Debreceni Egyetem
Természettudományi Kar
Matematikai Intézet

Diplomamunka

Párhuzamosítható változó hosszú hash függvény

készítette:

Bucsay Balázs

témavezető: Tengely Szabolcs

Debrecen, 2012. 04. 18.

Tartalomjegyzék

Köszönetnyilvánítás

Ez a diplomamunka nem jöhetett volna létre a megértő és támogató családom és közeli szeretteim nélkül.

Szeretnék köszönetet mondani az egyetemi éveim alatt tanáraimnak, mind a Miskolci Egyetemen a Programtervező Informatikus BSc és a Debreceni Egyetemen a Matematikus MSc képzésben tanítóknak a sok átadott tudásért, a barátaimnak a biztatásért és elismerésért, illetve a munkáltatóimnak a rugalmas hozzáállásukért és támogatásukért.

1. Bevezető

A következő diplomamunka a Párhuzamosítható változó hosszú hash függvényről avagy a Paralellizable Variable Length Hash Function-ről fog szólni, vagyis a PVL hash függvényről. Ez a függvény a matematika tudományág egyik diszkrét tudományágában a kriptográfiában helyezkedik el azon belül is a kriptográfiai hash függvények között.

A kriptográfia a titkosítás tudománya, ami matematikai eszközöket használ fel az üzenetek titkosítására, hogy többek között illetéktelen szemektől az üzenetet megóvja illetve megőrizze az integritását. A kriptográfia széles eszköztárával ma már valószínűsíthetően vagy éppen bizonyítottan biztonságosan kommunikálhatunk, ellenőrizhetjük az üzenet érintetlenségét, validálhatjuk a címzettet illetve a feladót és így tovább. Ezeket az eszközöket a mindennapjainkban állandó jelleggel használjuk a tudtunkon kívül is. Amikor telefonálunk a GSM, 3G hálózatokon titkosított adatfolyammal kommunikálunk, a weboldalak egy része titkosítást igényel a megnyitásuktól kezdve, a banki tranzakciók titkosított környezetben futnak, a számítógépeink frissítése hitelesítve történik, így próbálva megvédeni minket azoktól akik az adatainkat szeretnék megszerezni.

A kriptográfiai hash függvények a kriptográfia egy kis szelete, ami alapvetően a matematikai hash függvényekből alakult ki (lásd: 3.1). Ezek a függvények különböző adatstruktúrák kezelésére nagyon jól használhatók, nagyon hasznosak és lényegük, hogy minden bemenetükre determinisztikusan egy értéket számolnak ki. A kriptográfiai hash függvények (lásd 3.1) ezeknek a tovább fejlesztéseik, ugyanígy hash függvények viszont több megkötés vonatkozik rájuk, erősebb tulajdonságaik vannak. Két nagyon fontos említésre méltó tulajdonságok, hogy PRNG-ként (Pseudo Random Number Generator - Pszeudó véletlen szám generátor) kell viselkedniük de mind ezt determinisztikusan és ütközésmentesnek kell lenniük. Ez a két tulajdonság azt jelenti, hogy semmilyen körülmények között még csak becslést sem lehet tenni arra, hogy egy adott bemenethez milyen kimenet tartozik illetve, hogy minden kimenet egyedi és nem lehet vagyis rendkívül nehéz két ugyanolyat készíteni más-más bemenethez. A PVL hash függvénynek ezeket a feltételeket kell többek között teljesíteni, már a tervezési folyamata is erre éleződött ki.

A diplomamunka 2. fejezete (Ismertető) nagy vonalakban egy történelmi áttekintést ad arról, hogy milyen történeti előzményei is vannak a hash függvény

tervezésének, milyen alapokat szolgáló hash függvények és elméleti háttér volt adott ennek a tervezéséhez, illetve részletezi a tervezési célokat pontokba szedve. A 3. fejezet (Tervezési alapkövek) részletesen kifejti az alapvető és már kevésbé triviális elméleti hátterét a figyelembe vett tervezési ciklusnak, kitérve olyan alapkőnek mondható elméletekre mint a Merkle-Damgård konstrukció, lavina effektus vagy Nandi tétele. A 4. fejezet (Kriptográfiai hash függvények ismertetése) történeti áttekintést nyújt a tudományággal ismerkedők számára, részletezi az MD család két tagját, az SHA család jelenlegi két gyermekét illetve a RIPEMD-160 és HAS-V algoritmusokat is. Az 5. fejezetben (Függvény specifikációja) a PVL hash függvény elméleti háttere van részletezve ábrákkal, konstans és függvény definíciókkal. Ebben a fejezetben a kész algoritmus teljes körű leírása található meg sematikus struktúrájától kezdve a részletgazdag jellemzéséig. A függvény követi az elődei által leírt konvencionális utat, azok pár hibáját javítva, illetve pár újdonsággal megfűszerezve. Újdonság például a változó hossza a készült lenyomatnak. A függvény segítségével bizonyos korlátok között változó méretű lenyomatokat készíthetünk, így növelve a biztonságot.

A dolgozat számos referenciát is feltüntet amik súlyos szerepet játszottak a felkészülési és tervezési fázisban és a legvégén a függelékben egy C-ben írt forráskód is található ami a hash függvény teszt implementációja.

A végcél, vagyis egy változó kimenettel rendelkező hash függvény megalkotása sikeresnek mondható, hiszen a dolgokat egy kész hash függvényt ír le. Ez a hash függvény kell, hogy teljesítse a felsorolt követelményeket és remélhetőleg semmilyen elméleti gyengeséget nem tartalmaz.

2. Ismertető

2.1. PVL Hash ismertetése nagy vonalakban

A Paralellizable Variable Length kriptográfiai hash függvény egy változó hosszúságú kimenettel rendelkező iterált hash függvény. Az 1990-ben megjelent Ronald Rivest által megalkotott MD4 algoritmus nagyon jó alappal szolgált a mai napig használatban lévő, fejlesztett és fejlesztés alatt lévő hash függvényeknek. Az MD4 hibáiból és erősségeiből tanulva jöttek létre az MD5 (a következő generáció az MD csalában), az SHA-1 és SHA-2 algoritmusok is amiket az USA szabadalmi hivatala (NIST) fejlesztetett ki és szabadalmazott. A diplomamunka írása közben folyik az SHA-3-ra jelölt algoritmusok versenyeztetése, amelyek egy része hasonlóképp épül fel, mint a diplomamunkában ismertetett PVL hash függvény illetve az előzőleg is említett hash függvények. A PVL hash függvény, mint ahogy említett az iterált hash függvény, MD5 és SHA-1 felépítéséhez hasonlóan 4 körben, összesen 64 lépésben titkosítja a bemenetet. A bemenet először a Merkle-Damgård alapelveknek illetve annak megerősített változatának megfelelő módon kiegészíti (padding) az üzenetet és a végére írja az üzenet eredeti hosszát. Ezen felül minden egyes blokkot elejénél fogva kiegészíti megint csak az üzenet eredeti hosszával így biztosítva, hogy az algoritmus védett legyen a length extension (üzenet kiegészítő) támadás ellen. A bemenet blokkjait a meghatározott módon iterált módon összekeverve az inicializációs vektorokkal (IV-k) 2, 4 illetve 6db különböző hash függvényt készít. Feltételezve, hogy az iterált hash függvény ütközésmentes és Nandi tételét használva ugyanazon bemenetből különböző hash függvényeket állíthatunk elő determinisztikusan, ezeket az előállított hash függvényeket előre definiált módon összekeverjük és ezzel állítjuk elő a végleges hasht. A hash mérete függ az üzenet (bement) hosszától, a választott hashelési módtól valamint a számítógépes architektúrától. Ezek alapján az algoritmus képes 32bites platformokon 160, 320 illetve 480bites hasheket előállítani illetve 64bites architektúrán 320, 640 és 960bites hasheket.

2.2. Tervezési célok

A PVL Hash algoritmus tervezésekor a következő pontok, mint célok merültek fel és kerültek megvalósításra:

- Robusztusság és gyorsaság
  A hash függvényeknél nagyon fontos tényező az idő és tárbonyolultság. A legtöbb hash függvény konstans tárral és lineáris időben tudja generálni a hasht természetesen determinisztikus módon, célom, hogy a diplomamunkában leírt hash függvény is hasonlóképpen konstans tárral és lineáris időben oldja meg a problémát, esetleg az időbonyolultsága csökkenthető legyen a bemenetek hosszával való összefüggésben. Ezen felül szükséges robusztusnak lennie vagyis probléma mentesen, gyorsan és determinisztikusan kell működnie.

- Egyszerűség
  Rendkívül fontos, hogy legyen egyszerű, az egyszerűség hozzásegíti az algoritmust a könnyű elemezhetőség és implementálhatósághoz így a hibái, sebezhetőségei könnyedén és gyorsan kibuknak. Egyszerű struktúra ami az MD4 és ráépülő hash algoritmusokat jellemzi elősegíti a tervezést, hogy a jó tulajdonságokat könnyűszerrel meg tudjuk tartani és rossz tulajdonságokat, eddig talált sebezhetőségeket pedig könnyedén elkerüljük. Tudtában a tanulmányoknak és eredményeknek elkerülhetünk triviális baklövéseket és a lényegre koncentrálhatunk.

- Univerzalitás
  Rugalmasság és az univerzalitás meghatározó elv. SHA-2 családban bemutatott hash függvények ugyanúgy működtek, a különbség bennük a skálázhatóságban rejlő apróságok. Ez a cél itt is megfogalmazódott inkább sémát, mint hash függvényt fog leírni a diplomamunka, az értékek és egyéb belső függvények könnyen skálázhatóak illetve cserélhetőek lesznek. Ezzel egy rugalmas, könnyen alakítható hash függvény készíthető.

- Változó hash méret
  A hash függvény változó hosszúságú hasht képes generálni egy azon üzenethez. A felhasználó képes ezt szabályozni, így a számára megfelelő, ekvivalens biztonsággal képes különböző hosszú lenyomatokat készíteni.

- Párhuzamosíthatóság
  Az algoritmus tervezésekor figyelembe kell venni a párhuzamosíthatóságot. Nem szabad, hogy ez gyengítse az algoritmust bármilyen módon, inkább erősítenie kell. A párhuzamosítás segít, hogy ugyanannyi idő alatt több adatot dolgozzon fel az algoritmus, így kvázi gyorsabban számolja a lenyomatot.

- Ütközésmentesség
  Az ütközésmentesség a következőkben lesz definiálva, viszont alapvető követelmény egy hash-hez legalább $O(2^{n/2})$ számítás szükséges, hogy megtalálja az eredeti vagy az ütköző üzenetet.

- Előkép ellenállóság és második-előkép ellenállóság
  Akárcsak az ütközésmentesség az előkép és második-előkép ellenállóságot is a következőkben definiálom, viszont fontos, hogy ezek a tulajdonságok teljesüljenek.

