SZAKDOLGOZAT

Kathi Ferenc

Debrecen

2009

Debreceni Egyetem
Informatikai Kar

Hash függvények

Témavezető:
Prof. Dr. Pethő Attila
Egyetemi tanár, Dékán, Tanszékvezető

Készítette:
Kathi Ferenc
Mérnök Informatikus (Bsc.)

Debrecen
2009

Tartalomjegyzék

Bevezetés 3

1. Kriptográfiai hash függvények 4

   1.1. Definíció 4

   1.2. Hash függvények csoportosítása 4

      1.2.1. Csoportosítás működésük elve alapján 4

      1.2.2. Alapvető tulajdonságok és definíciók 5

   1.3. Alternatív terminológia 6

   1.4. Lavina hatás 6

2. Hash függvény konstruálása 8

   2.1. Általános modell 8

   2.2. Merkle-Damgård konstrukció 9

   2.3. Bitkitöltési formák 10

3. MDC konstruálása 11

   3.1. Blokk kódolón alapuló előállítás 11

      3.1.1. Davies-Meyer 13

      3.1.2. Matyas-Meyer-Oseas 14

      3.1.3. Miyaguchi-Preneel 15

      3.1.4. MDC-2 16

      3.1.5. MDC-4 17

      3.1.6. Hirose 18

4. MAC konstruálása 19

   4.1. CBC blokk kódolóból 19

   4.2. CMAC 20

   4.3. HMAC (MDC segítségével) 22

5. A kriptográfiai hash függvények néhány változata 24

   5.1. MD család 24

      5.1.1. MD2 24

      5.1.2. MD4 24

      5.1.3. MD5 25

      5.1.4. MD6 28

   5.2. SHA 29

5.2.1. SHA-0, SHA-1 29
5.2.2. SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512) 31
5.2.3. SHA-3 34
5.3. RIPEMD 35
5.4. WHIRLPOOL 38
5.5. A bemutatott hash függvények példa lenyomatai 40
6. Támadások a hash függvények ellen 41
6.1. Brute Force 41
6.2. Előre legenerálás 42
6.3. Time-Memory Trade-Off módszerek 42
6.3.1. Szivárvány táblák 43
7. Legfontosabb alkalmazási területek 44
7.1. Digitális aláírás 44
7.2. Jelszavak 44
7.3. Üzenet integritás 45
Irodalomjegyzék 46

Bevezetés

A hash függvények, az 1970-es évekbeli megjelenésük óta fontos szerepet töltenek be a kriptográfiában. Nevezik őket üzenet összefoglaló vagy egyirányú titkosító algoritmusoknak is. Nagyon hasznos építőelemei a különféle biztonsági problémák megoldásának. Ahogy minket azonosít az ujjlenyomatunk, vagy a DNS-ünk, az adatok hash értéke is egy, csak rá jellemző bitsorozat, jó hash függvény estében. Ezért a hash kódokat gyakran nevezik digitális ujjlenyomatnak is.

A hash függvény a bementére érkező adatból tehát egy fix értéket állít elő, ami csak rendkívül kis valószínűséggel nem lesz egyedi. A bemenet mérete lehet akár hatalmas is (pl.: 2^{64} bit nagyságrendű), a kimenet ilyenkor hozzá képest jelentéktelen hosszúságú (pl.: 512 bit). Viszont a kisméretű bemenetekből (néhány bit) is elő tudja állítani az előbbi fix méretű kimenetet. Az előbbi tulajdonságaik egyes alkalmazások során előnyt, míg más helyen hátrányt jelentenek.

A szakdolgozatban bemutatom a hash függvények tulajdonságait, főbb fajtáikat, megalkotásuk módszereit, néhány ismertebb és gyakrabban használt függvény működését, a hash függvények ellen bevethető támadásokat, és a legfontosabb alkalmazási területeket.

1. fejezet

Kriptográfiai hash függvények [1], [10]

1.1 Definíció

A hash függvény egy üzenetet dolgoz fel és előállít belőle egy értéket, egy hash kódot. D tartomány és R értékkészlet esetén h: D→R, |D|>|R| több az egyhez hozzárendelés. Ezért elkerülhetetlen hogy ütközések történjenek. Ez azt jelenti, hogy két különböző bemenethez azonos kimenet társul. Az ütközések valószínűsége azonban lecsökkenthető, ha bizonyos megszorításokat alkalmazunk.

H függvény hash függvény, ha minimálisan rendelkezik a két következő tulajdonsággal:

1. tömörítés: egy tetszőleges, véges hosszúságú x bemenethez egy rögzített, n hosszúságú kimenetet, h(x) rendel hozzá.
2. könnyű kiszámíthatóság: adott h és adott x bemenet esetén h(x) értékét könnyű kiszámítani.

1.2 Hash függvények csoportosítása:

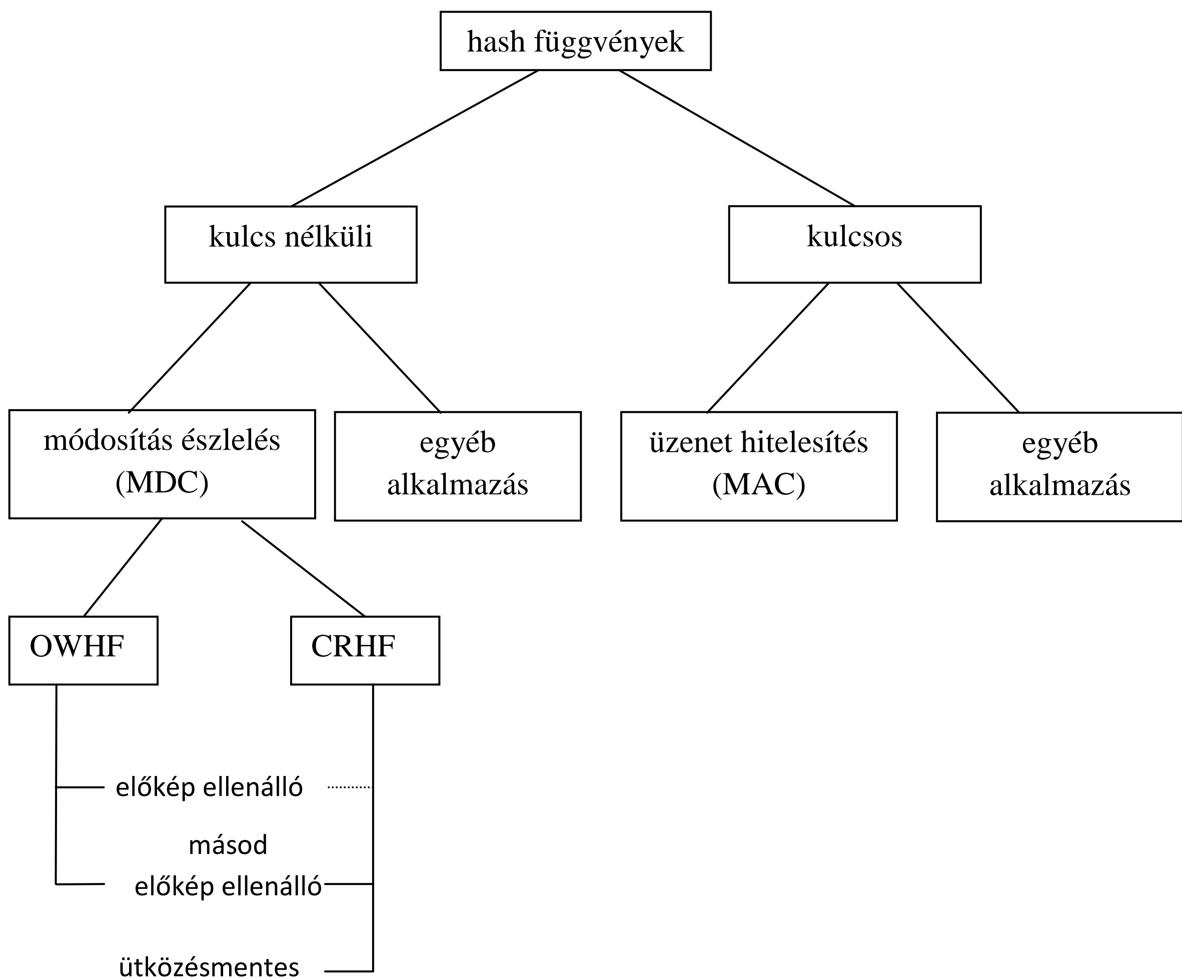
1.2.1 Csoportosítás működésük elve alapján:

a. MDC (modification detection code)
   Módosítást észlelő kód: Célja hogy az üzenetre jellemző képet vagy hash értéket nyújtson. A kulcsnélküli hash fv.-ek közé tartozik.
   Csoportosíthatóak:
   i. egyirányú hash függvény (OWHF): Nehéz találni egy olyan üzenetet melynek hash értéke megegyezik egy előre meghatározott hash értékkel.
   ii. ütközésmentes hash függvény (CRHF): Nehéz találni két olyan bemenetet melyek hash értéke megegyezik.
b. MAC(message authentication code)
   Üzenet hitelesítési kód: Célja hogy bármilyen egyéb mechanizmus nélkül biztosítsa az üzenet forrásának hitelességét és az üzenet sértetlenségét. Két

elkülöníthető paramétere van: az üzenet és a titkos kulcs. A kulcsos hash fv.-ek közé tartozik.

1.2.2 Alapvető tulajdonságok és definíciók

a. preimage resistance (előkép ellenálló): Minden adott kimenethez „nehéz” találni bemenetet melynek hash értéke éppen az adott kimenet.
b. 2nd-preimage resistance (másod előkép ellenálló): Nehéz találni olyan bemenetet amelynek ugyan az lesz a kimenete mint egy adott bemenet, azaz adott x esetén megtalálni $x' \neq x$ hogy $h(x)=h(x')$
c. collison resistance (ütközéssel szemben ellenálló): Nehéz találni két olyan különböző bemenetet (x, x'), amelyek hash értéke megegyezik, azaz $h(x)=h(x')$



1. ábra: A kriptográfiai hash függvények egyszerűsített osztályozása

A fentiekben szerepeltek a könnyű és a nehéz szavak. Könnyűnek nevezzük a probléma megoldását, ha az polinomiális időn belül keresztülvihető, azaz valamennyi műveletet elvégezve, valamennyi idő elteltével lefut az algoritmus. Nehéz, más néven számítás útján keresztülvihetetlen, (computationally infeasible) egy algoritmus, ha lefutásához rendkívül sok idő vagy tárkapacitás szükséges.

1.3 Alternatív terminológia

A korábban szereplő fogalmak a különféle szakirodalmakban változatos formában fordulhatnak elő. Az alábbi megfeleltetések érvényesek:

1. előkép ellenálló = egyirányú (preimage resistance = one way)
2. másod előkép ellenálló = gyengén ütközésmentes (2nd-preimage resistance = weak collision resistance)
3. ütközésmentes = erősen ütközésmentes (collision resistance = strong collision resistance)

1.4 Lavina hatás [9]

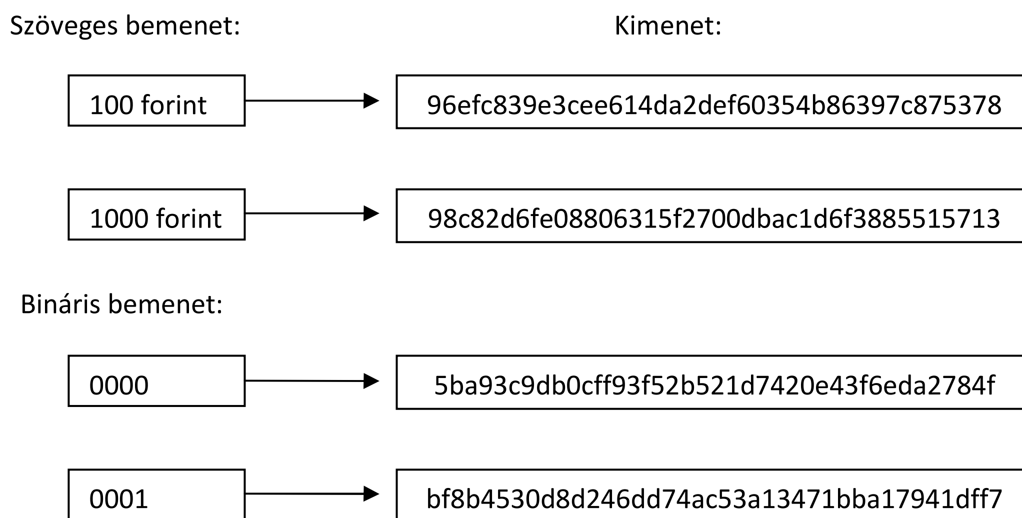
A lavina hatás a blokk kódolók és a kriptográfiai hash függvények egy kívánatos tulajdonsága. A bemenet kismértékű megváltoztatása esetén a kimenet jelentősen megváltozik. A változtatás lehet akár a bemenet egy bit értékének módosítása is, ilyenkor is jelentős mértékben kell megváltozniuk a kimeneti bitek értékeinek. A kifejezést több mint 30 évvel ezelőtt Horst Feistel használta először egy amerikai tudományos magazin hasábjain. Ha a hash függvény nem mutat erőteljes lavina hatást, gyengén véletlenszerűsít, akkor nem tekinthető erősnek. A kimenet alapján megsejthető a bemenet.

