Kriptográfiai hash algoritmusok elemzése

Bucsay Balázs

Miskolci Egyetem Gépészmérnöki és Informatikai Kar

Témavezető: Dr. Kovács László egyetemi docens
Konzulens: Dr. Fegyverneki Sándor egyetemi docens

2009. május 13.

Tartalomjegyzék

1. Bevezetés 3

2. Hash algoritmusok matematikai szempontból 5

3. Hash algoritmusok kriptográfiai szempontból 7
3.1. Bevezetés . . . 7
3.2. Alkalmazásuk . . . 8
3.3. Kriptográfiai tulajdonságai . . . 8
3.4. Hasonló algoritmusok . . . 9
3.5. Egy-irányú törmörítés . . . 10
3.5.1. Davies-Meyer . . . 12
3.5.2. Matyas-Meyer-Oseas . . . 13
3.5.3. Miyaguchi-Preneel . . . 13
3.6. Merkle-Damgård elképzelés . . . 14
3.7. Block cipherek . . . 15
3.8. Block cipherekre alapuló hash algoritmusok . . . 15
3.9. Kriptográfiai hash függvények összefűzése . . . 15
3.10. Avalanche effect (Lavina effektus) . . . 16
3.11. Plusz fogalmak . . . 16

4. Hash algoritmusok az informatikában 18

5. MySQL-323 algoritmus 20

6. MD5 algoritmus 23

7. FreeBSD MD5 algoritmus 28

8. SHA-1 algoritmus 31

9. Algoritmusok összehasonlítása 35

10. Hash törési módszerek **36**
10.1. Bruteforce 36
10.2. Külső szabályok (External rules) 37
10.3. Szólisták (wordlists) 38
10.4. Előgenerálás 38
10.5. Trade-off módszerek 39
10.5.1. Cryptanalitic time-memory trade-off 39
10.5.2. Rainbow Tables (Szivárvány táblák) 40
10.5.3. Markov szűrő 41

11. John the Ripper rövid bemutatása **43**

12. Optimalizálás **45**
12.1. Architektúrális optimizálás 46
12.2. Optimalizálás memóriával 47
12.3. Optimalizálás hash csonkolással 48

13. Tesztelés és végszó **50**

A. CD-Használati útmutató **53**
A.1. Források és fordításuk 53
A.2. John The Ripper, modulok forrása és fordításuk 54
A.3. John The Ripper és a modulok használat 54

1. fejezet

Bevezetés

A szakdolgozat a kriptográfiában és az informatikában használatban lévő kriptográfiai hash algoritmusokról, azok optimalizálásáról, kulcsok hash formából való visszakeresés módszereireiről szól és tesz javaslatokat rá.

Napjainkban nagyon nagy szerepet kap az informatikán belül a biztonságtechnika, aminek szerves része a kriptográfia. Kriptográfia egy kisebb részhalmaza az egyirányú kódolások az úgynevezett hash algoritmusok. Ezek az algoritmusok képesek arra, hogy bármekkora bemenetről egy fix méretű egyedi kimenetet képezzenek le, ami csak arra lesz jellemző, ideális esetben. Ezzel a módszerrel tudjuk biztosítani azt, hogy a legkisebb eltéréssel rendelkező bemenetek is különböző kimentet kapjanak az intervallumban (egyezések történhetnek, de a valószínűségük elég kicsi). Így bármilyen implementációs szinten tudunk könnyedén eltérést ellenőrizni (Információelmélet: Hibadetektálás, Biztonságtechnika: Jelszavak biztonságos tárolása, stb.) Rendkívül könnyen, jól használható technika ez és rendkívül elterjedt is. Előnye, hogy csak egy irányba működik, így ha megpróbáljuk megfordítani az algoritmust, véges sok variációt kapunk véges hosszúságú bementekre.

Rengeteg rendszer használja authentikációnál azt a módszert, hogy a megadott jelszónak illetve jelmondatnak egy specifikált hash algoritmus szerinti kimenetét tárolja el, és ha ezekre kerül a sor akkor csak újra elkódolja a megadott algoritmus szerint, ha egyezés van akkor megegyezik a bemenet is, sikeres az azonosítás, ha nem akkor nem egyezett meg, így sikertelen azonosítás.

Digitális aláírások is így működnek. A megadott bemenetről (bináris fájlok, szöveges üzenetek, e-mailek) készít az algoritmus egy hash-t, majd a bemenet mellé csatolja. A címzett miután megkapta az üzenetet ismét megnézi mi a kapott tartalom hash-e, és ha egyezik akkor módosítatlanul jött az üzenet, ha nem egyezik akkor esetleg hamisított példányt kapott.

A következő fejezetek bemutatni hivatottak a legelterjedtebb vagy legegyszerűbb hash algoritmusokat, azok felépítését, esetleges hibáit, módszereket ütközés vagy részleges ütközés találására, tár- illetve időkapacitás spórolásra így gyorsabb és hatékonyabb hash megfejtést eredményezve. Ezen felül bemutatom a hagyományos és újabb módszereket a hashelt kulcsok megfejtésére.

2. fejezet

Hash algoritmusok matematikai szempontból

A hash algoritmusoknak több fajtája van, az egyik a matematikában használt hash függvények. Ezek a függvények bármilyen adatot megadott méretű számmá alakítanak, így használhatóak például indexeknek tömbökhöz. Természetesen mint látható is szorosan kötődnek más fogalmakhoz, mint például a lenyomatokhoz, hiba detektáló és javító kódokhoz, kriptográfiai hash algoritmusokhoz. Hash függvények használatával létrehozhatunk hash táblákat, amiben kulcsokat tárolhatunk. A kulcsokat a hash függvény által visszaadott index alatt tároljuk el.

Egy ideális matematikai hash függvény jellemzői:

- Alacsony költség: a függvény költsége nem haladhatja meg bármelyik másik kereső függvény költségét. Még a bináris keresés $O(\log n)$ -el dolgozik, addig egy jó hash függvénynek a kereséshez vagy akár a beszúráshoz is csak $O(1)$-re van szüksége.
- Determinisztikusság: Bármilyen bemenetről ugyanazt a kimenetet kell, hogy adja, bármennyi próbálkozás után is.
- Uniformitás: Fel kell mérni a hash függvény megalkotása előtt a bemeneti halmazt, és aszerint megalkotni. Minden kimenetre ugyanazon valószínűséggel kell leképeznie, vagyis a kimenetek eloszlása egyenletes kell hogy legyen. Ha az eloszlás nem egyenletes, akkor egy rekeszbe akár több kulcs is kerülhet, így megnő a beszúrás illetve a keresés költsége is.

Egy jól felmért helyzetben, egyenletes eloszlású hash függvényt létrehozva, $O(1)$-el tudunk írni és olvasni egy hash táblába.

Ha nem jól mérjük fel az igényeket, vagy szimplán rosszul alkotjuk meg a hash függvényt akkor akár $O(n)$ is lehet a beszúrás és a keresés is.

Ezen követelmények tovább bővítésével alkothatóak meg a kriptográfiai hash algoritmusok.

3. fejezet

Hash algoritmusok kriptográfiai szempontból

3.1. Bevezetés

Kriptográfiai hash algoritmusok is a matematikai hash függvényeken alapulnak. Ezek a típusú függvények csak pár kritérium után kapják meg a kriptográfiai hash algoritmus besorolást. Maga az algoritmus egy transzformáció ami bármekkora méretű bemenetből egy fix méretű egyedi kimenetet eredményez, ezt hívjuk hash-nek (hash értéknek).

3.1.1. Definíció (Kriptográfiai hash függvény)**.** [2] Legyen $m \in \{0,1\}^*$ (a kódszavak halmaza). $H(m)$ pedig egy hash függvény. $H : \{0,1\}^* \to \{0,1\}^n$, ahol $n > 0, n \in \mathbb{N}$.

Ezeket a hash algoritmusok széles körben használják, például:

- file tartalom ellenőrzésre
- üzenet integritás vizsgálat
- authentikáció

A hash algoritmusok egy elviekben akár végtelen hosszú üzenetet várnak bemenetként és egy algoritmus által előre definiált méretű kimenetet hash-t generálnak. Bármekkora méretben is változik az üzenet, a hash értéknek is elvileg változnia kell vele. Így bárki kezébe kerül a hash, az üzenetet nem tudja belőle meghatározni, viszont akinek megvan az üzenet, reprodukálhatja könnyedén a hozzá tartozó hash-t.

Egy ideális kriptográfiai hash algoritmus (továbbiakban szimplán hash algoritmus) három tulajdonsága: hihetetlenül egyszerű (gyors) hash értéket számítani vele hihetetlenül nehéz, szinte lehetetlen hash érték után megmondani a bemenetet (úgy hogy előzőleg nem ismerjük) magasan valószínűtlen két olyan bemenetet találni, aminek a hash értéke megegyezik. Jellemzői még, hogy a működésének a lehető legjobban kell hasonlítania egy (pszeudó) random függvényhez, de mégis determinisztikusnak kell maradnia.

Az algoritmus nem biztonságosnak tekinthető, ha:

- Előzőleg nem látott üzenet megtalálhatunk a hash értékből.
- Ütközéseket találunk két nem egyező üzenetből generált hash-nél: $m_1 \neq m_2, H(m_1) = H(m_2)$

Ha bárki képes ezek előállítására, akkor hamisíthat vagy jogosulatlanul küldhet üzenetet bárkinek például.

3.2. Alkalmazásuk

Alkalmazásuk rengeteg helyen elterjedt. Pl. üzenet integritás megőrzéséhez használják. Az üzenet az éteren keresztül könnyen eltorzulhat és pár esetben a hiba javító/detektáló kódok sem segítenek ezen. Az üzenet megérkezése után könnyen ellenőrizhetjük épségét egy hash értékkel amit a küldés előtt generáltak. Ha a két érték egyezik, akkor biztos hogy ugyanazt az üzenetet kaptuk meg. Ezt a módszert használják a unix rendszerek csomagkezelői, forráskód management rendszerek, windows alatt a drivereknél mint digitális aláírás.

Peer-to-peer rendszereknél a csomagok érintetlen, hiba mentes érkezését nézve.

Jelszavas beléptetéseknél, ahol nyilvánvaló okokból nem plaintextben (titkosítatlan formában) tárolják azt, hanem hashelt formában. Ha a megadott jelszó hash-e és az eltárolt hash egyezik, akkor sikeres az authentikáció.