- Üzenet kiterjesztő támadás elleni védettség
  Szükséges, hogy az üznet kiterjesztés ellen védett legyen az algoritmus, így ne lehessen hamisítani a hash-el aláírt üzeneteket.

- Könnyű implementálhatóság
  Az MD4 és ráépülő hash algoritmusok struktúrájának, illetve a Merkle-Damgård alapelveknek köszönhetően az algoritmus könnyen implementálható illetve készíthető hozzá specifikus hardver is.

3. Tervezési alapkövek

A következő pontokban ismertetett fogalmak, definíciók, tételek alapköveit jelentették az algoritmus tervezésének, ezek megértése elengedhetetlen a a PVL hash függvény teljes átlátásához. A fejezet megírásához és az alaposabb megértéshez sokat segített Johannes A. Buchmann: Introduction to Cryptography [14] és Bart Preneel: Analysis and Design of Cryptographic Hash Functions [11] című könyve, illetve Wolfram MathWorld portál, Donghoon Chang, Mridul Nandi, Jesang Lee, Jaechul Sung, Seokhie Hong: Hash Function Design Principles Supporting Variable Output Lengths from One Small Function [10], Merkle: Secrecy, authentication, and public key systems [1] és Damgård A Design Principle for Hash Functions. In Advances in Cryptology publikációi [2].

3.1. Hash függvények jellemzése

A matematikai hash függvényeknek rengeteg felhasználási területe van, könnyedén lehet velük hash táblákat készíteni amiben $O(1)$ időbonyolultsággal optimális esetben lehet keresni illetve hasonlóan optimális esetben $O(1)$ időbonyolultsággal írni. A kriptográfiában [14] is felütötték a fejüket a hash függvények, bár az matematikai hash függvények önmagukban nem ajánlottak használatra (túl gyengék) tovább fejlesztve őket könnyedén számos célra felhasználhatók.

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

A kriptográfiai hash függvények a matematikai hash függvény alapjain nyugszanak, azonban több fontos és elengedhetetlen tulajdonsággal bírnak annak érdekében, hogy felhasználásuk biztonságosabb legyen. Egyik fontos jellemzőjük, hogy egyirányúak.

3.1.2. Definíció. (Egyirányú függvény) [15]. Legyen f egy függvény amit egyirányúnak nevezünk ha a következők teljesülnek:

- Ha x adott $f(x)$ kiszámítása egyszerű,
- ha y adott, akkor x kiszámítása nehéz, ahol $y = f(x)$,
- $f(x)$ függvény legyen publikus.

3.1.3. Definíció. (Kriptográfiai hash függvény). Legyen $m \in \{0,1\}^*$ (a kódszavak halmaza). $H(m)$ pedig egy hash függvény. $H : \{0,1\}^* \to \{0,1\}^n$, ahol $n > 0, n \in \mathbb{N}$. $H(m)$-et kriptográfiai hash függvényenk nevezzük ha kielégíti a következőket:

- H egyirányú függvény,
- előkép ellenálló (Preimage resistant): adott y-oz nehéz találni x-et úgy, hogy teljesüljön a $y = H(x)$,
- másod előkép ellenálló (Second preimage resistant): adott y_1 -hez nehéz y_2 -őt találni úgy, hogy $H(y_1) = H(y_2)$ és $y_1 \neq y_2$,
- ütközésmentes (Collision resistant) nehéz y_1 és y_2-őt találni úgy, hogy teljesüljön a $H(y_1) = H(y_2)$

A diplomamunka egy a Kriptográfiai hash függvény definíciójának megfelelő függvény elkészítését veszi célba.

3.2. Permutációk

Egy halmaz permutációjának hívjuk azt a függvényt ami egy halmazt egy meghatározott módon rendez.

3.2.1. Definíció. Legyen X egy véges halmaz. Legyen $\sigma : X \to X$ bijektív függvényeket a halmaz permutációinak nevezzük. Ezek összességét S_X-el jelöljük, S_n pedig $\{1, 2, \cdots, n\}$ összes permutációja aminek elem száma $n!$.

Egyszerű jelölése a permutációknak pl. S_5 egy elemének:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}.$$

Az σ függvény az első sort a 2. sorba képzi.

3.2.2. Definíció. Legyen σ egy permutáció. Azt mondjuk, hogy σ n-ed rendű ha $\sigma^n = \sigma$, ahol $n > 0$ legkisebb amire teljesül.

3.3. Lavina effektus

Kriptográfiai hash függvényeknél elengedhetetlen a lavina effektus tulajdonsága vagyis, hogy a kimenet minden bitje szükséges hogy a bemenet minden bitjétől függjön. Ha egyetlen egy bitben megváltoztatjuk a bemenetet, az üzenetet aminek a lenyomatát akarjuk venni, akkor a függvény oly esetben teljesíti csak a lavina effektus tulajdonságot, ha legalább a bitek fele megváltozik.

Elengedhetetlen ehhez egy metrika bevezetése:

3.3.1. Definíció. Legyen $D : x \times y \to \mathbb{N}$, ahol $x, y \in \{0,1\}^n, n > 0$,

$$D(x,y) := \sum_{i=1}^{n} |x_i - y_i|$$

ekkor H függvény egy metrika és Hamming távolságnak nevezzük.

1. Állítás. *Legyen D a Hamming távolság függvény, H pedig egy hash függvény ami eleget tesz a lavina effektusnak, ha*

$$D(H(x), H(y)) \to \frac{n}{2}$$

ahol n a lenyomat hossza, $x, y \in \{0,1\}^n$ és $D(x,y) = 1$.

Vagyis ha két üzenet csak egy bitben tér el egymástól akkor a lenyomatok Hamming távolságának konvergálnia kell a lenyomat hosszának a feléhez.

3.4. Prefix kódok

Kódelméletben egy $X = \{x_1, x_2, \cdots x_n\}$ nem üres, véges halmazt ábécének nevezünk, és az elemeit pedig betűknek. A betűkből (az X halmaz elemeiből) szavak állíthatók elő, a szavak hosszán a benne szereplő betűk számát értjük. Legyen $Y = \{y_1, y_2, \cdots, y_m\}$ nem üres, véges halmaz amit kódábécének nevezünk és X* az X elemeiből előállított véges szavak halmaza. Ha létezik egy $f : X^* \to Y^*$ leképzés ami szavakból kódszavakat állít elő, akkor f egy kódolás, vagyis f minden X^*-beli szóhoz egy Y^*-beli kódszavat kódol.

3.4.1. Definíció. Legyen y_1 és y_2 két kódszó, y_1-et y_2 prefixének nevezzük és y_1 $<=$ y_2 -vel jelöljük ha y_2 kódszó y_1-el kezdődik.

3.4.2. Definíció. Legyen Y egy kód. Ha teljesül minden $y_1, y_2 \in Y^*$ -ra, hogy $y_1 \preceq y_2 \iff$ ha $y_1 = y_2$, akkor Y prefix-mentes vagy más néven prefix kód.

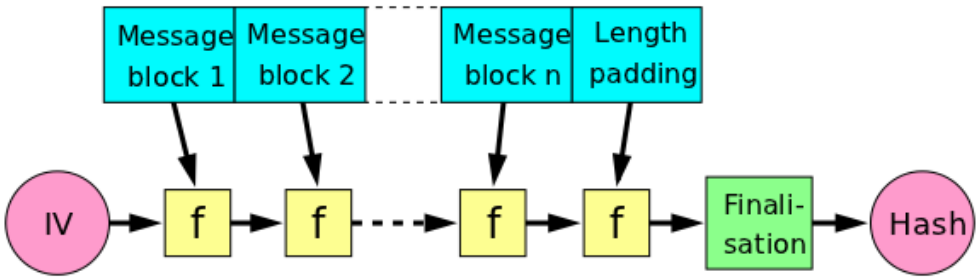
Prefix kódok jó tulajdonsága az, hogy akár változó hosszal is igaz rájuk, hogy egy kód folyamban folytonosan félreértéseket mellőzve felismerhetők a kódszavak, és így determinisztikusan visszafejthetők is.

3.5. Merkle-Damgård konstrukció

1979-ben Ralph Merkle a PhD tézisében [1] kifejtette, hogy a kulcs méretének nem feltétlen kell kisebbnek lennie, mint az üzenetének. Ha az üzenet rövidebb, akkor bizonyos transzformációk után ugyanolyan biztonságot érhetünk el a titkosítással, mint az előzőleg bevált módszerekkel. Ivan Damgård ekkor, teljesen függetlenül [2] Merkle-től bizonyította, hogy ha egy tömörítő függvény ütközésmentes, akkor

megfelelően kiegészítve az üzenetet a hash függvény ami erre a tömörítő függvényre alapszik is ütközésmentes marad. Ezeket az eredményeket azóta is számtalan helyen hivatkozzák és használják fel, mint hasznos kutatási eredményt. Többek között az SHA-3-ra pályázó algoritmusok is alapoznak erre a bizonyításra.

A Merkle-Damgård konstrukció kimondja, hogy milyen szempontokat is érdemes szem ügyre venni mielőtt hash függvényt tervezünk. Először is szükséges egy tömörítő függvényt keresni aminek több változója is van és ütközésmentes illetve inicializációs vektorokat lefixálni amikkel az első lépésben a hash függvényünk (a tömörítő függvény) indulni fog. A hash függvény elhagyhatatlan változóját nevezzük a következőkben üzenetnek. Az üzenetet fix hosszúságú blokkokra kell bontanunk, majd a tömörítést minden blokkon el kell végeznünk. Ha az üzenet utolsó blokkja nem elég hosszú, akkor úgy kell kiegészíteni, hogy a blokk hosszának megfeleljen. Kiegészítés után az első blokkot az IV-kkel együtt kiszámítjuk, majd a többi blokkot már (az IV-t helyettesítve) a kiszámolt értékekkel számoljuk ki. Így a blokkokat egy láncba felfűzve újra és újra titkosítjuk a tömörítő függvénnyel. A legutolsó lépés a véglegesítés ahol a végleges értéket egy függvény számolja ki az utolsó tömörítő függvény eredményéből. A 3.1 ábrán látható a konstrukció, a részei a bővebben, a 3.5.1, 3.5.2 és a 3.5.3 pontokban lesznek kifejtve.