A lavina effektus éppen ezért egyike a legfontosabb tervezési szempontoknak. Hogy teljesítsék, a hash függvények nagy adatblokkokon végzik műveleteiket. Minden kis változás gyorsan elterjed az algoritmus iterációiban, így a kimenet minden bitjének függeni kell a bemenet összes bitjétől, mielőtt véget ér az algoritmus.

Az 1985-os CRYPTO konferencián A. F. Webster és Stafford E. Tavares bevezetett 2 precízebb fogalmat a lavina hatásra. Ezeket a DES kódoló algoritmus helyettesítő dobozaira (S-box) vezették be, de értelmezhetőek a hash függvényekre is:

1. szigorú lavina kritérium (strict avalanche criterion (SAC)): Bármelyik kimeneti bit értéke ½ valószínűséggel változik meg ha, egy bemeneti bit értéke megváltozik.
2. bit függetlenségi kritérium bit independence criterion (BIC): Legyen a kimenet két bitje j és k. Ezek értékének egymástól függetlenül kell megváltozni ha, egy bemeneti bit értéke megváltozik.

A lavina effektus szemléltetésére álljon itt egy ábra: [19]



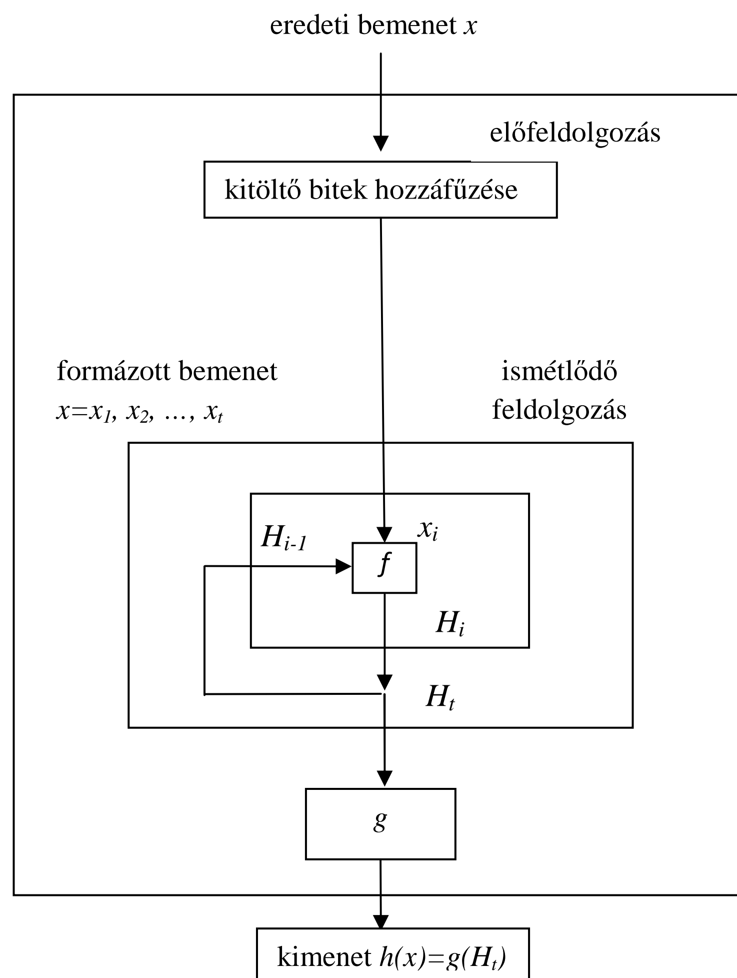
2. ábra: A lavina hatás szemléltetése az SHA-1 függvény használatával

2. fejezet

Hash függvény konstruálása [1]

2.1 Általános modell

A legtöbb kulcs nélküli hash függvény működése egy ismétlődő folyamaton alapul, amely során a tetszőleges eredeti bemenetet egymást követő rögzített méretű blokkokra bontják fel. Ezt szemlélteti a következő ábra:



3. ábra: Hash függvény általános modellje

Első lépésként a bemenetet r hosszúságú blokkokra osztjuk (x_i). Ehhez nagy valószínűséggel szükség lesz bitkitöltésre, úgy hogy a teljes hosszúság a blokkhosszúság többszöröse legyen. Minden egyes x_i blokk a bemenete lesz egy belső f hash függvénynek, ami a h függvény

tömörítő függvénye. f kimenete egy n hosszú köztes érték, bemenete pedig az előző köztes érték illetve a soron következő x_i blokk.

A folyamat a következőképpen írható fel:

$$H_0=IV; \qquad H_i=f(H_{i-1},x_i), \quad 1\leq i\leq t; \qquad h(x)=g(H_t)$$

- H_0: előre definiált kezdeti érték (*initializing value IV*)
- H_{i-1}: n bit hosszú változó az i-1 és az i szakasz között
- g: opcionális transzformáció, gyakran egy önazonosság $g(H_t)= H_t$.

Bármilyen ütközéssel szemben ellenálló tömörítő függvény kiterjeszthető ütközésmentes hash függvénnyé. Ez hatékonyan megtehető a Merkle-Damgard konstrukció segítségével. Ezzel a hash függvény keresésének problémája redukálódik tömörítő függvény keresésére.

2.2 Merkle-Damgård konstrukció [11]

Bemenet: f tömörítő függvény. Kimenet: h hash függvény. Mindkettő ütközéssel szemben ellenálló.

Lépések:

- Tegyük fel hogy f az $(n + r)$ bites bemenetből n bites kimenetet állít elő. (pl.: n=128, r=512).
- A b bithosszúságú x bemenetet $x_1x_2\ldots x_t$ blokkokra daraboljuk, melyek r hosszúságúak. Az utolsó blokkot szükség szerint 0 bitekkel feltöltjük.
- Definiálunk egy extra utolsó blokkot x_{t+1} –et amely tartalmazza b jobbra zárt bináris reprezentációját. Feltételezzük, hogy ($b<2^r$). b az eredeti üzenet hossza.
- Jelentse 0^j j db 0-ból álló bitmintát, x hash értéke pedig:

$$h(x)=H_{t+1}=f(H_t||x_{t+1})$$

$$H_0=0^n; \qquad H_i=f(H_{i-1}||x_i), \quad 1\leq i\leq t+1$$

Merkle és Damgård, hogy bizonyítsák a konstrukció biztonságosságát, javasolták hogy a 3. lépésben az üzenetet az eredeti üzenet hosszának kódolt formájával kell kitölteni. Ezt hossz kiterjesztésnek (length padding) vagy Merkle-Damgård megerősítésnek (MD strengthening) nevezzük.

2.3 Bitkitöltési formák

Az előbbi blokk alapú hash-elési módszereknél általában szükség van arra, hogy extra biteket fűzzünk a bemenethez, ezáltal felduzzasztva azt a blokkméret egész számú többszörösére. Ezt többféleképpen is megtehetjük:

- Az üzenet végéhez a lehető legkevesebb számú 0 bitet csatolunk, úgy hogy az így kapott üzenet a blokkméret többszöröse legyen.
  pl: blokkméret 8 byte, a kitöltött üzenet: hashbeme net00000
- Az üzenet végéhez egy darab 1-es bitet és a lehető legkevesebb számú 0 bitet csatolunk úgy hogy az így kapott üzenet a blokkméret többszöröse legyen.
  pl: blokkméret 8 byte, a kitöltött üzenet: hashbeme net10000
- Az üzenet végéhez extra blokkot is csatolhatunk, így csökkentve a támadások sikerességét (MD megerősítés)
- Az előzőek kombináció is lehetségesek

Inicializációs értékek

A hash érték generálása és későbbi ellenőrzése esetén is ugyanazt a kezdeti inicializációs értéket kell használni, legyen az fix vagy véletlenszerűen generált, esetleg a bemenet függvényében kiszámolt érték. Ha előzetesen nem ismert az ellenőrző személy számára, akkor az üzenettel együtt el kell küldeni neki.

3. fejezet

MDC konstruálása [1], [12]

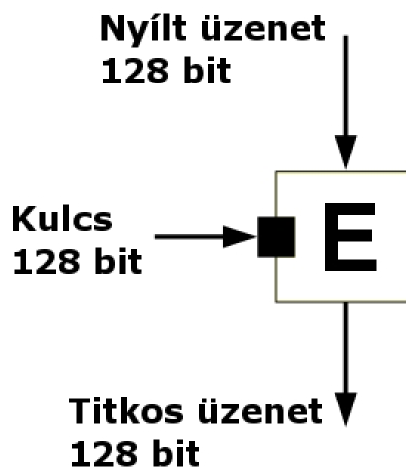
A kulcs nélküli hash függvények alcsoportját alkotó módosítást detektáló kódokat kategorizálhatjuk a belső tömörítő függvényük működése alapján. Ebből a szempontból nézve három fő kategória létezik:

- blokk kódolón alapuló hash függvények
- moduláris aritmetikán alapuló fügvények
- testre szabott, eredetileg is hashelésre tervezett függvények

3.1 Blokk kódolón alapuló előállítás

Ennek a módszernek nagy előnye, hogy a már szoftveresen vagy hardveresen megvalósított blokk kódolóból kis többlet számítási költséggel készíthető hash függvény.

Először nézzük meg hogy egy blokk kódoló hogyan épül fel:



4. ábra: Egy tipikus blokk kódoló

A blokk kódoló bemenetként megkap két fix méretű adatok, (az egyirányú tömörítő függvényhez hasonlóan) nyílt szöveget és kulcsot, kimenete pedig a nyílt szöveggel azonos méretű titkosított üzenet. A blokk kódoló azonban csak részben egyirányú. Adott nyílt és tikos üzenetekből nem állapítható meg a titkosító kulcs, de a blokk kódoló dekódoló függvényével adott titkos üzenetből a kulcsot ismerve könnyen megfejthető a nyílt üzenet.

Ezért ahhoz hogy a blokk kódolóból egyirányú tömörítő függvényt állítsunk elő, extra műveletek szükségesek.

Blokk kódolóból hash függvény előállítására többfajta módszer is létezik:

- Egyszeres blokkhosszúságú tömörítő függvények: kimenetük ugyanannyi bites lesz amennyit az alapul szolgáló blokk kódoló feldolgoz.
  - Davies-Meyer
  - Matyas-Meyer-Oseas
  - Miyaguchi-Preneel
- Kétszeres blokkhosszúságú tömörítő függvények: kimenetük a blokk kódoló által feldolgozott bitek számának kétszerese.
  - MDC-2
  - MDC-4
  - Hirose

A blokk kódoló használatának hátránya hogy általában lassabb, mint a speciálisan hash-elésre tervezett egyirányú tömörítő függvények. Ennek oka, a kulcsütemezés az üzenet blokkjai számára. A gyakorlatban azonban a sebessége elfogadható. Bizonyított, hogy lehetetlen olyan egyirányú tömörítő függvényt konstruálni, amely egy fix kulccsal csak egyszer hívja meg a blokk kódolót.

Blokk kódoló használatának előnye hogy olyan helyeken ahol titkosítást és hash függvényt is használni kell (pl.: smart kártyák, kisméretű beágyazott rendszerek) kód tárhelyet lehet megtakarítani.