3.3. Kriptográfiai tulajdonságai

Bár nincsenek lefektetett követelmények egy ilyen algoritmustól, viszont az biztos, hogy ezek betartása a minimum, a biztonságos függvény megalkotáshoz:

- Előkép ellenálló (*Preimage resistant*), adott h-hoz nehéz legyen, olyan x-et találni, ami teljesíti az $H(x) = h$-et.
- Másod előkép ellenálló (*Second preimage resistant*): adott m_1-hez nehéz legyen olyan m_2-őt találni amire igaz, hogy: $H(m_1) = H(m_2)$, de $m_1 \neq m_2$
- Ütközés ellenálló (*Collision resistant*): nehéz bármilyen m_1-et és m_2-őt találni, úgy hogy a következő állítás teljesedjen: $H(m_1) = H(m_2)$

Bár ezek a kritériumok mind segítenek a hash algoritmusok biztonságossá tételéhez, még mindig nem mondhatjuk, hogy egy hash algoritmus biztonságos ami ezekkel a tulajdonságokkal rendelkezik. Sok függvény ezek ellenére is sebezhető marad, pl a hossz-kiterjesztő támadás ellen.
Ha létezik m üzenetünk és H algoritmusok és tudjuk a $H(m)$ hasht és a $length(m)$ hosszt, akkor könnyedén meg tudjuk mondani egy választott m' -re a következőt (|| a konkatenációt jelenti):

$$H(m||m')$$

3.4. Hasonló algoritmusok

Még a kriptográfiai hash algoritmusoknak van pár fix kikötése az elkészítésben, van pár hasonló felépítésű és alkalmazású hash függvény, amik bár nem ugyanazt a biztonságot garantálják, de a felhasználásuknál ezt nem is követelik meg.

Ilyen pl a CRC algoritmus (cyclic redundancy check), régebben használták a biztonságtechnikában is, pl WEP-nél (Wired Equivalent Privacy), de viszonylag hamar találtak benne sebezhetőséget.

Hasonló elven működik a MIC (Message Integrity Code) is, ami bármely hosszú üzenetből egy hash-t készít, amit az üzenet mellé csatolhatunk. Ez hash biztosítja az integritását az üzenetnek, vagyis azt, hogy senki sem változtatta meg. A MIC-nek nem árt egy titkosított csatornán utaznia, hogy ha az üzenetet hamisítják, legalább a MIC-et ne tudják, mivel egy hamisított üzenetből könnyedén előállíthatjuk az annak megfelelő MIC-et.

A MIC kiterjesztése a MAC avagy MAIC (Message Authentication and Integrity Code) ami az üzenet integritásán kívül még az authentikusságát is magában hordoz. Az algoritmus két bemenettel dolgozik, az egyik egy titkos kulcs, a másik maga az üzenet. Ezekből egy olyan hash-t generál,

amivel bármikor tudjuk igazolni a megfelelő kulccsal (autentikusság) magát a üzenet módosítatlan tartalmát (integritás). Még a hagyományos hash algoritmusok nem használnak kulcsokat, a MAC használ. Ezért az üzenet közlés feleinek, előre le kell tisztázni a titkos kulcsot, amit az algoritmusnál használni fognak.

A MAC-et tovább gondolva elég sok hasonló algoritmust fejlesztettek ki. Ezek közül az egyik a HMAC vagy KHMAC (Keyed-Hash Message Authentication Code). Ennek előnye, hogy nem csak egy titkos kulcsot, hanem egy szabadon választható kriptográfiai algoritmust is használ. Mint a MAC, a HMAC is integritást és autentikusságot garantál. Maga a generált kód, csak annyira lesz erős, mint az algoritmusban használt hash algoritmus, ha egy kriptográfiailag gyenge hash algoritmust használunk, akkor a HMAC kódot is könnyen hamisíthatjuk, találhatjuk meg a hozzá tartozó eredeti üzenetet és a kulcsot.

A HMAC függvény a következő:

$$HMAC(m,k) = H((k \oplus opad)||H((k \oplus ipad)||m))$$

m az üzenetet tartalmazza, k a titkos kulcsot, H az előre definiált kriptográfiai hash függvény. Ha a titkos kulcs hossza kissebb mint a hash algoritmus adatblokkmérete, akkor ki kell bővíteni a kulcsot a adatblokkméret hosszára zérusokkal. Ugyanígy kell megadnunk az $opad$ot és az $ipad$ot is. Az opad-nak *0x5C*, az $ipad$-nak pedig *0x36* hexadecimális értékkekből kell állniuk adatblokkméret hosszon.

3.5. Egy-irányú törmörítés

Bár tömörítés ez is, de nincs semmi köze az adat tömörítéshez. Az a fajta tömörítés reverzibilis, vagyis a bemenetet megkaphatjuk a kimenetből, még az egy-irányú tömörítés[6] inreverzibilis.

Az egy-irányú tömörítés az egy-irányú függvények családjába tartozik, amire jellemzőek a következők:

- a két bemenet alapján könnyű kiszámítani a kimenetet
- ha egy támadó tudja a kimenetet, akkor közel kivitelezhetetlen kiszámítani a bemeneteket
- ha egy támadó tudja a kimenetet, és csak az egyik bemenetet, akkor is legyen kiszámíthatatlan a másik bemenet.

- legyen ütközés ellenálló

Az egy-irányú tömörítő függvények két bemenetet vesznek és egy kimenetet adnak. Ezeket a bemeneteket keverik össze és adják eredményül a kimenetet. A bemenetek és a kimenetek hossza mindig fix, de nem kell hogy megegyezzenek. Vegyünk egy 256bites és egy 128bites bemenet, és a függvény belőlük egy 128bites kimenetet készít, vagyis 384bitből 128bitet.

Keverésnek a lavina effektust eredményeznie kell, vagyis minden bemenő bit hatással kell hogy legyen az összes kimenő bitre.

Sok egy-irányú tömörítő függvény blokk cipherekből készül. Még régi blokk cipherek teljes mértékben reverzibilisek, a modernebbeket már részlegesen konstruálják meg csak reverzibilisnek, hiszen csak a titkos kulcs ismeretében lehet mind a két irányba transzformálni az adatot, ezért lehetséges csak és érdemes a modernebbeket felhasználni egy-irányú tömörítő függvény készítésére. Pár transzformációval bármelyik blokk cipher egy-irányú tömörítő függvénnyé alakítható.

Pár konstrukció:

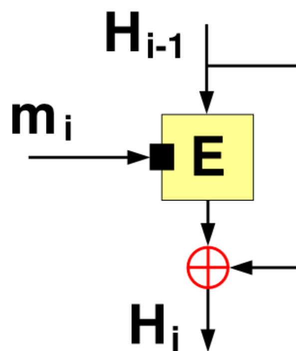
- Davies-Meyer
- Matyar-Meyer-Oseas
- Miyaguchi-Preneel
- MDC-2
- MDC-4.

A modern kriptográfiai hash függvények legtöbbje ezen a koncepciók minimum egyikét használja.

Egy hash függvény csak akkor tekinthető biztonságosnak ha: a blokk cipher nem rendelkezik különleges tulajdonságokkal ami megkönnyítené a megfejtést és elég nagy hash méretet eredményez (128bit valószínűleg, a jelenlegi teljesítményű számítógépek mellett elég). És az utolsó adatblokk kibővített méretben (adatblokknak megfelelően) kerül a hash algoritmus bementére (Lásd: Merkle-Damgård elképzelés).

3.5.1. Davies-Meyer

A David-Meyer egy-irányú tömörítő függvény az üzenetblokk egy darabjából és az előző állapotból állítja elő a kimenetet. Első lépésben nem ismerjük a 0. hash értéket, így egy előre meghatározott fix H_0 értékkel dolgozunk.



3.1. ábra. Davies-Meyer

Következő képpen írhatjuk le:

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

A kizáróvagy (XOR) operáció nem kötött, használhatunk más operátort is a függvényben.

Hiába használunk a függvényben teljesen biztonságos blokk ciphert, ennek ellenére találhatunk a bármelyik m_i-hez olyan h-t amire igaz:

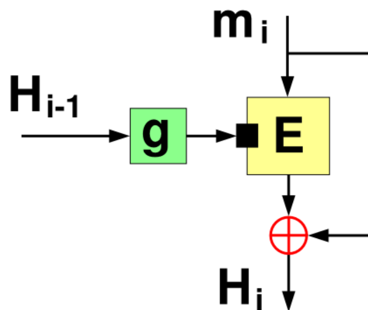
$$E_m(h) \oplus h = h, h \in H$$

Ez az állítás a számításhoz felhasznált változók fix méretéből ered.

A blokkméret a blokk cipher-től függ, mind a titkos kulcsnak (H), mind az adatblokk darabok méretének egyezniük kell.

3.5.2. Matyas-Meyer-Oseas

Ez a módszer pontosan a Davis-Meyer ellentéte avagy duálisa:



3.2. ábra. Matyas-Meyer-Oseas

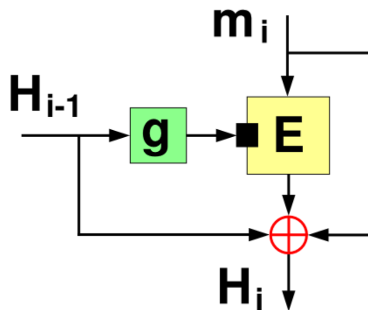
Leírva:

$$H_i = E_{g(Hi-1)}(m_{i-1}) \oplus m_{i-1}$$

Ha a blokk cipher más méretet kezel a titkos kulcsnál (hash értéknél), mint amekkora az adatblokk darabja, akkor azt muszáj átkonvertálnunk, kiegészítenünk. Erre használjuk a g függvényt az összefüggésben.

3.5.3. Miyaguchi-Preneel

A Miyaguchi-Preenel módszer a Matyas-Meyer-Oseas kiegészítettje. Az elv teljesen ugyanaz, mindenben megegyezik, egy dolgot kivéve. Ez pedig az utolsó lépés, ahol a kapott értéket, még az előző hash értékkel meg kell XOR-olnunk.



3.3. ábra. Miyaguchi-Preneel

Leírva:

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

3.6. Merkle-Damgård elképzelés

A Merkle-Damgård elképzelés[7] egy javaslat, módszer a kriptográfiai hash függvények elkészítésére. Manapság minden elkészített és gyakorlatban használt, elterjedt hash algoritmus ezeket a lefektetett alapelveket követi, így próbál biztonságosabb konstrukciót létrehozni. 1989-ben Ralph Merkle és Ivan Demgard egymástól függetlenül bizonyította, hogy ha a hash algoritmusban használt tömörítő függvény ütközés-ellenálló, akkor a hash függvény is az. Az is igaz, hogy a hash függvény csak annyira lesz ütközés ellenálló, amennyire a tömörítő függvény.