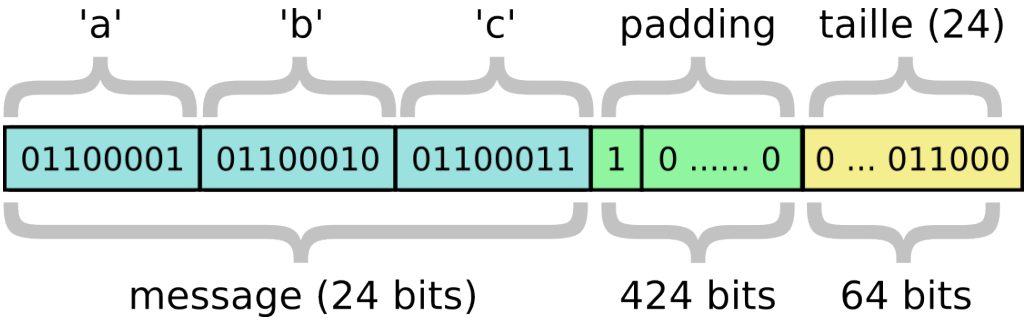
3.1. ábra. Merkle-Damgård konstrukció (http://en.wikipedia.org/wiki/File:Merkle-damgard.png)

3.5.1. Padding

Merkle [1] egy nagyon egyszerű kiegészítési (padding) módszert javasolt vagyis, hogy az üzenetet bináris számrendszerben úgy egészítsük ki, hogy a végére egy 1-est hozzáfűzünk, majd kitoldjuk 0-ákkal amíg a blokk tart. Ennél hasznosabb ha még a az eredeti üzenet hosszát is az üzenet végére írjuk, így lezárva az üzenetet és elkerülve a length extension támadás egyik válfaját (erről a következő pontban bővebben) Kifejtve:

$$M_1 = b(M_0)||1||0^{(l(s)-(l(b(M_0)+1)\%s)-l(b(l(s))))}||b(l(s))$$

ahol M_1 az új kiterjesztett üzenet, M_0 az eredeti üzenet, || a konkatenáció, l a hosszúságot megadó függvény, b pedig a bináris alakot adja meg és s a blokkok hosszát jelenti.



3.2. ábra. Példa a paddingolásra (http://fr.wikibooks.org/wiki/Les_fonctions_de_hachage_cryptographiques)

3.5.2. Iteráció

3.5.1. Definíció. Legyen f egy tömörítő függvény. Ebből a tömörítő függvényből előállítható egy $H(x)$ iterált hash függvény:

$$H(x) = f(h_i, x_i), \text{ ahol } h_i = \begin{cases} IV & \text{ha } i = 0 \\ f(h_{i-1}, x_{i-1}) & \text{ha } i > 0 \end{cases}$$

A H hash függvény egy iterált függvény ami az f tömörítő függvényt iterálja végig az üzenet x_i blokkjain. Első lépésben h_i az előre definiált IV majd a többi lépésben az előző lépésben kiszámolt f értéke. Egy jó hash algoritmusnál az induló értékek, az IV-k értéke lényegtelen.

3.5.3. Véglegesítés

A legutolsó f értéken egy t transzformációs függvény hattatunk és ebből kapjuk meg a végleges értéket. Ez a lépés elhanyagolható, inkább csak formális jelentősége van. Itt lehet egy ember számára könnyebb formának megfeleltetni az értéket, vagy levágást (truncation-t) végrehajtani rajta amivel ugyan rövidül a hash mérete de ellenállóvá teszi bizonyos támadások ellen.

3.6. Length extension attack

Üzenet kiterjesztő támadásnak nevezzük azt az egyszerű támadási módot, ahol a támadó ismeri a $H(x)$ értéket illetve az x hosszát, de x-et nem. Ekkor könnyedén előállítható egy $H(x||\dot{x})$ érték ahol $\dot{x}$ értéke bármi lehet. Bár az x üzenetet ezzel

nem tudjuk rekonstruálni, de például egy ilyen módszerrel támadható hash függvény által aláírt email-t kiegészítve hiteles aláírással tudunk ellátni. Az MD5, SHA-1 algoritmusok ezt a támadást részben védik csak ki, részben pedig sebezhetők. Az üzenet hosszának és $H(x)$ tudtában nem $H(x||\dot{x})$ értéket tudjuk előállítani hanem $H(padding(x)||\dot{x})$ értéket. Ezzel hasonlóképpen sebezhetők az algoritmusok, de a sebezhetősége sokkal gyengébb, mint eredeti esetben, bár ez sem megbocsátható. A támadás könnyen kivitelezhető, méghozzá az IV-k lecserélésével a $H(padding(x))$ hash értékekre.

3.7. Nandi tétele

Egyik lehetőség a változó hosszúságú hash függvények építéséhez ha DBL (Double Block Length) függvényeket alkalmazunk. Ezek a függvények egy permutáció és egy tömörítő függvény segítségével megduplázzák a kimenet hosszát úgy, hogy ha a tömörítő függvény ütközésmentes, akkor az eredményük is az marad. Ezt a tételt Nandi bizonyította a [10] publikációjában.

3.7.1. Tétel (Nandi)**.** *Legyen $F : \{0,1\}^m \to \{0,1\}^{(2*n)}$ egy tömörítő függvény, $P : \{0,1\}^m \to \{0,1\}^m$ egy permutáció aminek nincsen fix pontja és P^2 az identitás permutáció, $f : \{0,1\}^m \to \{0,1\}^n$ pedig egy tömörítő függvény ami ütközésmentes. Ekkor F függvény egy Double Block Length függvénynek definiálható a következőképpen:*

$$F(X) = f(X)||f(P(X))$$

Ahol $||$ a konkatenálást jelzi. Ha F a feltételeket kielégíti akkor ütközésmentes.

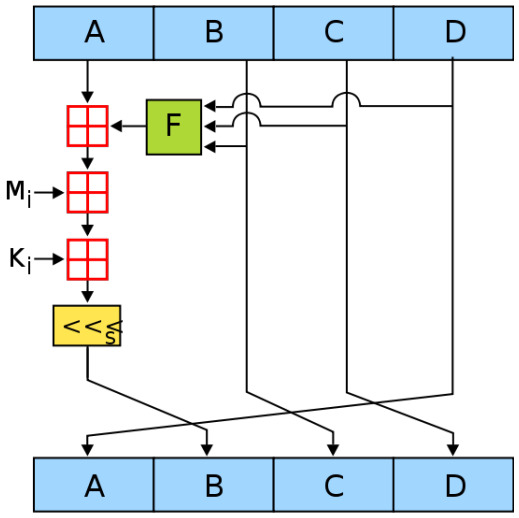
4. Kriptográfiai hash függvények ismertetése

A következőkben pár méltán híres és részben használatban lévő hash függvényről lesz szó. Az MD4 [3] algoritmust már 1991-ben leváltotta a biztonságosabb MD5 [4]. Az MD5 és SHA-1 [5] mai napig aktívan használatban van az SHA-2 [6] variánsokkal együtt. A RIPE-MD [7] és HAS-V [8] variánsok inkább kísérleti hasheknek mondhatók, nagy érdeklődésre az iparban nem, viszont a kriptográfusok körében tettek szert. Részletesebb leírásért érdemes elolvasni az 1320-as RFC-t az MD4-ről [3], RFC 3174-et az MD5-ről [4] illetve az RFC 3174-et az SHA-1 hash függvényről [5], amikben a függvények mint szabványok vannak lefektetve. Ezen felül az SHA-2 és SHA-1 függvényekről és követelményeiről bővebben, bár néhol elég szűkszavűan ír a FIPS 180-3 [6], amit az NSA készített. A párhuzamosításhoz illetve a változó hosszú kimenethez nagy segítséget adott a RIPEMD-160 algoritmus és a HAS-V kóreai hash függvény, ezek leírása illetve elemzései megtalálhatók a Hans Dobbertin, Antoon Bosselaers, Bart Preneel: RIPEMD-160: A Strengthened Version of RIPEMD [7], Nan Kyoung Park, Joon Ho Hwang, Pil Joong Lee: HAS-V: A New Hash Function with Variable Output Length [8] és Florian Mendel and Vincent Rijmen: Weaknesses in the HAS-V Compression Function [9] publikációkban.

4.1. MD4 algoritmus

Ronald Linn Rivest 1990-ben kifejlesztette az MD4 függvényt, ez az algoritmus maximum közel 2^{32} hosszúságú üzeneteket transzformált 128bites hashekké. A hash függvény megjelenése nagy áttörést jelentett a hash algoritmusok körében és óriási tömegeket ösztönzött a kriptográfusok között. Ennek eredményeként aktívan kutatott algoritmus lett és számos hiányosságra fény derült az idő elteltével.

Az MD4 [3] algoritmus 128bites hash-t produkál a bemenetből amit 512bites blokkokra vág illetve a Merkle-Damgård konstrukció szerint készít el. Ezeket a blokkokat 48 lépésben (3 körön) keresztül transzformálja a következő függvények-



4.1. ábra. MD4 egy lépése (http://en.wikipedia.org/wiki/MD4)

kel:

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$

Ezek a függvények 32bites változókkal dolgoznak amik kezdetben a 4db előre definiált konstans IV és az üznet blokkok darabjai később a transzformált üzenet blokkok.

Végül a transzformált 4db 32bites értéket a egy véglegesítő függvény konkatenálja és ez az érték lesz az MD4 hash függvény végleges értéke.

4.2. MD5 algoritmus

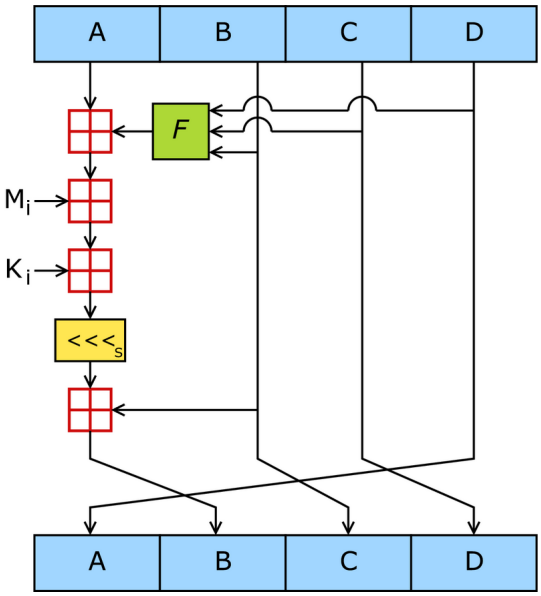
Az MD5 [4] is Ronald L. Rivest kutatásainak eredménye. Az MD4 algoritmus hibáiból tanulva, erősségeit megtartva készült el a megújult tagja az MD (Message Digest) családnak. Természetesen ez az algoritmus is rengeteg kriptográfust és matematikus mozgatott meg, így rendkívül sok támadás és publikáció található róla. Mára elég sok legalább elméleti szintű támadás jött ki ami a jó pár éve erősnek hitt algoritmust teljesen eltemette, így használatát már sehol nem javasolják.

Az algoritmus struktúrája szinte teljesen kövei elődjét. 512bites blokkokkal operál és 128bites hasht készít. A blokkokat 32bites részekre osztja és 64 lépésben (4.2 ábra), 4 körben minden körnek megfelelő különböző transzformációt hajt

végre.

$$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$
$$G(X,Y,Z) = (Z \wedge X) \vee ((\neg Z) \wedge Y)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$$

Ezek a transzformációk összetettek, részben állnak a körnek megfelelő függvényből (F, G, H, I) illetve annak eredményét rotálják bitenként és egy a lépésnek megfelelő konstans értékkel is modulárisan növelik.