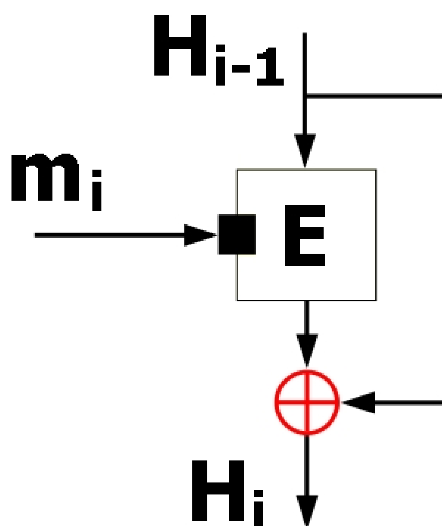
Különböző tömörítő függvényeken alapuló hash függvények hatékonyságát jellemezhetjük a hash rátával. A ráta egy arányszám a blokk kódoló műveletek és a kimenet között.

$$R_{h=}\frac{|m_i|}{s \cdot n}$$

Jelentse:

- n a blokk kódoló kimeneti bithosszát,
- m a bemenetet,
- s pedig a blokk kódoló műveletek számát.

3.1.1 Davies-Meyer



5. ábra: Davies-Meyer féle séma

A Davies-Meyer egyszeres blokkhosszúságú egyirányú tömörítő függvény az üzenet blokkjait (m_i) a blokk kódolóba kulcsként küldi, az előző hash érték (H_{i-1}) pedig a nyílt üzenetként kerül feldolgozásra. A kódoló kimenetéül kapott értéket és az előző hash értéket a kizáró vagy művelettel (XOR) összeadjuk és így megkapjuk a következő hash értéket (H_i). Az első körben, előző hash érték hiányában egy inicializációs értéket (H_0) kell használni.

Formulával kifejezve:

$$H_i = E_{m_i}(H_{i-1}) \oplus H_{i-1}$$

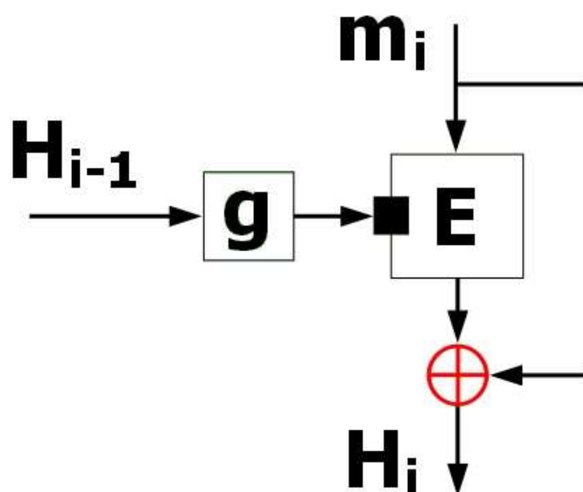
A séma hash rátája: $R_{DM} = \frac{k}{1 \cdot n} = \frac{k}{n}$ k = kulcsméret, azaz $|m_i|$

A blokk kódolóknak lehetnek olyan jellegzetességei, melyek nem jelentenek biztonsági kockázatot titkosítás esetén, azonban ha hash függvény konstruálásához használjuk fel őket, ezek hátrányos tulajdonságokká válnak. Így lehetővé válik a kimenetek befolyásolása a bemenetek manipulálásával. Ilyen tulajdonság pl.: a fix pontok létezése, azaz ha $E_{kulcs}(x) = x$. Ennél a konstrukciónál ha H_{i-1} egy fix pont az m_i kulcsra nézve, ($E_{m_i}(H_{i-1}) = H_{i-1}$) akkor a tömörítő függvény kimenete: $H_i = 0$. A fix pontok segítenek a támadónak előképet találni,

azonban jelenleg nincs olyan hatékony támadás, ami ezen a tulajdonságon alapul. Ez a probléma a véletlenszerű függvényeknél nem jelentkezik A biztonság fokozható a MD megerősítés használatával.

Az MD5 függvény és az SHA függvénycsalád is ez a konstrukciót használja.

3.1.2 Matyas-Meyer-Oseas



6. ábra: A Matyas-Meyer-Oseas féle séma

A Matyas-Meyer-Oseas konstrukció szintén egyszeres blokkhosszúságú, és az előző séma fordítottjának tekinthető. Itt az üzenet egyes blokkjai (m_i) lesznek a nyílt üzenetek. Hogy megkapjuk a következő hash értéket (H_i), az üzenet és a titkos üzenet között XOR műveletet kell végrehajtani. Az előző hash érték lesz a kódolás kulcsa. Ha a kódoló blokk és kulcsmérete különböző, akkor az előző hash értéke egy *g* függvény bemenetére kerül, ami elvégzi a szükséges átalakításokat.

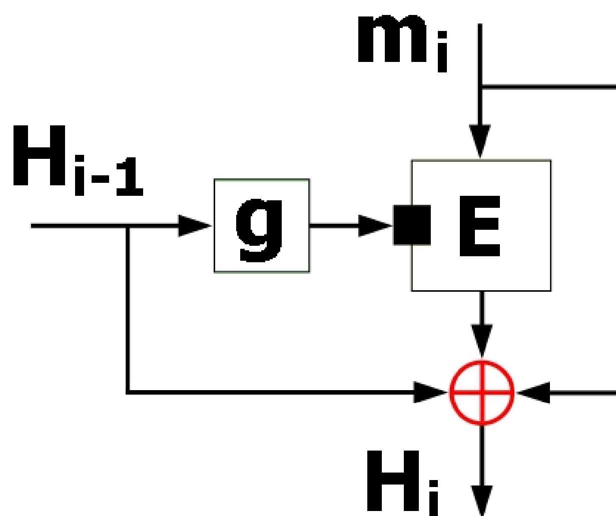
A műveletek formulával kifejezve:

$$H_i = E_{g(H_{i-1})}(m_i) \oplus m_i$$

A konstrukció hash rátája:

$$R_{MMO} = \frac{n}{1 \cdot n} = 1$$

3.1.3 Miyaguchi-Preneel



7. ábra: Miyaguchi-Preneel séma

Szintén egyszeres blokkhosszúságú konstrukció, a Matyas-Meyer-Oseas séma kiterjesztése. Annyiban tér el attól, hogy ahhoz hogy megkapjuk a következő hash értéket, az üzenetblokk és a tikos szöveg összeadása után, az így kapott értékhez hozzáadjuk az előző hash értéket.

Formulával kifejezve:

$$H_i = E_{g(H_{i-1})}(m_i) \oplus H_{i-1} \oplus m_i$$

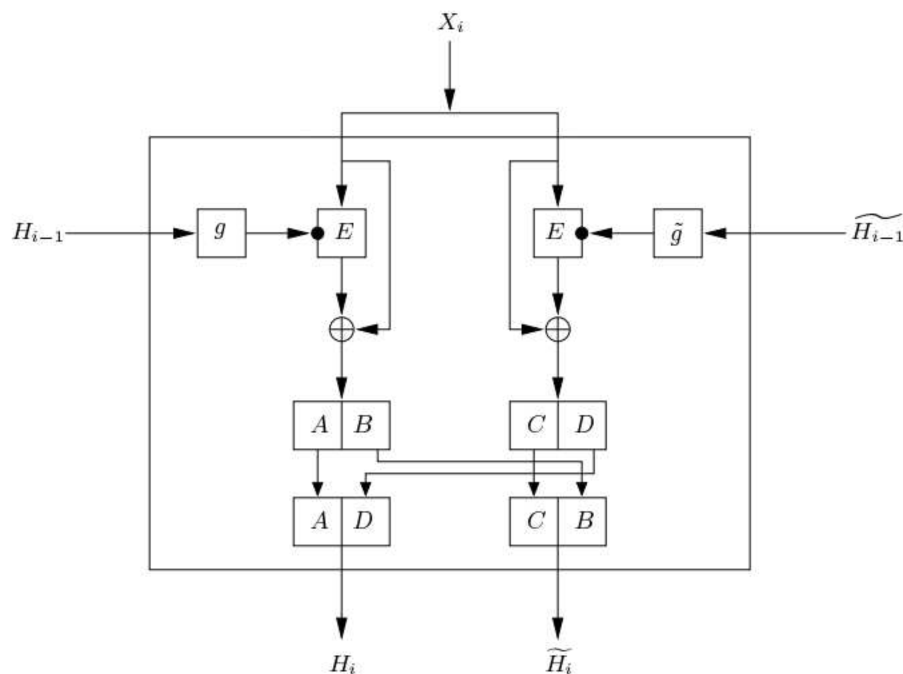
Arányszáma: $R_{MP} = R_{MMO} = \frac{n}{1 \cdot n} = 1$

A konstrukció egyik lehetséges módosítása, hogy az üzenetet kulcsként, az előző hash értéket pedig nyílt üzenetként kapja meg, és így a Davies-Meyer séma kiterjesztésének tekinthető.

Az első konstrukciónál egy adott *m* üzenethez találni olyan *m'* üzenetet melyre teljesül, hogy hash(*m*)=hash(*m'*), 2^k hosszú üzenet esetében $3 \cdot k \times 2^{\frac{n}{2}+1} 2^{n-k+1}$ időn belül lehetséges, a második és harmadik sémánál ez az érték $k \times 2^{\frac{n}{2}+1} 2^{n-k+1}$.

3.1.4 MDC-2

Az MDC-2 hash függvény a bemenet egy blokkjának feldolgozásához két blokk kódolót vesz igénybe. Két darab Matyas-Meyer-Oseas konstrukciót alkalmaz, így lesz a kimenete kétszeres hosszúságú. A blokk kódolója eredetileg a DES (Data Encryption Standard) 64 bites kódoló, 56 bites kulccsal.



8. ábra: MDC-2 függvény

A g és $\tilde{g}$ függvény egy bizonyos szisztéma szerint a 64 bites értékből előállítja az 56 bites kulcsokat a DES számára. A bemenetet, ami a 64 egész számú többszöröse lehet, blokkokra osztjuk.

A műveletek formulával kifejezve:

$$H_0 = IV; \quad k_i = g(H_{i-1}); \quad C_i = E_{k_i}(x_i) \oplus x_i; \quad H_i = C_i^L || \widetilde{C_i^R}$$

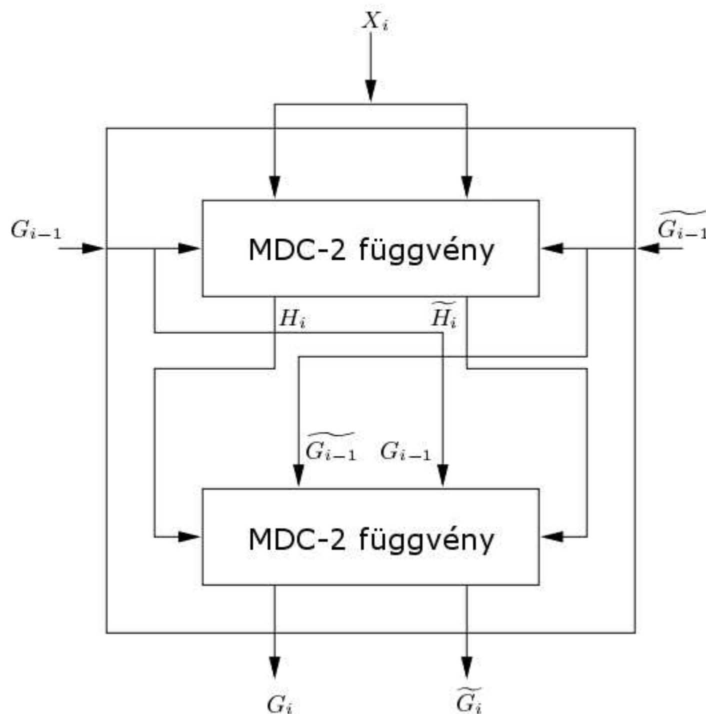
$$\widetilde{H_0} = \widetilde{IV}; \quad \widetilde{k_i} = \tilde{g}(\widetilde{H_{i-1}}); \quad \widetilde{C_i} = E_{\widetilde{k_i}}(x_i) \oplus x_i; \quad \widetilde{H_i} = \widetilde{C_i^L} || C_i^R$$

C_i 32 bites jobb és bal felét jelentse C_i^L és C_i^R, a kimenet: $h(x) = H_t || \widetilde{H_t}$ $\quad 1 \leq i \leq t$

Hash rátája DES használatával: $R_{MDC-2} = \frac{1}{2}$

3.1.5 MDC-4

Az MDC-4 MDC-2 függvényekből épül fel. Egy MDC-4 iteráció két, egymást követő MDC-2 függvény végrehajtásából áll.



