Masarykova univerzita
Fakulta informatiky

Kontrola integrity dat hašovacími funkcemi

Bakalářská práce

Jan Krhovják

2003

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Brno 8.1.2003

Děkuji Dr. Václavu Matyášovi za odborné vedení bakalářské práce a poskytování cenných rad při jejím zpracování.

Shrnutí

Cílem této bakalářské práce je seznámení se s kryptografickými hašovacími funkcemi, s jejich praktickým využitím, a naprogramování open-source utility využívající těchto funkcí pro zajištění kontroly integrity dat. V první části uvedeme základní pojmy a definice potřebné k rozboru, popisu a klasifikaci hašovacích funkcí a metod kontroly integrity dat. Druhá část práce se bude zabývat již samotným popisem funkčnosti a realizace programu.

Klíčová slova

Hašovací funkce, jednocestné hašovací funkce, bezkolizní hašovací funkce, integrita dat, autentizace zpráv, MAC, MDC, SHA, FIPS PUB 180-1, FIPS PUB 180-2.

Obsah

Kapitola 1

1 Úvod

V dnešní době hrají *kryptografické hašovací funkce* významnou roli, a to jak v komunikačních a bezpečnostních protokolech, přihlašovacích a autentizačních procedurách, certifikátech a digitálních podpisech, tak také v oblasti zachování integrity dat. Zatímco standardní hašovací funkce, které nacházejí uplatnění především v nekryptografických aplikacích, pouze převedou řetězce libovolné délky na řetězce pevné délky, musejí kryptografické hašovací funkce splňovat i několik dalších kritérií. V této práci se budeme zabývat převážně kryptografickými hašovacími funkcemi[1] a jejich aplikací. Důraz bude kladen především na zachování a kontrolu integrity dat.

Co tedy vlastně hašovací funkce je? Nejedná se o nic jiného, než o nějaké zobrazení z víceprvkové množiny všech různých konečných řetězců na méněprvkovou množinu řetězců pevné délky. Z čehož vyplývá, že zobrazení není injektivní (a tedy ani bijektivní). Čili nějaké dva různé řetězce z větší množiny se mohou zobrazit na jeden řetězec z množiny menší, a v důsledku toho budou existovat kolize. Proto jedna z podmínek, které budou muset hašovací funkce splňovat, je bezkoliznost (tj. výpočetní obtížnost nalezení kolizí). Druhou podmínkou bude jednocestnost (tj. nemožnost zpětného určení vstupu).

Jak však lze využít takovýchto funkcí k ochraně integrity dat? Základní myšlenka spočívá ve vytvoření haše[2], který „jednoznačně“ koresponduje s daty. Ten je uložen a nějakým způsobem chráněn. Mechanizmus ověření integrity dat je založen na vytvoření nového haše, který je následně porovnán s již uloženým a chráněným hašem. Jsou-li oba stejné, data nebyla změněna. Tím se zredukoval problém ochrany potenciálně velkých dat na problém ochrany velmi malého haše s konstantní délkou.

Odlišnou třídu hašovacích funkcí tvoří *Message Authentication Codes*[3]. Tyto funkce mají dva vstupy (zprávu a tajný klíč) a výstup, který bez znalosti tohoto klíče nejsme schopni určit. Lze je tedy použít k ochraně integrity dat i k symetrické autentizaci.

K práci je přiložen program využívající hašovacích funkcí SHA-1, SHA-256, SHA-384 a SHA-512 pro zajištění kontroly integrity dat. Vstupem mu jsou soubory určené k ochraně integrity. Výstupem je kontrolní soubor obsahující názvy a haše jednotlivých zadaných vstupních souborů, či pouze *xor* těchto hašů. Na konec kontrolního souboru je navíc přidán jeho vlastní haš, který zaručuje jeho integritu. Je-li vstupem programu kontrolní soubor, pak po ověření jeho vlastní integrity proběhne kontrola integrity chráněných souborů. Tato aplikace je vyvinuta pro Unix/Linux v gcc 3.2 a pro MS Windows v MS Visual C++ 6.0. Na obou platformách pracuje v textově orientovaném prostředí. V MS Windows je navíc přidána podpora grafického uživatelského rozhraní.

[1] V dalším textu budeme hašovacími funkcemi téměř vždy rozumět kryptografické hašovací funkce.
[2] Výstup hašovací funkce. Typicky hexadecimální řetězec fixní délky.
[3] MACs – autentizační kódy zpráv, někdy také označovány jako ověřovací kódy zpráv.

Kapitola 2

2 Klasifikace a struktura hašovacích funkcí

V následujícím textu budeme vycházet především z [1].

2.1 Označení a terminologie

h, h_k, k	- hašovací funkce, klíčovaná hašovací funkce, tajný (soukromý) klíč
x	- data či zpráva určená k ochraně
$y=h(x)$	- haš (reprezentativní obraz) dat, někdy označován jako: hash code, hash result, hash value, imprint, digital fingerprint, message digest
$N, N_0, (n \in N)$	- množina všech přirozených čísel bez nuly, s nulou, (n je prvkem množiny N)
$\|N\|$	- mohutnost (kardinalita) množiny N
$\|\|$	- binární operátor zřetězení
Σ	- abeceda – v našem případě binární, čili $\Sigma = \{0,1\}$.
Σ^*, Σ^n	- množina všech binárních slov (řetězců) libovolné délky, a délky právě n
Σ^{*n}	- množina všech řetězců maximální délky n

2.2 Základní rozdělení

Definice 1: *Hašovací funkce* (v obecném pojetí) je taková funkce $h: \Sigma^{*m} \rightarrow \Sigma^n$, která splňuje alespoň následující dvě vlastnosti:

1. *Komprese – h* převádí vstup *x* libovolné konečné délky na výstup *h(x)* pevné délky *n*.
2. *Snadná vypočitatelnost* – hodnota *h(x)* je snadno vypočitatelná.

Hašovací funkce dělíme do dvou základních skupin podle toho, mají-li vstup s jedním nebo se dvěma parametry. *Bezklíčové hašovací funkce* dostávají na vstup pouze data, zatímco *klíčované hašovací funkce* dostávají na vstup navíc ještě tajný klíč. Obvykle je však užitečnější podrobnější rozdělení hašovacích funkcí založené na dalších jejich vlastnostech (viz obr. 1). V této práci se zaměříme především na následující dvě třídy hašovacích funkcí.

2.2.1 Modifikační detekční kódy (MDCs – Modification Detection Codes)

Někdy se také označují jako *manipulační detekční kódy* nebo také *kódy zaručující integritu zpráv* (MICs – Message Integrity Codes). Účel MDC je (neformálně řečeno) vytvořit haš dat a usnadnit zabezpečení integrity dat, které je požadováno některými specifickými aplikacemi. MDCs jsou podtřídou bezklíčových hašovacích funkcí a lze je rozdělit na:

a) jednocestné hašovací funkce;
b) bezkolizní hašovací funkce.

Jejich definice budou uvedeny později.

2.2.2 Ověřovací kódy zpráv (MACs – Message Authentication Codes)

MACs jsou podtřídou klíčovaných hašovacích funkcí. Jejich účel je (neformálně řečeno) zajistit integritu a autenticitu zprávy zároveň. MAC detekuje jakoukoliv úmyslnou či neúmyslnou změnu ve zprávě, čímž zajišťuje integritu zprávy, avšak současně také autentizuje jejího původce, neboť ten jediný (vyjma příjemce) mohl znát tajný klíč.

2.3 Základní vlastnosti

Jak již bylo naznačeno v úvodu, kryptografické hašovací funkce musí splňovat oproti standardním nekryptografickým hašovacím funkcím i některé další podmínky. V této části si uvedeme další[1] kritéria pro bezklíčové hašovací funkce.

