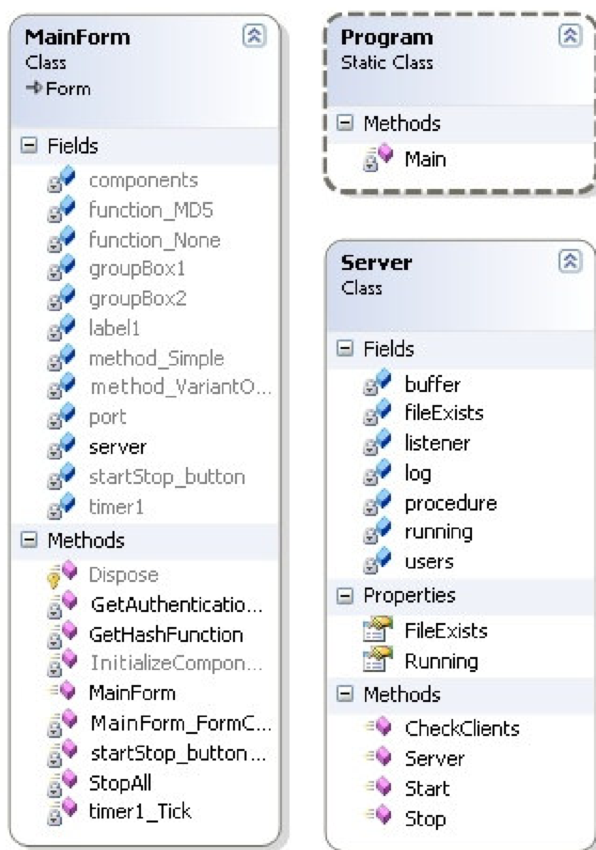
Obr. 12: Schéma tříd AuthenticationProcedure, AdvancedProcedure, SimpleProcedure



Obr. 13: Schéma tříd HashFunction, IdentityFunction a MD5



Obr. 14: Schéma tříd Client, MainForm a Program u aplikace Client



Obr. 15: Schéma tříd Server, MainForm a Program u aplikace Server

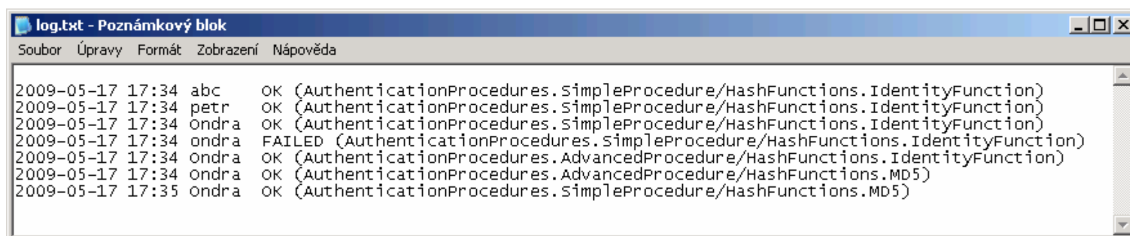
7.1.4 Logování, načítání dat z textového souboru

Logování

Aplikace Server ukládá některé informace z proběhlé komunikace do souboru log.txt. Tento soubor se nachází ve složce \Authentication\bin\Debug, nebo \Authentication\bin\Release. Záleží jestli ve Visual Studiu 2008 vybereme kompilování ladicí (Debug), nebo finální (Release) verze. Je několik rozdílů, k ladící verzi jsou připojeny tzv. ladicí informace k programu. U ladící verze jsou obvykle vypnuty optimalizace, to sice zpomaluje běh výsledného kódu, ale tímto umožňuje programátorovi lepší ladění programu, například zobrazování jednotlivých proměnných apod.

Logování může být velmi užitečné při budoucím využití, pro odhalování chyb či například pro statistické vyhodnocení výsledku apod. Aplikace ukládá datum a čas pokusu o autentizaci. Dále jméno uživatele, který se snažil autentizovat, zda byl úspěšný. Do souboru se také ukládá použitá autentizační metoda a hashovací funkce. Na Obr. 16 je ukázka výpisu ze souboru log.txt. U prvních třech pokusů o autentizaci jsou použita tři různá přihlašovací jména, na čtvrtém řádku je vidět, že Server rozlišuje u jmen a hesel velká a malá písmena

a autentizace nebyla úspěšná. Na dalších třech řádcích jsou pak vidět další kombinace autentizačních metod a hashovacích funkcí.



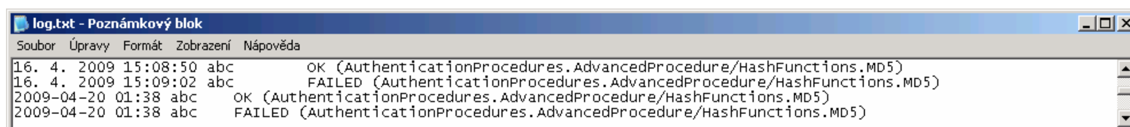
log.txt - Poznámkový blok

Soubor Úpravy Formát Zobrazení Nápověda

```
2009-05-17 17:34 abc    OK (AuthenticationProcedures.SimpleProcedure/HashFunctions.IdentityFunction)
2009-05-17 17:34 petr   OK (AuthenticationProcedures.SimpleProcedure/HashFunctions.IdentityFunction)
2009-05-17 17:34 Ondra  OK (AuthenticationProcedures.SimpleProcedure/HashFunctions.IdentityFunction)
2009-05-17 17:34 ondra  FAILED (AuthenticationProcedures.SimpleProcedure/HashFunctions.IdentityFunction)
2009-05-17 17:34 Ondra  OK (AuthenticationProcedures.AdvancedProcedure/HashFunctions.IdentityFunction)
2009-05-17 17:34 Ondra  OK (AuthenticationProcedures.AdvancedProcedure/HashFunctions.MD5)
2009-05-17 17:35 Ondra  OK (AuthenticationProcedures.SimpleProcedure/HashFunctions.MD5)
```

Obr. 16: Ukázka logování do souboru log.txt

Zajímavostí je, že formát systémového času, který aplikace přebírá, se v různých regionálních nastaveních MS Windows liší. To se poté promítne i do logování. Na následujícím obrázku Obr. 17 ukázka souboru log.txt, kde první dva pokusy proběhly v anglickém regionálním nastavení Microsoft Windows XP a další dva v českém nastavení Microsoft Windows XP.