9. ábra: MDC-4 függvény

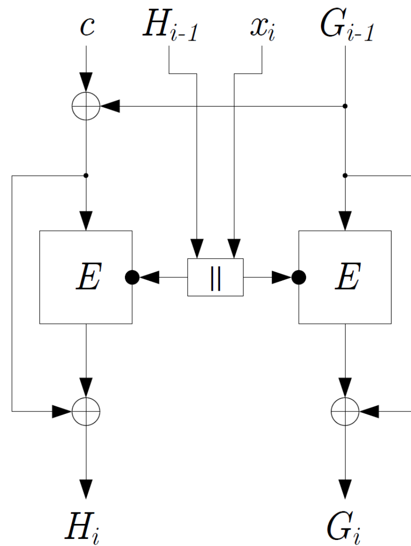
A bemenet, *x* szintén 64 bit többszöröse lehet, a kimenet pedig *x* 128 bites hash értéke.

Működése:

- Az első MDC-2 tömörítő függvény 2 üzenet bemenetére ugyanaz a következő 64 bites üzenetblokk kerül
- Az első MDC-2 tömörítő függvény kulcsai az MDC-4 tömörítésből számítódnak ki.
- A második MDC-2 függvény kulcsai az előző MDC-2 függvény kimeneteiből származnak.
- A második MDC-2 függvény adat bemenetei az előző MDC-4 tömörítés ellentétes oldaláról származnak.
- $h(x) = G_t || \widetilde{G_t}$

Arányszáma: $R_{MDC-4} = \frac{1}{4}$

3.1.6 Hirose



10. ábra: Hirose függvény

Shoichi Hirose tervezte meg 2006-ban. A konstrukció tartalmaz két blokk kódolót és egy permutációt. Blokk kódolóját úgy kell megválasztani, hogy a *k* kulcsméret nagyobb legyen az *n* blokkméretnél. Erre a célra megfelel az AES kódoló 128 bites blokkmérettel és 256 bites kulcsmérettel. Az egyes körökben az üzenet *n-k* hosszú blokkja kerülnek feldolgozásra.

Az algoritmus lépései

- A soron következő üzenetblokk, x_i és az előző belső állapot, H_{i-1} összefűzése. Így megkapjuk a blokk kódolók aktuális kulcsát
- $G_i = E_{K_i}(G_{i-1}) \oplus G_{i-1}$

  $H_i = E_{K_i}\big(p(G_{i-1})\big) \oplus p(G_{i-1})$
- p egy permutáció, $p(x) = x \oplus c \quad\quad c \neq 0$ konstans

Ennél a sémánál is a Davies-Meyer konstrukció jegyei tűnnek fel, azonban a többivel szemben ennek az előnye, hogy mindkét titkosítás ugyanazt a kulcsot használja és ezért a kulcsütemező rész közösen használt.

A végső hash érték: $H_t||G_t$

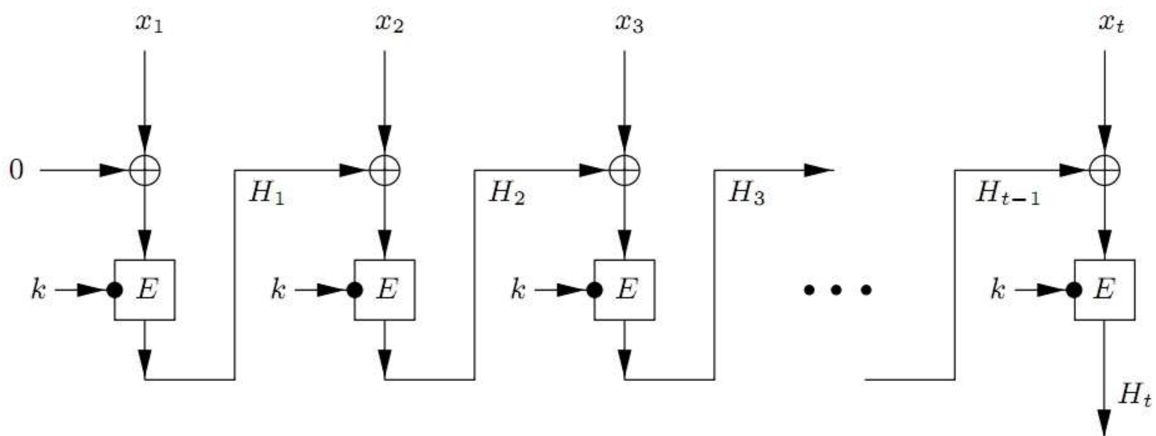
Arányszáma: $R_H = \frac{k-n}{2 \cdot n}$

4. fejezet

MAC konstruálása [2]

4.1 CBC-MAC

A leggyakrabban használt MAC algoritmusok a blokk kódoló láncolás (CBC Cipher Block Chaining) technikáján alapulnak.



11. ábra: MAC konstruálás láncolás technikával

Legyen a blokk kódolók blokkmérete *n*. Az esetlegesen bitkitöltött bemenet *n* bites blokkokra darabolódik szét ($x_1,\dots,x_t$). A *k* jelöli a titkos kulcsot. A belső állapotok és a végleges érték a kiszámítása formálisan:

$$H_1 = E_k(x_1) \quad H_i = E_k(H_{i-1} \oplus x_i) \quad 2 \le i \le t$$

Az inicializációs érték: *IV*=0, a végeredmény pedig az *n* bites H_t érték.

A konstrukció további megerősítéseként a folyamat végére illeszthető egy másik kulccsal (*k'*) történő dekódolás, majd az eredeti kulccsal elvégzett kódolás:

$$H_t' = E_{k'}^{-1}(H_t), \qquad H_t = E_k(H_t')$$

A CBC-MAC csak fix hosszúságú üzenetekre biztonságos, változó hosszúságúakra nem. Ha a támadó ismer két üzenet-MAC párost, (*x,y*) és (*x',y'*), tud alkotni olyan *x''* harmadik üzenetet, melynek MAC értéke szintén *y'*. Ezt a következőképpen teheti meg: *x'* első blokkja és *y* között XOR műveletet hajt végre és az így módosított *x'*-t *x* után fűzi.

Formálisan: $x'' = x||[(x'_1 \oplus y)||x'_2|| \dots ||x'_t]$.

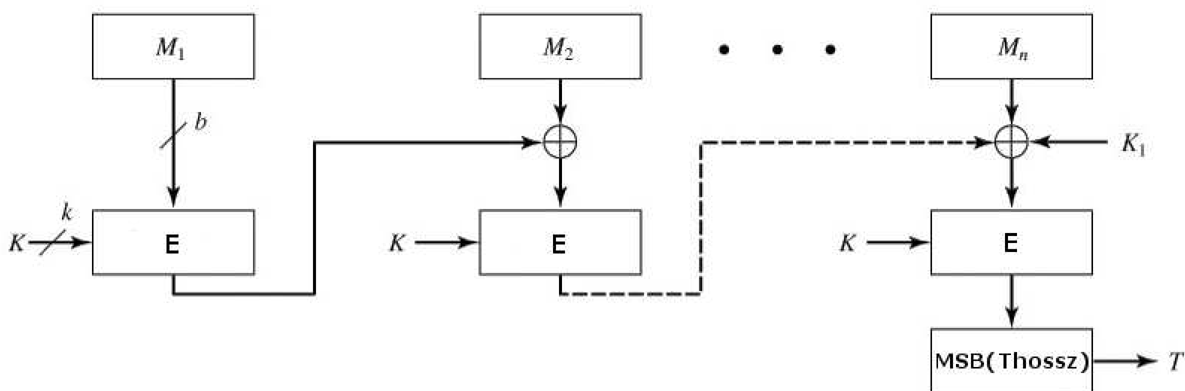
Emiatt született meg a CMAC konstrukció, amely kiküszöböli a problémát.

4.2 CMAC (Cipher-Based MAC)

A CBC-MAC problémája megoldható, ha három kulcsot használunk. Az egyik kulcs legyen *k* hosszú, a másik kettő pedig *n* hosszúságú, *k* a kulcs hossza, *n* pedig a blokk kódoló mérete. A konstrukciót később módosították, úgy hogy a két *n* bites kulcs kiszámítható a tikosító kulcsból, így nem kell az algoritmus számára külön szolgáltatni. A blokk kódoló lehet AES vagy triple DES.

Két eset lehetséges:

- Az üzenet hossza a blokkméret, *b* egész számú, *n* többszöröse. AES esetében *b*=128, a triple DES változatnál *b*=64



12. ábra: CMAC az első esetben

Az üzenetet felosztjuk *n* darab blokkra: $M_1, \ M_2, \dots M_n$

Az algoritmus használ egy *k* bit hosszú *K* kulcsot és egy n bit hosszú K_1 konstans értéket. AES esetében *k* lehet 128, 192, 256, triple DES-nél pedig 112 vagy 168 bit.

A működése formálisan:

$C_1 = E(K, M_1)$

$C_2 = E(K, [M_2 \oplus C_1])$

...

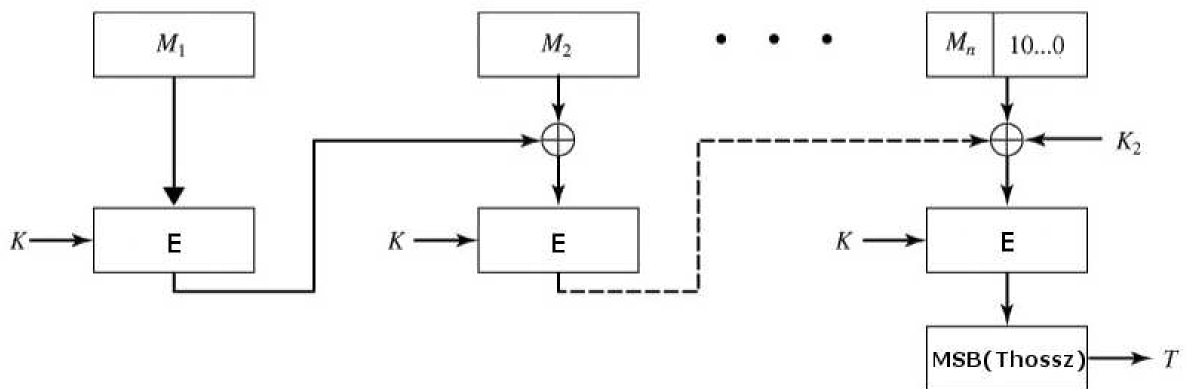
$C_n = E(K, [M_n \oplus C_{n1} \oplus K_1])$

$T = MSB_{Thossz}(C_n)$

Jelölések:

- *T*: MAC kód
- *Thossz*: *T* bithosszúsága
- $MSB_S(X)$: *X* legbaloldalibb *s* darab bitje.

- Ha az üzenet nem a blokk kódoló méretének egész számú többszöröse, akkor az utolsó blokkot feltöltjük egy 1-es bittel és annyi 0-val, hogy az utolsó blokk mérete megegyezzen a blokk kódoló méretével. Az algoritmus működése megegyezik az előző esettel, annyi különbséggel, hogy itt K_1 helyett K_2 kulcsot használunk.



13. ábra: CMAC a második esetben

K_1 és K_2 kiszámítása:

$$L = E(K, 0^n)$$

$$K_1 = L \cdot x$$

$$K_2 = L \cdot x^2 = (L \cdot x) \cdot x$$

Az *L* érték kiszámításához kódolunk egy csupa 0 bitből álló blokkot. K_1-et megkapjuk, ha az *L* értéket 1-el balra shiftelt értéke és egy *b*-től függő konstans érték között XOR műveletet hajtunk végre. A K_2 hasonlóan számolható ki.

A konstans értékek: AES esetében $x^{128} + x^7 + x^2 + x + 1$, triple DES-nél $x^{64} + x^4 + x^3 + x + 1$

Itt a · művelet a GF(2^n) véges test elemei közötti szorzást jelenti, *x* és x^2 pedig első, ill. másodrendű polinomok. Binárisan ábrázolva *x* n-2 darab 0-ból és az 1,0 bitekből áll, x^2 pedig n-3 darab 0-ból és az 1,0,0 bitekből áll.