4.2. ábra. MD5 egy lépése (http://en.wikipedia.org/wiki/MD5)

Minden lépés az ábrának megfelelően kerül végrehajtásra. A 64 lépés után újabb blokkot dolgoz fel az algoritmus, vagy ha nincs több blokk, akkor véglegesíti az értékeket az MD4 algoritmusnál leírtaknak megfelelően.

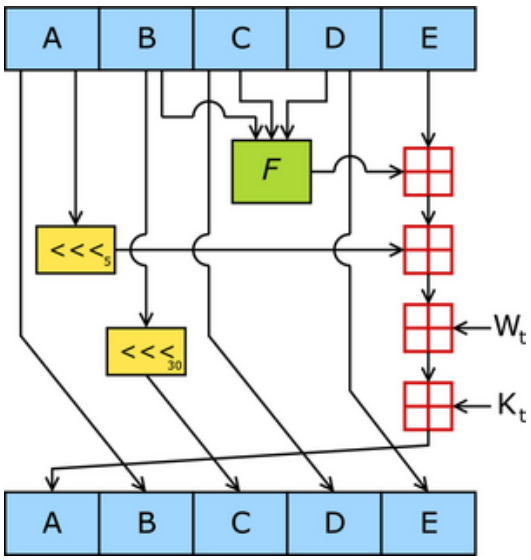
4.3. SHA-1 algoritmus

Az SHA-1 [5] algoritmust (Secure Hash Algorithm) az National Security Agency fejlesztette ki az USA-ban. 1995-ben publikálták, miután hibát találtak az SHA-0 algoritmusban 2 év után. Az algoritmus nagyon kicsit tér csak el az SHA-0 hash algoritmustól. Ezt a hash függvényt a mai napig rengeteg helyen használják annak ellenére, hogy mára már az MD5-höz hasonlóképp rengeteg publikáció jelent meg a sebezhetőségeiről. Az MD5 után ez volt a következő populáris hash függvény ami nagy rivalda fényt kapott. Az alapjait az MD4, MD5-ből kapta,

így felépítése, struktúrája rendkívül hasonlít az előzőekben ismertetett hash függvényekre. Lényeges újítás és különbségek az MD5-höz képest ahhoz, hogy bár 512bites blokkokkal dolgozik a függvény, 160bites kimenetet generál. Az 512bites blokkokat felduzzasztja 2560bites (80*32bit) blokkokra, majd 80 lépésben ezeket transzformálja. A transzformáció rendkívül hasonlít az MD5-ére, bár pár függvényt kicseréltek illetve a belső állapotokból sem csak egyet változtat hanem kettőt (transzformációs függvények és rotáció az állapotokon).

$$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$
$$G(X,Y,Z) = X \oplus Y \oplus Z$$
$$H(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$I(X,Y,Z) = X \oplus Y \oplus Z$$

Jobban megfigyelve a g és j függvények megegyeznek illetve az f és g (így az i is) megtalálható az MD5 függvényei között



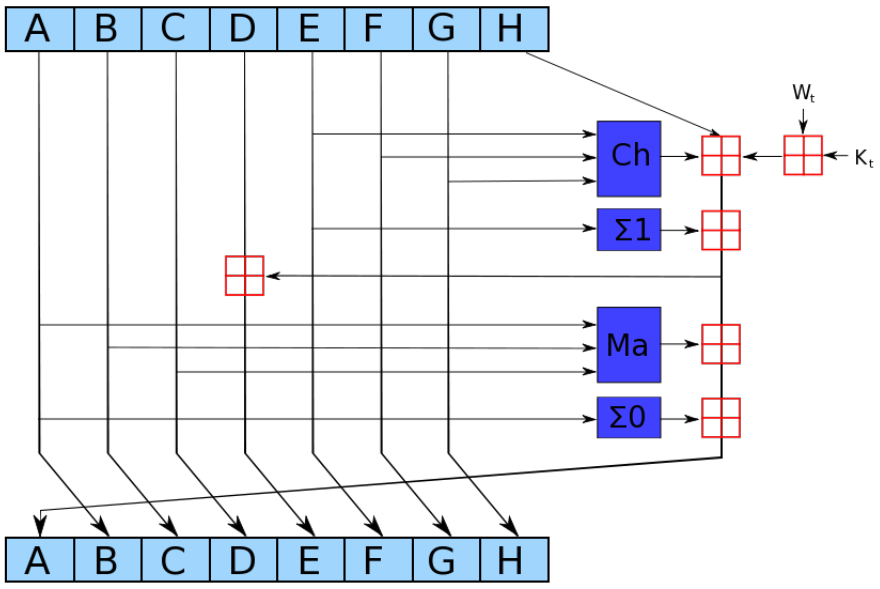
4.3. ábra. SHA-1 egy lépése (http://en.wikipedia.org/wiki/SHA-1)

4.4. SHA-2 algoritmus család

Bár még a legtöbb helyen MD5 és SHA-1 van elterjedve, az ipar és kormányzati szektorban főleg SHA-2 –t használnak már. Az SHA-2 –őt [6] az NSA fejlesztette ki, mint az SHA-1 –et. Mára már az SHA-2 –őt ajánlja a NIST is mivel 2005-ben az SHA-1 találtak egy elméleti matematikai hibát aminek már csak a létezése is meggyengítette a bizalmat a hash függvényben. Ugyan ez a sebezhetőség (bár az SHA-2 hasonló struktúrájában az elődjéhez) mégsem található meg a hash

függvényben. Még az előzőleg felsorolt nevek mind egy hash függvényt foglalnak magukban, az SHA-2 rendhagyó módon egy függvény családot. A család tagjai struktúrailag teljesen megegyeznek két fő különbség van köztük. Egyrészt a belső reprezentálás vagyis blokk méret ebből kifolyólag a maximális mérete a bemenetnek illetve a kimenet hossza ami részben az előzőektől illetve a véglegesítő függvénytől függ. A család két tagja 512bites blokkokra bontja a bemenetet illetve 32bites részblokkokra. A titkosítás után létrejön egy 256bites forma amit az SHA-2/256 meghagy az eredeti alakjában az SHA-2/224 pedig az utolsó 32bitet levágva véglegesíti. Ugyanígy az SHA-2/512 és SHA-2/384 1024bites blokkokra és 64bites részblokkokra bontja a bemenetet. Ebből a titkosítás után egy 512bites hash jön ki amit az egyik algoritmus megtart a másik pedig 128bitet elhagy a végéről.

SHA2	Lenyomat hossz	Blokk méret	Szó méret	Lépések száma
SHA-224	224	512	32	64
SHA-256	256	512	32	64
SHA-384	384	1024	64	80
SHA-512	512	1024	64	80



4.4. ábra. SHA-2 egy lépése (http://en.wikipedia.org/wiki/SHA-2)

Mint előbb említve volt a struktúra nagy hasonlóságot mutat elődeivel. Ez a hash függvény is iterációs hash függvény, viszont olyan értelemben vett körökről, mint az MD4, MD5 és SHA-1 –nél láttuk nem beszélhetünk. Itt egy nagy kör van ami 64 illetve 80 iterációból áll. Az iteráción belül 4db előre definiált függvény van

amik a biteket keverik össze illetve cserélik fel a belső reprezentációban szereplő változók értékeit.

A 4.4 ábrán található függvények:

$$Ch(X,Y,Z) = (X \wedge Y) \oplus (\neg X \wedge Z)$$
$$Ma(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$
$$\Sigma_0(X) = (X^{\ggg 2}) \oplus (X^{\ggg 13}) \oplus (X^{\ggg 22})$$
$$\Sigma_1(X) = (X^{\ggg 6}) \oplus (X^{\ggg 11}) \oplus (X^{\ggg 25})$$

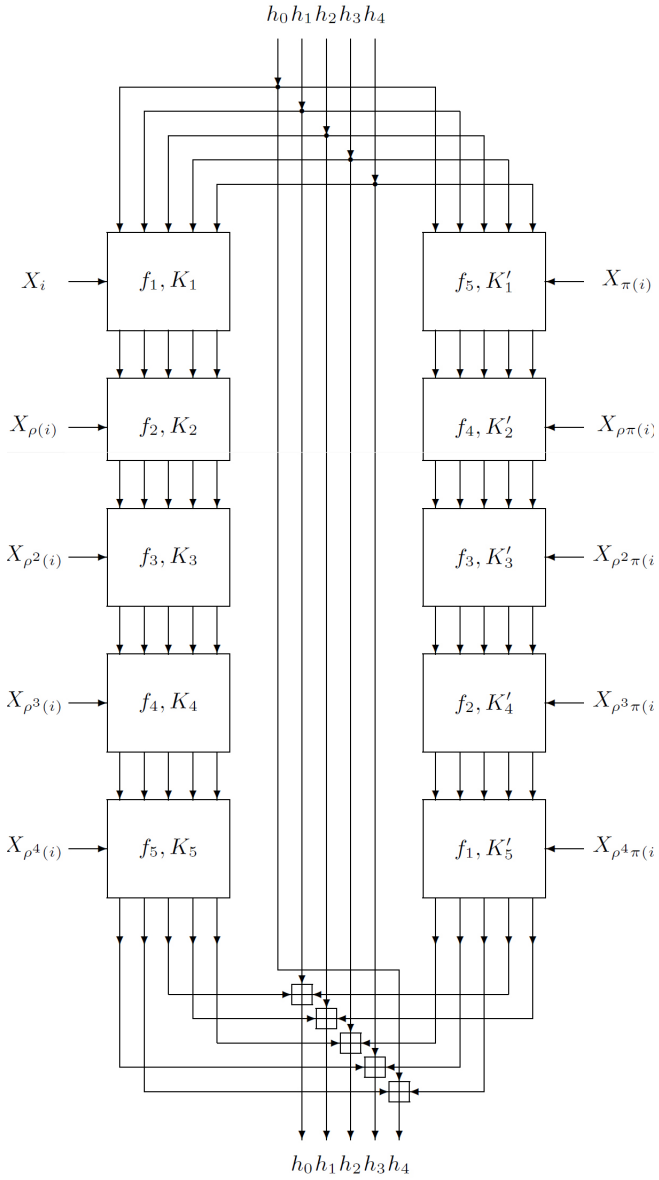
A család természetesen teljesíti a Merkle-Damgård által lefektetett alapelveket. Érdekessége még a bemenet blokkjainak ($512/32 = 16$ illetve $1024/64 = 16$) részéből előállít 48 másik részblokkot amit a titkosítás folyamatán összekever az eredeti részblokkokkal. Az SHA-2/224 illetve SHA-2/384 véglegesítő algoritmusa olyan szempontból is érdekes, hogy a végelhagyásos módszer (truncation) védelmet nyújt önmagában is a hossz kiterjesztő támadások ellen.