log.txt - Poznámkový blok

Soubor Úpravy Formát Zobrazení Nápověda

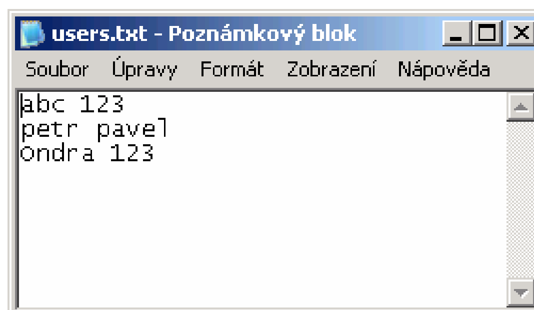
```
16. 4. 2009 15:08:50 abc       OK (AuthenticationProcedures.AdvancedProcedure/HashFunctions.MD5)
16. 4. 2009 15:09:02 abc       FAILED (AuthenticationProcedures.AdvancedProcedure/HashFunctions.MD5)
2009-04-20 01:38 abc    OK (AuthenticationProcedures.AdvancedProcedure/HashFunctions.MD5)
2009-04-20 01:38 abc    FAILED (AuthenticationProcedures.AdvancedProcedure/HashFunctions.MD5)
```

Obr. 17: Různé verze operačních systému MS Windows a jejich vliv na logování

Načítání dat z textového souboru

Abychom nemuseli přidávat či pozměňovat uživatelská jména a hesla přímo ve zdrojovém kódu, aplikace Server se stará o načítání těchto údajů z textového souboru. Tento soubor má název users.txt a je nutné jej umístit (nebo je již umístněn) buď ve složce \Authentication\bin\Debug, nebo \Authentication\bin\Release. Opět záleží, která verze je pro kompilování nastavena ve Visual Studiu 2008.

Na Obr. 18 je ukázka souboru users.txt. Jméno a heslo je při načítání do proměnné typu dictionary rozeznáváno pomocí mezery mezi nimi. Není tedy možné používat jako uživatelské jméno například jméno a příjmení oddělené mezerou. Jednotliví uživatelé jsou pak odděleni znakem dalšího řádku, tedy řádkováním.

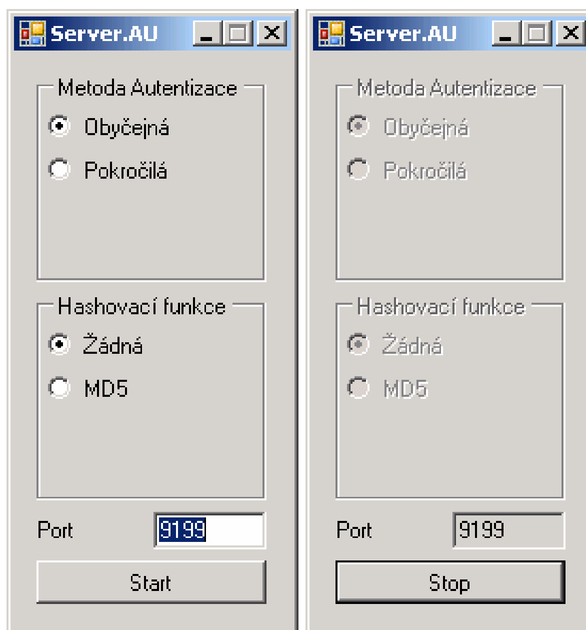


Obr. 18: Ukázka obsahu souboru users.txt

7.2 Server, uživatelské rozhraní

Uživatelské rozhraní aplikace Server je velice jednoduché. Obsahuje dvakrát dva přepínače, textové pole a tlačítko Start/Stop. Přepínače slouží pro výběr požadované metody autentizace a pro výběr požadované hashovací funkce. Do textového pole lze pak definovat na jakém konkrétním portu bude aplikace tzv. poslouchat. Číslo předem vyplněného portu jsem zvolil 9199, protože tento port by neměl být obsazen žádnou běžně využívanou aplikací a neměl by jej zneužívat žádný obecně známý a rozšířený virus, který by mohl narušovat průběh komunikace.

Tlačítkem Start se spustí instance třídy Server. Zkontroluje se textové pole „Port“, jestli obsahuje číslice a jestli jsou ve správném rozsahu. Dále se aplikace pokusí načíst textový soubor se jmény a hesly a spustí časovač, který pravidelně kontroluje, jestli se nesnaží navázat na zvoleném portu klient komunikaci. Nakonec se přepínače a textové pole přepnou do režimu ReadOnly, aby je nebylo možno při běhu Serveru měnit. To by totiž nemělo žádný efekt, protože stav přepínačů a textového pole je zjišťován jen při stisku tlačítka Start. U tlačítka Start se změní popisek na Stop a je možné tímto tlačítkem zastavit běh Serveru, přičemž dojde i k zapsání řádku logů do textového souboru log.txt.

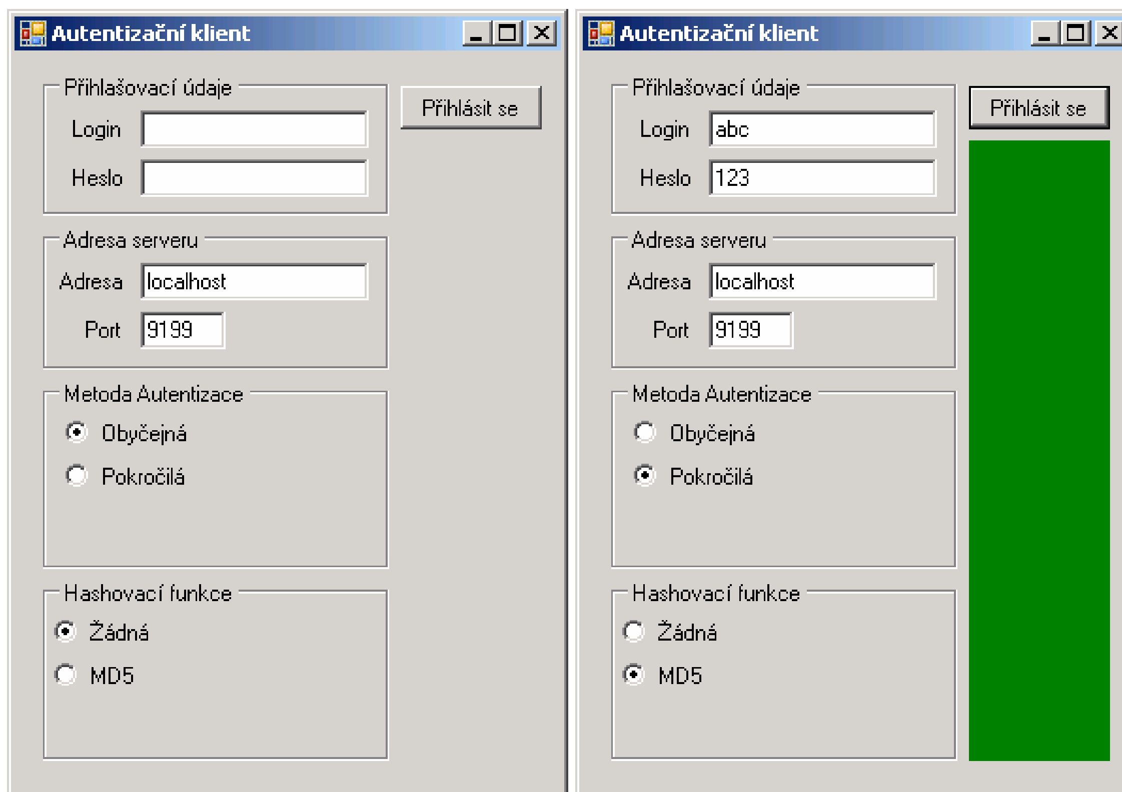


Obr. 19: Ukázka uživatelského rozhraní aplikace Server

7.3 Client, uživatelské rozhraní

Uživatelské rozhraní obsahuje také dvakrát dva přepínače, stejné jako u aplikace Server. Dále dvě textová pole, jedno pro číslo portu, které musíme zadat stejné jako u aplikace Server. Do dalšího textového pole nazvaného „Adresa“ musíme zadat IP adresu zařízení (či doménové jméno zařízení), na kterém je spuštěna aplikace Server. Do textových polí „Login“ a „Heslo“ zadáváme přihlašovací jméno a heslo, ty se poté podle zvolené metody autentizace zpracují a pošlou pro další zpracování aplikaci Server.