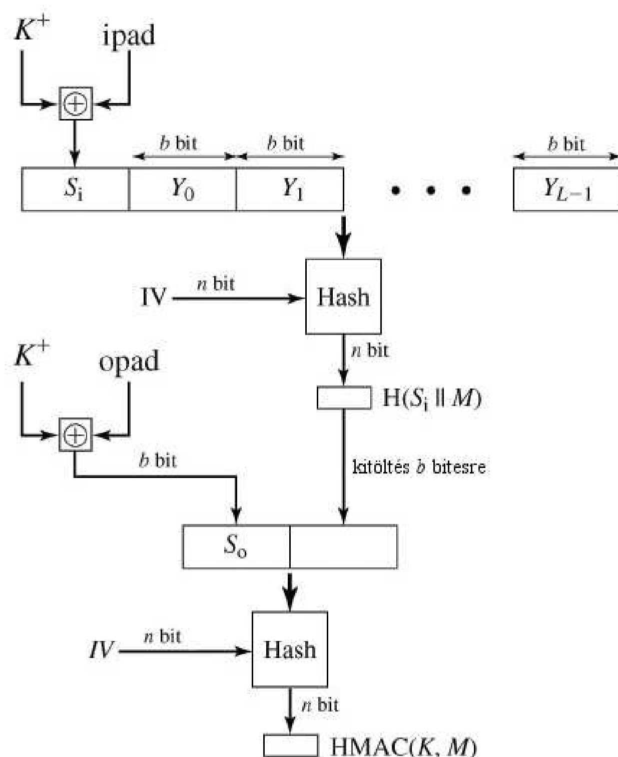
4.3 HMAC (MDC segítségével)

A HMAC hash függvényen alapuló MAC algoritmust jelent. Főleg az utóbbi években kerültek a figyelem középpontjába. Ennek oka egyrészt, hogy az olyan hash függvények, mint pl.: az MD5 és az SHA-1 általában gyorsabbak, mint a szimmetrikus blokk kódolók, másrészt eljáráskönyvtáruk is széles körben elérhető.

HMAC tervezésénél követendő célok:

- Ingyenesen és széles körben elérhető jól megtervezett hash függvényt kell használni.
- A beágyazott hash függvény szükség esetén könnyen cserélhető legyen egy gyorsabb vagy biztonságosabb változatra.
- Jelentősen nem csökkenhet a hash függvény eredeti teljesítménye.
- Kulcsok kezelése egyszerű legyen.

Ezek a szempontok akkor valósulnak meg, ha a HMAC úgy tekint a beágyazott hash függvényre, mint egy fekete dobozra. A hash függvény egy modul a rendszeren belül és ez a modul cserélhető, ha szükséges.



14. ábra: HMAC struktúra

Az algoritmusban szereplő elemek:

- H: beágyazott hash függvény
- IV: a H kezdeti értéke
- M: A HMAC üzenet bemenete
- Y_i : M i-edik blokkja $0 \leq i \leq (L-1)$
- L: az üzenetblokkok száma
- b: a blokk bithosszúsága
- n: H által produkált érték hossza
- K: titkos kulcs
- K^+: K nulla bitekkel balról kiterjesztett módosítása, hossza b
- *ipad*: $00110110_B = 36_H$ b/8–szer ismétlődve
- *opad*: $01011100_B = 5C_H$ b/8–szer ismétlődve

$$HMAC(K, M) = H[(K^+ \oplus opad)||H[(K^+ \oplus ipad) \,||\, M]]]$$

Az *ipad* és *opad* értékekkel végzett XOR műveletek átbillentik K bitjeinek felét. Ezért azzal hogy S_i és S_o áthalad a hash algoritmus tömörítő függvényén, előállítódik két álvéletlen, K-ból számított kulcs.

A HMAC algoritmusnak nagyjából annyi idő alatt kell lefutni, mint a beágyazott hash függvénynek, ha az üzenet hosszú.

A HMAC egy hatékonyabb verziója, amikor két értéket előre kiszámítunk:

$$f(IV, (K^+ \oplus ipad))$$

$$f(IV, (K^+ \oplus opad))$$

f a hash függvény tömörítő függvénye. Ezeket az értékeket elég a kiinduló szakaszban kiszámítani, illetve amikor megváltozik a kulcs. Ennél a megvalósításnál csak egy plusz tömörítő függvény adódik a folyamathoz, és akkor van különösen előnye, ha az üzenetek rövidek.

A HMAC biztonsága a beágyazott hash függvény biztonságától függ.

5. fejezet

A kriptográfiai hash függvények néhány változata

5.1 MD család

5.1.1 MD2 [13]

Az MD2 hash függvényt Ronald Rivest alkotta meg 1989-ben. 1992-ben specifikálták az RFC 1319 számú dokumentumában. Az algoritmust 8 bites processzorra optimalizálták.

Az algoritmus az üzenetből 128 bites lenyomatot készít.

Működése:

- az üzenet kiterjesztése úgy hojgy a bájtokban mért hossza a 16 egész számú többszöröse legyen. A kitöltés minimum 1, maximum 16 byte-al történik.
- az üzenet 16 byte-os ellenőrző összege a kitöltött üzenethez kerül hozzácsatolásra.
- a bemenet minden 16 byte-os részének feldolgozásakor egy 256 byte-os S-box szerint permutációt hajt végre a 48 byte-os átmeneti tárolón, X-en
- a kimenet: $X[0\ldots15]$

Az MD2 tömörítő függvényében már 1997-találtak ütközéseket, 2004-ben pedig bebizonyosodott az algoritmus gyengesége az előkép támadással szemben, az algoritmus nem egyirányú. Az MD2 tehát nem biztonságos, ezért több biztonsági protokollból is kitiltották.

5.1.2 MD4 [14]

Az MD4-et szintén Rivest tervezte 1990-ben. Az MD4 az előfutára volt a későbbi, hozzá hasonló hash függvényeknek, mint például MD5, SHA, RIPEMD függvények.

Az 512 bites üzenetblokkokat 32 bites szavanként dolgozza fel, az elkészült lenyomat pedig 128 bites. Az algoritmus az MD megerősítésnek megfelelően, bitkiterjesztéssel indul, majd az üzenet minden blokkját feldolgozza a mögöttes tömörítő függvény. A függvény a 4 darab 32 bites változó értékét, az első üzenetblokk hashelése előtt egy fix értékkel, majd az aktuális hash értékkel tölti fel. A tömörítő függvények 3 körre vannak osztva, egyenként 16 lépéssel.

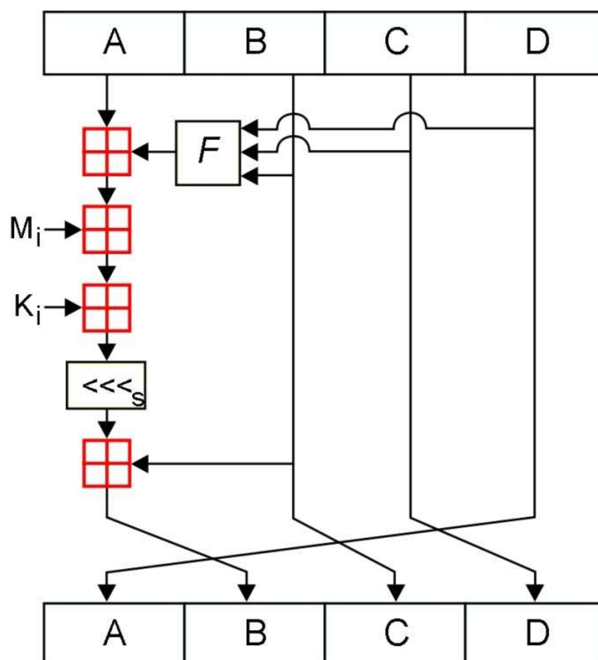
Biztonságossága: Az MD4 gyengeségeit már 1991-ben felfedezték, és 1996-ban találtak először ütközést. 2004-ben tovább finomult az ütközéses támadása, így napjainkban már nagyon könnyű támadást intézni a függvény ellen.

Ennek ellenére ma is használatban van több helyen, pl.: fájlcserélő hálózatoknál alkalmazzák.

5.1.3 MD5 [15]

Az MD5 függvényt is Ronald Linn Rivest tervezte 1991-ben. Célja az MD4 leváltása egy biztonságosabb függvény által. Megalkotása óta többször is előjöttek az algoritmus biztonsági problémái. Már 1993-ban találtak két különböző inicializációs vektort, amelyek ugyanazt a lenyomatot szolgáltatták. 1996-ban találtak először ütközést, és ezután az SHA-1 függvényt kezdték el ajánlani a kriptográfiával foglalkozó szakemberek, mint biztonságos hash függvényt. Az ütközés kereső algoritmusok olyan szinten fejlődtek, hogy 2006-ban már percek kérdése volt ütközést találni. Ennek ellenére az MD5 biztonságosnak tekinthető és napjainkban is széles körűen elterjedt a használata.

Az algoritmus működése:



15. ábra: MD5 művelet

Az üzenetből készülő lenyomat az előzőekhez hasonlóan itt is 128 bites lesz. A bemenetet 512 bites darabokra osztjuk, majd bitkiterjesztést alkalmazunk: egy 1-es bitet annyi 0-s követ, hogy az így kapott üzenet hossza 512-64 bites legyen. A fennmaradó helyre az üzenet hosszát jelképező 64 bites érték kerül.

Az algoritmus belső állapota 128 bites, 4 db 32 bites szóból áll: *A, B, C* és *D*. Ezek kezdeti értékei rögzített konstansok. Az üzenet blokkjainak feldolgozása 4 hasonló szakaszból áll, ezeket köröknek nevezzük. Egy körön belül a *15.ábrán* látható művelet 16-szor hajtódik végre. A művelet tartalmaz egy *F* nem lineáris függvényt, modulo összeadást $\boxplus$, és balra forgatás műveletet:$\lll_S$. A forgatás függvény olyan, hogy a bitsorozatot *S* bittel eltolja balra és a baloldalon kilépő biteket a jobb oldalon visszahelyezi.

A négy kör számára négy különböző *F* létezik:

- $F(X,Y,Z) = (X \land Y) \lor (\neg X \land Z)$
- $G(X,Y,Z) = (X \land Z) \lor (Y \land \neg Z)$
- $H(X,Y,Z) = X \oplus Y \oplus Z$
- $I(X,Y,Z) = Y \oplus (X \lor \neg Z)$

Az algoritmus működése pszeudo kóddal felírva:

```
A számítások során használt segédváltozók
var int[64] r, k

Az r tömb feltöltése az egyes körökben szükséges siftelések
számával
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,
7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,
5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,
4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,
6, 10, 15, 21}

A k tömb feltöltése a szinusz függvény felhasználásával. pow a
hatványfüggvényt jelöli.
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × (2 pow 32))

inicializációs vektorok:
var int h0 := 0x67452301
```

```
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

bitkitöltés műveletek:
"1" bit hozzáfűzése az üzenethez.
"0" bitek hozzáfűzése, amíg az üzenet hossza ≡ 448 (mod 512)
az eredeti  üzenet bithosszúságának hozzáfűzése

az üzenet minden 512 bites darabjára a következő műveletek
végrehajtása:
w[i]: az 512 bites rész egy 32 bites részlete, i≤0≤15
for each 512-bit chunk of message

 Az aktuális üzenetdarab számára a kezdeti értékek beállítása
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3

    A 64 művelet végrehajtása
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then
            f := (b and c) or ((not b) and d)
            g := i
        else if 16 ≤ i ≤ 31
            f := (d and b) or ((not d) and c)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47
            f := b xor c xor d
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63
            f := c xor (b or (not d))
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
        b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
        a := temp


    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d

var int digest := h0 append h1 append h2 append h3
```

5.1.4 MD6 [5], [6]

Az MD5 hash függvény 1991-es tervezése óta nagyot változott az informatika világa. Egy mai hash függvénynek többet kell nyújtania, hogy kielégítse a kor elvárásait. Rivest professor ezért a 2008-as CRYPTO konferencián bemutatta az utódot, az MD6 hash függvényt, amit az USA-beli Massachusetts Institute of Technology (MIT) műszaki egyetemen alkotott meg az ő általa vezetett csoport.

Az MD6 függvényt benevezték az NIST (National Institute of Standards and Technology) által 2007 végén kiírt pályázatára, melynek célja hogy találjanak egy minden igényt kielégítő modern hash függvényt. A nyertes függvény neve SHA-3 lesz.