Ahhoz, hogy ezzel a módszerrel hash-t tudjunk előállítani több dologra van szükségünk. Első sorban egy üzenetre, amit a blokk ciphereknek megfelelő adatblokkméretekre kell felvágnunk. Ha az üzenet hossza nem zérust ad maradékul a blokkmérettel való osztáskor, akkor azt ki kell egészítenünk. Ezt a kiegészítést nevezzük *length-padding*-nak vagy *Merkle-Damgård strengthening*-nek. A módszer még egy javaslatot tesz arra is, hogy hogyan toldjuk ki az üzenet végét. Ha a megadott kritériumnak nem felel meg, akkor az üzenet végére egy lezáró értéket kell helyeznünk, majd az üzenet hosszát. A maradék hosszt zérusokkal kell feltölteni. Az kitoldott üzenet blokkokon kívül szükségünk van egy blokk cipher-re, ami a lényegi transzformációt végzi (lásd következő fejezet). A blokk ciphereknek minimum 2, vagy annál több bemenetük van, és egy kimenetük. Ezeket a bemeneteket transzformálják, keverik össze és azok alapján adják meg a kimenetet. Inicializációs Vektorokra (IV), amik az 1. lépésben adják a blokk ciphereknek a bemenetet az üzenet első blokkja mellé, többi lépésben nem a tiszta IV-ket használjuk az üzenetblokkok mellé, hanem az előző lépésben kapott, transzformáltakat.

Ha minden üzenetblokkon végigmentünk, megkapjuk az üzenetblokkokkal transzformált IV-t. Ezt a vektort még egy utolsó transzformációnak vetjük alá, a véglegesítő függvénynek (*finalisation function*). Ez a függvény véglegesíti a vektort, adja a tényleges hash-t, ez a lépés még jobban megerősíti a hash algoritmust, segít hogy jobban összekevertek legyenek a bitek és az algoritmus eleget tegyen a lavina effektnek vagy csak szimplán egy egy-irányú tömörítő függvény összetömöríti a kevesebb bit-re.

3.7. Block cipherek

A blokk cipher egy szimmetrikus kulcsos titkosító függvény. Két bemenettel dolgozik, a titkos kulccsal és a adatblokkméterekre darabolt üzenet egy blokkjával. Ezek alapján adja meg a fix méretű kimenetet. Minden blokk ciphernek létezik inverz függvény, szóval reverzibilisek. Ha egy üzenetet így titkosítunk, akkor a titkos kulcs ismeretében meg is tudjuk fejteni a titkosított üzenetet.

3.7.1. Definíció (Blokk cipher)**.** [8] Legyen E a blokk cipher függvény, K a titkos kulcs, M az üzenet egy blokkja, akkor:

$$E_K^{-1}(E_K(M)) = M$$

Minden kulcsra nézve E egy permutációt képez M-hez, E egy bijektív leképzés. Ez a leképzés 2^n-iken permutációval rendelkezik, ahol n a blokkméret. A tipikus blokkméret 64 és 128bit volt, manapság 128bit-el dolgoznak a blokk cipherek. Minimum 80bites titkos kulcsot ajánlanak, így elgéséges védelmet nyújt a brute-force támadások ellen.

Azokat a blokk ciphereket amiket sokszoros egymás utáni használatra terveztek iterációs blokk ciphereknek vagy termelő ciphereknek (*product cipher*) nevezünk. Az ilyen iterációkat köröknek nevezzük. Általában 4 és 32 között szokott lenni a körök száma.

3.8. Block cipherekre alapuló hash algoritmusok

Manapság használt hash algoritmusok meghatározó részét blokk cipherekből rakják össze. Ezeknek az algoritmusoknak a felépítését, a blokk cipherek egymás után fűzését bizonyos elvekre fektetik (*Modes of operation*). Ezek az elvek foglalják össze, hogy a blokk cipherek milyen sorrendben követik egymást, és a bemenetüket mi képzi. A legismertebb hash függvények mint pl az *MD4, MD5, SHA-1* is blokk cipherekre alapulnak.

3.9. Kriptográfiai hash függvények összefűzése

Két különböző algoritmus által generált hash összefűzése, konkatenálása egy erősebb biztonságosabb hash-t adna az elképzelés szerint, de igazából a képzett hash nem lesz erősebb mint a két komponense. Példa:

$$H(x) = SHA1(x)||MD5(x)$$

Gyengesége a konkatenációnak az, hogy az iterációs hash generálási technikából következik, hogy ha 2-collision -t találhatunk a hash algoritmusokban akkor könnyedén képezhető n-collision is (n db különböző üzenetre ugyanazt a hash generálja). Ha n-collision -t tudunk képezni akkor elég nagy a valószínűsége, hogy találunk olyan üzenetet, ami a másik hash algoritmusnál is collisiont okoz.

Ennek ellenére hibás közeg átvitelnél tökéletesen biztonságosan lehet használni a módszert, ha azt akarjuk, hogy legalább az egyik hash legyen helyes.

3.10. Avalanche effect (Lavina effektus)

A kriptográfiában a lavina effektus[9] egy jellemző, amit elvárnak az algoritmustól, pl a blokk cipherektől és a hash algoritmusoktól. Ez az elvárás pedig nem más mint, hogyha akár egy bitje is a bemenetnek megváltozik, akkor a kimenet szignifikáns része (legalább a fele) kell hogy változzon. Maga a fogalom Horst Feistel-hez kötődik aki 1973-ban fektette ezt le, bár maga az elképzelés jóval jobban visszanyúl, méghozzá Claude Elwood Shannonhoz.

Ha egy kriptográfiai hash függvény nem teljesíti ezt az elvárt követelményt, akkor gyengének mondjuk. Az állandósága, illetve a gyenge változása miatt megbecsülhető, az előző lépés így könnyen megfejthető részben vagy egészben az eredeti bemenet (üzenet) is.

A lavina effektus az első és egyik legfontosabb kritérium amit szem előtt kell tartani egy blokk cipher, vagy hash algoritmus tervezésénél, pontosan ebből az okból kifolyólag nevezzük a legtöbb blokk ciphert, termelő (produt) ciphernek, hiszen több iteráción átesnek, így jól megforgatják a biteket, és ebből az indokból kifolyólag dolgoznak a hash algoritmusok nagy adatblokkokkal.

3.11. Plusz fogalmak

3.11.1. Definíció (Szigorú lavina kritérium (Strict avalanche criterion))**.** A kritérium (SAC) egy boolean függvény jellemző, ami a függvény teljességére épít, vagyis bármelyik bemeneti változó komplementerét vesszük hatással kell hogy legyen a kimenetre, minden bitnek 0,5 valószínűséggel kell megváltoznia.

3.11.2. Definíció (Bit függetlenségi kritérium (Bit independence criterion))**.** Ha a bemenetből, i bit megváltozik, a kimenetből választott két bit: j és k

vizsgálva egymástól függetlenül kell hogy változzon. Ennek igaznak kell lennie bármelyik i, j és k -ra.

3.11.3. Definíció (Plaintext)**.** A plaintext a transzformáció előtti információ reprezentációja. (Plaintextnek nevezzük az összes információt amit a küldő a vevőnek akar küldeni.)

3.11.4. Definíció (Ciphertext)**.** A titkosított információ, a transzformálás utáni formája az információnak.

3.11.5. Definíció (Diffusion)**.** A plaintext és a ciphertext közötti összefüggés.

3.11.6. Állítás. Egy jó diffusion-nal rendelkező ciphernek telejsítenie kell a szigorú lavina kritériumot (SIC).

3.11.7. Definíció (Confusion)**.** A titkos kulcs és a ciphertext közötti összefüggés.

3.11.8. Definíció (Hamming távolság)**.** [1] Legyen $x, y \in \{0, 1\}^n$, akkor

$$d(x, y) = \sum_{j=1}^{n} |x_j - y_j|$$

jelölés alatt a Hamming távolságot értjük.

4. fejezet

Hash algoritmusok az informatikában

Mint bevezetőben is említett, az informatikában a kriptográfiai hash függvényeket elég széles skálán használják.

Az MD4 bizonyítottan nem biztonságos, ennek ellenére rengeteg helyen használatban van. Viszonylag gyorsan számol hash-t a megadott értékekből és a számolt hash mérete sem nagy, így teljesen ideális, az átvitelt sem terheli. Peer-to-peer hálózatok előszerettel használja az MD4-et, checksum számításra, vagyis az átvitt adatok konzisztenciájának ellenőrzésére. A Microsoft Windows, NT-től kezdve használja a saját fejlesztésű hash algoritmusában, az NTLM-ben.

Az MD sorozatbeli MD4 után a következő lépés az MD5 volt. Ez a kriptográfiai hash algoritmus talán a legjobban elterjedt ebben az időben. Hasonlóan gyorsan számítható mint az MD4, és biztonságosabb is. Bár már találtak benne több hibát, még mindig nem mondható olyan módszer ami indokot adna az algoritmus teljes kivonására az informatikából. Legtöbb helyen, jelszavakat tárolnak vele, de több módosított változata is piacra került mint például a FreeBSD MD5.

A FreeBSD MD5 egy MD5 függvényre alapuló hash algoritmus. Az MD5-öt mint blokk ciphert használja több lépésen át, többszörösen kódolva vele az adatot. Az algoritmus csak annyira biztonságos mint maga az alapja vagyis az MD5. Nagy előnye a magas processzor igény, ami alatt előállítja a hasht.

Az SHA családot az NSA formálta, ezek közül az SHA-1 a legelterjedtebb. Biztonságos algoritmusnak készítették, és egyenlőre tartja is magát. Elég sok titkosító algoritmusban, protokollban használják, pár példa: SSH, IPsec, TLS, SSL, PGP. Ugyanígy használják checksum számításhoz is pl. Git-nél.

A továbbiakban még a MySQL régi algoritmusa lesz ismertetve. A régi MySQL-323 algoritmus csak az adatbázisban használt authentikációra való. Az adatbázis tervezők fejlesztették ki, így nem is lett túlságosan jó. Tanulva a hibáikból nem új hash algoritmust fejlesztettek, hanem egy jól beváltat választottak, az SHA-1 -et és azt módosították. Az új algoritmus neve is MySQL-SHA-1 lett ami hasonlóan a FreeBSD MD5-höz, csak felhasználja az SHA-1 -et a hash generálásban. Ez az algoritmus csak az authentikációhoz való.

5. fejezet

MySQL-323 algoritmus

A MySQL323 algoritmus egy specifikus hash algoritmus. A MySQL adatbázishoz fejlesztették ki a tervezők, a szimpla authentikációs protokolljuk kiegészítéseként, vagyis hogy ne tárolják a jelszavakat plaintext alakban.

Az algoritmusra neve, a MySQL323 jelölés a programot, az adatbázis szervert és a megjelenés verziószámát jelöli. Az függvény a 3.23-as MySQL verzióban bukkant fel először. Később ezt le is váltották a MySQL-SHA-1 -re, ami jóval nagyobb biztonságot ad a felhasználóknak, hiszen ez az SHA-1 algoritmus módosított változata. Az algoritmus implementációját megtalálhatjuk a MySQL server forráskódjában, hiszen open-source.