Při kliknutí na tlačítko „Přihlásit se“ aplikace zkontroluje, jestli je zadáno v textovém poli „Port“ číslo a jestli je ve zvoleném rozsahu 1 až 65535. V opačném případě pokus o autentizaci neproběhne a objeví se zpráva s textem "Port musí být číslo v rozsahu 1 – 65535". Pokud je port zadán dobře, pokusí se aplikace navázat komunikaci na tomto zvoleném portu a zvolené IP adrese s aplikací Server. Pokud se z nějakého důvodu otevřít socket nepodaří, objeví se zpráva s textem "Nelze se připojit k serveru." Pokud se socket vytvořit podaří, dojde k pokusu o autentizaci dle zvolené metody. Pokud Server potvrdí správnost jména a hesla, autentizace proběhla úspěšně a dojde k obarvení obdélníku pod tlačítkem přihlásit na zelenou barvu. V opačném případě je obdélník obarven na červenou barvu. Pokud klikneme přímo na obdélník, vrátí se jeho barva zpět na barvu pozadí formuláře. To je ilustrováno na následujícím obrázku Obr. 20. Vlevo stav po spuštění, vpravo úspěšná autentizace indikována zeleným pruhem vpravo.



Obr. 20: Ukázka uživatelského rozhraní aplikace Client

7.4 Třída AuthenticationProcedure

Od třídy AuthenticationProcedure se odvozují třídy jednotlivých autentizačních metod. Obsahuje dvě abstraktní metody ClientProcedure a ServerProcedure. To znamená, že pokud vytvoříme novou třídu autentizační metody, která bude potomkem třídy AuthenticationProcedure, uživatelské rozhraní nám tyto dvě metody nabídne k přepsání (override) a nemusíme tedy tuto jakousi kostru třídy psát ručně. Dále třída AuthenticationProcedure obsahuje metody, které budeme v třídách autentizačních procedur používat. Tato třída je tedy stvořena primárně za účelem budoucí možnosti využití dalších metod autentizace.

Popis důležitých metod:

Send

Společně s Receive nejdůležitější a nejvíce využívaná metoda. Nejdříve zkontroluje, zda je otevřen proud dat, pokud tomu tak je, hodnotu odešle, vstupní hodnotu tedy umístí do proudu dat. Aby této metodě bylo možno jako vstupní parametr dát datový typ string i pole bytů, je tato metoda přetížená, akceptuje tedy oba datové typy.

Receive

Tato metoda slouží k přijímání dat. Kontroluje, jestli není aktivní proud dat, pokud ano, přečte z něj data, přetypuje je na datový typ string a přiřadí do proměnné data. Tato hodnota potom tvoří výstup z metody.

ReceiveLoginPasswordHash

Volá metodu Receive, provede tedy to stejné, ale navíc rozdělí přijatá data pro rozlišení jména a hesla, oddělovacím znakem je „\n“ (nový řádek).

GetHash

Tato metoda vypočítá ze vstupní hodnoty hashový kód pomocí funkce, která je vybrána přepínačem. Metoda vrací datový typ pole bytů.

GetHashAsString

Stejné použití jako u metody GetHash, tato však vrací datový typ string.

CombineToByteArray

Metoda slouží ke zkombinování jména a hesla, či jeho hashového kódu do pole bytů. Login od hesla, či jeho hashového kódu oddělí znakem „\n“, aby šlo na straně serveru poznat, kde končí jedna část a začíná druhá. Metoda je přetížena, to znamená, že vstupními hodnotami mohou být jméno datového typu string a heslo, či jeho hashový kód datových typů string, či pole bytů.

TryToAuthenticate

Metoda vytvořená spíše jako ukázka, jak pracovat s datovým typem dictionary. Na ukázku je využita v autentizační metodě s názvem „obyčejná“. Porovná jméno a hashový kód hesla přijaté od klienta se jménem a hashovým kódem, které načteme z textového souboru users.txt. Tato metoda ale není univerzálně použitelná a například v pokročilé autentizační metodě je porovnání psáno již přímo ve zdrojovém kódu této metody.

7.5 Třída SimpleProcedure, obyčejná metoda autentizace

Tuto metodu autentizace jsem nazval „obyčejná“. Metoda je založena na prostém poslání jména a hesla pomocí Clienta Serveru. Server tyto hodnoty porovná s hodnotami uloženými v textovém souboru a pošle odpověď klientovi, jestli byla autentizace úspěšná, či nikoliv. V případě zatržení přepínače MD5 dojde k poslání hashového kódu zadaného hesla a Server v tomto případě uložené heslo v textovém souboru také hashuje a porovnává přijatou hodnotu až s tímto hashovým kódem.

Tuto metodu lze napsat jednodušeji, chtěl jsem ji ale využít pro ukázku použití nabízených metod třídy AuthenticationProcedure.

Zdrojový kód metody ClientProcedure třídy SimpleProcedure:

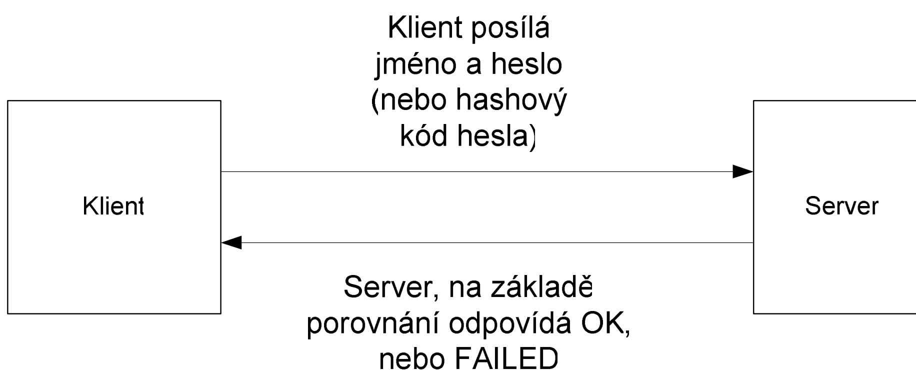
```
byte[] hashedPassword = GetHash( password );
byte[] data = CombineToByteArray( login, hashedPassword );
Send( data );
string response = Receive();
return response == "OK" ? true : false;
```

V této metodě se jméno a heslo posílá jako pole bytů, to není nutné, lze je poslat jako string načtený z textového pole. Proměnné typu string by ale nemusela vždy vyhovovat budoucím možným využitím jiných metod autentizace a proto je zde možnost poslat načtená data i jako pole bytů.