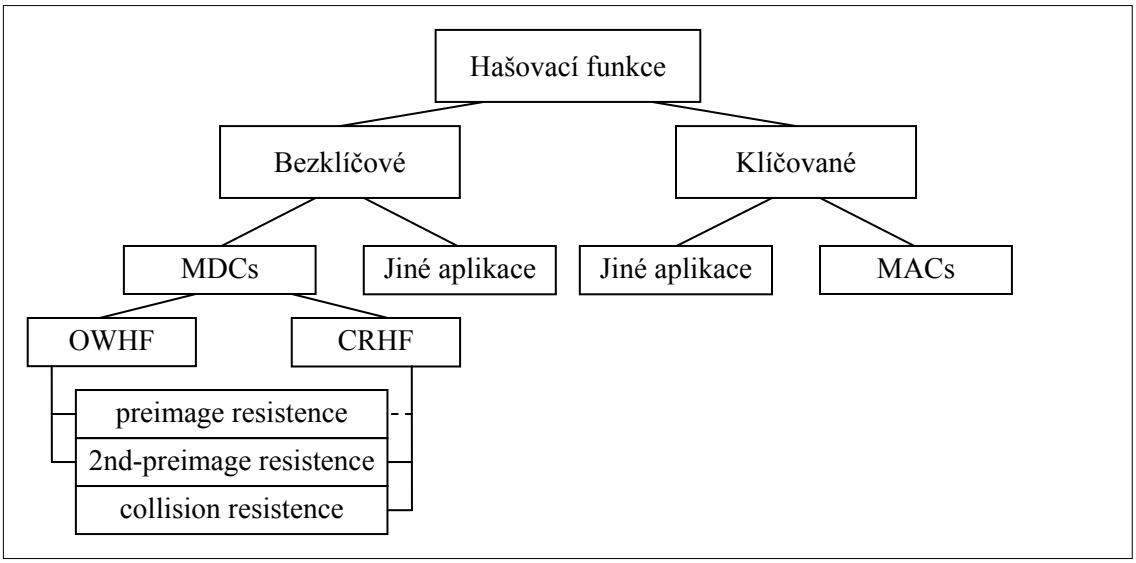
1. *Odolnost proti nalezení vzoru* (preimage resistence) – pro libovolné předem dané y je výpočetně nemožné nalézt vzor x', pro něž platí $h(x') = y$.

2. *Odolnost proti nalezení druhého vzoru* (2nd-preimage resistence) – pro dané x je výpočetně nemožné nalézt x' takové, aby platilo, že $h(x) = h(x')$.

3. *Odolnost proti kolizím* (collision resistence) – je výpočetně nemožné nalézt dva různé vstupy ($x \neq x'$) takové, že jejich hašovaná hodnota bude stejná ($h(x) = h(x')$).

Definice 2: *Jednocestná hašovací funkce* (OWHF – One Way Hash Function) je hašovací funkce h splňující navíc[1] kritéria odolnost proti nalezení vzoru a odolnost proti nalezení druhého vzoru.

Definice 3: *Bezkolizní hašovací funkce* (CRHF – Collision Resistant Hash Function) je hašovací funkce h splňující navíc[1] kritéria odolnost proti nalezení druhého vzoru a odolnost proti kolizím.



Obr. 1.: Zjednodušená klasifikace kryptografických hašovacích funkcí.

[1] Dvě kritéria (komprese a snadná vypočitatelnost) již byly uvedeny v definici 1.

Definice 4: Algoritmus pro *ověřovací kódy zpráv* (MACs) je třída funkcí h_k parametrizovaná tajným klíčem k s následujícími vlastnostmi:

1. *Snadná vypočitatelnost* – pro známou funkci h_k, hodnotu k a vstup x by mělo být snadno vypočitatelné $h_k(x)$. Výsledek se nazývá MAC-hodnota nebo pouze MAC.
2. *Komprese* – h_k převádí vstup x libovolné konečné délky na výstup $h_k(x)$ pevné délky n.
3. *Výpočetní složitost* – i za předpokladu, že je předem známo libovolně velké množství párů text-MAC $(x_i, h_k(x_i))$, kde $i \in N_0$, avšak není znám klíč k, je výpočetně nemožné spočítat další pár text-MAC $(x, h_k(x))$ pro nový vstup $x \neq x_i$.

Poznámka 1: Podmínka *odolnost proti kolizím* negarantuje podmínku *odolnost proti nalezení vzoru*. Nechť g je hašovací funkce odolná vůči kolizím převádějící řetězec libovolné konečné délky na n-bitový řetězec. Definujme funkci $h: \Sigma^{*m} \rightarrow \Sigma^{n+1}$ takto:

$$h(x) = \begin{cases} 1 \parallel x, & \textit{jestliže } x \textit{ má bitovou délku } n \\ 0 \parallel g(x), & \textit{jinak}. \end{cases}$$

Funkce h má tedy $(n+1)$-bitový výstup, a je odolná vůči kolizím. Není však odolná proti nalezení vzoru. (Identita na řetězci pevné délky sice splňuje odolnost vůči kolizím, ale už ne odolnost proti nalezení vzoru.)

Poznámka 2: Z podmínky *odolnost proti kolizím* plyne podmínka *odolnost proti nalezení druhého vzoru*. Stačí uvážit kontrapozitivní formu tohoto tvrzení a je ihned zřejmé, že pokud neplatí podmínka odolnost proti nalezení druhého vzoru (tj. je výpočetně možné k danému x nalézt jiné x' takové, že $h(x) = h(x')$), tak je také porušena i podmínka odolnost proti kolizím (tj. je výpočetně možné nalézt dva různé vstupy $(x \neq x')$ a platí, že jejich hašovaná hodnota bude stejná $(h(x) = h(x'))$).

Poznámka 3: Někdy se můžeme setkat s alternativní terminologií námi uvedených pojmů:

a) odolnost proti nalezení druhého vzoru ≡ *slabá odolnost vůči kolizím*;
b) odolnost proti kolizím ≡ *silná odolnost vůči kolizím*;
c) jednocestná hašovací funkce ≡ *slabá jednocestná hašovací funkce*;
d) bezkolizní hašovací funkce ≡ *silná jednocestná hašovací funkce*.

Kapitola 3

3 Integrita dat a autentizace zpráv

3.1 Základní motivace, definice a rozdělení

Tato část práce, vycházející opět z [1], se zabývá využitím hašovacích funkcí k ochraně integrity dat a k zjištění původu dat (autentizaci zpráv). Typicky bývá požadováno obojí, a to zda data pocházejí od předpokládaného zdroje (data origin authentication) a zda se nezměnil jejich obsah (data integrity). Tyto dva požadavky však nemohou být odděleny. Byla-li totiž data úmyslně pozměněna, pak mají jistě jiný zdroj. Naopak, nemůžeme-li zcela jistě určit zdroj a odkazovat se na něj, nemá ani smysl ptát se, zdali byla data změněna či nikoliv.

Definice 5: *Integrita dat* (data integrity) je vlastnost, která nám zaručuje, že data nemohou být změněna neautorizovaným způsobem, a to po celou dobu od jejich vzniku, přenosu, či uchovávání na autorizovaném místě.

Definice 6: *Ověření původu dat* (data origin authentication) je typ autentizace, s jehož pomocí je potvrzen originální zdroj daných dat vytvořených někdy v minulosti. Zaručuje nám také integritu dat.

Definice 7: *Autentizace zpráv* (message authentication) je termín používaný analogicky s *ověřením původu dat*. Zajišťuje nám jak ověření původu dat se zohledněním původního zdroje zprávy, tak také integritu dat.

Existuje mnoho metod využívajících hašovacích funkcí a zajišťujících nám ověření původu dat. Nejčastěji se však setkáme s následujícími třemi:

1. Ověřovací kódy zpráv (MACs).
2. Digitální podepisovací schémata.
3. Přidání (před zašifrováním) tajné autentizující hodnoty k zašifrovaným datům.

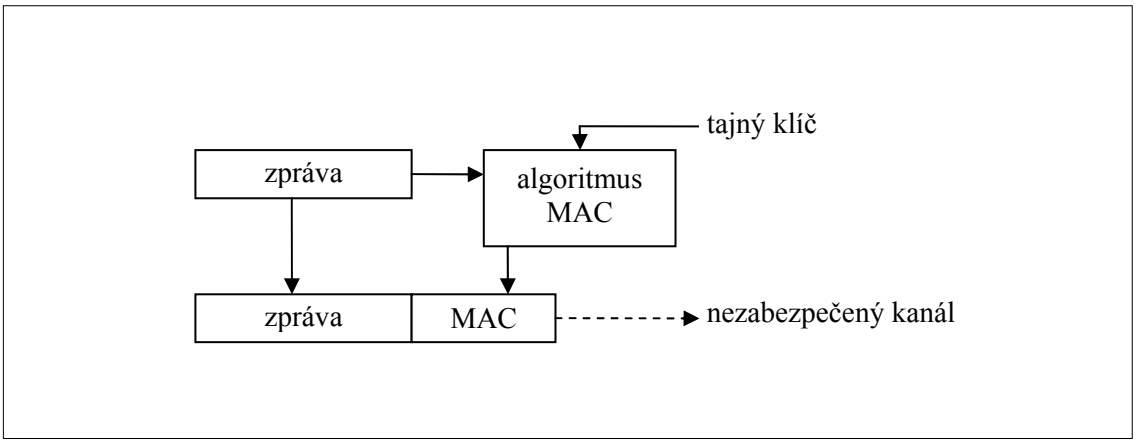
Mechanizmy ověřování původu dat založené na sdílení tajných klíčů (tj. MACs) nepovolují rozlišovat původ dat mezi stranami sdílejícími klíč, a tudíž nemohou potvrdit originální původ dat (oproti například digitálním podpisům). Každá ze stran sdílejících tajný klíč totiž mohla danou zprávu vytvořit. Ačkoliv MACs společně s digitálními podpisy již mohou být použity k potvrzení toho, že data byla vytvořena určitou stranou, neposkytují nám žádné záruky jedinečnosti a aktuálnosti. K tomuto účelu slouží tzv. *transakční autentizace*.