C implementációja (forrás: MySQL Server forrás):

```
void hash_password(ulong *result,
                   const char *password,
                   uint password_len)
{
  register ulong nr = 1345345333L, add = 7,
                nr2 = 0x12345671L;
  ulong tmp;
  const char *password_end = password + password_len;
  for (; password < password_end; password++)
  {
    if (*password == ' ' || *password == '\t')
      continue;
    tmp = (ulong) (uchar) *password;
    nr ≙ (((nr & 63) + add) * tmp) + (nr « 8);
    nr2 += (nr2 « 8) n̂r;
    add += tmp;
```

```
    }
    result[0] = nr & (((ulong) 1L « 31) -1L);
    result[1] = nr2 & (((ulong) 1L « 31) -1L);
}
```

Egy kétszer 32bites tömböt, a jelszót és a jelszó hosszát várja bemenetnek. Mint az algoritmusból elsõ ránézésre látszik, se a space se a tabulátor karakterekkel nem foglalkozik, kihagyja õket a titkosításból. Ez azt eredményezi, hogy bármennyi szóközt használhatunk a jelszóba, ugyanazt kapjuk eredményül, mintha nem raktunk volna bele egyet sem. Ez a tulajdonság a MySQL híres felhasználóbarátságából adódik.

Az algoritmus 2db 32bites IV-vel rendelkezik:

- 1345345333
- 0x12345671

A feldolgozás közben, a ciklus a jelszó minden karakterét felhasználja a transzformációkhoz. Egy-irányú tömörítőt nem használ, viszont blokk ciphernek egy saját fejlesztésű bináris transzformációt igen:

$$nr = nr \oplus (((nr\&63) + add) * tmp) + (nr << 8)$$

$$nr2 = nr2 + (nr2 << 8) \oplus nr$$

Mind a két transzformáció tartalmazza a XOR utasítást, ezért nem megbecsülhető az elõzõ állapot könnyedén. A titkos kulcs tudatában természetesen meg tudjuk mondani, itt az nr előző és új állapotait. Kisebb tanulmányozás után kiderül, hogy az nr hatással van az $nr2$-re, így $nr2$ függ az nr-től, viszont visszafelé ez nem igaz. Vagyis az algoritmus kimenetének az első 32bitétől függő lesz a második 32bit viszont ez fordítva már nem igaz. Ez a tulajdonság a következőkben lesz fontos.

Az algoritmus elég távol van a Merkle-Damgård elképzeléstől, hiszen nem bontja üzenetblokkokra a jelszót, és ezért nem is toldja ki, majd zárja le a blokkot.

Véglegesítő függvénynek nevezhetnénk az utolsó két sort, ami feltölti a tömböt.

Lavina effektushoz pár teszt érték:

uzenet0	0443 3721 4056 3c22
00001000100001100110111001000010	10000000101011000111100010000000
uzenet1	0443 34d2 4056 3bd3
00001000100001100110100110100100	10000000101011000111011110000000
0uzenet	3210 a8c1 0061 e596
01100100001000010101000110000010	00000000110000111100110000000000
1uzenet	7e36 f06a 5b2a b6d8
11111100011011011110000011010100	10110110010101010111000000000000

Az első kettő és a második kettő üzenet között csak 1-1 bit eltérés volt, ami a kritérium szerint minimum a bitek felét meg kellett volna hogy cserélje, ennek ellenére a bitcserék száma:

- elsõ kettő esetében: 12
- második kettő esetében: 29

az elvárt 32 helyett. Az is látszik, hogy a cserélődés száma a hossztól és a csere pontjának helyiértékétől is függ.

A kis teszt eredményei láttán kimondható, hogy az algoritmus gyenge.

6. fejezet

MD5 algoritmus

Az MD5 algoritmus az MD család tagja és az MD4 utódja. Az algoritmus 1991-ben készült el, Ronald L. Riverst az RSA egyik feltalálója keze által. Az MD5 az MD4-et volt hivatott leválltani, hiszen 1991-ben, az algoritmus már nem volt biztonságosnak mondható.

A publikáció után, már 2évvel rögtön találtak is hibát az algoritmusban. Pszeudó-ütköztetés lehetséges, ha az inicializációs vektorokat tetszés szerint változtatjuk. 1996ban jött a következő, de erősebb csapás az algoritmusra, ami megmutatta, hogy az MD5 által használt tömörítő függvény ütköztethető. Ettől a ponttól tartották az algoritmust kevésbé megbízhatónak és ajánlották, hogy SHA-1 -et használjanak helyette.

Mai napig számos támadás létezik az MD5 ellen, de egyik sem használható ki úgy, hogy komolyabb károkat lehessen vele okozni. A szakdolgozat írása alatt megjelent az MD6 algoritmus is, ami az SHA-3-ra pályázik, amire az NSA írt ki pályázatot.

Az MD5 talán jelenleg a legelterjedtebb, és leghíresebb kriptográfiai hash algoritmusnak tekinthető. Internetes szabványnak is bejegyezték, és megtalálható az RFC 1321 hivatkozási szám alatt.

A MySQL-323 algoritmushoz képest, az MD5 elképesztően nagy ugrás. Az algoritmus 128bites kimentet generál, bármilyen bemenetből. A bemenetet 512bites blokkokra bontja, és azokat transzformálja. A Merkle-Damgård elképzelésben lefoglalt blokkosításon kívül, még a *length-padding*-ot is magában foglalja, vagyis kiegészíti a bemenetet, hogy 512-vel osztható legyen a hossza. Ezen kívül rendelkezik *finalization* függvénnyel, ami lezárja az utolsó blokkot, beleírja a bemenet hosszát és 0x80 karakterrel lezárja. Ez a koncepció teljesen megegyezik a Merkle-Damgård elképzeléssel.

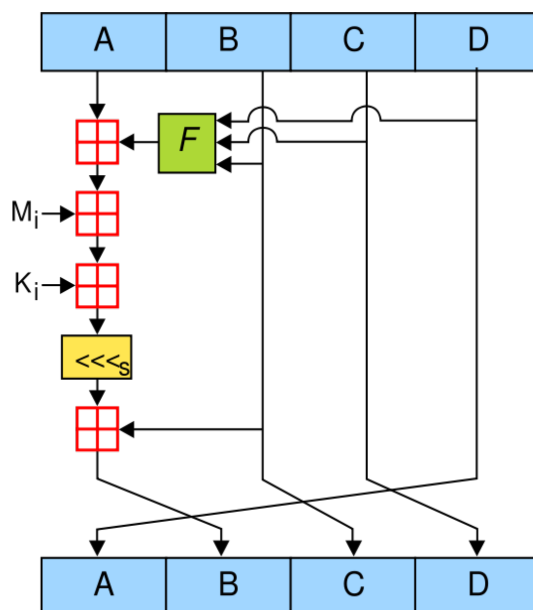
A hashelési algoritmus pszeudó kódja:

```
// 64 méretű tömbök a bitforgatáshoz és más transzfor-
  mációkhoz
var int[64] r, k

// az itt megadott mértékekkel fogja körbeforgatni a
  biteket az algoritmus a megadott lépésben
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}

// Algoritmus érdekessége, hogy a sinus és a 2
  hatványait használja fel a transzformációhoz,
  mint konstansok.
for i from 0 to 63
  k[i] := floor(abs(sin(i + 1)) * (2 pow 32))

// Inicializációs Vektor, 4elemmel
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

// Merkle-Damgård előírások teljesítése
length-padding
finalization

// A bemenet 512bitenként való feldolgozása:
  // 16db 32bites változó feltöltése az
    512bitnyi bemenettel
  for i from 0 to 15
    w[i] := (int)M[0]
  // belső IV-k inicializálása
  var int a := h0
  var int b := h1
  var int c := h2
  var int d := h3
```

```
  // szabvány szerinti 64lépés, 4váltó függvénnyel,
     változók cseréje
  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

  // a 64lépés után az IV-k változtatása, növelése a
     transzformált IV-kkel
  h0 := h0 + a
  h1 := h1 + b
  h2 := h2 + c
  h3 := h3 + d

// bitforgató függvény (ROL+ROR)
leftrotate (x, c)
  return (x « c) or (x » (32-c));
```

Az algoritmus az inicializáció, előkészítés után 64lépést tesz meg minden egyes blokkon. A megadott IV-ből kiindulva az 512bitet a definiált függvények szerint transzformálja, és 128bitbe tömöríti, ami sorban az új, a következő blokk IV-jét adja meg. A 64lépés, 4db nem lineáris, moduláris aritmetikán alapuló (túlcsordulással nem kell foglalkozni) függvényt használ:

Az algoritmus képpel szemléltetve:



6.1. ábra. MD5 algoritmus

$$
\begin{aligned}
F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
G(X,Y,Z) &= (X \wedge Y) \vee (Y \wedge \neg Z) \\
H(X,Y,Z) &= X \oplus Y \oplus Z \\
I(X,Y,Z) &= Y \oplus (X \vee \neg Z)
\end{aligned}
$$

Ezeket a függvényeket nevezhetjük egyenként az algoritmus blokk ciphereinek, és a körökben a termelő ciphereknek, hiszen a *product cipher*ek definíciója, hogy iterációban használt blokk cipherek legyenek.

Az algoritmus még használ egy *rotate* függvényt is, ami a bináris *ROR* és *ROL* függvény keverékeként fogható fel. Az elején megadott mátrixból, minden lépésben a megfelelő értéket kiválasztva, elcsúsztatjuk az IV megadott részét jobbra, illetve balra, így bitveszteség nélkül megkapjuk a megforgatott rész IV-t.

Végső pillanatban, amikor az összes blokkon lefutott az összes kör, már csak az 4db transzformált IV-t kell egymáshoz fűzni, és megkapható a hash végleges értéke.

Lavina effektushoz pár teszt érték:

uzenet0	a627719604b8ef5566de4fe21d8f93a0
01101001100011101110010001100101	10101010111101110001110100100000
01000111111100100111101101100110	00000101110010011111000110111000
uzenet1	717df6094a3bcd8936a08583be4d196a
10010000011011111011111010001110	10010001101100111101110001010010
11000001101000010000010101101100	01010110100110001011001001111101

Hamming távolság: 63

Egyetlen egy karakter változása is jól jellemzi, hogy a Lavina effektust, mint elvárást az algoritmus teljesíti.

7. fejezet

FreeBSD MD5 algoritmus

Ezt az algoritmust Poul-Henning Kamp híres FreeBSD fejlesztő fejlesztette ki. Az algoritmus az MD5-re alapszik. A FreeBSD MD5 lényege, hogy eléggé költséges, 1000 iterációt tartalmaz, amiben MD5 algoritmussal hasheli el a saját maga által gyártott hasht. Az algoritmus egy igen egzotikus kriptográfiai algoritmus, így jellemezni is nehéz. Pontosan annyira biztonságos, mint amennyire az MD5, hiszen az az alapvető építő pillére, úgyis jellemezhető mint blokk (*product*) cipher. A legtöbb Linux disztribúció, és természetesen a FreeBSD is ezt a hash algoritmust használja előszeretettel a jelszavak titkosításához, ebben a formátumba tárolják a shadow illetve master.passwd fileokban.