Zdrojový kód metody ServerProcedure třídy SimpleProcedure:

```
string[] loginPassword = ReceiveLoginPasswordHash();
bool result = TryToAuthenticate( loginPassword[0],
                loginPassword[1], users );
Send( result ? "OK" : "FAILED" );
login = loginPassword[0];
return result;
```

Ověřovací procedura může být vepsána přímo v kódu metod třídy SimpleProcedure, zde je však využito metod ReceiveLoginPasswordHash a TryToAuthenticate, popsaných u třídy AuthenticationProcedure. Proměnná login se vytváří kvůli vytváření logu.



Obr. 21: Schématické znázornění komunikace u obyčejné metody autentizace

7.6 Třída AdvancedProcedure

Tuto metodu jsem nazval „pokročilá“. Klient pošle své jméno, načež server odpoví náhodně vygenerovanou číselnou hodnotou. Tuto hodnotu klient přidá k heslu, z takto vytvořené hodnoty vytvoří hashový kód dle zvolené hashovací funkce a teprve výsledný hashový kód posílá serveru. Server vyhledá podle zadaného jména odpovídající řádek v textovém souboru,

načte z tohoto řádku uložené heslo, přidá k němu tutéž náhodně vygenerovanou hodnotu a porovná výsledný hashový kód s hodnotou přijatou od klienta. Tato autentizační metoda velmi znesnadňuje zjištění hesla při odposlouchávání. Útočníkovi se stěžují útoky pomocí předem vygenerovaných tabulek, například pomocí rainbow tables. Útočníkovi zbývá na každý odposlechnutý hashový kód použít útok hrubou silou, což je velmi výpočetně náročné. Délka hesla se navíc přidáním náhodné hodnoty prodlužuje, to ještě více znesnadňuje případný útok.

Zdrojový kód metody ClientProcedure třídy AdvancedProcedure:

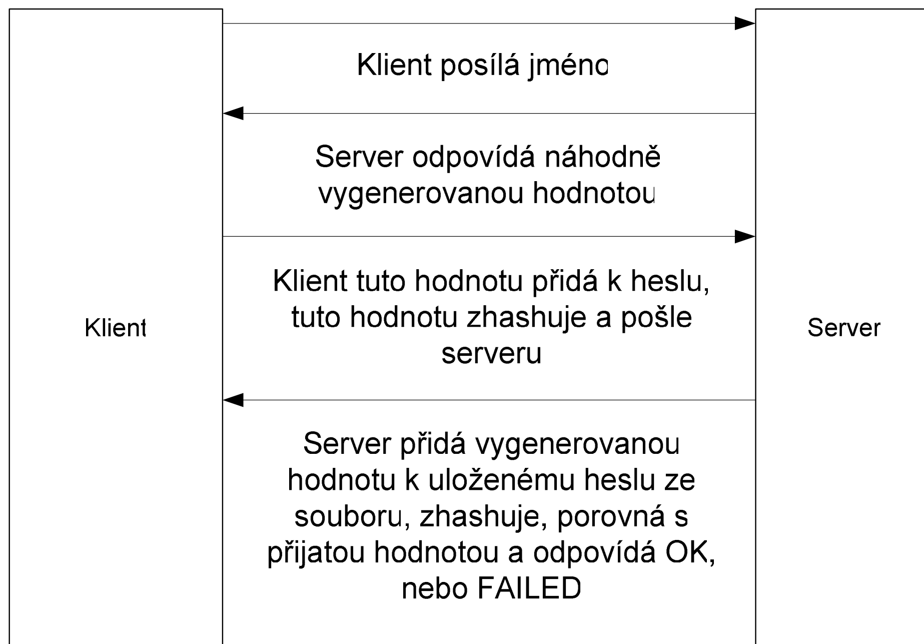
```
Send(login);
string randfromserver = Receive();
string hashedPassword=GetHashAsString(password+randfromserver);
Send(hashedPassword);
string response = Receive();
return response == "OK" ? true : false;
```

V této metodě se zpracovávají hodnoty jako datový typ string. Je zde využita metoda GetHashAsString z třídy AuthenticationProcedure. Tato metoda vrací hashový kód rovnou jako datový typ string, není potřeba jej přetypovávat.

Zdrojový kód metody ServerProcedure třídy AdvancedProcedure:

```
login = Receive();
string r = rand.Next().ToString();
Send(r);
string finalhash = Receive();
if (users.ContainsKey(login) && finalhash ==
    GetHashAsString(users[login] + r))
{
    Send("OK");
    return true;
}
else
{
    Send("FAILED");
    return false;
}
```

Jak je vidět, zdrojový kód ověření u Serveru je napsán přímo ve třídě samotné autentizační metody. V podmínce je kontrolováno, jestli je v textovém soubor přijaté jméno obsaženo a pokud ano, jestli se shoduje vypočtený hashový kód s přijatým.



Obr. 22: Schématické znázornění komunikace mezi u pokročilé metody autentizace

7.7 Třídy HashFunction, IdentityFunction, MD5, Server, Client, MainForm

Třída HashFunction

Tato třída je kostrou, která předepisuje formu metody pro výpočet obecné hashovací funkce. Od ní se dědí další konkrétní hashovací funkce.

Třída IdentityFunction

V této třídě je definována funkce, odvozená z třídy HashFunction. Zde se konkrétně jedná o funkci, která vrací hodnotu shodnou s hodnotou vstupující.

Třída MD5

Další funkce odvozená z třídy HashFunction. Zde už se jedná o hashovací funkci a to konkrétně MD5. Funkce tedy ze vstupní hodnoty vypočítává hashový kód.

Třída Server

Hlavním cílem této třídy je přijímání příchozích spojení. O to se stará hlavně metoda CheckClients, ta testuje, zda se nepokouší klient navázat spojení a pokud ano, tak jej přijímá a dále předává vybrané autentizační metodě. Tato třída také osahuje obslužný kód pro ukládání údajů do logu. Třída se stará i o načítání obsahu souboru users.txt, ve kterém jsou uložena jména a hesla.

Třída MainForm u aplikace Server

Třída obsahuje obslužný kód pro uživatelské rozhraní, tedy rozhoduje, který přepínač je aktivní, reaguje na akci kliknutí u tlačítka Start, respektive Stop, kontroluje jestli je port napsaný v textovém boxu ve správném formátu a stará se zobrazení chybových hlášení. Obsahuje také komponentu timer1, která každých 100ms spouští metodu CheckClients. Často se pro tento případ používá nekonečná smyčka, rozhodl jsem však pro časovač, protože takovéto řešení tolik neblokuje výpočetní výkon počítače a hlavně ovládání formuláře samotného programu.