4.5. RIPEMD-160 algoritmus

A Race Integrity Primitives Evaluation Message Digest avagy a RIPEMD [7] algoritmust Hans Dobbertin, Antoon Bosselaers és Bart Preeneel fejlesztette ki az MD4-et, mint alapot felhasználva. Ebben az algoritmusban is jelentős tervezési hibákat találtak, így röviddel utána több változata is megjelent a RIPEMD-nek így a RIPEMD-160, is ami egy erősített változata az eredeti függvénynek. Felépítése egy kiegészített és részekben lecserélt MD4. 48 lépés helyett 80 lépésben transzformálja a bemenetet és 4 kör helyett 5 körben. Az öt körhöz 5 különböző transzformációt használ fel, amiből kettő teljesen megegyezik az MD4-be felhasznált transzformációs függvényekkel. Ezen felül ami nagy újdonsággal bír az algoritmusban, hogy rendkívül jól párhuzamosítható, hiszen a padding után az algoritmus két szálra válik és pár permutációt felhasználva ez jól látszik a 4.5 ábrán is, illetve a transzformációs függvény sorrendjét felcserélve kvázi ugyanazokat a transzformációkat hajtja végre.

$$F(X,Y,Z) = X \oplus Y \oplus Z$$
$$G(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$H(X,Y,Z) = (X \vee \neg Y) \oplus Z$$
$$I(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$J(X,Y,Z) = X \oplus (Y \vee \neg Z)$$

Az F és G transzformációs függvények teljesen megegyeznek az MD4-nél felsorolt F és H függvénnyel illetve az I nagy hasonlóságot mutat az G függvénnyel. A két párhuzamos szál miután véget ér az értékeket szimpla moduláris összeadással összeadja és véglegesíti, ez a véglegesített érték lesz a hash.

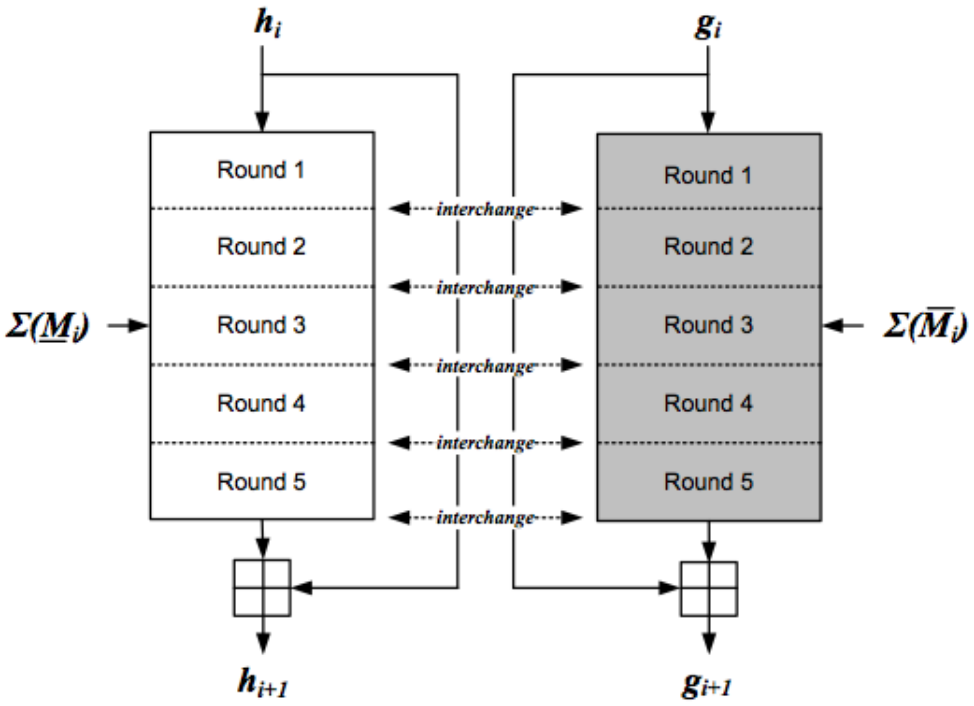


4.5. ábra. RIPEMD-160 tömörítő függvénye (http://ru.wikipedia.org/wiki/RIPEMD-160)

4.6. HAS-V algoritmus

A HAS-V [8] algoritmus egy változó hosszal rendelkező koreai hash függvény. Ez a függvény is erős hasonlóságot mutat az előzőekben felsoroltakkal, ráadásul érdekességként megjegyezhető, hogy az Grazi Egyetem kutató publikálták a [9] cikket amiben az algoritmus sebezhetőségeit tárgyalják és az egyik szerző az NIST

által versenyeztetett SHA-3 függvény címért induló Gröstl függvény kutatója.



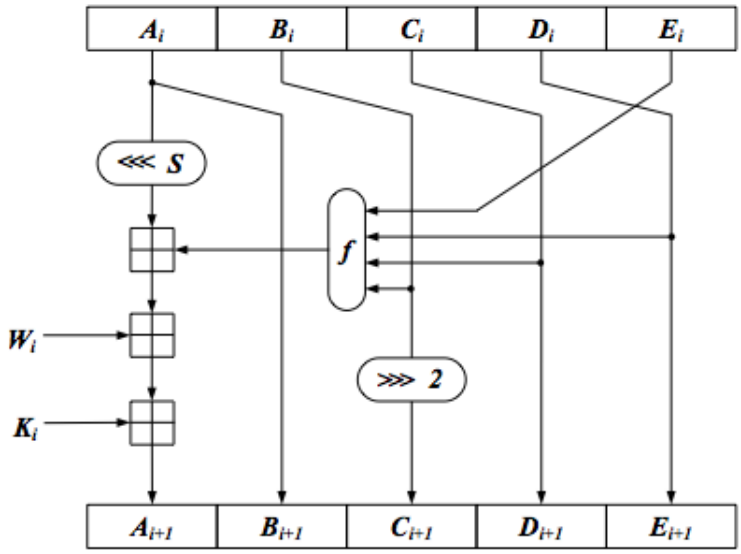
4.6. ábra. HAS-V tömörítő függvénye [9]

A függvény 1024bites blokkokra bontja az üzenet majd abból 128, 160, 192, 224, 256, 288 ill 320bites hash értéket készít. Mindezt a RIPEMD-hez hasonlóan könnyen párhuzamosítható, mivel két szálon titkosítja a bemenetet és minden egyes kör után kicseréli a szálak között az eredményeket.

Az algoritmus 5 kör alatt 100 lépésben transzformálja az üzenet blokkokat, minden körben más függvénnyel, mint az előző hash függvényeknél is kivéve az SHA-2 családot. A függvények a következőek:

$$F(B,C,D,E) = (B \wedge C) \oplus (\neg B \wedge D) \oplus (C \wedge E) \oplus (D \wedge E)$$
$$G(B,C,D,E) = (B \wedge D) \oplus C \oplus E$$
$$H(B,C,D,E) = (B \wedge C) \oplus (\neg B \wedge E) \oplus D$$
$$I(B,C,D,E) = B \oplus (C \wedge D) \oplus E$$
$$J(B,C,D,E) = (\neg B \wedge C) \oplus (B \wedge D) \oplus (C \wedge E) \oplus (D \wedge E)$$

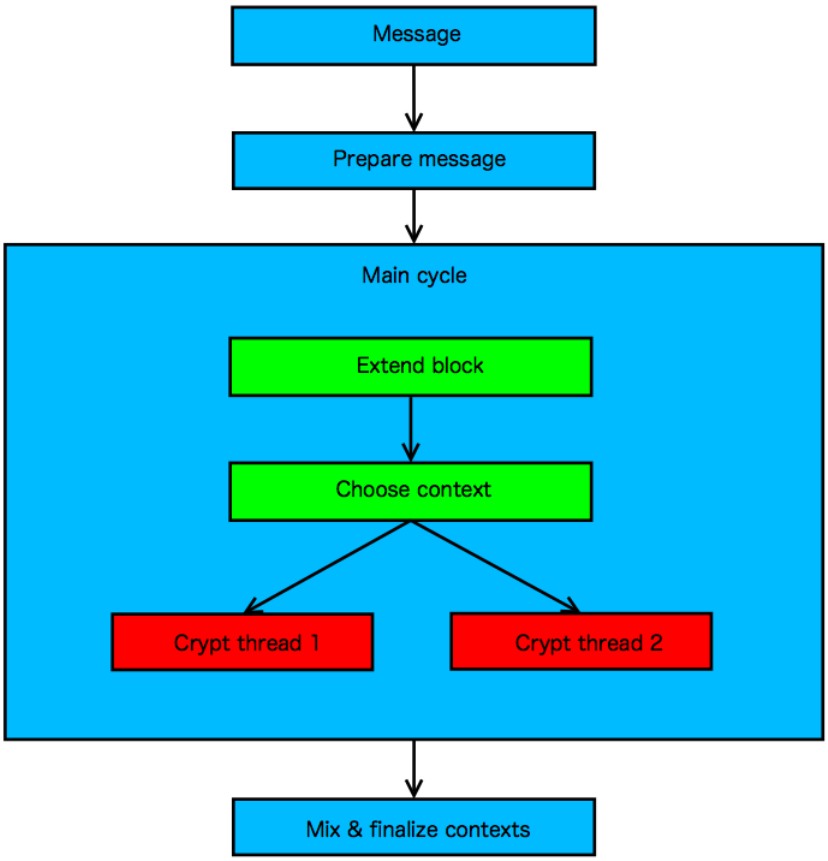
Hasonlóságok az előzőleg említett algoritmusok transzformációival összehasonlítva is találhatók, de mivel eggyel több dimenziósak a függvények ezért nem teljesen egyeznek meg.



4.7. ábra. HAS-V egy lépése [9]

5. Függvény specifikációja

A Paralellizable Variable Length Hash függvény magában hordozza az MD4, MD5 függvény jó tulajdonságait, egy az SHA-1-ben található dúsító algoritmushoz hasonló függvényt illetve az SHA-2-höz hasonló függvény családot hoz létre, minthogy inkább egy egyedül álló függvényt. A PVL magasan skálázható, a struktúrája könnyen engedi a változtatásokat. Egyszerűen implementálható és remélhetőleg ütközésmentes és magas szintű biztonságot ad a preimage és second preimage támadások ellen. Védelmet nyújt a message length extension ellen illetve változó hosszal rendelkező hash-t generál.



5.1. ábra. Parallelizable Variable Length Hash függvény sematikus ábrázolása

5.1. Változó hash méret

Az algoritmus, mint az előzőleg ismertetett hash függvények is, közel áll a számítógépes architektúrákhoz, szándékosan hiszen így egy könnyen implementálható és gyorsan használható lineáris időbonyolultsággal és konstans tárral rendelkező algoritmust kapunk. A tervezés 32bites illetve 64bites Intel x86 illetve x64-es architektúrákat vesz figyelembe, viszont az algoritmus könnyen átírható más egyéb architektúrákra is, amelyek más szó hosszúságokkal rendelkeznek. Pontosan ebből az okból kifolyólag az algoritmus 32bites részblokkokkal (továbbiakban szavakkal) illetve 64bites szavakkal végzi a műveleteket. 32bites architektúrán az algoritmus három különböző hosszúságú hash-t képest generálni

- 160bit, az üzenet minimum hossza: 0byte
- 320bit, az üzenet minimum hossza: 59byte
- 480bit, az üzenet minimum hossza: 123byte

Ugyanígy 64bites architektúrán 64bites szavakkal operálva a duplája igaz a kimenet hosszára:

- 320bit, az üzenet minimum hossza: 0byte
- 640bit, az üzenet minimum hossza: 59byte
- 960bit, az üzenet minimum hossza: 123byte

A következőkben a feltételezett architektúra a egyszerűség kedvéért a 32bites Intel architektúra lesz, és az alapértelmezett hossza a hash-nek pedig 160bit-es, ahol ettől eltér a diplomamunka ott jelölve van.