A FreeBSD MD5 *salt*olt (sózott algoritmus). A sózott algoritmusok lényege, hogy egy plusz adattal sózza, bolondítja, fűzi össze a bemenetet és azzal hasheli el. Így azt érhetjük el, hogy hiába vesszük kétszer ugyanazt a bemenetet, más lesz a hash értéke a kimenet, ha más volt a salt is. Unix rendszerek a saltot véletlenszerűen generálják a jelszó hashelésénél. Jelszavak, adatok újrahashelésénél, a salt tudatában, ugyanazon módon előállítható a meglévő hash (marad determinisztikus az algoritmus). újrahashelésénél, hogy a tárolt, eredeti hash-t kapjuk vissza.

Pszeudó kód:

```
md5_update(password)
md5_update("$1$")
md5_update(salt)

md5_update(password)
md5_update(salt)
md5_update(password)
```

```
md5_digest(hash)
md5_update(hash)

md5_digest(hash)

for i from len(password) to 0
   if (i & 1) == 1 then
      md5_update(0)
   else
      md5_update(password[0])

for i from 0 to 999
   if (i & 1) == 1 then
      md5_update(password)
   else
      md5_update(hash)

   if (i % 3) != 0 then
      md5_update(salt)

   if (i % 7) != 0 then
      md5_update(password)

   if (i & 1) == 1 then
      md5_update(hash)
   else
      md5_update(password)
   md5_digest(hash)

finalhash := transform_to64(hash)
saltedhash: = "$1$"+salt+"$"+finalhash
```

Az algoritmus rengetegszer, egy szisztéma szerint elhasheli saját magát illetve a saltot. *md5_update* függvény a hashelni való blokkokat rendezi, mindig hozzáírja a megfelelő bemenetet, az *md5_digest* pedig kiszámolja az md5 hasht. További lépésekben nem csak a bemenetet, a saltot, hanem a hash-t is elhasheli mégegyszer és egy 1000 lépéses iterációban még megismétli. A végén a szerző saját maga által definiált vektorban található 64. karakter szerint átalakítja a kapott MD5 hash-t és beformázza a megadott alakra.

Az átalakítás a következő:

- fogunk meghatározott 3db byteot az MD5 hashből
- ezt 4db-ra 6-6 bitekre vágjuk és ennek megfelelőket kikeressük a vektorból

előző két lépést 6-szor megismételjük (utolsóban lépésben 1byte-tal, nem 3-mal)

Mivel a FreeBSD MD5 az MD5 algoritmusára támaszkodik, ezért minden kriptográfiai jellemzőjével rendelkezik. A lavina effektust garantálja az alap hash algoritmus, ezért ennek is rendelkeznie kell vele.

8. fejezet

SHA-1 algoritmus

Az SHA-1 algoritmus az SHA függvények családjába tartozik. Az SHA rövidítés megfelelője a Secure Hash Algorithm, vagy Biztonságos Hash Algoritmus. Az SHA függvényeket az NSA (National Security Agency) fejleszti, ellenőrzi és a NIST (National Institute of Standards and Technology) publikálja, vezeti be. 1993-ban publikálták az első SHA függvényt, de titokzatos körülmények között rögtön vissza is hívták, és javítottak a benne lévő tömörítő függvényben egy bitforgatást. A javítást azóta sem indokolták. Ez a javított algoritmus kapta az SHA-1 nevet, és a módosítatlan, eredetire pedig SHA-0 -val szokás hivatkozni.

Az SHA-1 algoritmus hasonló felépítésű mint MD5, ennek oka az MD4, hiszen mind a két algoritmus ebből született, és erre építkezik. Még az MD5 128bites hash-t generál, az SHA-1 160biteset. Az MD5 viszontagságai miatt, az SHA-1 volt a legmegbízhatóbb hash algoritmus, de mára ez is elmúlt. A NIST 2010-re tervezi az SHA-1 leváltását SHA-2 variánsokra, mert mára már az SHA-1-ben is számos matematikai gyengeséget valószínűsítettek.

Az algoritmus mint említett volt bármekkora bemenetből 160bites kimenetet generál. A bemenetet 512bites darabokra bontja, majd a Merkle-Damgård elképzelés szerint kibővíti a megmaradt utolsó darabot nullákkal, lezárja és beleírja a bemenet eredeti hosszát.

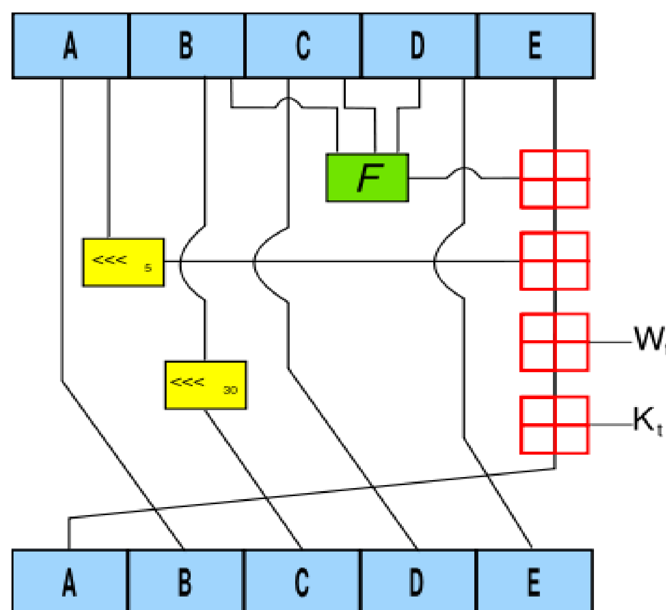
A hashelési algoritmus pszeudó kódja:

```
// Inicializációs Vektor, 5elemmel
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
```

```
h4 = 0xC3D2E1F0

// Merkle-Damgård előírások teljesítése
length-padding
finalization

// A bemenet 512bitenként való feldolgozása:
for each 512-bit chunk of message
  // 16db 32bites változó feltöltése az 512bitnyi
    bemenettel
  for i from 0 to 15
    w[i] := (int)M[0]
  // még 64db 32bites szó készítése az eredeti
    bemenetből
  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)

  // belső IV-k inicializálása
  a = h0
  b = h1
  c = h2
  d = h3
  e = h4

  // szabvány szerinti 80lépés, 4váltó függvénnyel,
    változók cseréje
  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

// a 80lépés után az IV-k változtatása, növelése
    a transzformált IV-kkel
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
```

Az algoritmus képpel szemléltetve:



8.1. ábra. SHA-1 algoritmus

Az algoritmus 4db IV-vel rendelkezik, ezek inicializálás után a bemenetet, 512bites darabokra vágja, majd a Merke-Damgård elképzelésnek megfelelően felkészíti a transzformációra. Az 512bitet 16db 32bites változóban tárolja el, és ezen felül még készít 64db 32bites értéket az eredeti 16ból. Az eltárolt 80értéket 80lépésen keresztül, 20-20 lépésben 4db blokk

cipherrel transzformálja. A 80lépés mindegyikében felcseréli a változókat, így biztosítva a teljes mozgatást, és indukálja a lavina effektust. 80lépés után az eredeti IV-hez moduláris aritmetikával hozzáadja az 5db transzformált értéket, így kapva a kimenetet az 5*32, 190bites hash-t.

Lavina effektushoz pár teszt érték:

uzenet0	93b2af4852073975b3a76144ac3f145de6429ea3
00010010111101010100110111001001	10101110100111001110000001001010
00100010100001101110010111001101	10111010001010001111110000110101
11000101011110010100001001100111	
uzenet1	faeae2bf8cc311fcb7d054dc9711e8018aa9e61f
11111010100011101010111010111111	00111111100010001100001100110001
00111011001010100000101111101101	10000000000101111000100011101001
11111000011001111001010101010001	

Hamming távolság: 84

Egyetlen egy karakter változása is jól jellemzi, hogy a Lavina effektust, mint elvárást az algoritmus teljesíti.

9. fejezet

Algoritmusok összehasonlítása

Három ismeretett algoritmus pár adata táblázatba foglalva:

Név	Bit hossz	Blokk méret	Lépések száma	Hamming távolság
MySQL-323	64	8	$O(n)$	29
MD5	128	512	64	63
SHA-1	160	512	80	84

Jól látható, hogy a MySQL-323 a kis teszt alapján nem teljesíti a Lavina effektust, ezen felül kellően kicsi kimenettel bír. Még az MD5 és az SHA-1 mind a kettő teljesíti, és nem csak a Lavina effektust, a Merkle-Dåmgard előírásokat is, az SHA-1 nagyobb kimeneti hosszal rendelkezik.

10. fejezet

Hash törési módszerek

A szakdolgozat csak az előkép (*preimage*) ellenállósággal foglalkozik, illetve annak gyengeségével, módszerek alkalmazásával, amik segítenek megtalálni a $H(x)$ hash értékhez az x értéket. Jelenleg elég sok módszer van a jelszavak megfejtésére. A legegyszerűbbektől kezdve, vagyis hogy minden lehetséges variációt (vagy legalábbis egy nagyobb részhalmazát) kipróbálunk, a bonyolultabbakig ahol szociológiai, pszichológiai modellekre alapozva redukáljuk a kulcsok, plaintextek lehetőségeinek számát és ez eredeti halmazhoz képest viszonyítottan kicsi részhalmazt próbálunk végig. Jelenlegi legbonyolultabb metódusok a Trade-off módszerek, ahol az alapvető erőforrásokat mint a tárhely és teljesítmény ötvözve használjuk ki.

10.1. Bruteforce

Ez a módszer a lehető legegyszerűbb. Mint neve is mutatja a nyers erőre, a teljesítményre vonatkozunk csak. Egy vagy több hash birtokában fel kell mérnünk a kritériumokat, amik a következők megválasztásából áll:

Karakterkészlet: l
Kezdő- és véghossz: s, f

Ezek tudtában könnyen kiszámolható a variációk lehetősége:

$$\sum_{i=s}^{f} |l|^i$$

A variációk számából, illetve a függvényből jól látható, hogy egy exponenciális problémával állunk szemben. A kulcsok hosszának növekedésével arányosan nő a kiszámolandó hash-ek száma.

GRAPH

Ha a karakterkészlet csak az angol ABC kis- és nagybetűiből és számokból áll (*mixed-alpha-numeric*), akkor 1től 8 hosszig a variációk száma:

$$\sum_{i=1}^{8}(26*2+10)^i = 221919451578090$$

9 hosszig:

$$\sum_{i=1}^{9}(26*2+10)^i = 13759005997841642$$

10.2. Külső szabályok (External rules)

Jobb jelszótörő alkalmazások engedik kulcs generálási szabályok felállítását. Johnban is megtehető ez a john.conf szerkesztésével.