Třída Client

Tato třída je velice krátká. V podstatě se zde autentizační metodě pouze předává kanál (v jazyce C# datový proud), pomoci kterého může komunikovat se Serverem a spouští se zde vybraná autentizační metoda.

Třída MainForm u aplikace Client

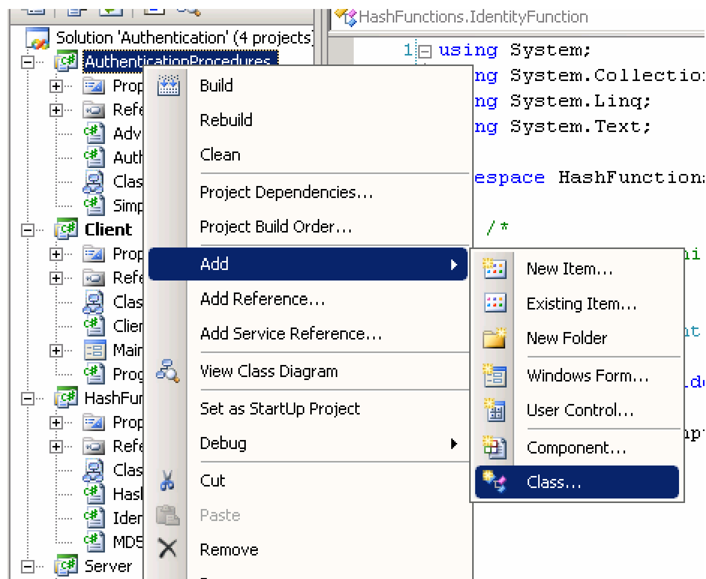
Třída obsahuje obslužný kód pro uživatelské rozhraní, tedy rozhoduje, který přepínač je aktivní. Dále reaguje na akci kliknutí u tlačítka „Přihlásit se“, kontroluje, jestli je port napsaný v textovém poli ve správném formátu. Také se stará o zobrazení chybových hlášení a obsahuje rozhodovací podmínku, která podle výsledku autentizace obarvuje obdélník ve formuláři na určenou barvu.

7.8 Přidání autentizační metody a hashovací funkce

Přidání autentizační metody

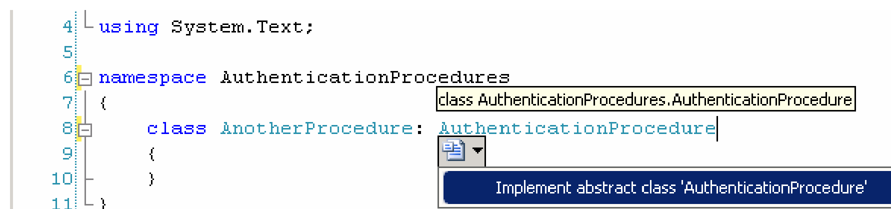
Pro vyzkoušení nových metod autentizace je možné pozměnit metody stávající. Další možností je přidání metody nové. Řešení s touto variantou počítá a díky rozvržení tříd je na ni připraveno. Pro názornost uvedu postup velmi podrobně a krok za krokem.

Do projektu AuthenticationProcedures přidáme novou třídu. Zde ji pojmenujeme například AnotherProcedure. Na Obr. 23 je ukázán postup pro přidání třídy.



Obr. 23: Postup přidání nové třídy

V této nové třídě je potřeba definovat, že její rodič je třída AuthenticationProcedure. To provedeme připsáním dvojtečky za název definice názvu třídy a dopsáním AuthenticationProcedure. Po kliknutí na název rodiče, Visual Studio nabídne implementování všech abstraktních metod. Viz. Obr. 24.



Obr. 24: Postup jak implementovat abstraktní metody

Tímto postupem se připravili metody potřebné pro novou autentizační metodu, viz. Obr. 25.

```
public class AnotherProcedure: AuthenticationProcedure
{

    public override bool ClientProcedure(string login, string password)
    {
        throw new NotImplementedException();
    }

    public override bool ServerProcedure(Dictionary<string, string> users, out string login)
    {
        throw new NotImplementedException();
    }
```

Obr. 25: Ukázka implementovaných abstraktních metod u nové třídy

Abychom autentizační metodě zpřístupnili jmenný prostor HashFunctions, na začátek třídy dopíšeme řádek:

```
using HashFunctions;
```

Autentizační metoda dále potřebuje ke své práci hashovací funkci. O její uchování se stará třída AuthenticationProcedure. Musíme ale každé nové autentizační metodě vytvořit stejné rozhraní, aby mohla hashovací funkci vůbec přijmout a zavolat konstruktor rodičovské třídy, v jejímž zdrojovém kódu je vše potřebné definováno. Do třídy je tedy potřeba ještě přidat následující řádek:

```
public NázevNověVytvořenéTřídy(HashFunction f) : base(f) { }
```

Zdrojový kód třídy AnotherProcedure pak tedy bude v našem příkladě vypadat následovně:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using HashFunctions;

namespace AuthenticationProcedures
{
    class AnotherProcedure:AuthenticationProcedure
    {
        public AnotherProcedure(HashFunction f) : base(f) { }

        public override bool ClientProcedure(string login, string password)
        {
            throw new NotImplementedException();
        }

        public override bool ServerProcedure(Dictionary<string, string> users, out string login)
        {
            throw new NotImplementedException();
        }
    }
}
```

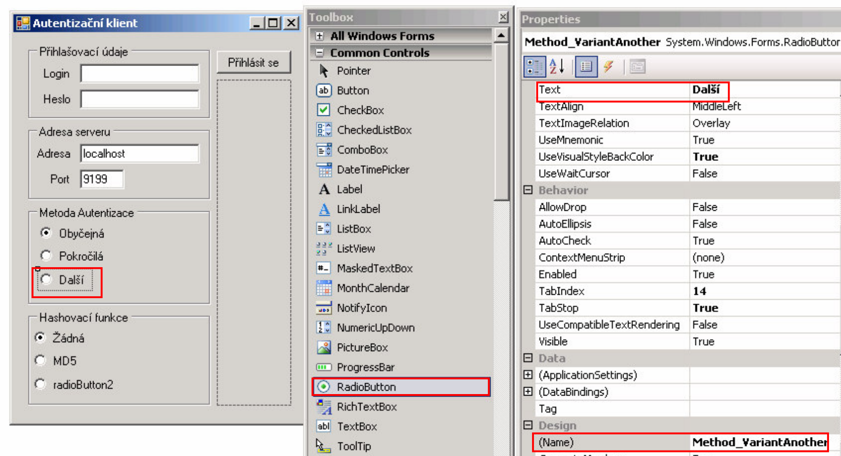
Obr. 26: Zdrojový kód nové třídy

Tímto je nová třída připravena pro vepsání kódu, pomocí kterého můžeme vyzkoušet další metody autentizace. V metodě ClientProcedure se implementuje obslužný kód pro komunikaci a zpracování dat aplikace Client. V metodě ServerProcedure se implementuje obslužný kód pro komunikaci a zpracování dat aplikace Server.

Výstupní hodnotou metody ServerProcedure je i proměnná login, tedy uživatelské jméno uživatele, který se pokouší o autentizaci. Ta je dále zpracována v třídě Server za účelem logování a musí tedy být v této metodě definována.

Pro úspěšné použití nové autentizační metody je ještě potřeba ji svázat s uživatelským rozhraním aplikací.

Do formulářů aplikací Server a Client tedy přidáme prvek RadioButton. Ve vlastnostech právě přidaného přepínače vyplníme v záložce Appearance text prvku, zde například Další. V záložce Design lze vyplnit název (Name) přepínače. Můžeme nechat původní název radioButton1, pro lepší přehlednost je vhodnější zvolit název vlastní. Zde například Method_VariantAnother. Na Obr. 27 ukázka přidání přepínače u aplikace Client.



Obr. 27: Přidání a nastavení přepínače

Posledním krokem je připsání obslužného kódu pro výběr přepínače a s tím související volbu metody. To opět musíme provést u aplikací Server i Client. Kód se nachází v třídách MainForm jednotlivých aplikací, konkrétně u metody GetAuthenticationProcedure.

Do této metody je potřeba přidat následující řádky:

```
else if ( NázevPřepínače.Checked)
{
    return new NázevTřídyNovéMetody(f);
    }
```

Zdrojový kód metody v třídě MainForm po přidání přepínače Method_VariantAnother pak bude například vypadat následovně:

```
private AuthenticationProcedure GetAuthenticationProcedure()
{
    HashFunction f = GetHashFunction();

    if ( method_Simple.Checked )
    {
        return new SimpleProcedure( f );
    }
    else if ( method_VariantOne.Checked )
    {
        return new AdvancedProcedure(f);
    }
```

```
else if ( Method_VariantAnother.Checked)
{
    return new AnotherProcedure(f);
}

return null;
}
```

Nyní je již pro implementaci nové autentizační metody vše připraveno.

Přidání hashovací funkce

Přidání nové hashovací funkce je ještě jednodušší. Stejně jako u přidávání nové autentizační metody, vytvoříme novou třídu. Tentokrát ale v projektu HashFunctions. Této třídě definujeme, že je potomkem třídy HashFunction. Poté již postupujeme analogicky jako u přidávání autentizační metody. Implementujeme abstraktní metody, přidáme přepínače do uživatelského rozhraní aplikací. Do metod GetHashFunction u tříd MainForm jednotlivých aplikací přidáme obslužný kód pro přepínače. Nyní je již vše připraveno pro implementaci nové hashovací funkce.

7.9 Ukázka komunikace

Pro odchycení komunikace mezi aplikacemi Server a Client jsem použil program Wireshark, který je volně ke stažení s licencí freeware. Odchycením paketů lze názorně ukázat, že řešení funguje dle předpokladů a lze na nich i ukázat konkrétní data, která si mezi sebou aplikace posílají u obou ze zvolených metod.

Na Obr.28 a Obr.29 jsou v aplikacích zvoleny „obyčejná" autentizační metoda a „IdentityFunction" jako hashovací funkce, jejíž výstup se rovná hodnotě vstupní. Lze tedy vidět, jak aplikace Client (zde konkrétně spuštěna počítači s IP 192.168.100.17) posílá aplikaci Server (zde konkrétně spuštěna na počítači s IP 192.168.100.75) uživatelské jméno a heslo, zde konkrétně jméno: abc a heslo: 123. Server po porovnání údajů odpovídá OK, autentizace tedy byla úspěšná.

```
4 0.001196   192.168.100.17   192.168.100.75   TCP   vsat-control > 9199 [PSH, ACK] Seq=1 Ack=1 Win=65535
5 0.080840   192.168.100.75   192.168.100.17   TCP   9199 > vsat-control [PSH, ACK] Seq=1 Ack=15 Win=6552

0000  00 17 08 30 1f 96 00 00  00 00 0a bc 08 00 45 00   ...0.... ......E.
0010  00 36 af a6 40 00 80 06  01 6e c0 a8 64 11 c0 a8   .6..@... .n..d...
0020  64 4b 07 58 23 ef 9b de  0d 01 90 75 b3 b4 50 18   dK.X#... ...u..P.
0030  ff ff 87 be 00 00 61 00  62 00 63 00 0a 00 31 00   ......a. b.c...1.
0040  32 00 33 00                                        2.3.
```

Obr. 28: Uživatelské jméno a heslo v odchyceném paketu

```
4 0.001196   192.168.100.17   192.168.100.75   TCP   vsat-control > 9199 [PSH, ACK] Seq=1 Ack=1 Win=65535
5 0.080840   192.168.100.75   192.168.100.17   TCP   9199 > vsat-control [PSH, ACK] Seq=1 Ack=15 Win=6552