5.2. Hash függvény sematikus leírása

A PVL hash függvény nagyon hasonló struktúrával rendelkezik, mint az előzőleg ismeretetett hash függvények. Részben azért mert ettől eltérni nem érdemes, másrészt pedig a Merkle-Damgå konstrukció miatt. Ezt a sematikus leírást a 5.1 is nagyon jól szemlélteti.

5.2.1. A bemenet előkészítése

A hash függvény megkapja a bemenetet amit a feldolgozáshoz, titkosításhoz először elő kell készítenie. Az előkészített üzenetnek muszáj oszthatónak lennie a blokk mérettel, vagyis 512bit (64bites módban 1024bit) többszöröse kell legyen. Ha ez nem teljesül, akkor ki kell egészíteni az üzenetet. A Merkle-Damgård konstrukció szerint, az üzenet végére kell írni egy 1-es bitet amivel az üzenet lezárásra

kerül, majd feltölteni 0-ával úgy, hogy az utolsó blokkban maradjon még hely ahova bekerül az eredeti üzenet hossza. Ez a konstrukció ugye megerősíti a bemenetet és biztonságosabb, kevésbé sebezhetővé teszik bizonyos támadások ellen, viszont az üzenet kiterjesztő támadás (length extension atack) ellen nem hatásos. A length extension attack ellen két nagyon egyszerű módszer is védelmet nyújt:

- Truncation: az üzenetből készült hash megcsonkítása, bizonyos számú bit levágása vagy elhagyása a készült hashből. Így a támadó az egész hash-t nem tudja felhasználni a következő blokk titkosításakor IV-knek
- Prefix kódok: Ha minden üzenetet úgy konstruálunk meg, hogy az üzenet első blokkja önmagában csak az üzenet hossza lesz, akkor ezzel elérjük, hogy az üzenet prefix kódként viselkedik, hiszen egyik előkészített üzenet sem lesz egy másik prefixe.

Ezekkel a módszerekkel amik megtalálhatók a Merkle-Damgaåd Revisited : how to Construct a Hash Function-ben könnyűszerrel megakadályozható az üzenet kiterjesztő támadás, a második a prefix kódos módszer buktatója, hogy például stream-en érkező üzenetet nem tudunk vele hashelni, mivel nem tudjuk mekkora a végleges mérete, hiszen a paddingoláshoz szükséges, hogy rögtön az elején tudjuk az üzenet hosszát. Ennek ellenére, a könnyű megvalósíthatósága és a támadás kivédése miatt a tervezéskor felhasználásra kerül. A truncation vagyis az üzenet megcsonkításának (truncation) módszere nem kerül felhasználásra a tervezés során, viszont ezzel az algoritmus bármikor percek alatt kiegészíthető.

A bemenet előkészítése így a következő lépésekből áll:

- megállapítjuk az üzenet hosszát majd létrehozunk egy blokkot ami csak a bináris reprezentációját tartalmazza az üzenet hosszának. Ez lesz az első blokkja a kiterjesztett üzenetnek.
- az üzenetet a blokkok hosszára vágjuk, majd az utolsó blokkot úgy egészítjük ki, hogy a végére egy 1-es bitet írunk, majd kiegészítjük annyi 0-ás bittel, hogy 32bites módban 480bit hosszú legyen illetve 64bites módban 960bites. A maradék 32 illetve 64bitre pedig beírjuk az üzenet hosszát ismét, bináris reprezentációban.

Lényegében a módszer megegyezik a 3.4.1-es pontban ismertetettekkel, csak az első blokk megjelenése az újdonság.

5.2.2. Iteratív ciklus

A hash algoritmus fő része a ciklusban végrehajtódó belső függvények, ezek a függvények végzik a tényleges titkosítása, az üzenet blokkok és azon részblokkjainak összekeverését. A ciklus előre meghatározott lépés számban hajtódik végre, ami a blokkok számával egyezik meg:

2. Állítás. *Legyen i az iterációk száma, M az eredeti üzenet mérete, b a blokk méret és c pedig az architektúra szavának a mérete, akkor*

$$i = \begin{cases} (M/b) + 1 & ha\ M(mod\ b) \leq b - c \\ (M/b) + 2 & ha\ M(mod\ b) > b - c \end{cases}$$

i értéke az üzenet méretétől függ csak, az iterációk számára két eset lehetséges, és ezek az esetek 1 iterációban térnek csak el, még pedig azért mert az üzenet hossza lehet a paddingolásnál használt 1, 0 bitek és hossz konkatenálásával túllépné a megengedett blokk méretet. A következő lépések az összekeverés illetve a véglegesítés kivételével mind a ciklusban futnak.

5.2.3. Blokkok előkészítése

Mivel az algoritmus 4 körben és 64 lépésben titkosítja az üzenetet ezért minden egyes lépésben más rész blokkra (továbbiakban szóra) lesz szükség. Architektúrától függetlenül a szavak száma egy blokkban csak 16, így szükség van még 48 másikra. Az SHA-1-ben erre megoldásként azt használták fel, hogy az új szavakat 4 eredeti vagy előzőleg előállított szó bináris "kizáró vagyolásából" állítják elő. Az általam használt módszer hasonló, az eredeti szavakból 48 új szó kerül előállításra a következő módon.

3. Állítás. *Legyen W egy rendezett halmaz, amelynek elemei a blokk* 16 *eredeti szavai. Ezekre a szavakra egy index formájában tudunk hivatkozni, legyen W_i az i.-ik eredeti szó. Legyen N egy rendezett halmaz, ami az új 32 szót fogja tartalmazni, ez hasonlóan indexelhető, mint a W halmaz és*

$$\begin{aligned} &j = i\ (mod\ 16), \\ &h = i\ /\ 16\ \textit{egész része}, \\ &k = i/4\ \textit{egész része}, \\ &w_i \in W\ \textit{és}\ n_i \in N \end{aligned}$$

akkor az új szavak a következő képpen állnak elő:

$$\begin{aligned} n_k = (W_{P^h(j)} \lll 3) \oplus (w_{P^h(j+1)} \lll 3) \oplus (w_{P^h(j+2)} \lll 3) \\ \oplus (w_{P^h(j+3)} \lll 3) \qquad (i = 1, 2, \cdots, 192) \end{aligned}$$

Ahol $P \in S_{16}$ egy invertálható permutáció aminek a rendje minimum 12.

A P permutáció rendje szükséges, hogy minimum 12 legyen, mert az új 48 szót külön-külön 4 eredeti szó felhasználásával állítjuk elő. Ha a permutáció megfelel a feltételeknek, akkor garantálja, hogy minden új generált szó legalább 1 eredeti szóban eltérjen, így nem jöhet létre két megegyező.
A tervezéskor felhasznált permutáció aminek a rendje 15:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 3 & 7 & 5 & 9 & 13 & 11 & 15 & 2 & 16 & 4 & 8 & 12 & 10 & 14 & 1 & 6 \end{pmatrix}$$

5.2.4. Kontextus választás

Az algoritmus titkosító részet az 5.1 ábrán ábrázolja. A blokkokat az algoritmus két különböző szálnak adja át, amelyek 2x64 lépésen keresztül transzfomálja, majd az elkészült részhasheket visszaadja. A szálak strukturált halmazokat kapnak, amikben az IV-k, konstansok illetve az aktuális kiterjesztett blokk van. Ezeket a strukturált halmazok a következőkben kontextusoknak fogom nevezni. A szálakhoz tartozik külön-külön egy kontextus ami a szálnak megfelelő értékeket tartalmazza. Ebben a pontban három lehetséges állapot alakulhat ki annak függvényében, hogy mekkora hash-t szeretnénk generáltatni a függvénnyel. Mint fentebb említve volt, a függvény 3 módban képes lenyomatokat készíteni, 3 féle különböző hosszal, 160/320 biteseket, 320/640 bitekeset illetve 480/960biteseket ezek sorban 2, 4 illetve 6 lenyomatból állnak véglegesen össze, egy blokkból két lenyomatot generálunk, két szálon, viszont hosszabb bemenet esetén, hosszabb hasht igényelve több blokkal több 4 vagy 6 szálon generálva számukkal megegyező hasht keverjük össze az elkészült hash-eket. Mivel a szálak, a blokkok erejéig teljesen függetlenek egymástól ezért idő veszteség nélkül tökéletesen párhuzamosíthatók, így akár esetenként 3 blokkot, 6 szálon egymástól függetlenül egy időben számolhatunk. A blokkokhoz a következő módon történik a kontextus választás (a szálak külön kontextusát itt egyben kezelem, mivel ugyanazt a blokkot dolgozza fel a szál pár)

- 160/320bites hash: minden blokkhoz ugyanaz az első kontextust használja
- 320/640bites hash: minden páratlan blokkhoz az első, minden páros blokkhoz a páros kontextust használja
- 480/960bites hash: minden 3-mal való maradékos osztás 0-át adó maradék esetén az első, 1-et adó maradék esetén a második és minden 2-őt adó maradék esetén a harmadik kontextust használja

Vagyis sorban minden kontextusra jut egy blokk minden esetben, ha ezek titkosítása megtörtént az új blokkok feldolgozása jön sorban.