Definice 8: *Transakční autentizace* (transaction authentication) bývá označení pro autentizaci zpráv rozšířenou navíc o poskytnutí záruk na jedinečnost a aktuálnost dat (čímž se předchází detekci přeposílaných zpráv).

Požadavky na jedinečnost a aktuálnost dat bývají někdy souhrnně označovány jako *časově proměnné parametry* (TVPs – Time Variant Parameters). Tento pojem zahrnuje například náhodná čísla, sekvence či časová razítka. Zjednodušeně a neformálně řečeno tedy platí, že autentizace zpráv + TVPs = transakční autentizace.

3.2 Zajištění integrity dat pouze s využitím MAC

Dříve zmiňované ověřovací kódy zpráv (MACs) byly navrženy právě pro aplikace, kde je požadováno zachování integrity zpráv (ale již ne nutně důvěrnost). Autor takovéto zprávy x vypočítá s využitím svého tajného klíče k MAC $h_k(x)$. Tajný klíč je sdílen s předpokládaným příjemcem zprávy, jemuž je vzápětí odeslána zpráva i MAC (obvykle ve tvaru: $x \,||\, h_k(x)$). Příjemce oddělí přijatá data x' od ověřovacího kódu a spočítá si s využitím sdíleného klíče nový ověřovací kód $h_k(x')$. Následně porovná přijatý ověřovací kód $h_k(x)$ se svým právě vypočítaným ověřovacím kódem $h_k(x')$, a jsou-li stejné, znamená to, že data jsou autentická ($x = x'$) a jejich integrita je zachována (tj. byla vytvořena „druhou" stranou, která znala sdílený klíč a během přenosu nebyla pozměněna). Vytvoření a odeslání zprávy je graficky znázorněno níže (viz obr. 2).



Obr. 2.: Zajištění integrity dat s využitím hašovacích funkcí (MACs).

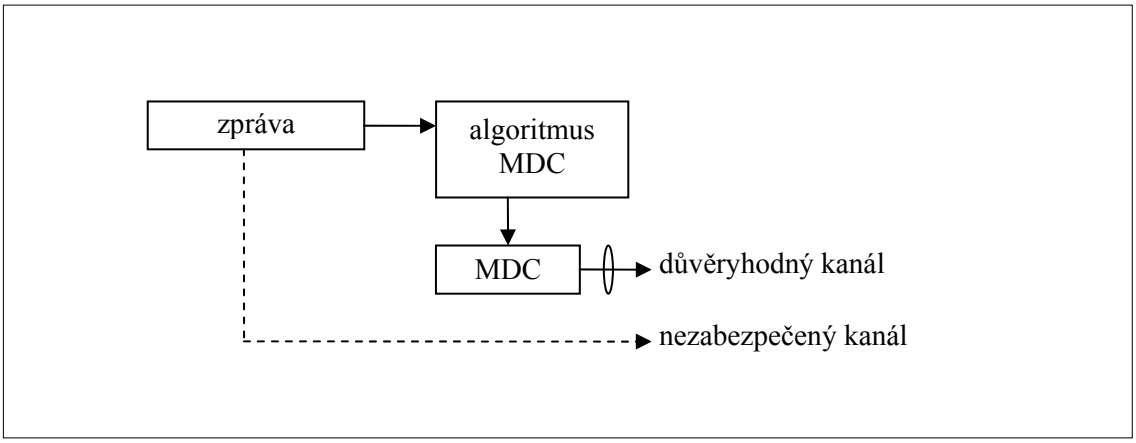
3.3 Zajištění integrity dat s využitím MDC a důvěryhodného kanálu

Při zajišťování integrity dat však není použití tajného klíče nezbytně nutné. Zpráva může být pouze zhašována a autenticita haše může být ochráněna použitím nějakého důvěryhodného či soukromého kanálu. Autor vypočítá za použití modifikačního detekčního kódu (MDC) haš dané zprávy. Tu pak pošle jejímu příjemci nezabezpečeným kanálem, zatímco její haš pošle nezávislou (důvěryhodnou) cestou. Tato cesta nemusí být nutně elektronická, může se jednat o kus papíru, disketu či kompaktní disk. Příjemce pak, podobně jako v předchozím případě, spočítá haš přijaté zprávy a porovná jej s hašem mu doručeným (v tomto případě důvěrným kanálem). Jsou-li stejné, je integrita zprávy zachována. V opačném případě je integrita zprávy porušena. Vytvoření a odeslání zprávy a haše je graficky znázorněno níže (viz obr. 3).

Příkladem zkombinování MDC a důvěryhodného kanálu za účelem zajištění integrity dat jsou schémata digitálních podpisů (jako například RSA[1]). Ty totiž většinou vyžadují užití MDC

[1]RSA (Rivest, Shamir, Adleman) – Asymetrická bloková šifra. Lze ji využít k elektronickému podepisování.

a důvěryhodný kanál je vytvářen asymetrickým podpisem. Takovýmto způsobem může být například distribuován software v nedůvěryhodných sítích.



Obr. 3.: Zajištění integrity dat s využitím hašovacích funkcí (MDCs) a důvěryhodného kanálu.

3.4 Zajištění integrity dat s využitím šifrování

Ačkoliv nám digitální podpisování zaručuje integritu i autentizaci dat, šifrování samotné nám nezaručuje ani jednu z těchto vlastností. V této části budeme zkoumat, jak můžeme využít hašovacích funkcí společně s šifrováním k zajištění integrity dat.

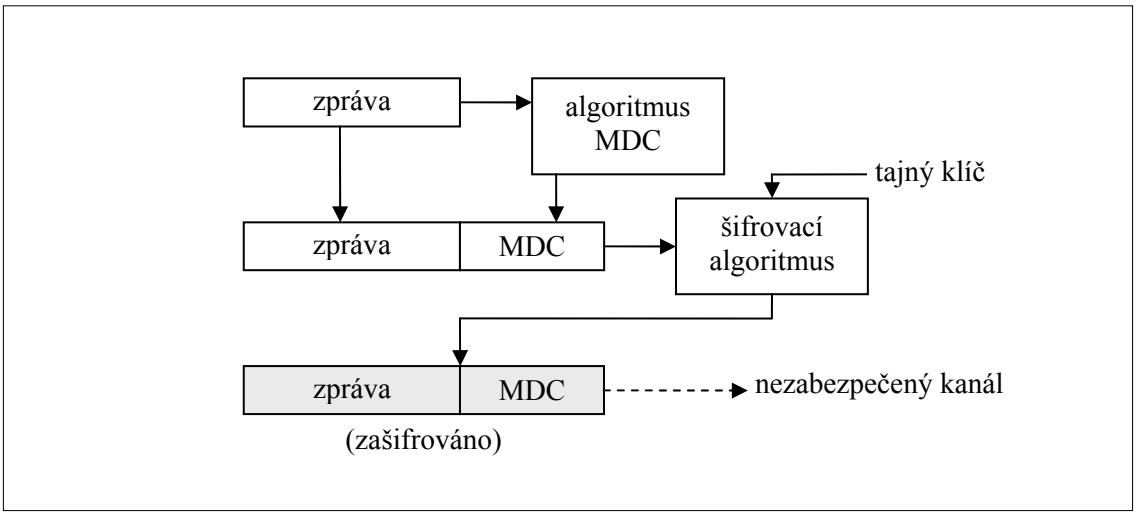
Poznámka 4: Šifrování nám skutečně nezaručuje integritu dat. Mohlo by se sice zdát, že pokud rozšifruji zprávu pocházející od osoby A, a ta dává smysl, že se jedná skutečně o nezměněnou zprávu od osoby A, protože ta jediná měla klíč k potřebný k zašifrování této zprávy. Ve skutečnosti však útočník ani nemusí klíč znát. Existují totiž nejrůznější techniky manipulace se zašifrovanými zprávami (například vhodná záměna a vkládání častěji se opakujících zašifrovaných bloků v ECB[1] modu apod.).