0000  00 00 00 00 0a bc 00 17  08 30 1f 96 08 00 45 00   ........ .0....E.
0010  00 2c 3d 47 40 00 80 06  73 d7 c0 a8 64 4b c0 a8   .,=G@... s...dK..
0020  64 11 23 ef 07 58 90 75  b3 b4 9b de 0d 0f 50 18   d.#..X.u ......P.
0030  ff f1 49 cc 00 00 4f 00  4b 00                     ..I...O. K.
```

Obr. 29: Server odpovídá OK, odchycený paket

Autentizační metoda byla přepnuta na pokročilou, jako hashovací funkce byla ponechána IdentityFunction, aby byli vidět konkrétní hodnoty, které si mezi sebou aplikace posílají. Na Obr.30 je vidět, že aplikace Client nejdříve pošle jen uživatelské jméno, zde konkrétně: abc. Načež Server odpovídá náhodně vygenerovanou hodnotou, zde: 1313758352. Obr. 31. Client připojí k heslu přijatou hodnotu, vznikne tak hodnota 1231313758352 a vytvoří hashovací kód této hodnoty. V tomto případě byla využita funkce IdentityFunction, která hodnotu nijak nezměnila a tato tedy byla poslána aplikaci Server. Viz. Obr. 32. Autentizace v tomto případě opět proběhla úspěšně a Server poté ještě odpověděl OK.

Obr. 30: Client posílá pouze uživatelské jméno

Obr. 31: Náhodně vygenerovaná hodnota posílaná aplikací Server

Obr. 32: Přidání hesla k náhodně vygenerované hodnotě

8 Závěr

Úvodní kapitola této diplomové práce se zabývá základními poznatky o hashovacích funkcích. Jsou zde uvedeny časem stále rostoucí požadavky na hashovací funkce. Dále jsou zde popsány konstrukční prvky, ze kterých je hashovací funkce poskládána a uvedeny příklady použití hashovacích funkcí. Tato teorie je velmi potřebná pro vysvětlení a hlavně pochopení textu v dalších kapitolách.

Práce se také zabývá útoky na hashovací funkce, a to jak útoky na samotné algoritmy, například pomocí kryptoanalýzy či metody hrubé síly, tak i útoky na uložený hashový kód, jako je například slovníkový útok nebo útok pomocí Rainbow tables.

Kapitola „Současné hashovací funkce" rozebírá dnes nejpoužívanější hashovací funkce LMHash, MD4, MD5, SHA-0, SHA-1 a hashovací funkce z rodiny SHA-2. Jsou zde uvedeny poznatky o konstrukci těchto funkcí a porovnává je i z bezpečnostního hlediska. V této kapitole je pak zajímavostí rozebrání hledání kolizí pomocí tzv. „tunelování" a mnohonásobné modifikace zpráv, které se ukázaly být velice efektivní a udaly nový a perspektivní směr v kryptoanalýze hashovacích funkcí. Z bezpečnostního hlediska pak z této kapitoly vyplývá, že z těchto funkcí se v praxi doporučuje používat již jen hashovací funkce z rodiny SHA-2.

Kapitola „Budoucnost hashovacích funkcí" se zaobírá všeobecnými problémy iterativní konstrukce dnešních funkcí a z textu vyplývá, že je potřeba hledat nové, bezpečnější hashovací funkce. Touto tématikou se pak zabývá druhá část této kapitoly. Konkrétně je zde popsána velmi aktuální soutěž o novou hashovací funkci SHA-3. Také je zde nabídnuto nahlédnutí na velice zajímavé hashovací funkce HDN typu SNMAC, autory označované jako „hashovací funkce nové generace"

Šestá kapitola se zabývá souvislostí hashovacích funkcí a jejich použití při autentizaci. Zde je rozebrána problematika ukládání hesel pomocí hashovacích funkcí a jejich využití i v autentizačních protokolech. Z této kapitoly mimo jiné vyplývá velký nedostatek u operačních systémů Windows verzí XP a starších. Doporučením je nepoužívat LMHash a LANMAN.

Poslední kapitola obsahuje dokumentaci k přiloženému praktickému řešení obecného autentizačního rámce. Řešení je implementováno v programovacím jazyce C#. S ohledem na zadání je dbáno na flexibilitu řešení a budoucí možnost využití jiných metod autentizace. Rozhodl jsem se pro aplikace typu server – klient. Klient získá od uživatele data potřebná pro autentizaci, podle vybrané autentizační metody je zpracuje a poté odesílá vybranému serveru. Server naslouchá na určeném portu, zpracovává přijatá data a po vzájemné komunikaci

se podle nich rozhoduje, zda klientova autentizace bude úspěšná, či ne. Svoje rozhodnutí poté zasílá klientovi, který ve svém formuláři ukáže uživateli výsledek. Pomocí těchto aplikací jsou úspěšně vyzkoušeny dvě vybrané autentizační metody.

Flexibility a možnosti budoucího rozšíření je dosáhnuto rozdělením programu do několika projektů a následně tříd. Pro vytvoření nové autentizační metody, či přidání nové hashovací funkce stačí jen přidat novou třídu do určeného projektu nebo pozměnit stávající. O samotné posílání dat, načítání jména a hesla z textového souboru, obsluhu uživatelského rozhraní, vytváření logu a podobně, se již postará zdrojový kód ostatních tříd.

Použitá literatura

[1] KLÍMA, V. Hašovací funkce, principy, příklady kolize [online]. [cit. 2005-19-03]. Dostupné z WWW: *www.cryptofest.cz/slides/klima_cryptofest_2005.pdf*

[2] SMITH, RICHARD, E. Authentication: From Passwords to Public Keys. Addison-Wesley Professional , 2001. ISBN 978-0201615999

[3] MENEZES, A., PAUL, C. VAN OORSCHOT, SCOTT, A. V. Handbook of applied cryptography. CRC Press, 2001. ISBN 0-8493-8523-7

[4] X. WANG, D. FENG, X. LAI, H. YU. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. [online]. [cit. 2004-17-08]. Dostupné z WWW: *http://eprint.iacr.org/2004/199*

[5] RIVEST, R. RFC 1320 – The MD4 Message-Digest Algorithm [online]. [cit. duben 1992]. Dostupné z WWW: *http://tools.ietf.org/rfc/rfc1320.txt*

[6] DOBBERTIN, H. Cryptanalysis of MD4, Fast Software Encryption LNCS, Vol. 1039, Springer-Verlag, 1996. ISBN:3-540-60865-6

[7] RIVEST, R. RFC 1321 – The MD5 Message-Digest Algorithm [online]. [cit. duben 1992]. Dostupné z WWW: *http://tools.ietf.org/rfc/rfc1321.txt*

[8] H. Dobbertin. Cryptanalysis of MD5 Compress. Presented at the rump session of Eurocrypt '96. [online]. [cit. 1994-14-05]. Dostupné z WWW: *http://citeseer.ist.psu.edu/dobbertin96cryptanalysis.html*

[9] KLÍMA, V. Finding MD5 Collisions – a Toy For a Notebook [online]. [cit. 2005-08-03]. Dostupné z WWW: *http://eprint.iacr.org/2005/075*

[10] KLÍMA, V. Tunely v hašovacích funkcích: kolize MD5 do minuty [online]. [cit. 2006-18-03]. Dostupné z WWW: *http://crypto-world.info/klima/2006/tunely.pdf*

[11] FIPS PUB 180 - SECURE HASH STANDARD [online]. [cit. 1993-11-05]. Dostupné z WWW: *security.isu.edu/pdf/fips180.pdf*

[12] FIPS PUB 180-1 - SECURE HASH STANDARD [online]. [cit. 1995-17-04]. Dostupné z WWW: *cryptography.com/resources/whitepapers/misc/FIPS180-1.txt*