5.2.5. IV-k és round key-ek

Az inicializációs vektorok és az algoritmusban használt konstansok nagyon fontosak az algoritmusban, viszont az értékük lényegtelen, hiszen bármilyen kezdő értékkel ugyanolyan biztonságot kell, hogy nyújtson a függvény. Ezek a vektorok és konstansok értékei, máskülönben a π hexadecimális formában ábrázolt

számjegyei lesznek megfelelő sorrendben. A működéshez 10db IV-re, 5-5 IV a 2 párhuzamos szálnak illetve 24db konstans érték (továbbiakban round key) a 4 körhöz minimum 3 és maximum 6 szálhoz. Minden körben és minden szálban más értéket használva körönként. A felhasznált konstansok listája:

- 32bites IV-k és round keyek:

$$
\begin{aligned}
IV_0 &= 0x7CC43B89 \\
IV_1 &= 0x473215D9 \\
IV_2 &= 0x165FA266 \\
IV_3 &= 0x80957705 \\
IV_4 &= 0x93CC7314 \\
IV_5 &= 0x211A1477 \\
IV_6 &= 0xE6AD2065 \\
IV_7 &= 0x77B5FA86 \\
IV_8 &= 0xC75442F2 \\
IV_9 &= 0xFB9D35CF
\end{aligned}
$$

$$
\begin{aligned}
K = \{\ & 0x3C971814, 0x6B6A70A1, 0x687F3584, 0x52A0E286, 0xB79C5305, \\
& 0xAA500737, 0x3E07841C, 0x7FDEAE5C, 0x8E7D44EC, 0x5716F2B8, \\
& 0xB03ADA37, 0xF0500C0D, 0xF01C1F04, 0x0200B3FF, 0xAE0CF51A, \\
& 0x3CB574B2, 0x25837A58, 0xDC0921BD, 0xD19113F9, 0x7CA92FF6, \\
& 0x94324773, 0x22F54701, 0x3AE5E581, 0x37C2DADC\ \}
\end{aligned}
$$

- 64bites IV-k és roundkeyek:

$$
\begin{aligned}
IV_0 &= 0x4B7A70E9B5B32944 \\
IV_1 &= 0xDB75092EC4192623 \\
IV_2 &= 0xAD6EA6B049A7DF7D \\
IV_3 &= 0x9CEE60B88FEDB266 \\
IV_4 &= 0xECAA8C71699A17FF \\
IV_5 &= 0x5664526CC2B19EE1 \\
IV_6 &= 0x193602A575094C29 \\
IV_7 &= 0xA0591340E4183A3E \\
IV_8 &= 0x3F54989A5B429D65 \\
IV_9 &= 0x6B8FE4D699F73FD6
\end{aligned}
$$

$$K = \{\ 0xC75442F5FB9D35CF, 0xEBCDAF0C7B3E89A0,$$
$$0xD6411BD3AE1E7E49, 0x00250E2D2071B35E, 0x226800BB57B8E0AF,$$
$$0x2464369BF009B91E, 0x5563911D59DFA6AA, 0x78C14389D95A537F,$$
$$0x207D5BA202E5B9C5, 0x832603766295CFA9, 0x11C819684E734A41,$$
$$0xB3472DCA7B14A94A, 0x1B5100529A532915, 0xD60F573FBC9BC6E4,$$
$$0x2B60A47681E67400, 0x08BA6FB5571BE91F, 0xF296EC6B2A0DD915,$$
$$0xB6636521E7B9F9B6, 0xFF34052EC5855664, 0x53B02D5DA99F8FA1,$$
$$0x08BA47996E85076A\ \}$$

5.2.6. Titkosítás

A titkosítás vagyis a lényegi rész az 5.2 ábra szerint hajtódik végre. A teljes titkosítás egy szálon 64 lépésből áll, amit 4 körbe rendezünk, minden körben más és más F függvényt használunk, illetve különböző rész blokkokat (szavakat). A 4 körben 4 különböző F függvényen kívül 4különböző round key-t is használunk, amik szálonként természetesen változnak. A 4 körnek megfelelő 4 függvény sorrendben:

$$F(A,B,C,D) = A \oplus B \oplus C \oplus D$$
$$G(A,B,C,D) = (B \wedge D) \oplus (A \wedge C) \oplus (A \wedge D)$$
$$H(A,B,C,D) = (A \wedge B) \oplus \neg B \oplus D$$
$$I(A,B,C,D) = (B \wedge C) \vee (\neg D \wedge A) \oplus C$$

Mint a 3.5.1 pontban ismertetve volt egy iterációs hash függvény a következőképpen kell, hogy kinézzen:

$$H_1(x) = f(h_i, x_i), \text{ ahol } h_i = \begin{cases} IV & \text{ha } i = 0 \\ f(h_{i-1}, x_{i-1}) & \text{ha } i > 0 \end{cases}$$

Az első lépésben a blokk egyik szavát az IV-vel titkosítjuk, majd egy másikat a már kiszámolt értékkel és így tovább még el nem fogynak a lépések. Ha végigértünk mind a 4 körön, 64 lépésen, akkor megkapjuk a végleges hash-t. Ezt kicsit kiegészítve egy R függvénnyel ami egy permutáció, különböző és lenyomatot kapunk a következő feltételekkel:

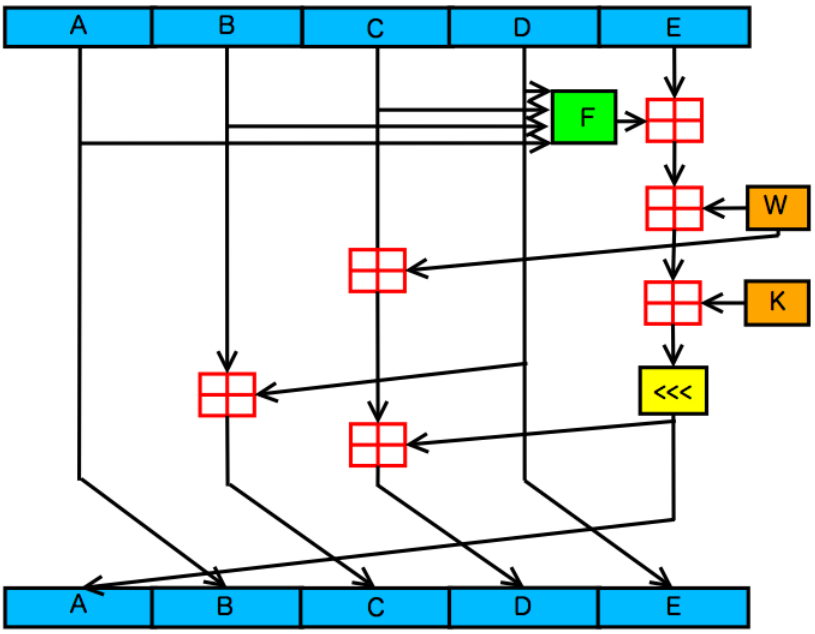
4. Állítás. *Legyenek f_r tömörítő függvények ($r = 1,2,3,4$), $H_2(X)$ egy iterált hash függvény, $R \in S_n$ egy invertálható permutáció úgy, hogy $R^2 = id$ úgy hogy n az adott architektúra szavának mérete, legyen*

$$H_2(x) = f_r(h_i, R(x_i)),\ ahol\ h_i = \begin{cases} IV & ha\ i = 0 \\ f_r(h_{i-1}, R(x_{i-1})) & ha\ i > 0 \end{cases}$$

Ekkor $H_1(x) \neq H_2(x)$ és ha a két hash függvény ütközésmentes, akkor $H_1(x)||H_2(x)$ is ütközésmentes marad Nandi tétele szerint.

Mivel a két hash konkatenálva illetve külön külön is ütközésmentes, ezért az összegük sem lesz különböző.

A szál pároknál, amik ugyanazt a bemenetet kapják a különbség az IV és round key-eken kívül még az, hogy minden egyes blokkra a 2. szál egy előre definiált R permutációval is rendelkezik, ami a blokk szavaiban a biteket sorrendjét megváltoztatja így elérjük, hogy a két szál ugyanazzal a bemenettel két különböző hash-t generáljon, ugyanolyan biztonsággal.



5.2. ábra. Parallelizable Variable Length Hash függvény egy lépése

A titkosítóról szóló ábra 5.2 minden lépésben transzformálja a belső 5 állapotot (A, B, C, D, E), ehhez csak az állapotok, a körnek megfelelő $F()$ függvény, K round key, R forgatás mennyisége bitekben illetve a soron következő W blokkrész kell (A balra forgatást a következőképpen jelöljük: $\lll$, matematikai megfelelője: $x \lll s := x * 2^s + \frac{x}{2^{((n-s) mod (n))}}$):

$$
\begin{aligned}
B &= A \\
C &= B + D \\
D &= C + W + (F(A, B, C, D) + E + W + K) \lll R \\
E &= D \\
A &= (F(A, B, C, D) + E + W + K) \lll R
\end{aligned}
$$

Így minden lépésben az értékek egy belső állapottal jobbra csúsznak (A a B-be, D az E-be) illetve összekeverednek egymással, a round key-ekkel és az részblokokkal. A tervezéskor felhasznált 32bites illetve 64bites permutációk a következők:

$$P_{32} = \begin{pmatrix} 1 & 12 \\ 12 & 1 \end{pmatrix} \begin{pmatrix} 2 & 19 \\ 19 & 2 \end{pmatrix} \begin{pmatrix} 3 & 21 \\ 21 & 3 \end{pmatrix} \begin{pmatrix} 4 & 13 \\ 13 & 4 \end{pmatrix} \begin{pmatrix} 5 & 15 \\ 15 & 5 \end{pmatrix} \begin{pmatrix} 6 & 31 \\ 31 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 7 & 8 \\ 8 & 7 \end{pmatrix} \begin{pmatrix} 9 & 24 \\ 24 & 9 \end{pmatrix} \begin{pmatrix} 10 & 20 \\ 20 & 10 \end{pmatrix} \begin{pmatrix} 11 & 29 \\ 29 & 11 \end{pmatrix} \begin{pmatrix} 14 & 25 \\ 25 & 14 \end{pmatrix} \begin{pmatrix} 16 & 22 \\ 22 & 16 \end{pmatrix} \begin{pmatrix} 17 & 28 \\ 28 & 17 \end{pmatrix}$$

$$\begin{pmatrix} 18 & 27 \\ 27 & 18 \end{pmatrix} \begin{pmatrix} 23 & 32 \\ 32 & 23 \end{pmatrix} \begin{pmatrix} 26 & 30 \\ 30 & 26 \end{pmatrix}$$

$$P_{64} = \begin{pmatrix} 1 & 61 \\ 61 & 1 \end{pmatrix} \begin{pmatrix} 2 & 60 \\ 60 & 2 \end{pmatrix} \begin{pmatrix} 3 & 35 \\ 35 & 3 \end{pmatrix} \begin{pmatrix} 4 & 62 \\ 62 & 4 \end{pmatrix} \begin{pmatrix} 5 & 55 \\ 55 & 5 \end{pmatrix} \begin{pmatrix} 6 & 48 \\ 48 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 7 & 11 \\ 11 & 7 \end{pmatrix} \begin{pmatrix} 8 & 52 \\ 52 & 8 \end{pmatrix} \begin{pmatrix} 9 & 46 \\ 46 & 9 \end{pmatrix} \begin{pmatrix} 10 & 64 \\ 64 & 10 \end{pmatrix} \begin{pmatrix} 12 & 43 \\ 43 & 12 \end{pmatrix} \begin{pmatrix} 13 & 34 \\ 34 & 13 \end{pmatrix} \begin{pmatrix} 14 & 42 \\ 42 & 14 \end{pmatrix}$$

$$\begin{pmatrix} 15 & 53 \\ 53 & 15 \end{pmatrix} \begin{pmatrix} 16 & 40 \\ 40 & 16 \end{pmatrix} \begin{pmatrix} 17 & 30 \\ 30 & 17 \end{pmatrix} \begin{pmatrix} 18 & 25 \\ 25 & 18 \end{pmatrix} \begin{pmatrix} 19 & 63 \\ 63 & 19 \end{pmatrix} \begin{pmatrix} 20 & 59 \\ 59 & 20 \end{pmatrix} \begin{pmatrix} 21 & 31 \\ 31 & 21 \end{pmatrix}$$

$$\begin{pmatrix} 22 & 26 \\ 26 & 22 \end{pmatrix} \begin{pmatrix} 23 & 58 \\ 58 & 23 \end{pmatrix} \begin{pmatrix} 24 & 45 \\ 45 & 24 \end{pmatrix} \begin{pmatrix} 27 & 49 \\ 49 & 27 \end{pmatrix} \begin{pmatrix} 28 & 47 \\ 47 & 28 \end{pmatrix} \begin{pmatrix} 29 & 51 \\ 51 & 29 \end{pmatrix} \begin{pmatrix} 30 & 17 \\ 17 & 30 \end{pmatrix}$$

$$\begin{pmatrix} 31 & 21 \\ 21 & 31 \end{pmatrix} \begin{pmatrix} 32 & 44 \\ 44 & 32 \end{pmatrix} \begin{pmatrix} 33 & 38 \\ 38 & 33 \end{pmatrix} \begin{pmatrix} 36 & 54 \\ 54 & 36 \end{pmatrix} \begin{pmatrix} 37 & 39 \\ 39 & 37 \end{pmatrix} \begin{pmatrix} 41 & 57 \\ 57 & 41 \end{pmatrix} \begin{pmatrix} 50 & 56 \\ 56 & 50 \end{pmatrix}$$