3.4.1 Použití modifikačního detekčního kódu (MDC)

Je-li kromě integrity dat vyžadováno i utajení, můžeme k tomu využít modifikační detekční kódy (funkce h). Autor zprávy vypočítá haš zprávy $H = h(x)$, který připojí k datům a společně s nimi zašifruje. K šifrování využije symetrickou šifru E se sdíleným klíčem k (funkce E_k). Výsledkem je tedy zašifrovaná zpráva i haš: $C = E_k(x \,||\, h(x))$. Tato zpráva je doručena příjemci, který ji s využitím klíče k dešifruje. Následně pak oddělí z dešifrovaných dat zprávu x' od haše H' a spočítá nový haš $h(x')$. Rovnost hašů H' a $h(x')$ nám potvrdí integritu dat. V opačném případě je integrita zprávy porušena. Vytvoření a odeslání zašifrované zprávy a haše je graficky znázorněno níže (viz obr. 4).

Základní myšlenka spočívá v ochraně haše a obtížnosti změnit (bez znalosti klíče) zašifrovanou zprávu C tak, aby si i po této změně haše odpovídaly (tj. obtížnost vytvoření nové zašifrované zprávy $C' = E_k(x' \,||\, h(x'))$).

[1] ECB (Electronic Codebook) je jedna z metod blokového šifrování. Každý z bloků je šifrován nezávisle na ostatních. Dva stejné bloky jsou stejným klíčem šifrovány vždy stejně.



Obr. 4.: Zajištění integrity dat s využitím hašovacích funkcí (MDCs) a šifrování

3.4.2 Použití ověřovacího kódu zpráv (MAC)

V některých případech je doporučeno použít MAC namísto MDC. Předchozí myšlenka tak zůstane nezměněna, pouze místo $h(x)$ bude použito $h_k(x)$. Výsledná zašifrovaná zpráva i haš bude mít tedy podobu: $C = E_k(x \,||\, h_k(x))$. Toto vylepšení nám přináší výhodu v tom, že pokud šifra selže, tak nám MAC stále zaručuje integritu dat. Ovšem nevýhodou je nutnost znát dva klíče. Vzhledem k tomu, že oba dva zabezpečovací mechanizmy by měly být absolutně nezávislé, je vhodné volit i klíče k a k' různé.

3.5 Neúmyslná ohrožení integrity dat

Až doposud jsme se zabývali zajištěním integrity dat z pohledu spíše bezpečnostního (tj. ochrana proti úmyslným změnám dat). Je však třeba zmínit, že mnoho chyb, které se mohou vyskytnout, mají charakter spíše náhodné chyby. Tyto chyby bývají nejčastěji způsobovány poruchami a šumem v přenosových kanálech, a mohou tak zapříčinit, že k nám data přicházejí pozměněná. Veškeré výše uvedené metody jsou samozřejmě schopny tyto chyby odhalit, avšak nejsou na to navrženy (tj. jejich algoritmy jsou vesměs poměrně složité, a šifrovat zprávu s hašem jen kvůli náhodným chybám také není zcela praktické).

Proti takovýmto poruchám v přenosu se v praxi používají nekryptografické techniky (umožňující mj. rychlejší výpočet a nepoužívající žádné tajné klíče či důvěryhodné kanály), nejčastěji *kontrolní součty* (checksums). Tyto kontrolní součty bývají většinou volně přiloženy k datům, k nimž se vztahují, a jsou společně s nimi šířeny. Získáme-li takováto data, ihned se můžeme přesvědčit, proběhl-li přenos správně či nikoliv. Takovéto kontrolní součty mohou být i součástí některých protokolů, které při detekci chyby mohou nejen nechat zopakovat přenos, ale dokáží i samy opravit určitý typ chyb bez nutnosti opětovného přenosu. Pak se většinou nazývají *Error Correcting Codes* (ECCs). V praxi se většinou užívají *Cyclic Redundancy Codes* (CRCs), které jsou založené na bázi polynomů.

Kapitola 4

4 Útoky na hašovací funkce

4.1 Narozeninové útoky

Narozeninové útoky (birthday attacks) patří do skupiny obecných útoků nezávislých na specifických vlastnostech algoritmu dané funkce. Podstatná je pouze délka haše a doba trvání jeho výpočtu. Při útoku hrubou silou na odolnost proti nalezení druhého vzoru se vypočítávají haše náhodných zpráv tak dlouho, než některý z nich není roven haši původní zprávy, čímž vznikne kolize. Pravděpodobnost nalezení takovéto náhodné zprávy je $1/2^n$, kde n je délka haše dané hašovací funkce. Oproti tomu narozeninové útoky využívají takzvaného *narozeninového paradoxu*. Jeho základní myšlenka spočívá v tom, že pravděpodobnost narozenin dvou lidí ve stejný den ve skupině např. 23 lidí je asi $(1-(365!/(365-23)!)/365^{23})) = 0{,}5072 \approx 50{,}7\%$. S přibývajícím počtem lidí tato pravděpodobnost ještě roste (pro 30 lidí je asi 70,6% a pro 40 lidí je 89,1%).

Jeden z prvních útoků na n-bitové hašovací funkce využívajících *narozeninového paradoxu* provedl Yuval [2]. K originální zprávě x_1 zkonstruoval padělek, zprávu x_2. Pak vygeneroval $2^{n/2}$ mírných modifikací x_1' zprávy x_1, a ty uschoval spolu s jejich nově vypočítaným n-bitovým hašem $h(x_1')$. Pak vytvořil mírné modifikace x_2' zprávy x_2, a jejich haše $h(x_2')$. Spočítané haše porovnal, a když se shodovaly ($h(x_1') = h(x_2')$), získal dvě kolidující zprávy x_1' a x_2'.

Existence takovýchto kolizí by se dala poměrně snadno zneužít. Stačí uvážit, že by se mírné modifikace skládaly pouze z různých změn „netisknutelných“ znaků (tj. mezery by se nahrazovaly tvrdými mezerami, přidávaly a odebíraly by se prázdné řádky apod.), nebo z nahrazování podobných slov tak, aby text dával stále smysl. Takto by se daly předem získat dvě kolidující zprávy s různým, ale smysluplným textem (třeba nějaké smlouvy). A vzhledem k tomu, že hašovacích funkcí se poměrně často[1] užívá v souvislosti s elektronickým podpisem, mohl by „útočník“ podepsat pro sebe méně výhodnou zprávu (tj. její haš). Později by pak mohl kdykoliv tvrdit opak a za originál vydávat zprávu druhou, výhodnější (jejíž haš je totožný).

4.2 Útoky na zřetězení

Útoky na zřetězení jsou založeny na iteračním charakteru hašovacích funkcí a využívají některých jejích proměnných (týkají se především MDCs). Zde budeme vycházet z [1] a [3].

4.2.1 Meet in the middle

Typickým příkladem může být útok *meet in the middle*. Tento útok je do značné míry podobný Yuvalovu narozeninovému útoku, pouze s tím rozdílem, že kolize jsou hledány na mezivýsledcích (tj. zřetězených proměnných H). Útok je směrován nejen proti odolnosti nalezení kolizí, ale

[1] Většinou nebývá elektronicky podepisována celá dlouhá zpráva, ale pouze její n-bitový haš.

i proti odolnosti nalezení druhého vzoru, což původní Yuvalův útok neumožňoval. Útočník v tomto případě vygeneruje s mírnými modifikacemi různé možnosti první poloviny zprávy a různé možnosti druhé poloviny zprávy. Hašování prvních polovin pak probíhá od začátku až do určitého kroku *i* a hašování druhých polovin probíhá inverzně od konce zpět do kroku *i*. Testuje se shoda mezivýsledků v zřetězené proměnné H_i. Obrana proti tomuto druhu útoku spočívá v lepším a důmyslnějším návrhu hašovací funkce tak, aby nebylo možné počítat inverzní kroky.

4.2.2 Pevný bod

Pevný bod (fixed point) kompresní funkce je dvojice (H_{i-1}, x_i) pro kterou platí $f(H_{i-1}, x_i) = H_{i-1}$. Nalezení takovéhoto bodu znamená, že do funkce může být vložen libovolný počet bloků x_i a výsledný haš se nezmění. Tímto je napadena odolnost proti nalezení druhého vzoru a také bezkoliznost.