2009. július 1.-én Rivest professor úr megírta a NIST-nek hogy az MD6 még nem áll készen, hogy az SHA-3 címre pályázzon, hivatalosan azonban a függvény nem vonult vissza. Az MD6 nem jutott be az SHA-3 bajnokság második körébe.

Az MD6 főbb jellegzetességei:

- A bemenet mérete maximálisan 2^{64}-1 bit lehet.
- Az elkészült lenyomat 1-512 bites lehet.
- A kívánt lenyomatmérettől függetlenül 1 tömörítő függvényt használ. A nagyobb méretet a körök számának növelésével éri el.
- Működése fa szerkezetű, az adatok egy négyes fa (a csomópontoknak maximum négy gyermekük van) levélelemei, a hash érték pedig a gyökérelemnél számítódik ki.
- A fa felépítés miatt nagymértékben párhuzamosítható.
- A köztes értékek (láncolt változók) 1024 bites értékként kerülnek a fa felsőbb szintjére, a végső kimenet pedig a gyökérnél megjelenő 1024 bites érték csonkítása a kívánt lenyomatméretre.
- Lehetőség van automatikusan MAC kód kiszámítására, amennyiben használjuk a tömörítő függvény kiegészítő kulcs bemenetét, ami 512 bit hosszú.
- Az algoritmus csak az AND, XOR és SHIFT műveleteket és az ezekből könnyen származtatható műveleteket használja, ezért a függvény egyszerű.

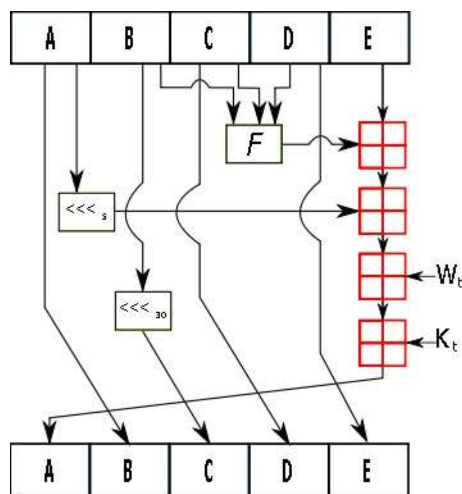
Habár az MD6 már sose lesz SHA-3, de a dokumentáció és a forráskódok ingyenesen letölthetők az MD6 project honlapjáról.

5.2 SHA [4], [16]

Az SHA (Secure Hash Algorithm) függvényeket az NSA (National Security Agency) tervezte, és a NIST szabványosította. Három változatuk létezik, amelyek felépítése különböző. Ezek: SHA-0, SHA-1, és SHA-2. Az utóbbinak több alfajtája is létezik, amik egymástól az elkészült lenyomat méretében különböznek.

5.2.1 SHA-0 és SHA-1

A legelső változatot (SHA-0) 1993-ban szabványosították, azonban később biztonsági okokra hivatkozva visszavonták, és helyét 1995-ben átvette az SHA-1. Ez elődjétől csak egyetlen bitelforgatásban különbözik. Az SHA-1 az üzenetből 160 bites lenyomatot állít elő.



16. ábra: SHA-1 művelet

Az algoritmus a már megszokott bitkiterjesztés művelettel kezdődik. Az üzenet végéhez először egy 1-es bitet, majd annyi bitet csatolunk, hogy az így kapott üzenet hossza: $l + 1 + k \equiv 448 \, mod \, 512$, l az eredeti üzenet hossza, k a 0 bitek száma. Ezután következik l binárisan leírt értékének csatolása. Az így kapott üzenet hossza 512 többszöröse lesz.

A működés pszeudó kóddal leírva:

```
Változók kezdőértékének beállítása
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
```

Bitkitöltés műveletek:
"1" bit hozzáfűzése az üzenethez.
"0" bitek hozzáfűzése, amíg az üzenet hossza ≡ 448 (mod 512)
az eredeti üzenet bithosszúságának hozzáfűzése

Az üzenet minden 512 bites darabjára a következő műveletek végrehajtása:
w[i]: az 512 bites rész egy 32 bites részlete, i≤0≤15

```
for each 512-bit chunk of message

   A 16 db 32 bites szó kiterjesztése 80 bitesre
   for i from 16 to 79
     w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16])
     leftrotate 1

   a = h0
   b = h1
   c = h2
   d = h3
   e = h4

   for i from 0 to 79
       if 0 ≤ i ≤ 19 then
         f = (b and c) or ((not b) and d)
         k = 0x5A827999
       else if 20 ≤ i ≤ 39
         f = b xor c xor d
         k = 0x6ED9EBA1
       else if 40 ≤ i ≤ 59
         f = (b and c) or (b and d) or (c and d)
         k = 0x8F1BBCDC
       else if 60 ≤ i ≤ 79
         f = b xor c xor d
         k = 0xCA62C1D6

       temp = (a leftrotate 5) + f + e + k + w[i]
   e = d
   d = c
   c = b leftrotate 30
   b = a
   a = temp

   h0 = h0 + a
   h1 = h1 + b
   h2 = h2 + c
   h3 = h3 + d
   h4 = h4 + e

digest = hash = h0 append h1 append h2 append h3 append h4
```

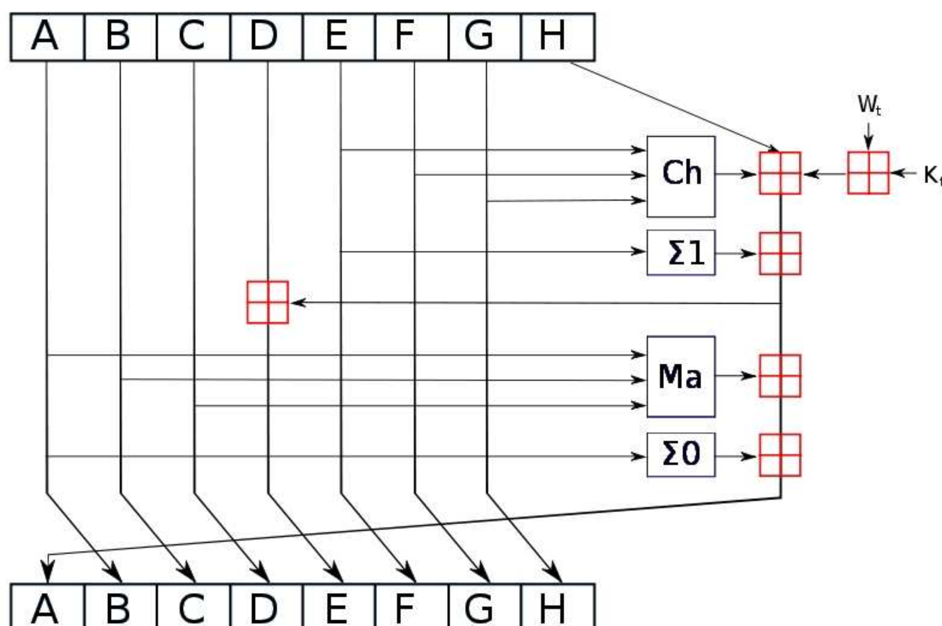
5.2.2 SHA-2

Az SHA-2 algoritmusokat 2001-ben publikálták a FIPS PUB 180-2 számú dokumentumban. A változatok az előállított lenyomat hosszának megfelelően: SHA-224, SHA-256, SHA-384, and SHA-512. A 224 bites változat csak később, 2004-ben jelent meg.

Az SHA-1-hez képest nagyobb biztonságot nyújtó függvény nem terjedt el olyan széleskörűen, mint elődje. Az okok között szerepel, hogy még az SHA-1-nél sem találtak ütközést, és a Windows XP SP2-es vagy régebbi operációs rendszerek nem támogatják az SHA-2 használatát.

Az üzenet előkészítése:

- SHA-224, SHA-256: Az üzenet az SHA-1-nél megismert módszer szerint kerül feldolgozásra
- SHA-384, SHA-512: Szintén hasonlít az SHA-1 módszerére annyi különbséggel hogy itt a $l + 1 + k \equiv 896 \; mod \; 1024$ kongruenciának kell teljesülni l az eredeti üzenet hossza, k a 0 bitek száma. Az így kapott üzenet hossza 1024 többszöröse lesz.



17. ábra: SHA-2 művelet

Az SHA-256 működése pszeudó kóddal leírva:

```
Változók kezdőértékének beállítása
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Konstansok értékének beállítása:
k[0..63] :=
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x59f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967,
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354,
0x766a0abb, 0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
0xa4506ceb, 0xbef9a3f7, 0xc67178f2


Az üzenet minden 512 bites darabjára a következő műveletek
végrehajtása:
w[i]: az 512 bites rész egy 32 bites részlete, i≤0≤15
for each 512-bit chunk of message

   A 16 db 32 bites szó kiterjesztése 64 bitesre
      for i from 16 to 63
         s0 := (w[i-15] rightrotate 7) xor (w[i-15]
         rightrotate 18) xor (w[i-15] rightshift 3)
         s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate
         19) xor (w[i-2] rightshift 10)
         w[i] := w[i-16] + s0 + w[i-7] + s1

      a := h0
      b := h1
      c := h2
      d := h3
```

```
        e := h4
        f := h5
        g := h6
        h := h7

        for i from 0 to 63
            s0 := (a rightrotate 2) xor (a rightrotate 13) xor (a
            rightrotate 22)
            maj := (a and b) xor (a and c) xor (b and c)
            t2 := s0 + maj
            s1 := (e rightrotate 6) xor (e rightrotate 11) xor (e
            rightrotate 25)
            ch := (e and f) xor ((not e) and g)
            t1 := h + s1 + ch + k[i] + w[i]
        h := g
        g := f
        f := e
        e := d + t1
        d := c
        c := b
        b := a
        a := t1 + t2

        h0 := h0 + a
        h1 := h1 + b
        h2 := h2 + c
        h3 := h3 + d
        h4 := h4 + e
        h5 := h5 + f
        h6 := h6 + g
        h7 := h7 + h

digest = hash = h0 append h1 append h2 append h3 append h4
append h5 append h6 append h7
```

Az SHA-224 működése ehhez nagyon hasonló, csak a h0,…,h7 kezdőértékek eltérőek és a lenyomathoz nem fűzi hozzá h7 érékét.

Az SHA-512-nél a számok 64 bit hosszúak, a k tömb 80 elemű, mások az inicializációs értékek, a fő ciklus 80 körből áll, és a különbözik a forgatott, ill. eltolt bitek száma.

Az SHA-384 működése az 512 bites változathoz nagyon hasonló, csak a h0,…,h7 kezdőértékek eltérőek és a lenyomathoz nem fűzi hozzá h6 és h7 érékét.

5.2.3 SHA-3 [5]

A NIST hogy kiválassza az SHA-2 algoritmusok kiegészítésére szánt új kriptográfiai hash függvényt, pályázatot írt ki 2007. november 2.-án. Ez az új függvény kapja majd meg az SHA-3 nevet és alkalmas lesz arra, hogy mind az Egyesült Államok kormánya, a privát szektor és az egész világ ingyenesen használja.

A jelöltek 2008. október 31.-éig nyújthatták be pályázatukat. A 64 jelöltből 51-et engedtek tovább az első körbe december 10.-én.

Az első SHA-3 konferenciára 2009. február 25.-től került sor. Elhatározták, hogy 2009 nyarára tovább szűkítik a pályázók számát. Július 24.-én bejelentették a 14 algoritmust, amelyek továbbjutottak a verseny második körébe. Ezek: BLAKE, Grøstl, Shabal, BLUE MIDNIGHT WISH, Hamsi, SHAvite-3, CubeHash, JH, SIMD, ECHO, Keccak, Skein, Fugue, Luffa.

A pályázóknak ezután szeptember 15.-ig adtak időt, hogy finomítsák függvényeiket, kijavítsák a problémákat, ha szükségesnek érzik.

A NIST akkortól számítva 1 évig várja, hogy a nagyközönség analizálja az algoritmusokat, és az esetleges hibákra fény derüljön.

A következő SHA-3 konferenciát 2010. augusztus 23.-n tervezik megtartani, hogy a második körös pályázók újra bemutathassák algoritmusaikat. A konferencia után kb. 5 döntőst fognak kiválasztani a verseny végső köre számára.