[13] F. CHABAUD, A. J. Differential Collisions in SHA-0. Advances in Cryptology - Crypto'98, LNCS 1462, 1998. ISBN 978-3-540-64892-5

[14] WANG X., YIN L., YU H. Collision Search Attacks on SHA1 [online]. [cit. 2005-13-02]. Dostupné z WWW: *http://people.csail.mit.edu/yiqun/shanote.pdf*

[15] WANG X., YIN L., YU H. Finding Collisions in the Full SHA-1 [online]. [cit. 2005-10]. Dostupné z WWW: *people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf*

[16] FIPS PUB 180-2 - SECURE HASH STANDARD + Change Notice to include SHA-224 [online]. [cit. 2002-01-07]. Dostupné z WWW: *csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf*

[17] SANADHYA, K. S., SARKAR, P.. New Collision attacks Against Up To 24-step SHA-2. [online]. [cit. 2008]. Dostupné z WWW: http://eprint.iacr.org/2008/270.pdf

[18] KLÍMA, V. Zcela nový koncept hašovacích funkcí [online]. [cit. 2006-13-11]. Dostupné z WWW: www.*root.cz/clanky/vlastimil-klima-zcela-novy-koncept-hasovacich-funkci*

[19] JOUX, A.: Multicollisions in iterated hash functions. Application to cascaded constructions. Proceedings of Crypto 2004 [online]. [cit. 2004]. Dostupné z WWW: http://web.cecs.pdx.edu/~teshrim/spring06/papers/general-attacks/multi-joux.pdf

[20] Kelsey, J., Schneier, B. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work [online]. [cit. 2004-15-11] Dostupné z WWW: http://eprint.iacr.org/2004/304

[21] The SHA-3 Zoo [online]. [cit. 2008-12-07]. Dostupné z WWW: *http://ehash.iaik.tugraz.at/index.php/The_SHA-3_Zoo*

[22] KLÍMA, V. Rodina speciálních blokových šifer DN a hašovacích funkcí nové generace HDN typu SNMAC [online]. [cit. leden 2007]. Dostupné z WWW: *http://cryptography.hyperlink.cz/SNMAC/DN_HDN_CZ.pdf*

[23] KLÍMA, V. New Concept of Hash Functions SNMAC Using a Special Block Cipher and NMAC/HMAC Constructions, IACR ePrint archive Report [online]. [cit. říjen 2006]. Dostupné z WWW: *http://eprint.iacr.org/2006/376.pdf*

[24] Jongsung, K., Biryukov, A., Preneel B., Hong, H. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1" [online]. [cit. 2006].Dostupné z WWW:*http://homes.esat.kuleuven.be/~kjongsun/papers/scn02006.pdf*

[25] LEVICKÝ, D. Kryptografia v informačnej bezpečnosti. elfa, s.r.o. Košice 2005. ISBN 80-8086-022-X

[26] GLASS, E. The NTLM Authentication Protocol and Security Support Provider [online]. [cit. 2006]. Dostupné z WWW: *http://davenport.sourceforge.net/ntlm.html*

[27] KRAWCZYK, H., BELLARE, M., CANETTI, R. HMAC: Keyed-Hashing for Message Authentication [online]. [cit. únor 1997]. Dostupné z WWW: *ftp://ftp.isi.edu/in-notes/rfc2104.txt*

[28] KOHL, J., NEUMAN, C. RFC 1510 - The Kerberos Network Authentication Service (V5) [online]. [cit. září 1993]. Dostupné z WWW: http://www.ietf.org/rfc/rfc1510.txt

[29] RIGNEY, C., WILLENS S., LIVINGSTON. RUBENS. MERIT. SIMPSON, W. RFC 2865 - Remote Authentication Dial In User Service (RADIUS) [online]. [cit. červen 2000]. Dostupné z WWW: http://www.ietf.org/rfc/rfc2865.txt

[30] KLIMA, V., KNAPSKOG, S. J., EL-HADEDY, M., GLIGOROSKI, D., AMUNDSEN, J., MJØLSNES. J., FRODE, S. BLUE MIDNIGHT WISH [online]. [cit. říjen 2008]. Dostupné z WWW: http://people.item.ntnu.no/~danilog/Hash/BMW/Supporting_Documentation/BlueMidnightWishDocumentation.pdf

[31] RIJMEN, V., DAEMEN, J. The Design of Rijndael: AES - The Advanced Encryption Standard. Springer-Verlag, 2002. ISBN 3-540-42580-2

[32] PILLER, I. Práce se sockety v jazyce C#. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, vedoucí práce Ing. Jan Kacálek, 2007

Seznam obrázků

Obr. 1: Odolnost proti kolizi prvního řádu 12
Obr. 2: Odolnost proti kolizi druhého řádu 13
Obr. 3: Damgård-Merklova konstrukce s vyznačenou kompresní funkcí 15
Obr. 4: Porovnání použití blokové šifry u šifrování a hashování 16
Obr. 5: Davies-Meyerova konstrukce kompresní funkce 16
Obr. 6: Princip ukládání tabulky u útoku Rainbow Tables 20
Obr. 7: Ukázka jedné operace hashovací funkce MD4 24
Obr. 8: Kolize u MD4 25
Obr. 9: Ukázka jedné operace funkce MD5 s vyznačeným rozdílem od MD4 26
Obr. 10: Kompresní funkce SHA-1 28
Obr. 11: Ukázka nalezení multikolize s nižší složitostí 32
Obr. 12: Schéma tříd AuthenticationProcedure, AdvancedProcedure, SimpleProcedure 43
Obr. 13: Schéma tříd HashFunction, IdentityFunction a MD5 44
Obr. 14: Schéma tříd Client, MainForm a Program u aplikace Client 44
Obr. 15: Schéma tříd Server, MainForm a Program u aplikace Server 45
Obr. 16: Ukázka logování do souboru log.txt 46
Obr. 17: Různé verze operačních systému MS Windows a jejich vliv na logování 46
Obr. 18: Ukázka obsahu souboru users.txt 47
Obr. 19: Ukázka uživatelského rozhraní aplikace Server 48
Obr. 20: Ukázka uživatelského rozhraní aplikace Client 49
Obr. 21: Schématické znázornění komunikace u obyčejné metody autentizace 51
Obr. 22: Schématické znázornění komunikace mezi u pokročilé metody autentizace 53
Obr. 23: Postup přidání nové třídy 55
Obr. 24: Postup jak implementovat abstraktní metody 55
Obr. 25: Ukázka implementovaných abstraktních metod u nové třídy 55
Obr. 26: Zdrojový kód nové třídy 56

Obr. 27: Přidání a nastavení přepínače 57
Obr. 28: Uživatelské jméno a heslo v odchyceném paketu 59
Obr. 29: Server odpovídá OK, odchycený paket 59
Obr. 30: Client posílá pouze uživatelské jméno 60
Obr. 31: Náhodně vygenerovaná hodnota posílaná aplikací Server 60
Obr. 32: Přidání hesla k náhodně vygenerované hodnotě 60

Obsah CD

Přiložené CD obsahuje:

- Tuto práci v elektronické podobě: DiplomovaPracePiller.pdf
- Zdrojové kódy k praktické realizaci obecného autentizačního rámce v programovacím jazyce C#: složka \Authentication.

Pro úspěšné spuštění přiložených aplikací v OS MS Windows je potřeba mít nainstalovánu platformu .NET Framework. Je možné vyzkoušet prostředí Mono, které by mělo umožnit tyto aplikace spouštět i v OS Linux, to však nebylo testováno. Aplikace byli implementovány, laděny, spouštěny a vyzkoušeny v OS MS Windows XP SP3 s nainstalovanou platformou .NET Framework verze 3.5.

Pro vyzkoušení je možné najít aplikaci Client např. ve složce \Client\bin\Release a aplikaci Server ve složce \Authentication\bin\Release, kde najdeme i podpůrné soubory users.txt a log.txt. Je také možné využít zástupce těchto programů, uložené přímo v kořenovém adresáři CD.