4.2.3 Diferenciální kryptoanalýza

Diferenciální kryptoanalýza (differential cryptanalysis) je velmi silný nástroj sloužící k útokům nejen na hašovací funkce (včetně MACs), ale i na různé blokové šifry. V podstatě se hledají rozdíly mezi vstupem do kompresní funkce a výstupem z ní. Nalezené anomálie se pak statisticky vyhodnotí.

Kapitola 5

5 Popis funkčnosti programu

5.1 Motivace a využití

Cílem praktické části této bakalářské práce je naprogramování open-source utility využívající kryptografických hašovacích funkcí pro zajištění kontroly integrity dat. Program využívá hašovacích funkcí z třídy SHA-*x*, kde *x* označuje příslušnou verzi funkce. Vstupem mu jsou buď soubory určené k zajištění integrity či samotný kontrolní soubor, který uchovává haše chráněných dat. Výstupem je v prvním případě kontrolní soubor a v druhém případě výpis informující o stavu integrity chráněných dat. Tato aplikace je vyvinuta jak pro operační systémy typu Unix/Linux, tak také pro MS Windows. Kromě zajištění a ověření integrity dat může být program spolu s využitím asymetrické kryptografie použit i k ověření pravosti a původu dat.

5.2 Typy a principy použitých hašovacích funkcí

Program podporuje použití čtyř hašovacích funkcí z rodiny funkcí SHA (Secure Hash Algorithm), jedná se o SHA-1, SHA-256, SHA-384 a SHA-512. Poslední (chronologicky vlastně první) pátou funkci, původně označovanou pouze SHA nebo SHA-0, nepodporuje, protože tato funkce již není považována za bezpečnou. V roce 1998 na ni byl v [4] popsán teoretický útok založený na diferenciální kryptoanalýze (jeho časová složitost je 2^{61}), což je mnohem lepší výsledek než nám dává narozeninový útok (v tomto případě, pro délku haše 160 bitů, je jeho časová složitost 2^{80}).

Všechny výše zmiňované hašovací funkce lze zařadit do třídy bezklíčových MDCs. Stejně jako většina hašovacích funkcí využívají i funkce typu SHA kompresní funkci *f* iterativně. Čili po rozdělení vstupu *M* do bloků M_1 až M_n požadované délky, jejich doplnění (padding) a nastavení inicializačního vektoru *IV*, proběhne výpočet:

1. $H_0 = IV$
2. $H_i = f(H_{i-1}, M_i)$, kde $i = 1..n$
3. $H(M) = g(H_n)$

Výsledná transformace *g* bývá většinou identita $g(H_n) = H_n$. Kompresní funkce *f* a *IV* jsou samozřejmě u každé hašovací funkce definovány jinak, ale princip iterací zůstává.

5.2.1 SHA-1

Funkce SHA-1 byla schválena americkým úřadem pro normalizaci[1] a jako standard byla 17.4.1995 oficiálně vyhlášena v dokumentu FIPS PUB[2] 180-1 [5]. Nahradila tak původní funkci SHA-0 z roku 1993, definovanou ve FIPS PUB 180. U SHA-1 zatím nejsme schopni nalézt

[1] NIST – National Institute of Standards and Technology.
[2] FIPS PUB – Federal Information Processing Standards Publication.

kolize rychleji než s použitím narozeninového útoku (přičemž obě hašovací funkce se od sebe liší pouze o jednu jednobitovou rotaci vlevo přidanou do kompresní funkce v SHA-1). Vstupem SHA-1 je zpráva délky menší než 2^{64} bitů a výstupem je řetězec dlouhý 160 bitů. Kompresní funkce zpracovává data po blocích délky 512 bitů, čili celková délka zprávy musí být dělitelná 512. Je-li kratší, provádí se doplnění zprávy do požadované délky. Toto doplnění se provádí, i když je zpráva dělitelná 512 (tj. doplňuje se například i prázdná zpráva).

5.2.2 SHA-256, SHA-384 a SHA-512

Tyto hašovací funkce se objevily v těsném závěsu za vyhlášením standardu AES[1] a jsou popsány v oficiálním dokumentu FIPS PUB 180-2 [6]. Tím byl doplněn současný standard hašovacího algoritmu SHA-1 o nové algoritmy SHA-256, SHA-384 a SHA-512. Číslo za názvem algoritmu udává délku jeho binárního výstupu. Tyto tři funkce nám nabízejí zabezpečení na úrovni 128, 192 a 256 bitů, což jsou právě délky klíčů podporované současným standardem AES. SHA-256 pracuje, podobně jako SHA-1, se vstupy délky menší než 2^{64} bitů a její kompresní funkce po blocích délky 512 bitů. I zde musí být délka zprávy dělitelná 512, a provádí se proto doplnění do požadované délky. SHA-512 pracuje až se vstupy délky 2^{128}-1 bitů a její kompresní funkce po blocích délky 1024 bitů. Délka zprávy musí být proto dělitelná 1024, a opět se provádí její doplnění. Obdobně jako SHA-512 pracuje i SHA-384, která vznikne pouze zkrácením jejího výstupu na 384 bitů. Má však samozřejmě jiné inicializační hodnoty, a tudíž i jejich haše jsou zcela odlišné (tj. výsledný haš funkce SHA-384 není pouze předponou haše vyprodukovaného funkcí SHA-512).

5.3 Metody detekce chyb

Kromě samotných hašovacích funkcí ovlivňuje funkčnost programu také tvar kontrolního souboru. Na něm závisí, co vše bude o chráněných datech uchováváno a při jejich následné kontrole zpětně využito. Asi nejdůležitější je otázka, zda v kontrolním souboru uchovávat názvy souborů či nikoliv. Každá odpověď na ni má svá pro i proti. Někdy může být podstatná i délka výsledného kontrolního souboru, proto je nutné také zvážit, zda ukládat všechny haše souborů či pouze jejich výsledný *xor*. V závislosti na zodpovězení předcházejících otázek může být navrženo poměrně mnoho specifických metod s různými vlastnostmi detekce chyb. Ať již se však rozhodneme pro jakoukoliv metodu, bude potřeba zajistit ještě integritu samotného kontrolního souboru. Mohla by totiž nastat situace, kdy bychom měli správná data, ale poškozený kontrolní soubor. Aplikace by pak buď kontrolní soubor vůbec nezpracovala, nebo by ohlásila porušení integrity chráněných (v tomto případě však správných) dat. Její chování by pak záviselo na povaze a rozsahu poškození kontrolního souboru.

Většina dostupných volně šířitelných aplikací zabývajících se ochranou integrity dat pracuje tak, že do kontrolního souboru ukládá jména chráněných souborů a za ně mezerou oddělený kontrolní součet. Jsou tedy závislé na načítání jmen souborů z kontrolního souboru. Kontrolní součet bývá navíc většinou vypočítáván pomocí CRC kódů a integrita samotného kontrolního souboru nebývá také nijak chráněna. Hlavním cílem bylo tedy navrhnout aplikaci tak, aby všechny výše zmíněné „nedostatky“ odstraňovala, a navíc dávala uživateli možnost volby mezi různými metodami detekce chyb a typů hašovacích funkcí. Již samotné použití funkcí typu SHA nám rozšiřuje možnosti využití programu (spolu s využitím asymetrické kryptografie) o ověření pravosti a původu dat.

[1] AES (Advanced Encryption Standard) specifikuje symetrický šifrovací algoritmus, který může být použit k ochraně elektronických dat. V současné době je to algoritmus Rijndael, specifikovaný v dokumentu FIPS PUB 197.

5.3.1 Navrhované metody

Tím, že jsme se rozhodli neukládat si jména chráněných souborů do kontrolního souboru, vznikla potřeba stanovit nějaký závazný mechanizmus načítaní souborů při ověřování integrity dat. Ten byl stanoven tak, že se budou vždy načítat a kontrolovat soubory umístěné v aktuálním adresáři, vyjma kontrolního souboru samotného. V důsledku toho byly navrženy tři metody detekce chyb:

1. *Nízká úroveň detekce* (víme, že někde je nějaká chyba, nikdy nevíme jaká):
   Na začátek kontrolního souboru (viz obr. 5) se uloží aktuální konfigurace programu (tj. informace o tom, jaká metoda a typ funkce SHA jsou použity). Bude se hašovat po jednom souboru, výsledné haše budou *xor*-ovány a výsledný řetězec bude uložen. Na konec kontrolního souboru se uloží haš kontrolního souboru, který nám zajistí jeho integritu. Tato metoda nám zaručí integritu dat, ale ne názvů souborů. Bude-li integrita libovolného jednoho či více souborů narušena, tak se to sice dozvíme, avšak informaci o tom, který ze souborů je narušen, nám tato metoda nepřinese. Nedozvíme se ani, je-li narušen pouze jeden soubor, nebo zda jich je narušeno více. Pouhou změnu názvu souborů nebudeme schopni ani detekovat. Chybné výsledky také obdržíme přidáním či smazáním nějakého ze souborů z aktuálního adresáře.

konfigurace
xor hašů
haš tohoto souboru

Obr. 5.