A konfig fileban megadható, bármilyen generálási szabály, ami szerint a kulcsok egymás után következi fognak. Ezt két féleképpen tehetjük meg a Johnba. Egyik a reguláris kifejezésekkel való leírás, ahol megadható, hogy a legenerált kulcsot miszerint változtassa meg és így kapjunk új kulcsot. Kibővíthetjük, levághatunk belőle részeket, vagy akár fix hosszon változtathatjuk is. Az emberek jelszó megadási-megjegyzési módszereit tanulmányozva könnyen felállítható egy modellt, ami jelentősen redukálja a kulcsok számát. Így például:

- maximális hossz a 7 karakter
- kis és nagy betűk nincsenek keverve
- magán-, esetleg mássalhangzók számokra cserélése (leet speak -> 1337 5p34k)

A kis és nagy betűk keverése által, két különböző halmazt kapunk karakterkészletre:
kisbetűk + számok és nagybetűk + számok

A karakterkészlet részekrebontásával a kulcsok variációjának a száma rendkívül lecsökken:

$$2 * \sum_{i=1}^{7} 36^i$$

A kiszámolt kulcsokat egy reguláris kifejezéssel könnyen át lehet alakítani, úgy hogy a modellnek megfeleljen, vagyis a megfelelő karaktereket lecseréljük a kívántakra.

Reguláris kifejezéseken kívül, mint említett volt algoritmusokat is lehet írni, amivel más féleképpen képezhetünk kulcsokat. Johnnál ezt is a john.conf konfig fileba kell megadni, C-hez hasonló szintaktika szerint. Bár ez a lehetőség eléggé korlátolt, ennek ellenére sok minden megoldható benne. A konfigban megadott algoritmust a Johnban lévő belső interpreter értelmezi, majd fordítja byte-kódra. Ez módszer a generáláshoz igen lassú és programozási szempontból korlátozott is, hiszen nem bővelkedik funkcionalitásokban. Futás közben, egy köztes byte-kódra fordul ami lassítja is a generálási procedúrát.

John the ripper használatánál ez *–external=rulename* paraméterrel történik.

10.3. Szólisták (wordlists)

Statisztikákból következtethető, hogy az ember a jelszavát a saját anyanyelvéből választja ki a legszívesebben. Az interneten széles skálája elérhető a szólistáknak (wordlist) amiben az adott nyelv, legtöbbet használt szavai, esetleg összes szava megtalálható. A jelszótörő programnak megadva ezt a szótárfájlt kipróbálja, elhasheli az összes benne található szót.

John the ripper használatánál ez *–wordlist=filename* paraméterrel történik.

10.4. Előgenerálás

Előző pontokban leírt módszerek a CPU vagy más számítási eszközök sebességére alapoznak, így nem sok memória, vagy tárhely igényük van. Ez a módszert ezt hivatott felváltani. Még az előzőekben minden egyes alkalommal ki kellett számolni a hash-t és úgy összehasonlítani a meglévővel (megfejteni vágyottal), így ebben a módszerben ezt már csak egy lépésre redukáljuk, vagyis az összehasonlításra. Ahhoz, hogy csak ezt a lépést kelljen megtenni minden variációt egyszer le kell generálnunk, majd

eltárolni. A következőkben ezt az eltárolt adatbázist csak olvasnunk kell tudni, majd a kiolvasott sorokat össze hasonlítani a meglevő hashekkel.

Sajnos ennek a módszernek is vannak hátrányai. Először is nem működik hatékonyan a sózott (salt) hashek ellen, hiszen azoknak a száma a salt variációnak számával szorzódik. Másrészt, hatékony tárolás mellett is a példában 128biten hash és 8byteon tárolt plaintext mellett 1-től 8 hosszig a variációk saltolás nélkül 293petabyte-ot igényelnének. Saltolással ez még multiplikálódik a salt variációinak számával.

$$2^{|salt|} * \sum_{i=s}^{f} |l|^i$$

10.5. Trade-off módszerek

A trade-off módszerek lényege, hogy nem csak egy erőforrásra a számítási teljesítményre, vagy a tárolókapacitásra hagyatkozik, hanem több erőforrást keverve használ. Így az előző pontokban a bruteforce és az előgenerálás módszereket ötvözve fel lehet használni, vagy akár még pluszba, más módszereket is belekeverve hatékonyabbá tenni.

10.5.1. Cryptanalitic time-memory trade-off

A trade-off[3] módszerek alapját 1980-ban Martin Hellman fektette le. Feltalált egy olyan módszert, amivel időt és memóriát közösen költve ugyanazokat spórolhat meg. Az előzőekben felsorolt módszereket ötvözte és dolgozott ki egy módszert. 1982-ben ezt a módszert Ronald L. Rivest az MD család kiötlője tovább javított.

A módszer két függvényt használ, a hashelő eljárást: H és a redukciós eljárást: R. Ezekből a függvényekből, láncokat alkotunk és a lánc első és utolsó elemét eltároljuk:

$$x_1 \xrightarrow{H} h_1 \xrightarrow{R} x_2 \xrightarrow{H} h_2 \xrightarrow{R} \cdots \longrightarrow x_t \xrightarrow{H} h_t$$

Sajnálatos módon a láncok, még ha különböző kulcsnál is kezdődnek, keletkezhetnek ütközések és ezáltal több lánc össze is olvadhat, részben tartalmazhatja ugyanazt a részláncot. m a láncok száma, t a láncok hossza és N a lehetséges kulcsok száma:

Kulcs találati valószínűség:

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

A tábla hatékonysága a méretével gyorsan csökken. Ahhoz, hogy a sikeresebb legyen a módszer, több tábla generálása ajánlott. l darab táblával a találati valószínűség:

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

A táblákkal új redukciós függvény is jár így hiába ütköznek pontokban, sosem fognak összeolvadni egy másik tábla láncaival.
A megadott hash keresésének módszere a következő:
a keresendő hasht egy lánc első elemének tekintjük, és felépítünk belőle a táblák lánchosszúságának megfelelő hosszú láncot. Ha a lánc felépült és az elemei el vannak tárolva, könnyedén kereshető egyezőség a táblák végpontjai és az előbb készített lánc között egyezőség. Ha van egyezőség, akkor a hashez a keresett kulcs a tábla azon láncában van, ahol megtaláltuk a végpont egyezőséget. A talált láncot újra legenerálva megkapjuk a keresett kulcsot.

Hamis jelzéseknek nevezzük ha találunk olyan végpontot ami egyezik a láncunk egy részével, de a kulcs mégsincs a talált láncban, vagy ha a táblán belül a lánc egy másik lánccal összeolvad. Ennek eredménye, hogy lehetséges hogy nem csak egy láncot kell legenerálni.

Ronald L. Rivest a módszer fejlesztésében segítkezett, pár ötletet hozzátett. Az ötleteinek egyike, hogy a láncok végét jelöljék megkülönböztetett pontok, így könnyedén le lehet zárni a láncokat, és az esetlegesen egymásba olvadó láncokat is fel lehet ismerni.

10.5.2. Rainbow Tables (Szivárvány táblák)

A szivárvány táblák[4] az előző trade-off módszer tovább fejlesztése. Az eredeti alapötlet optimizálása, javítása, új ötletekkel való bővítése. Ezt az eljárást Philippe Oechslin találta fel és prezentálta a CRYPTO 2003 konferencián. Az eljárás olyan jól sikerült, hogy rengeteg helyen használják, számtalan publikáció és prezentáció jelent meg róla számos nyelven. Elosztott rendszereket írnak a táblák generálására és abban való keresésre. Nevét valószínűsíthetően a láncokról kapta, amit minden pontban megjelölve egy színnel, egy szivárvány szerű átmenet kapható.

A Rainbow Tables alapvető újítása az előző trade-off módszerhez képest, hogy t hosszúságú láncokat használ, mint elődje de $t-1$ redukciós függvénnyel.

$$x_1 \xrightarrow{H} h_1 \xrightarrow{R_1} x_2 \xrightarrow{H} \cdots \xrightarrow{R_{t-1}} x_t \xrightarrow{H} h_t$$

Ez azt eredményezi, hogy minden lépésben, más redukciós függvénnyel dolgozva, más más eredményt kap. Hiába van collision, a redukciós függvény a következő lépésben különbözni fog, így a készített plaintext is, egyetlen esetet kivéve, amikor két lánc ugyanabban pontjában történik az ütközés. Ha két lánc ugyanazon pontjában történik ütközés, aminek valószínűsége igen alacsony ($p = \frac{1}{t}$), akkor a végpontjaik meg fognak egyezni. Egyező, összeolvadt láncoknál a módszer eldobja az egyiket és újat generál.

A sikeres találat valószínűsége egy $m \times t$ -s táblában (m a láncok száma, t a láncok hossza).

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

ahol

$$m_1 = m, \quad m_{n+1} = N(1 - e^{-\frac{m_n}{N}})$$

A táblában keresés $O(t^2)$ számítási kapacitású. A keresett hash-ből, az $R_t - 1$ készítünk egy plaintext-et, ha ez megegyezik az egyik végponttal, akkor újragenerálva a láncot (tudva az 1. elemét) megtaláljuk a keresett plaintext-et ami a hash-hez tartozik. Ha nincs a végpontok halmazában, akkor előbbi redukciós függvénnyel végignézzük a láncot, és ezt addig tesszük, még nem találunk végpontot.

Pár előny az eredeti módszerhez képest:

- a felkeresések száma csak t-től függ.
- a végpontokból következtethető az összeolvadás
- fix hosszúak a láncok

10.5.3. Markov szűrő

Ezen módszer lényege az előzőek teljes keverése[10]. A szűrő egy hibrid algoritmus, ami több előzőleg ismeretett módszert alkalmaz, így:

- szociális, pszihológiai modellek (külső szabályok)
- trade-off módszerek, rainbow tables

10.5.1. Definíció (Markov lánc)**.** $\{X_n\}$ valószínűségi változók sorozatát, Markov láncnak nevezzük ha az alábbi feltétel teljesül rá:

$$P(X_{n+1} = x|X_n = x_n, ..., X_1 = x_1) = P(X_{n+1} = x|X_n = x_n),$$

ahol $n \in \mathbb{N}$, $x_i \in S$ $(i = 1..n), S \subseteq \mathbb{R}$

Egy meghatározott nyelvtannak megfelelően, Markov láncok segítségével meghatározza a szavak formázási szokásait, és aszerint alakíti ki értelmes, vagy értelmesnek látszó szavakat. Ezeket a szavakat véges determinisztikus automatákkal átalakítja. A módszer hatásossága abban relylik, hogy még a brute force és az eddigi trade-off módszerek, az egész kulcsteret magukba kellett, hogy foglalják, addig ez a módszer nem a teljességre, hanem a teljesítményre törekedik. A modellnek megfelelően legenerálja a valószínűsíthetően értelmes szavakat hossztól függetlenül és eltárolja. A tárolás módja a szivárvány tábláknál ismeretett módszer, pár különbséggel.