5.3 RIPEMD [3]

A RIPEMD család öt különböző hash függvényből áll. Ezek: RIPEMD, RIPEMD-128, RIPEMD-160, RIPEMD-256 és RIPEMD-320. Az előállított lenyomat méret rendre: 128, 128, 160, 256, 320 bit. Az üzenetet 512 bites blokkonként dolgozzák fel 16 darab 32 bites szóra osztva. Az üzenet kezdeti kitöltése a Merkle-Damgård megerősítésnek megfelelően történik. A függvények tömörítő függvényei 32 bites láncolt változókkal dolgoznak, melyek első körben fix értékekkel, a következő üzenetblokk számára pedig a köztes hash értékkel rendelkeznek.

A RIPEMD, RIPEMD-128, RIPEMD-160 függvények a láncolt változókból két lokális példányt készítenek és ezeket párhuzamosan, egymástól függetlenül dolgozzák fel. Minden körben a változók értékei az üzenetblokk újabb 32 bites szavával frissülnek. Ebből következik, hogy 16 kör után az üzenetblokk minden szava feldolgozásra kerül, újabb 16 kör után pedig megint az összes szó sorra kerül, de már más sorrendben. Az előző lépés 3-szor, 4-szer, 5-ször ismétlődik az algoritmustól függően. Az utolsó lépésben a láncolt változó értéke összeadódik a helyi változókkal, így megkapjuk a köztes hash értéket. Az üzenet minden 512 bites blokkjának feldolgozása során a köztes hash érték már a végső lenyomatot tartalmazza.

A RIPEMD-160 tömörítő függvénye:

$A \leftarrow h_0, A' \leftarrow h_0$
$B \leftarrow h_1, B' \leftarrow h_1$
$C \leftarrow h_2, C' \leftarrow h_2$
$D \leftarrow h_3, D' \leftarrow h_3$
$E \leftarrow h_4, E' \leftarrow h_4$
for i = 0 to 79 do
  $T \leftarrow (A + f_i(B, C, D) + W_{r(i)} + K_i)^{\lll s_i(r(i))} + E$
  $T' \leftarrow (A' + f_{79-i}(B', C', D') + W_{r'(i)}$
    $+ K_i')^{\lll s_i(r'(i))} + E'$
  $A \leftarrow E, A' \leftarrow E'$
  $E \leftarrow D, E' \leftarrow D'$
  $D \leftarrow C^{\lll 10}, D' \leftarrow C'^{\lll 10}$
  $C \leftarrow B, C' \leftarrow B'$
  $B \leftarrow T, B' \leftarrow T'$
$T \leftarrow h_1 + C + D'$
$h_1 \leftarrow h_2 + D + E'$
$h_2 \leftarrow h_3 + E + A'$
$h_3 \leftarrow h_4 + A + B'$
$h_4 \leftarrow h_0 + B + C'$
$h_0 \leftarrow T$

Jelentse:

- $h_0,\dots,h_4$ a láncolt változókat,
- $X^{\lll n}$ az X n bittel történő balra forgatását,
- $+$ modulo 2^{32} összeadást,
- $r(i)$ és $r'(i)$ a üzenetszavak indexeit,
- f_i nem lineáris függvényeket,
- s_i a shifteléseket,
- K_i és K_i' a konstansokat.

$r(i)$ és $r'(i)$ meghatározása:

ρ és π két permutáció:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(j)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

$\pi(j)=(9j+5)\text{mod}16$

	$0 \le i \le 15$	$16 \le i \le 31$	$32 \le i \le 47$	$48 \le i \le 63$	$64 \le i \le 79$
$r(i)$	i	$p(i-16)$	$p^2(i-32)$	$p^3(i-48)$	$p^4(i-64)$
$r'(i)$	$\pi(i)$	$p\pi(i-16)$	$p^2\pi(i-32)$	$p^3\pi(i-48)$	$p^4\pi(i-64)$

s_i értékek:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$0 \le i \le 15$	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
$16 \le i \le 31$	12	13	11	15	6	9	9	7	12	15	11	13	7	8	7	7
$32 \le i \le 47$	13	15	14	11	7	7	6	8	13	14	13	12	5	5	6	9
$48 \le i \le 63$	14	11	12	14	8	6	5	5	15	12	15	14	9	9	8	6
$64 \le i \le 79$	15	12	13	13	9	5	8	6	14	11	12	11	8	6	5	5

K_i *és* K_i*'* a konstansok:

	$0 \le i \le 15$	$16 \le i \le 31$	$32 \le i \le 47$	$48 \le i \le 63$	$64 \le i \le 79$
K_i	00000000_H	$5A827999_H$	$6ED9EBA1_H$	$8F1BBCDC_H$	$A953FD4E_H$
K_i'	$50A28BE6_H$	$5C4DD124_H$	$6D703EF3_H$	$7A6D76E9_H$	00000000_H

Az f_i nemlineáris függvények:

$f_i = (x, y, z) = x \oplus y \oplus z$ $\quad 0 \le i \le 15$

$f_i = (x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$ $\quad 16 \le i \le 31$

$f_i = (x, y, z) = (x \vee \neg y) \oplus z)$ $\quad 32 \le i \le 47$

$f_i = (x, y, z) = (x \wedge z) \vee (y \wedge \neg z)$ $\quad 48 \le i \le 63$

$f_i = (x, y, z) = x \oplus (y \vee \neg z)$ $\quad 64 \le i \le 79$

A RIPEMD-256 a RIPEMD-128-ból, a RIPEMD-320 pedig a RIPEMD-160 függvényekből származtathatóak kis átalakítással.

Az eredeti RIPEMD függvényt 1992-ben dolgozták ki. Az 1995-ös ütközéstalálat miatt vezették be a család többi tagját.

A 256 ill. 320 bites változatok nem nyújtanak magasabb fokú biztonsági szintet, a kisebb lenyomatméretűekhez képest, és az ütközést kereső támadások elleni védekezés miatt ajánlott a legalább 160 bites változat használata.

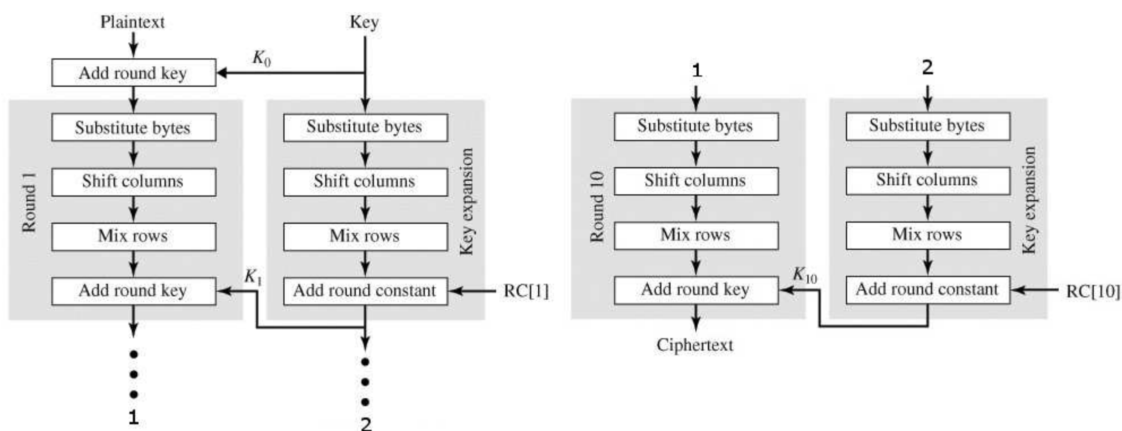
5.4 Whirlpool [2]

A Whirlpool első verzióját (Whirlpool-0) 2000-ben publikálta Vincent Rijmen és Paulo S. L. M. Barreto. Az algoritmus második verziója (Whirlpool-T) sikeresen pályázott az Európai Unió által finanszírozott NESSIE (**N**ew **E**uropean **S**chemes for **S**ignatures, **I**ntegrity, and **E**ncryption) projectre. A project célja az volt, hogy olyan kriptográfiai algoritmusokat gyűjtsön össze, amelyek gyengeségei minimálisak. A végső, javított változatot az ISO (International Organization for Standardization) nemzetközi szabványügyi hivatal az ISO/IEC 10118-3:2004 számú szabványban fogadta el.

A függvény tömörítő függvénye az AES -en(Advanced Encryption Standard) alapuló blokk kódoló és az üzenetről készült lenyomat 512 bites. Az üzenet hossza maximum 2^{256}-1 bit lehet.

Az algoritmus működésének lépései:

- Bitkiterjesztés: Az üzenet végéhez egy 1-es bitet és annyi 0-s bitet illesztünk, hogy az így kapott hossz 256 páratlan számú többszöröse legyen.
- A végére illesztjük az eredeti üzenet hosszának 256 biten kifejezett értékét. Az így kapott üzenet hossza 512 egész számú többszöröse.
- Az algoritmus egy 8x8 –s mátrixban tárolja az állapotokat. A mátrix egy cellájában 1 byte foglal helyet. Kezdetben csak 0-s biteket tartalmaz.
- Blokk kódolóját W-el jelölik és hasonlít az AES kódolóhoz, de több különbség is van közöttük.



18. ábra: A Whirlpool blokk kódolója

A titkosító algoritmus 4 különböző műveletet használ: Add round key (AK), Substitute bytes (SB), Shift clolumns (SC), Mix rows (MR). Az első kör előtt egyszer alkalmazzuk az AK műveletet, majd a 10 kör alatt mind a 4 művelet végrehajtódik.

A bementként kapott 512 nyílt üzenetblokkal sorfolytonosan feltölti a 8x8-s állapotmátrixot (CState). A kulcsot is, az előzőhöz hasonlóan egy mátrixba rakja bele (KState).

A 4 művelet működése:

- SB: Egy 16x16-os mátrixot használ, cellánként 1 byte-al. A táblázat megadja, hogy a CState minden egyes byte értékét milyen értékre kell kicserélni.
- SC: A CState oszlopain egy lefelé irányuló körkörös shift műveletet hajt végre. A 2., 3., …, 8. oszlopát rendre 1, 2, …7 értékkel tolja el lefelé, úgy hogy az alul „lepotyogó” byte-ok fent visszakerülnek a mátrixba. A művelet az első oszlopot változatlanul hagyja.
- MR: A CState minden egyes során egyfajta szétszórást hajt végre. Minden sor, minden byte-ja a sor mind a 8 byte-jának függvényében kerül lecserélésre.
- AK: a körkulcs és a CState között egy bitenkénti XOR műveletet hajt végre.

A whirlpool függvény teljesítménye jó, köszönhetően az AES-re alapuló kódolójának. Az SHA 512 bites változatához képest jobb teljesítményt nyújt, azonban ehhez több erőforrásra van szüksége.

5.5 A bemutatott hash függvények példa lenyomatai [19], [20]

x=”hashteszt” h(x):

```
MD2:
B5C34E7880BAF93F68B1EBD46CC44626

MD4:
E3068A7083D6876F3C5BF6A77DA7D278

MD5:
5E8D08C911F860309D527324340A8620

MD6:
D1515B9D5C206B559C42F7D2B2267573B82728D4049F2BC17503CE42C51883DDCA4D
CB58E79B23FDC168D9CAD63705221769B75A1AD11186B53574354E947FA2

RIPEMD-128:
08FAD35B3C3EE4D4F602F81C57E65AB1

RIPEMD-160:
61C286562F4D80A48AFC21BC3C30AACE66A5F266

RIPEMD-256:
86D8B18C2E2B183451853882C0BB082011A0A39798C470B47A197E5ABFADC616

RIPEMD-320:
3C16B2F8E15E51504A0BDD0EC4D98B5F7C9CEC269084F5F88AA77914299122029C67
10C155E60273

SHA-0:
9F9ABF85843463B9F88ACAAD4BE6EDA8759468EC

SHA-1:
3AB49EDBE06AEC81EC7075AF50E527A6B12546C6

SHA-224:
48641EE4108F61F342FA5ACCC05E88833538F9CBC9E9B255169A56BC

SHA-256:
D32B8278670C17C63BD932545606D8F389312E7C139A7661E2231028DD0B7697

SHA-384:
168FE3642E914FC0CE46E1E582DC31EA97D9148AF8E3C7228B1B934748267A619E47
C82C3011C77CFDC6CAA9BAC9401C

SHA-512:
339AADE889BBFA91D0833A9EC1C50C99B34CBB43BAD5F573CE8D49D572411AC0338E
C5D7C6D5C6596AE3C36A974FE5B9323F3331E456602734AD44FA6F4A576C

Whirlpool:
1BDBF234E4417D110560D1E16C4AB50CDBBABE89449A57C2D478701F8B4D01685597
9941842470271D40F485CFA08070AC6CAA43DC6B2F2388FD20A07D5C5D30
```

6. fejezet

Támadások a hash függvények ellen [21], [3]

A hash függvényt megtámadó személynek több célja is lehet: előkép megtalálása, másodelőkép találása, ütközések keresése, MAC kulcsának kikövetkeztetése. A következőkben az első cél elérésének lehetőségeit mutatom be, azaz amikor az ismert h(x) lenyomathoz meg szeretnénk tudni x-et, tehát az üzenetet.