2. *Střední úroveň detekce* (víme, kde je nějaká chyba, ne vždy víme jaká):
   Na začátku bude opět konfigurace. Bude se hašovat po jednom souboru, tedy každý soubor bude mít svůj haš a ten bude uložen (viz obr. 6). Za haši jednotlivých souborů bude ještě uložen jejich *xor*. Úplně na konci bude opět haš celého souboru. Tato metoda nám, oproti předchozí, detekuje porušení integrity dat u konkrétních souborů. Kontrola u této metody bude probíhat tak, že pokud bude souhlasit *xor* hašů, je vše v pořádku, pokud ne, pak se budou vždy postupně testovat jednotlivé haše. Nastane-li shoda s hašem uloženým v kontrolním souboru, je daný soubor v pořádku, jinak je porušen (nebo je v adresáři navíc). Pouhou změnu názvu souborů nebudeme schopni ani zde detekovat.

konfigurace
haš prvního souboru … haš *n*-tého souboru
xor hašů
haš tohoto souboru

Obr. 6.

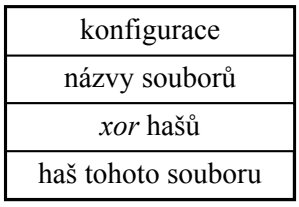
3. *Vysoká úroveň detekce* (víme, kde je chyba, a vždy víme jaká je to chyba):
   Tato metoda v podstatě kopíruje metodu druhou s tím rozdílem, že v kontrolním souboru je původní haš obsahu souboru nahrazen společným hašem názvu a obsahu souboru a za něj je ještě uložen haš samotného názvu souboru. Zde již jsme schopni detekovat i změnu názvu souborů. (Haš názvu a obsahu zde byl použit kvůli optimalizaci. V případě, kdy by nebyla integrita porušena, by se počítal pouze tento haš a shoda by byla nalezena nejpozději v *n*-tém pokusu, kde *n* je počet chráněných souborů. Byl-li by použit pouze haš obsahu, bylo by vždy nutno pro potvrzení zachování integrity počítat ještě haš názvu.)

Podobných metod by se dalo navrhnout i více. Prakticky použitelná se však jeví pouze metoda třetí, která jako jediná zajišťuje ochranu proti změně (a dokonce i záměně) názvů souborů. Cenou za vypuštění názvů souborů z kontrolního souboru je nám, kromě pouze jedné použitelné metody, také dosti veliké množství testů potřebných k dosažení shody. V druhé metodě je to v případě všech pozměněných souborů n^2 testů. V třetí metodě je to v případě pozměnění všech názvů souborů nejen n^2 testů, ale i výpočet dalších n hašů, což je v případě většího počtu souborů a funkce SHA-512 téměř nepoužitelné. Neúnosná se stala také velikost kontrolního souboru.

5.3.2 Podporované metody

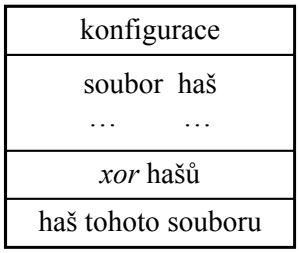
Po důsledném rozboru výhod a nevýhod navrhovaných metod jsme se rozhodli upustit od požadavku neukládat si jména chráněných souborů do kontrolního souboru. Tato modifikace nám přinesla celkové zjednodušení metod, a to jak po stránce návrhu, tak také po stránce implementační. Původní tři návrhy metod detekce chyb byly přepracovány, přičemž se zjistilo, že původně v podstatě jediná použitelná metoda je zbytečná, a může být plnohodnotně nahrazena novou metodou číslo dvě. Dalším poznatkem bylo, že již pro nás nemá význam zabývat se tolik tím, co se stane, je-li narušení integrity způsobeno přejmenováním jednoho či více souborů. Změna názvu souboru totiž způsobí, že není programem vůbec nalezen. Ovšem stále má smysl ptát se, co se stane, jsou-li pouze prohozena jména některých z chráněných souborů. Nové dvě metody detekce chyb vycházejí z původně navrhovaných metod. Jsou to:

1. *Nízká úroveň detekce*:
   Na začátek kontrolního souboru (viz obr. 7) se uloží aktuální konfigurace programu, následována volným řádkem a jmény chráněných souborů. Názvy souborů se uloží v pořadí, v jakém byly programem načteny, a opět budou následovány volným řádkem. Hašování bude probíhat po jednom souboru. Výsledné haše budou *xor*-ovány a výsledný řetězec, následovaný prázdným řádkem, uložen. Na konec se uloží haš kontrolního souboru, který nám zajistí jeho integritu. Oproti původní metodě jsme nyní navíc schopni zaručit integritu názvů souborů a nejsme závislí na přidávání či odebírání nechráněných souborů z aktuálního adresáře. Avšak záměnu názvů chráněných souborů nám ani tato metoda nedetekuje.



Obr. 7.

2. *Střední úroveň detekce*:
   Tato metoda je v podstatě ekvivalentní předchozí metodě, jen s tím rozdílem, že za každým názvem souboru je v kontrolním souboru uložen ještě jeho haš (viz obr. 8). Tato metoda nám oproti předchozí detekuje jak porušení integrity dat u konkrétních souborů, tak také záměnu názvů souborů (jak již bylo uvedeno výše, změna názvů je samozřejmě detekována také).



Obr. 8.

Po pečlivém zvážení všech uvedených navrhovaných i podporovaných metod je vidět, že požadavek na ukládání jmen souborů do kontrolního souboru, který byl původně označen jako „nedostatek“ určený k odstranění, je oprávněný a jeho odstraněním bychom kromě zvýšené časové složitosti nic nezískali.

Kapitola 6

6 Popis realizace a implementace programu

6.1 Textově orientovaná verze

Textově orientovaná verze je vytvořena jak pro operační systémy Unix/Linux, tak také pro MS Windows. Při implementaci pro Unix/Linux byly využity hašovací funkce SHA-1 od Steve Reida a SHA-256, SHA-384, SHA-512 od Aarona D. Gifforda. Kvůli problémům s kompilací posledních tří funkcí pod systémem MS Windows (jak v Borland C++, tak ve Visual C++) byly v tomto systému použity hašovací funkce Dr. Briana Gladmana. V systémech Unix/Linux je k aplikaci dodán skript, pomocí něhož je možno změnit nastavení souboru *Makefile*. Takto lze ještě před kompilací zvolit, zda bude aplikace provozována na systémech s architekturou typu little či big endian a zda bude použita angličtina či čeština. Rozdíl mezi little a big endianem je ve způsobu ukládání čtyř po sobě následujících bajtů v paměti. Orientaci little endian mají architektury typu CISC (Comlex Instruction Set Computer – např. procesory typu Intel, AMD), big endian mají architektury typu RISC (Reduced Instruction Set Computer). V systému Windows[1] je pro nastavení odlišné jazykové podpory nutné před kompilací změnit makro LANGUAGE (*0* pro angličtinu, *1* pro češtinu).

Utilita se volá z příkazové řádky a ovládá se pomocí parametrů zadávaných také z příkazové řádky. Není-li použito žádné nastavení, použijí se výchozí hodnoty a na vstupu jsou očekávány soubory. Název spustitelného souboru je *val*, popřípadě *val.exe*. K ovládání programu se požívá následující syntax: *val* [NASTAVENÍ] [SOUBORY], kde [NASTAVENÍ] = [{-s „xxx“, -m „x“, -o „soubor“, -c „soubor“, -i, -a „text“}] a význam jednotlivých parametrů je uveden níže:

-s „xxx“	-s (SHA) „xxx“ je typ funkce SHA či délka jejího výstupu. Možnosti: *1* | *160* | *256* | *384* | *512*. Výchozí hodnota „xxx“: *1*.
-m „x“	-m (method) „x“ udává typ metody, která bude použita. Možnosti: *1* | *2* | *3*. Výchozí hodnota „x“: *1*.
-o „soubor“	-o (output) „file“ je název výstupního kontrolního souboru. Výchozí hodnota „soubor“: *checksum.sha*.
-c „soubor“	-c (check) Je-li tento parametr použit, ostatní parametry jsou ignorovány a „soubor“ je název vstupního kontrolního souboru.
-i	-i (input) Je-li tento parametr použit, ostatní parametry kromě -s „xxx“ jsou ignorovány. Program bude čekat na zadání řetězce ze standardního vstupu a vrátí nám jeho haš. Není-li na standardní vstup nic přesměrováno, očekává zadání řetězce z klávesnice. Ukončovací znak je v UNIXU „^D“ v MS-DOS „^Z“.
-a „text“	-a (argument) Je-li tento parametr použit, ostatní parametry kromě -s „xxx“ jsou ignorovány. Program bude čekat na zadání řetězce „text“ jakožto argumentu a vrátí nám jeho haš.