Ha a modelleket jól mértük fel, és abból jól alkottuk meg a véges determinisztikus automatát, akkor egy megfelelő akár élő nyelvtan szerint, változó hosszúságokkal tudunk valószínűsíthetően értelmes szavakat generálni, amiket az emberek használnak, használhatnak. Ezek a variációk egy töredék részét foglalják magukba az egész kulcstérből, ezért a módszer sosem lesz teljes értékű, teljesen megbízható.

11. fejezet

John the Ripper rövid bemutatása

A John the Ripper egy nyílt forrású jelszó törő. Az Openwall project terméke, aminek a lényege a biztonságos nyílt operációs rendszerek (pl: Linux), és más nyílt forráskódú programok biztonságossá tétele. A John the Rippert (továbbiakban csak John, vagy jtr) Alexander Peslyak írta meg 1996-tól kezdve 2004-ig, és azóta is gondozza a forrását.

A John legnagyobb előnye, hogy egy hozzá szakértő, és szakmában lévő írta vagyis Alexander, aki máig az Informatikai Biztonságtechnikában dolgozik. Maga a John jelen pillanatban az 1.7.3.1-mas verziónál jár, amiben 6db hash algoritmus kódja található. A program platform független az összes elterjedt és pár mára már kihaló félben lévő operációs rendszerre, architektúrára is lefordítható. Operációs rendszerek listája:

- Linux
- FreeBSD
- OpenBSD
- NetBSD
- AIX
- Mac OS X
- Solaris (SunOS)
- Sco Linux
- HP-UX
- Irix

- DOS
- BeOS
- Windows (cygwin)

A program moduláltan íródott, így könnyedén készíthető hozzá bármilyen típusú hash algoritmus, kriptográfiai hash algoritmus C nyelven.

Könnyedén lehet több fajta módszerrel hasheket, hash listákat megfejteni vele. Pár fontosabb futási mód:

- test: A belefordított modulok tesztje, itt a test struktúrában található plaintexteket titkosítja majd hasonlítja össze a párukkal, a ciphertexttel (hashsel). Ezt megismétli számos alkalommal, ha nincs hiba akkor a modul hibátlannak mondható.
- format: Megadható paraméterként a modul neve, ami a használni kívánt hash algoritmust tartalmazza.
- wordlist: Paraméterként meg kell adni egy file-t, ami a szólista lesz. Ezekben található szavakat próbálja végig, külső szabályok szerint transzformálva, vagy eredeti formában (Lásd x. fejezet)
- session/restore: Session hozható létre, majd később a restore-al visszaállítható a megszakított állapot, így a törést nem kell újra kezdeni.
- external: Külső fileba megírható bármilyen kulcs (plaintext) generáló algoritmus, amit a john forrásában található interpreter értelmez, majd futtat. Így könnyedén készíthető, olyan külső modul ami csak bizonyos szabályok szerint generál kulcsokat.

Még manapság is nagyon felkapott ez a jelszótörő alkalmazás, hiszen ez a legjobban megírt, optimizált és ráadásul nyílt forrású jelszótörő program. A sikerének titka a modularizálás, bővíthetőség. Jómagam is jelenleg 2 javított (gyorsított) algoritmussal büszkélkedhetem a Johnhoz. Az eredeti (vanilla) forrás mellett található egy úgynevezett Jumbo patch is, ami a külső contributorok (közreműködők) által írt patcheket, modulokat gyűjti össze és integrálja a john legújabb verzióiba. Ez a patch ma is számos modult tartalmaz.

12. fejezet

Optimalizálás

Mind a szimpla és mind a kriptográfiai hash algoritmusoknál szükséges, hogy a metódus robosztus legyen. Szimpla hash algoritmusoknál megkívánható, hogy ne a rekeszek kiválasztásával menjen el a processzor idő, ugyanígy a hibadetektáló kódok generálásánál, lenyomatok készítésénél, ellenőrzésénél is fontos az idő mint tényező. A kriptográfiai hash algoritmusoknál ez egy jellemző is, amit megadnak az ismeretetőjükben is.

Vannak hash algoritmusok, mint pl a FreeBSD MD5 is, ahol direkt a lassúságra, törekedtek, így több mint ezer MD5 hashelést végez az algoritmuson belül a metódus, ezzel biztosítva a lassú generálást, újragenerálás. Ha egy FreeBSD MD5 hash-t kívánunk megfejteni bruteforce vagy hasonló módszerrel, akkor a kulcsokhoz rendelt hash generálása rengeteg időt igényel, így a lehetséges variációk mellett, rendkívül lassan tudható meg az eredeti kulcs.

Mint említett volt, a John The Ripper kiváló keretrendszert ad a hash algoritmusok implementálásához, és azok által gyártott hashek megfejtéséhez. A hash-ek megfejtésénél nem mindegy milyen algoritmussal vannak titkosítva, és főleg nem mindegy, hogy hogyan vannak azok az algoritmusok implementálva. Minél jobban, optimalizáltabban implementált egy algoritmus, annál több hash-t tud egységnyi idő alatt legenerálni.

Ezen felül a kriptográfiai hash algoritmusok nagy része rendelkezik sebezhetőségekkel, amik matematikailag bizonyíthatók, vagy már bizonyítottak. Ezen sebezhetőségek felfedezése, felismerése majd implementálása hihetetlenül meggyorsíthatja a törési folyamatokat.

12.1. Architektúrális optimizálás

Implementáláskor a legfontosabb szempont az architektúra. Ennek ismerete nélkül, nehézkes jól elvi optimizált kódot írni. Bár a gyakorlati optimizálást manapság a fordítók már nagyon jól elvégzik, mégsem bírálják a programozó által írt kódot felül.

Unix rendszerek alatt a legelterjedtebb C fordító a gcc. Ennek a fordítónak 4db optimalizálási szintje van[5]:

- 0: nincs optimalizálás, a lehető legjobban hasonlító kódot generálja a forráskódhoz hasonlítva. Debuggoláshoz ez a legjobb, hiszen könnyen felismerhető az programozó által írt kód.
- 1: a legelterjedtebb optimalizálásokat használja, speed-space trade-off-okat, így a binárisok kisebbek és gyorsabbak is lesznek mint 0-ás szinten.
- 2: a kettes szint további optimalizálást ad az egyeshez képest. A bináris méretét és memóriaigényét nem változtatja jelentős mértékben, viszont a fordításnál több idő és memóriát igényel. Itt már elemzi a kódot és az adatkezelést, majd olyan sorrendet határoz meg az instrukcióknak, amik alacsonyabb futási időt, viszont ugyanazon eredményt adják. (*Megjegyzés: GNU csomagok ezt a szintet használják alapértelmezetten*)
- 3: Előző kettő szint optimalizálási módszereit magában hordozza, viszont itt az elért eredmények költsége már drágább, több tárhelyet igényel a készült bináris. Többek között ez a szint eltünteti a függvények hívásait, a hívások helyére bemásolja a tartalmukat, így nem kell a processzornak, operációsrendszernek, stb. az ugrásokkal, kontextusváltásokkal törődni, szimplán csak szekvenciálisan futtatni a kódot.

A gcc a felsoroltak alapján tud optimalizálni, bár az utolsó szint kecsegtet a legjobb sebességgel, ami jelent helyzetben a legszükségesebb, mégis azt a veszélyt rejti magában, hogy esetenként lassabb kódot eredményez. Hiába próbál a gcc mint fordító mindent megtenni az optimalizálás területén, még mindig maradnak olyan elvi hibák amiket csak egy programozó javíthat.

Nem szabad elfelejteni, hogy a legtöbb hash (minden ismeretetett) algoritmus a végeredményt string formátumban adja, viszont az csak karaktertömbös reprezentációja a hashnek, ami egy számérték. Ha egy implementálni vágyott algoritmus is ilyen, akkor érdemes a kapott hasheket

számértékben tárolni, és a generált kulcsokat ugyanígy megkapni a hashelés után. Ha a két érték megegyezik, akkor megtaláltuk a kulcsunkat, vagy legalábbis egy mását, amivel ütközik. Összehasonlításra érdemes kivonást, vagy esetleg *XOR* (kizáró vagyot) használni, hiszen ezen utasítások a processzor alapjai így biztos, hogy gyorsan elvégezhetőek.

A lehető legjobb döntés a stringek és string műveletek, és több mint egy regiszter méretű adatok, struktúrák mellőzése. A processzor az alap műveletekkel, esetleg a plusz utasítákészletekkel tud a leggyorsabban eredményt adni, még a stringeknél vagy nagyobb adatoknál ez több műveletet igényel, így lassabb is. Ha mégis szükség van ilyen utasításokra, adatokra, adatstruktúrákra, akkor érdemes saját függvényt írni és használni, ami minden felesleges ellenőrzés nélkül csak a lényeget ellenőrzi. Használhatjuk a *memcmp*, *strcmp* függvényeket, de írhatunk sajátot is, hiszen a *strcmp* minden lépésben keresi a 0 értéket, ahol vége a stringnek, így lassítja a keresést. Ha csak azt akarjuk tudni, hogy különbözik-e a két string és semmi más plusz információt, akkor érdemes int-ekké alakítani a stringet, majd kivonni az értékeket egymásból.

Vannak esetek, amikor a kulcsokban lévő különbséget keressük, az előző kulcshoz képest mennyit változott az adott kulcs akkor string összehasonlítást kell végeznünk, majd megmondani, hogy hol változott a kulcs, erre érdemes saját függvényt írni, ami a rövidebb string hosszáig megy vagy változásig, és nézi mind két string adott pontjában az értékeket. Ez a függvény minimális lépésből, és instrukció számmal, előre tudott hosszal megmondja pontosan amit tudni akartunk.

Fontos megemlíteni kis kitérőként azokat az eseteket amikor nem egy, hanem több hash értéknek keressük a hozzá tartozó kulcsát. Ilyen esetekben a tárolás evidens módon több memóriát igényel, és keresés is több időt vesz igénybe. Drasztikusan tudja a növelni a törés idejét és instrukció számát a helytelen tárolás. A Jtr-ben létezik olyan lehetőség, hogy a keresett hash-eket hash táblában tároljuk el. Ha ezt megtesszük, akkor nem kell minden lépésben N hash esetén N összehasonlítást végezni, hanem csak $\frac{N}{r}$ számút ahol r a rekeszek száma, ha egyenletes eloszlással rendelkezik mind a hashtáblához hashelő függvény a hashekre nézve.

Ezen módszerek és a következő kettő pont implementálva található a csatolt mellékletben, teszt eredmények pedig a következő fejezetben.

12.2. Optimalizálás memóriával

Előzőekben ismertetett MySQL-323 hash algoritmus egyik nagy buktatója, hogy ha egy adott n hosszú kulcshoz ismerjük a hash értéket és egy válto-

zót (*add*), akkor a hasht IV-ként felhasználva, egy lépésben megmondható bármelyik $n + 1$ hosszú kulcs, aminek az első n karaktere megegyezik az eredeti kulccsal. Ennek ismeretében ha a karakterkészlet l hosszúságú, és n_0 hosszú kulcsokat törünk, felírható egy n_0 mélységű, l ágú teljes fa.