Az előkép megszerzésére irányuló támadásokat alapvetően két csoportba oszthatjuk. Az egyik csoportba azok tartoznak, amelyek csupán a rendszer számítási kapacitását használják a sikeresség érdekében. Ezek eredményessége nagymértékben elősegíthető, ha az üzenet megfelel bizonyos szabályoknak.

Mivel a számítógépes rendszer nem csak a központi számító egységből áll, hanem rendelkezik tárolókapacitással is, a két erőforrást együttesen kihasználó módszerek alkotják a támadások második csoportját.

6.1 Brute Force

Az elnevezés találó: „nyers erő”. A legpuritánabb módszer, csak a rendszer számítási kapacitását használja. x meghatározásához a karakterek összes variációjának hash értékét legeneráljuk és egyesével összehasonlítjuk a $H(x)$ értékkel. Ha megtaláltuk az egyezést, akkor megtaláltuk az üzenetet is. A lehetséges variációk száma függ az üzenet hosszától és attól, hogy az üzenet milyen karaktereket tartalmazhat, azaz a karakterkészlettől.

A variációk száma: $\sum_{i=a}^{b} k^i$

- a: minimális üzenethossz
- b: maximális üzenethossz
- k: használt karakterek száma

Például: ha csak a magyar abc kisbetűi szerepelnek egy maximálisan 6 karakter hosszúságú üzenetben, akkor a variációk száma:

$$\sum_{i=a}^{b} k^i = \sum_{i=1}^{6} 44^i = 7\ 425\ 065\ 340$$

Ha az üzenet hossza maximálisan 7 karakter lehet:

$$\sum_{i=a}^{b} k^i = \sum_{i=1}^{7} 44^i = 326\,702\,875\,004$$

Látható, hogy ha a felhasználó jelszavának már 1 karakterrel is hosszabb szót választ, a támadónak ezzel a módszerrel sokkal kisebb lesz a sikeressége.

A felhasználók túlnyomó része jelszavainak értelmes szavakat választ. Így ha rendelkezünk egy listával, amely tartalmazza az összes értelmes magyar szót, akkor a jelszó kitalálása lényegesen kevesebb időt vesz igénybe.

6.2 Előre legenerálás

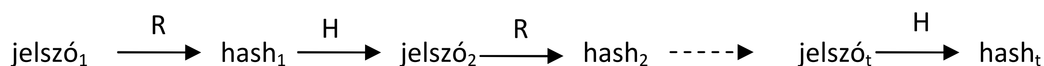
A szólistákat továbbgondolva tovább rövidíthető a jelszó feltörésének ideje, ha nem csak az értelmes szavakat tároljuk el, hanem a hash értéküket is, így a támadónak csak egyesével össze kell hasonlítania a *H(x)* éréket az előre legenerált hash értékekkel, így megspórolható a hash előállításához szükséges idő.

Ez a módszer már a fejlettebb támadások közé tartozik, mert a rendszer tárkapacitását is használja.

6.3 Time-Memory Trade-Off módszerek [7]

Az előgenerálás hátránya (a támadó szemszögéből nézve), hogy ha a tetszőleges karaktereket tartalmazó (kis-és nagybetűk, számok, különleges karakterek) értelmetlen jelszavakat is el akarjuk tárolni akkor nagyon nagy tárkapacitásra lenne szükség. A hash láncok csökkentik a szükséges tárterületet. A hash láncok ötlete az 1980-as évekből származik.

A módszer alkalmaz egy *R* redukciós függvényt, amely a hash értékeket a lehetséges jelszavak halmazára képezi le, és alkalmazza a *H* hash függvényt. A *H* és *R* egymás utáni alkalmazásával láncok alakulnak ki.



19.ábra: Hash lánc

A táblázat generálásához véletlenszerűen kiválasztunk jelszavakat, legeneráljuk a t hosszúságú láncokat, és eltároljuk a láncok első (kezdőpont) és utolsó (végpont) jelszavait.

Ezután *H(x)*-el kezdve kiszámítunk egy láncot. Ha a láncban valamelyik elem megegyezik a táblázatban tárolt végpontok egyikével, akkor az elemet felhasználva új láncot indítunk. Annak a valószínűsége, hogy az így kapott láncban megtaláljuk *H(x)* értékét és így a láncban pontosan előtte szereplő jelszót:

$$P \geq \frac{1}{N}\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1-\frac{it}{N}\right)^{j+1}$$

- *m*: láncok száma
- *t*: lánchosszúság
- *N*: lehetséges jelszavak száma

  l tábla esetén a sikeresség tovább növelhető:

$$P \geq 1-\left(1-\sum_{i=1}^{m}\sum_{j=0}^{t-1}\left(1-\frac{it}{N}\right)^{j+1}\right)^{l}$$

6.3.1 Szivárvány táblák [8], [17]

Az előző módszernek van egy hibája: ha két lánc valamelyik eleme megegyezik, akkor a láncok összeolvadnak és a táblázat kevesebb lehetséges jelszót fog tartalmazni. Mivel a láncok nincsenek teljes egészében tárolva, az ütközés nem vehető észre.

Ezen segítenek a szivárvány táblák: egy redukciós függvény helyett redukciós függvények sorozatát használják. Ütközés esetén a láncok végpontjai is megegyeznek és a duplikált lánc eltávolítható így a táblázatba új lánc kerülhet be. Még így sem garantált az ütközésmentesség, de a láncok nem olvadnak össze.

A sikeres találat valószínűsége:

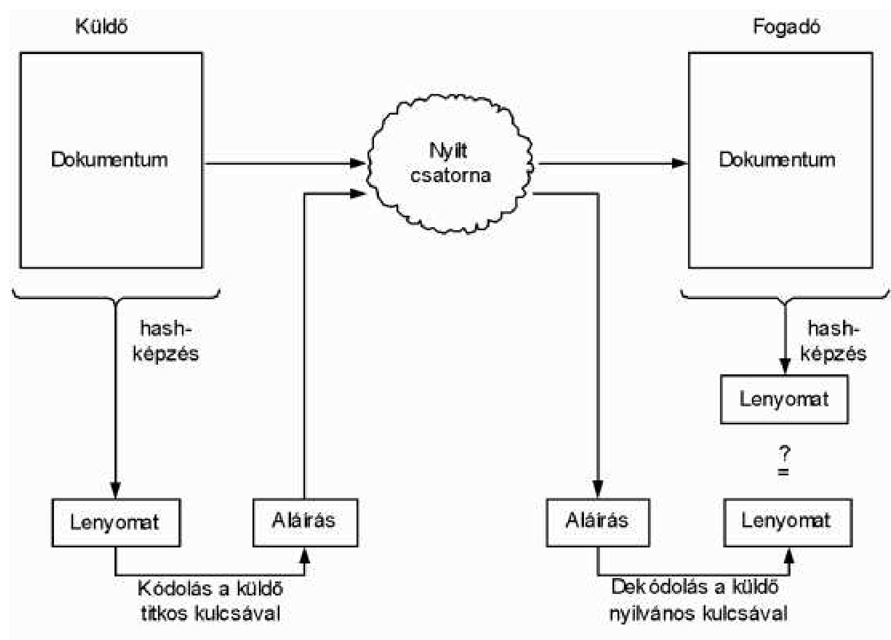
$$P = 1-\prod_{i=1}^{t}\left(1-\frac{m_i}{N}\right)$$

$$ahol\ m_{\ 1} = m\ és\ m_{n+1} = N\ \left(1-e^{-\frac{m_n}{N}}\right)$$

7. fejezet

Legfontosabb alkalmazási területek

7.1 Digitális aláírás [18]



20. ábra: Digitális aláírás folyamata

A digitális aláírás során az aláírandó dokumentumról először egy lenyomat készül. Ezután a hash értéket a küldő fél saját titkos kulcsával titkosítja. Az így kapott érték a digitális aláírás. Az eredeti dokumentumot és az aláírást elküldi a fogadó személynek. A vevő miután megkapta a dokumentumot szintén elkészíti a hash értéket (ugyanazzal a hash függvénnyel, mint a küldő személy) és összehasonlítja a kapott lenyomattal, amit a küldő nyilvános kulcsával dekódol. Ha a két érték megegyezik, akkor a dokumentumot nem módosították miközben áthaladt a csatornán. Az üzenetintegritáson túl a hash függvény használatának másik nagy előnye a méret jelentős lecsökkentése.

7.2 Jelszavak

Ha a rendszer a felhasználók jelszavait nyílt üzenetként tárolja, akkor azok, a rendszerhez hozzáférő támadó számára könnyen hozzáférhetőek. Ezért a jelszavakat csak hash érték formájában szabad tárolni. Amikor a felhasználó bejelentkezik a rendszerbe, a begépelt jelszavának hash értéke és a tárolt hash értékek lesznek összehasonlítva.

A támadó dolgát tovább nehezíti, ha a jelszó végéhez hozzáfűzünk egy véletlenszerűen generált értéket. Ezt sózásnak (salting) nevezzük. A sózás még az olyan kifinomult támadásokkal szemben is védelmet biztosít, mint a szivárvány táblák használata. Már egy 100 bites só is olyannyira megnöveli a lehetséges jelszavak számát, hogy lehetetlenné válik a hash érék visszafejtése.

7.4 Üzenetintegritás

Az üzenetek integritásvizsgálatát nem csak a digitális aláírásnál használják, hanem bármely olyan esetben ahol szeretnénk biztosítani, hogy a vevő ellenőrizhesse a kapott adatok épségét. Interneten keresztül elérhető nagyobb méretű adatok letöltése után, a hash kódot elkészítve, majd összehasonlítva a letöltés helyén közzétett értékkel, megbizonyosodhatunk róla hogy az adatok nem sérültek. Ez hasznos lehet pl.: Linux disztribúciók letöltése esetén, amikor ezt elvégezve megspórolhatunk egy üres CD-t, esetleg DVD-t.

Irodalomjegyzék

[1] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone : Handbook of Applied Cryptography, CRC Press, 1997

[2] William Stallings: Cryptography and Network Security Principles and Practices, Fourth Edition, Prentice Hall, 2005

[3] Henk C. A. van Tilborg: Encyclopedia of Cryptography and Security, Springer, 2005

[4] NIST: Federal Information Processing Standards Publication, FIPS PUB 180-3, 2008

[5] NIST: Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition, 2009

[6] Ronald L. Rivest : The MD6 hash function A proposal to NIST for SHA-3, 2009

[7] M. E. Hellman. A Cryptanalytic Time-Memory Trade-Off, 1980

[8] Philippe Oechslin: Making a Faster Cryptanalytic Time-Memory Trade-Off, 2003

[9] http://en.wikipedia.org/wiki/Avalanche_effect

[10] http://en.wikipedia.org/wiki/Cryptographic_hash_function

[11] http://en.wikipedia.org/wiki/Merkle-Damg%C3%A5rd_construction

[12] http://en.wikipedia.org/wiki/One-way_compression_function

[13] http://en.wikipedia.org/wiki/MD2_(cryptography)

[14] http://en.wikipedia.org/wiki/MD4

[15] http://en.wikipedia.org/wiki/MD5

[16] http://en.wikipedia.org/wiki/SHA_hash_functions

[17] http://en.wikipedia.org/wiki/Rainbow_table

[18] http://www.biztostu.hu/mod/resource/view.php?id=413

[19] http://www.fileformat.info/tool/hash.htm

[20] http://www.sharewareconnection.com/fsum-frontend.htm

[21] http://en.wikipedia.org/wiki/Password_cracking