[1] Systémy MS Windows jsou vytvořeny pro architekturu CISC.

Veškeré hodnoty uvedené v hranatých závorkách *[]* jsou nepovinné. Symbol ^ značí současné stisknutí klávesy *Ctrl*. Není-li použito ani jedno z nastavení {-a, -i, -c} a jsou-li předchozí nastavení zadána správně, očekává program seznam souborů k hašování ve tvaru: [SOUBORY] = soubor_1 [soubor_2 ... soubor_N]. Tedy povinný je alespoň jeden název souboru, jinak je vypsán text nápovědy.

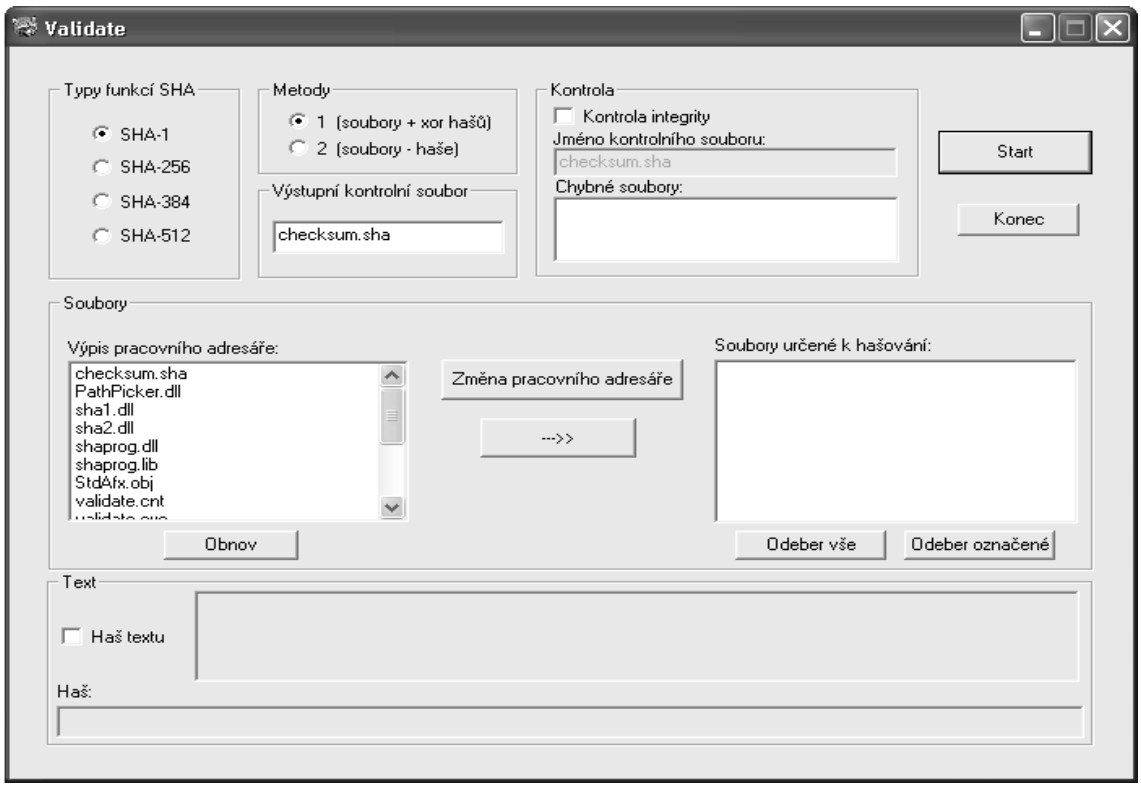
Příklady spouštění programu *val*:

val -a "text" -s 256	Vrátí haš (SHA-256) řetězce „text“.
val -c kontrola.crc	Zkontroluje integritu souborů uvedených v souboru *kontrola.crc*.
val -s 384 1.txt 2.txt	Do souboru *checksum.sha* bude uložen *xor* hašů souborů *1.txt* a *2.txt*.

Zbývá již jen dodat, že textově orientovaná verze byla naprogramována v jazyce C, a verze pro Unix/Linux byla zkompilována v gcc 3.2, zatímco verze pro MS Windows byla zkompilována ve Visual C++.

6.2 Verze GUI (Graphic User Interface)

Pro systém MS Windows byla také naprogramována verze s grafickým uživatelským rozhraním (GUI). K jejímu vytvoření bylo použito opět Visual C++ verze 6.0. Poněvadž se však vyskytly problémy s kompilací programu napsaného čistě v jazyce C a začleněného do C++ a přepisování všech zdrojových souborů by bylo náročné, byly z původního programu vytvořeny tři na sobě závislé dynamické knihovny. Tyto knihovny byly modifikovány pro snadnější použití v grafickém režimu v C++ (zejména byl předělán celý systém chybových a varovných hlášení). Grafická podoba programu *validate.exe* je znázorněna níže (viz obr. 9).



Obr. 9.: Grafické uživatelské rozhraní programu *Validate*.

6.3 Benchmark

Před provedením *benchmarku*[1] byl proveden test, který měl potvrdit vzájemnou kompatibilitu mezi oběma verzemi programu. Tento test však nedopadl úspěšně, protože se haše u funkcí SHA-384 a SHA-512 na objemnějších datech neshodovaly. Po podrobnější analýze problému jsme zjistili, že kritickou se stala hranice 512 MB (u menších souborů si haše odpovídaly). Aby mohla být chyba odstraněna, bylo potřeba zjistit, zda se nachází v implementaci Aarona D. Gifforda nebo Dr. Briana Gladmana. Za tímto účelem jsme vyhledali a nainstalovali volně šiřitelný program Hashish. Avšak i tento program vykazoval stejný typ chyby, a tedy haše všech tří programů se pro tak velké soubory lišily. V důsledku toho, byli kontaktováni oba dva autoři implementací hašovacích funkcí použitých v našem programu, a byli upozorněni na jejich vzájemnou nekompatibilitu. Jakožto podklad jsme k dopisu a k jejich zdrojovým kódům přiložili zdrojový kód programu, který generoval dva soubory o velikostech 536870911 a 536870912 (512 MB) bajtů. Přiloženy byly samozřejmě i námi zjištěné haše těchto souborů.

Abychom však nebyli závislí pouze na případné odpovědi autorů obou implementací, pokusili jsme se ještě jednou o sjednocení použitých funkcí (tj. nahrazení jedné z nich tou druhou). Vzhledem k tomu, že kompilace hašovacích funkcí od Aarona D. Gifforda se nám již dříve v systému MS Windows nepodařila, zbyla již jen poslední možnost, a to kompilace hašovacích funkcí Dr. Briana Gladmana v systémech Unix/Linux. Tato kompilace proběhla úspěšně, pouze se musely vyřešit architektonické záležitosti typu little a big endian. Tím jsme sice dosáhli korektního fungování programu, avšak nemohli jsme zaručit, zda je tato funkce implementována podle stanovených norem (tj. počítá-li správné haše).

Oba dva autoři se nám v následujících dnech ozvali. Chyba byla právě v implementaci Dr. Briana Gladmana. Spolu s poděkováním a vysvětlením, že oprava bude na jeho stránkách zveřejněna později společně s vylepšeným a rychlejším kódem pro SHA-256, na kterém právě pracuje, nám v příloze zaslal verzi, která námi zjištěnou chybu již nevykazovala. Analyzovali jsme tedy tuto novou verzi a zjistili, že chyba spočívala v pouhém přepisu. Při konečném upravování vypočítaného haše byla totiž jedna z rotací vpravo provedena pouze o 29 bitů (tj. stejně jako u SHA-256) namísto o 61 bitů. Po korekci příslušných rotací a následujícím úspěšném testu „kompatibility“ jsme již mohli začít provádět benchmark.