Ha a fát bővíteni szeretnénk, elég kikeresni a helyes ágakat, lejutni a fák leveleihez, majd onnan indulva felépíteni az újabb ágat, az új n_1 $(n_1 > n_0)$ hosszal. A levél keresését leszámítva $O(n_1)$ helyett, csak $O(n_1 - n_0)$-t kell megtenni.

Ezzel a módszerrel lényegében elérhető, hogy minden kulcs, inkrementális bruteforce módszere esetén csak egyetlen egy iterációt igényeljen, és ne a hosszával arányosat.

12.3. Optimalizálás hash csonkolással

Egy másik módszer amivel gyorsíthatjuk a törési folyamatot a hash csonkolás. Ez a módszer a processzor regisztereinek hosszából származtatható. Jelenleg a legelterjedtebb architektúra az x86-os, ami 32bites regisztereket használ. Természetesen vannak újabb és régebbi architektúrák is amik 64 vagy más hosszal dolgoztak, de az átlag embert elért architektúra az szinte mindig 32bites volt. Ennek eredménye, hogy a fejlesztők, hash algoritmusok feltalálói is az algoritmusokban 32bites csonkokra bontották a hasheket (mint minden ismeretett hash algoritmusnál), majd azokat a csonkokat (IV, *transzformált* IV) transzformálták, és a végén konkatenálva hexadecimális formátumban reprezentálják.

A leggyorsabb összehasonlítási módszer, ha két regiszter tartalmát kivonjuk egymásból, esetleg *XOR*-oljuk, és így ha 0-át kapunk akkor egyeztek. Viszont egy 32bites regiszterbe nem fér bele 64, 128, vagy 160bitnyi adat így kénytelenek vagyunk részenként összehasonlítani őket.

A törési folyamatot felgyorsíthatja ha több egymástól független ellenőrzést teszünk bele, és sikeres találat esetén ellenőrizzük sorban ezekkel a metódusokkal a hasheket. Ha valamelyik metódus nem találja egyezőnek a tárolt és a generált hash-t akkor folytatódik a keresés, ha minden metódus egyezőnek találja akkor megvan a keresett hash vagy egy neki megfelelő collision. Első lépésben elég az első 32bitet ellenőrizni. Ezzel a módszerrel egy normálisan megalkotott hash algoritmus már a lavina effektus miatt valószínűleg nem talál collisiont. Ha mégis talál, akkor végig kell nézni az összes 32bites blokkra.

Érdemes még analizálni a hash algoritmus és függetlenségeket, függőségeket keresni benne. A MySQL-323 algritmusban érdekes módon az *nr2*

érték mindig nr és *nr2*-től függ, viszont *nr* kiszámításához semelyik lépésben nem igényeltetik *nr2*. Ennek tudtában megalkotható az az algoritmus ami csak a hash első 32bitjét számolja ki, a második 32bitet nem. Ugyanezzel a módszerrel lecsokkenthető a 64lépéses MD5 algoritmus 61lépésre, hiszen az utolsó 3lépésben csak az utolsó 3 32bites blokkot számolja ki.

A két hash egyezőség ellenőrzéséhez megalkothatóak ezek csonkolt hash függvények, amik csak részben adnak megoldást, de pont annyit amennyivel tudunk számolni.
Lépések:

- csonkolt hash kiszámítása
- első ellenőrzés: tárolt hash első 32bite a csonkolt 32bites hashsel
- ha nem egyezik akkor másik kulccsal előlről
- teljes hash kiszámítása
- tárolt és generált hash összehasonlítása
- egyezőség esetén találat, egyébként előlről

Ezzel a módszerrel felgyorsíthatjuk a törési folyamatot, a eredeti hash algoritmus és a csonkolt hash algoritmus csonkolásának arányával.

13. fejezet

Tesztelés és végszó

A szakdolgozat készítése alatt, 3modul készült a John The Ripper programhoz. 1db módosított MD5 algoritmus, aminek az egy eredeti szerzője bartavelle becenév alatt működő programozó írt. Ezt az algoritmust optimizáltam, egészítettem ki, az előzőekben írtakkal. Két szintes ellenőrzés van benne, az első szinten csonkolt hash algoritmus fut le, ami csak az első 32bitet határozza meg. Második lépésben egyezés esetén a 2. hash algoritmus is lefut, ami a teljes 128bites hash-t határozza meg, majd hasonlítja össze a listában találhatókkal.

Ezen felül 2db MySQL-323 algoritmus modul íródott. Ezeket az algoritmusok teljesen újraírtam a MySQL adatbázis serverének forráskódja alapján. Az egyik modul a hash csonkolást alkalmazza, amely csak a hash felét számolja ki, majd hasonlítja össze a listában lévővel, így hasonlít az előzőleg ismerett MD5 modulra. A másik modul pedig a memóriával optimalizál, teljes fát igen költséges lett volna felépíteni, és a John nem inkrementális kulcs generálási módszerének köszönhetően ez nem is lett volna kifizetődő, így csak az utolsó kulcsot tárolja el, és az ahhoz tartozó értékeket. Ebből az értékből kiindulva számolja ki a következő kulcshoz számított hash-t.

A modulok megírásában Kasza Péter szaktársam segítkezett.

A Johnt futtatva a következő eredményeket érik el a modulok:

```
Benchmarking: Raw MD5 [raw-md5]... DONE
Raw: 2644K c/s real, 2866K c/s virtual

Benchmarking: Raw MD5 Szakdolgozat [raw-
md5_szakdoga]... DONE
Raw: 3372K c/s real, 4208K c/s virtual
```

```
Benchmarking: mysql [mysql]... DONE
Raw: 1590K c/s real, 1808K c/s virtual

Benchmarking: mysql_csonkolas_szakdoga [mysql-
csonkolas_szakdoga]... DONE
Raw: 13155K c/s real, 14079K c/s virtual

Benchmarking: mysql_szakdoga [mysql-szakdoga]... DONE
Raw: 7939K c/s real, 8329K c/s virtual
```

Az eredeti modul a Raw MD5, általam írt a Raw MD5 Szakdolgozat. Az optimizált MD5 algoritmus 1,2-szer gyorsabb az eredetinél.

MySQL-323-nál az eredetileg írt algoritmus neve a: mysql. Újra írtak a mysql-csonkolas_szakdoga, ahol a csonkolásos módszert alkalmazva közel 10szeres sebességnövekedést értem el, és a mysql-szakdoga ahol a memóriaval optimizált, fás kiterjesztés egy fajtáját implementáltam. Ez 5szörös gyorsulást eredményezett, de ez az alacsony érték, csak annak köszönhető, hogy a *test* mód a John The Ripperben a modul test struktúrájában megadott hasheket számos alkalommal titkosítja a modullal, így nem inkrementális módon vannak generálva a kulcsok, vagyis nem tesztelhető a modul sebessége ezen módon hitelesen.

Összesítés:

Név	Összehasonlítás/Másodperc	Optimalizálás
Raw-MD5 (eredeti)	2644K	0
Raw-MD5 (csonkolás)	3372K	1,275
MySQL-323 (eredeti)	1590K	0
MySQL-323 (csonkolás)	13155K	8,273
MySQL-323 (memória)	7939K	4,993

A szakdolgozat ismeretette a hash algoritmusok és a kriptográfiai hash algoritmusok alapjait. Bemutatott 4db változatos és használatban lévő kriptográfiában használt hash algoritmust, ismeretette a történetüket, felépítésüket, pszudó kódjukat, és esetlegesen hibáikat. Ezentúl rávilágított pár tervezési hibára, azok kihasználási lehetőségei. Szakdolgozattal egyetemben a módszerek implemetálva lettek és a mellékletben megtalálhatók.

Irodalomjegyzék

[1] T. W. Körner: *Coding and Cryptography*, 1998

[2] Hans Delfs, Helmut Knebl *Intruduction to Cryptography*, 2002

[3] M. E. Hellman. *A cryptanalytic time-memory trade off*, 1980

[4] Philippe Oechslin *Making a Faster Cryptanalytic Time-Memory Trade-Off*, 2003

[5] Brian J. Gough *An Introduction to GCC - for the GNU compilers gcc and g++*, 1994

[6] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone *Handbook of Applied Cryptography*, 2001

[7] R. C. Merkle *Secrecy, authentication, and public key systems. Stanford Ph.D. thesis, pages 13-15.*, 1979

[8] Horst Feistel *Tweakable Block Ciphers*, 2002

[9] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone *Cryptography and Computer Privacy*, 1973

[10] A. Narayanan, V. Shmatikov *Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff*, 2005

A. függelék

CD-Használati útmutató

A mellékelt CD-n találhatók a következők:

- A szakdolgozat maga pdf formátumban
- A szakdolgozat maga tex forrásban
- A szakdolgozat által tárgyalt hash algoritmusok implementált formában
- A szakdolgozat által ismeretett módszereket implementált modulok John The Ripperbe fordítva
- A szakdolgozat által ismeretett módszereket implementált modulok forrása John The Ripperbe forrásában

A.1. Források és fordításuk

A tárgyalt hash algoritmusok forrása a *sources*/ mappa alatt található, fordításuk UNIX rendszer alatt a következő:

```
# gcc mysql-323.c -o mysql-323
# gcc md5.c -o md5
# gcc freebsdmd5.c -o freebsdmd5
# gcc sha1.c -o sha1
```

Futtatásuk:

```
# ./mysql-323 [key]
# ./md5 [key]
# ./freebsdmd5 [key] [salt]
# ./sha1 [key]
```

A.2. John The Ripper, modulok forrása és fordításuk

A *john-szakdolgozat/src/* könyvtár alatt található a John The Ripper forráskódja és benne található installált modulok. Fordítása UNIX rendszerek alatt a következő:

```
# make
A megfelelő operációs rendszer és architektúra kiválasztása,
jelen esetben FreeBSD(x86):
# make freebsd-x86-sse2
```

Ezzel le is fordítódott a John, a binárisa a *john-szakdolgozat/run/* alatt található.

A.3. John The Ripper és a modulok használat

Modulok tesztelése:

```
# ./john -test
```

MySQL-323 Modulok használata (hash algoritmusok lefordítása után):

```
# echo test:`../../sources/mysql323 testhash» test.hash
# ./john -format=mysql test.hash
vagy
# ./john -format=mysql-csonkolas_szakdoga test.hash
vagy
# ./john -format=mysql-szakdoga test.hash
```

MD5 Modulok használata (hash algoritmusok lefordítása után):

```
# echo test:`../../sources/md5 testhash» test.hash
# ./john -format=raw-md5 test.hash
vagy
# ./john -format=raw-md5_szakdoga test.hash
vagy
```

A John The Ripper Windows operációs rendszerre lefordítva is megtalálható a CD-n a *binaries/* könyvtár alatt