Účelem benchmarku bylo zjistit, do jaké míry se výkonnostně liší obě použité implementace hašovacích funkcí, a podat uživateli alespoň částečnou informaci o době trvání hašování dat různých velikostí. Vzhledem k předchozímu úspěšnému zkompilování obou implementací hašovacích funkcí v systémech Unix/Linux byly testy prováděny na těchto platformách. K testování byla použita sestava založená na základní desce Aopen AX6BC, osazená procesorem Intel Celeron 666@833 (Coppermine) a 384 MB paměti SDRAM. Systémová sběrnice (FSB – Front Side Bus) byla 83 MHz a pevný disk IBM o kapacitě 40 GB podporující Ultra DMA 100 byl zapojen pouze v režimu Ultra DMA 33 (což bylo dáno omezením základní desky). Jako operační systém byl použit Red Hat Linux 7.3 (Valhalla). Testování probíhalo na datech o kapacitách 100 kB, 1 MB, 10 MB, 100 MB a 1 GB. Tato data byla vygenerována programem a na disk uložena v souvislém bloku (objektivnějšího výsledku testu bychom však dosáhli použitím dostatečně velkého RAM disku místo pevného disku).

Testování proběhlo celkem 20krát a trvalo devět hodin. Z výsledných časů byly spočítány aritmetické průměry. Výsledné průměry časů funkcí Dr. Briana Gladmana jsou uvedeny v tabulce č. 1 a výsledné průměry časů funkcí Aarona D. Gifforda jsou v tabulce č. 2. V obou tabulkách je také uvedena hašovací funkce SHA-1 od Steve Reida (tj. v tomto případě bylo provedeno 40 testů). Veškeré naměřené hodnoty jsou v sekundách.

[1] Benchmark – výkonnostní test.

Velikost dat	Průměrná doba výpočtu haše. [s]			
	SHA-1	**SHA-256**	**SHA-384**	**SHA-512**
100 kB	0,0035	0,0125	0,0230	0,0245
1 MB	0,0480	0,0990	0,2495	0,2500
10 MB	0,3905	0,9175	2,4745	2,4230
100 MB	4,6070	9,9445	24,9700	25,0655
1 GB	50,0820	104,7840	258,4800	259,0210

Tab. 1: Hašovací funkce implementované Stevem Reidem a Dr. Brianem Gladmanem.

Velikost dat	Průměrná doba výpočtu haše. [s]			
	SHA-1	**SHA-256**	**SHA-384**	**SHA-512**
100 kB	0,0040	0,0120	0,0250	0,0250
1 MB	0,0430	0,0940	0,1970	0,2225
10 MB	0,3790	0,8835	1,9585	1,9815
100 MB	4,5305	9,6215	20,4225	20,4515
1 GB	51,1150	102,0350	212,4995	210,8975

Tab. 2: Hašovací funkce implementované Stevem Reidem a Aronem D. Griffordem.

Z testů je patrné, že hašovací funkce SHA-384 a SHA-512 od Dr. Briana Gladmana jsou mnohem pomalejší. Z tohoto důvodu bylo upuštěno od sjednocování použitých funkcí a ve verzi programu pro Unix/Linux zůstaly použity rychlejší hašovací funkce od Aarona D. Gifforda. Stejně tak je pomalejší i funkce SHA-256, avšak zde již rozdíl činí pouhé dvě sekundy, a jak bylo zmíněno výše, v brzké době by se měla objevit rychlejší verze této funkce. Jistě nás také nepřekvapí poměrně nízká odchylka průměrných časů u funkcí SHA-384 a SHA-512, která je způsobena podobným návrhem těchto funkcí (viz 5.2.2). Vzhledem k velikosti dat lze u průměrných časů jednotlivých funkcí vypozorovat téměř lineární závislost, což je způsobeno iterativně pracující kompresní funkcí.

Kapitola 7

7 Závěr

Cílem práce bylo seznámit se se základním rozdělením kryptografických hašovacích funkcí a s jejich využíváním v oblasti zachování integrity dat. Hašovací funkce jsme si popsali a rozdělili do několika skupin. Viděli jsme, že správná hašovací funkce musí splňovat náročná bezpečnostní kritéria, aby byla schopna odolat případnému útoku, a byla tak použitelná v praxi. Představili jsme si také tři nové hašovací funkce, které by nám měly poskytnout požadovanou úroveň zabezpečení alespoň na příštích pětadvacet let.

V rámci práce byl vytvořen program využívající hašovacích funkcí třídy SHA k zaručení integrity dat. Oproti běžně dostupným aplikacím, zabývajících se ochranou integrity dat, byla navíc implementována ochrana integrity kontrolního souboru a možnost volby metody detekce chyb. Aplikace může být také využita (spolu s použitím asymetrické kryptografie) k ověření pravosti a původu dat.

Při testování rychlosti a kompatibility použitých algoritmů byla objevena chyba při výpočtu hašů v implementaci SHA-384 a SHA-512 Dr. Briana Gladmana. Tato chyba byla autorovi nahlášena a vzápětí byla i opravena.

Literatura

Knižní publikace:

[1] Menezes, A., Van Oorschot, P., Vanstone, S.: *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 0-8493-8523-7.

[2] Yuval, G.: *How to Swindle Rabin*. Cryptologia 3, 1979, s. 187-189.

[3] Preneel, B.: *The State of Cryptographic Hash Functions*. Lectures on Data Security, 1998, s. 158-182. Dokument dostupný na URL http://link.springer.de/link/service/series/0558/papers/1561/15610158.pdf.

[4] Chabaud, F. a Joux, A.: *Differential Collisions in SHA-0*. Extended abstract published in CRYPTO'98. Dokument dostupný na URL http://fchabaud.free.fr/English/Publications/sha.pdf.

[5] Federal Information Processing Standards (FIPS) Publication 180-1.: *Secure Hash Standard (SHA)*. April 1995. Dokument dostupný na URL http://cs-www.ncsl.nist.gov/publications/fips/fips180-1/fips180-1.pdf.

[6] Federal Information Processing Standards (FIPS) Publication 180-2.: *Secure Hash Standard (SHA)*. August 2002. Dokument dostupný na URL http://cs-www.ncsl.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

Internetové odkazy:

Aaron D. Gifford: http://www.aarongifford.com/computers/sha.html

Dr. Brian Gladman: http://fp.gladman.plus.com/cryptography_technology/sha/index.htm

Program Hashish: http://hashish.sourceforge.net/

Program Validate: http://www.fi.muni.cz/~xkrhovj/projekt/SBAPR.html

Příloha A – Testovací vektory hašovacích funkcí

SHA-1
Haš řetězce „*abc*“:
a9993e364706816aba3e25717850c26c9cd0d89d

Haš miliónkrát za sebou napsaného znaku „*a*“:
34aa973cd4c4daa4f61eeb2bdbad27316534016f

Haš miliónkrát za sebou napsaného řetězce „*Hello world!*“:
ba52b28fb230062c604c166ec0161700b8d709f2

SHA-256
Haš řetězce „*abc*“:
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Haš miliónkrát za sebou napsaného znaku „*a*“:
cdc76e5c9914fb9281a1c7e284d73e67f1809a48a497200e046d39ccc7112cd0

Haš miliónkrát za sebou napsaného řetězce „*Hello world!*“:
32afa05e8c1f4ee535da94cd553c3985e35c6bfa15a683c31707c782f8a2264d

SHA-384
Haš řetězce „*abc*“:
cb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc23
58baeca134c825a7

Haš miliónkrát za sebou napsaného znaku „*a*“:
9d0e1809716474cb086e834e310a4a1ced149e9c00f248527972cec5704c2a5b07b8b3dc38ecc4e
bae97ddd87f3d8985

Haš miliónkrát za sebou napsaného řetězce „*Hello world!*“:
f90a2bfb24e5238dcd7d58ceecd74fe77f4ee0ef3ea7096699ba5f3ce5c7c5ddf2a6abb49a82761c80
510d39b28d9a32

SHA-512
Haš řetězce „*abc*“:
ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a8
36ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f

Haš miliónkrát za sebou napsaného znaku „*a*“:
e718483d0ce769644e2e42c7bc15b4638e1f98b13b2044285632a803afa973ebde0ff244877ea60a
4cb0432ce577c31beb009c5c2c49aa2e4eadb217ad8cc09b

Haš miliónkrát za sebou napsaného řetězce „*Hello world!*“:
04eb95effa3e55fb3caf0e9c22b7509da0b206290015f32c4f09fe731acedee6141822b688cad822a
097b68d4bd839507b4152559ed099cc1b80a5f602d618e0