A ciklus itt befejeződik ha a blokkok elfogytak, ellenkező esetben az 5.2.3 ponttól újra előkészítjük a következő blokkot, blokkokat, kontextust választunk majd titkosítjuk.

5.3. Véglegesítés

A blokkok titkosításának végeztével 2, 4 vagy 6 szálhoz sorrendben 1, 2 vagy 3 kontextus tartozik, a szálok számának megfelelő lenyomatokkal. Ezekből a hashekből készül el a végleges hash amit az algoritmus úgy állít elő, hogy összekeveri azokat. Erre a keverésre használjuk fel a $J(X, i, s)$ függvényt ami minden egyes szál eredményéből kiválasztva egy részlenyomatot (belső állapotot) vagy esetleg többet és a megadott módon keveri össze az értékeket, majd adja vissza a végleges hash egy részét.

5.3.1. Definíció. Legyenek $J(X, i, s)$ keverő függvény, ahol X az elkészült részlenyomatok rendezett halmaza, i a részlenyomatokra hivatkozó index, s pedig a szálak száma, ekkor J-t a következőképpen definiáljuk:

$$J(X,i,s) = \begin{cases} X_i + X_{i+5} & \text{ha } s = 2, i = (1,2,\cdots,5) \\ (X_{i+15} \vee X_i) \oplus X_{i+10} \oplus \neg(X_{i+5} \wedge X_{i+17}) & \text{ha } s = 4, i = (1,2,\cdots,10) \\ (X_{i+15} \vee X_i) \oplus X_{i+20} \oplus \neg(X_{i+5} \wedge X_{i+17}) & \text{ha } s = 6, i = (1,2,\cdots,15) \end{cases}$$

X_i az $i/5$-ik szál i *mod* 5 belső állapotára (részhashére) hivatkozik. A szálak számának megfelelően így előállíthatunk három különböző méretű hash-t a következő módon:

5.3.2. Definíció. Legyenek $V(X, s)$ véglegesítő függvény, ahol X az elkészült részlenyomatok rendezett halmaza, s pedig a szálak száma, ekkor V-t a következő képpen definiáljuk:

$$V(X,s) = \begin{cases} \prod_{i=1}^{5} J(X,i,s) & \text{ha } s = 2 \\ \prod_{i=1}^{10} J(X,i,s) & \text{ha } s = 4 \\ \prod_{i=1}^{15} J(X,i,s) & \text{ha } s = 6 \end{cases}$$

Ahol $\prod$ a konketanációt jelöli.

5.4. A PVL Hash függvény

A $V(X, s)$ függvénnyel elkészül a végleges lenyomata az üzenetnek, amit először a megerősített Merkle-Damgård konstrukció szerint paddingolunk, illetve prefix kóddá alakítva szálakra bontva titkosítunk, majd a szálak végeredményét $J(X, i, s)$ függvénnyel összekeverjük. Így a függvény eleget kell, hogy tegyen a kriptográfiai hash függvények feltételeinek.

Irodalomjegyzék

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

[2] I. Damgård, *A Design Principle for Hash Functions. In Advances in Cryptology - CRYPTO '89 Proceedings*. Lecture Notes in Computer Science Vol. 435, G. Brassard, ed, Springer-Verlag, pp. 416-427, 1989

[3] Ronald L. Rivest, *Description of MD4, RFC 1320*. http://tools.ietf.org/html/rfc1320, 1990

[4] Ronald L. Rivest, *The MD5 Message-Digest Algorithm, RFC 1321*. http://tools.ietf.org/html/rfc1321, 1991

[5] National Security Agency, *US Secure Hash Algorithm 1 (SHA1), RFC 3174*. http://tools.ietf.org/html/rfc3174, 2001

[6] National Security Agency, *FIPS 180-3: Secure Hash Standard (SHS)*. http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf, 2008

[7] Hans Dobbertin, Antoon Bosselaers, Bart Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*. 1996

[8] Nan Kyoung Park, Joon Ho Hwang, Pil Joong Lee, *HAS-V: A New Hash Function with Variable Output Length*. 2001

[9] Florian Mendel and Vincent Rijmen, *Weaknesses in the HAS-V Compression Function*. 2007

[10] Donghoon Chang, Mridul Nandi, Jesang Lee, Jaechul Sung, Seokhie Hong, *Hash Function Design Principles Supporting Variable Output Lengths from One Small Function*. 2007

[11] Bart Preneel, *Analysis and Design of Cryptographic Hash Functions*. 2003

[12] Yong-Sork Her, Kouichi Sakurai, *A Design of Cryptographic Hash Function Group with Variable Output-Length Based on SHA-1*. 2002

[13] Jean-S'bastien Coron, Yevgeniy Dodis, C'cile Malinaud, and Prashant Puniya, *Merkle-Damgård Revisited: how to Construct a Hash a Function.* Advances in Cryptology – CRYPTO '05 Proceedings, Lecture Notes in Computer Science, Vol. 3621, Springer-Verlag, pp. 21–39., 2005

[14] Johannes A. Buchmann, *Introduction to Cryptography.* 2000

[15] Wolfram Research Inc., *One-Way Function.* http://mathworld.wolfram.com/One-WayFunction.html

A. Függelék: forráskód, pvlh.c

```
1  /*
2   * ----------------------------------------------------------------------
3   * "THE BEER-WARE LICENSE" (Revision 42):
4   * <earthquake[at]rycon[dot]hu wrote this file. As long as you retain
5   * this notice you can do whatever you want with this stuff. If we
6   * meet some day, and you think this stuff is worth it, you can buy
7   * me a beer in return.
8   * thanks Poul-Henning Kamp, for the license :)
9   * ----------------------------------------------------------------------
10  * This is just a TEST code, nothing serious.
11  * The code don't care about the endianess, about the internal
12  * representation, just a test implementation of the Parallelizable
13  * Variable Length hash's schema.
14  */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #include <math.h>
20
21 #if _WIN64 || __amd64__
22         #define USHORT  unsigned short
23         #define UINT    unsigned long int
24         #define ULONG   unsigned long long int
25         #define UCHAR   unsigned char
26         #define SIZEINTBIT      64
27         #define SIZEINTBYTE     8
28
29         #define ROUNDKEYS 0xC75442F5FB9D35CF, 0xEBCDAF0C7B3E89A0, \
30                           0xD6411BD3AE1E7E49, 0x00250E2D2071B35E, \
31                           0x226800BB57B8E0AF, 0x2464369BF009B91E, \
32                           0x5563911D59DFA6AA, 0x78C14389D95A537F, \
33                           0x207D5BA202E5B9C5, 0x832603766295CFA9, \
34                           0x11C819684E734A41, 0xB3472DCA7B14A94A, \
35                           0x1B5100529A532915, 0xD60F573FBC9BC6E4, \
36                           0x2B60A47681E67400, 0x08BA6FB5571BE91F, \
37                           0xF296EC6B2A0DD915, 0xB6636521E7B9F9B6, \
38                           0xFF34052EC5855664, 0x53B02D5DA99F8FA1, \
39                           0x08BA47996E85076A
40
41         #define IV0 0x4B7A70E9B5B32944
```

```
42          #define IV1 0xDB75092EC4192623
43          #define IV2 0xAD6EA6B049A7DF7D
44          #define IV3 0x9CEE60B88FEDB266
45          #define IV4 0xECAA8C71699A17FF
46          #define IV5 0x5664526CC2B19EE1
47          #define IV6 0x193602A575094C29
48          #define IV7 0xA0591340E4183A3E
49          #define IV8 0x3F54989A5B429D65
50          #define IV9 0x6B8FE4D699F73FD6
51
52  USHORT P2[64] = {60, 59, 34, 61, 54, 47, 10, 51, 45, 63, 6,
53          42, 33, 41, 52, 39, 29, 24, 62, 58, 30, 25, 57, 44,
54          17, 21, 48, 46, 50, 16, 20, 43, 37, 12, 2, 53, 38,
55          32, 36, 15, 56, 13, 11, 31, 23, 8, 27, 5, 26, 55, 28,
56          7, 14, 35, 4, 49,       40, 22, 19, 1, 0, 3, 18, 9};
57  #define PVLH "%016lx%016lx%016lx%016lx%016lx"
58
59  #else
60  #define PVLH "%08x%08x%08x%08x%08x"
61          #define USHORT  unsigned short
62          #define UINT    unsigned int
63          #define ULONG   unsigned long long int
64          #define UCHAR   unsigned char
65          #define SIZEINTBIT      32
66          #define SIZEINTBYTE     4
67
68          #define ROUNDKEYS 0x3C971814, 0x6B6A70A1, 0x687F3584, \
69                            0x52A0E286, 0xB79C5305, 0xAA500737, \
70                            0x3E07841C, 0x7FDEAE5C, 0x8E7D44EC, \
71                            0x5716F2B8, 0xB03ADA37, 0xF0500C0D, \
72                            0xF01C1F04, 0x0200B3FF, 0xAE0CF51A, \
73                            0x3CB574B2, 0x25837A58, 0xDC0921BD, \
74                            0xD19113F9, 0x7CA92FF6, 0x94324773, \
75                            0x22F54701, 0x3AE5E581, 0x37C2DADC
76
77          #define IV0 0x7CC43B89
78          #define IV1 0x473215D9
79          #define IV2 0x165FA266
80          #define IV3 0x80957705
81          #define IV4 0x93CC7314
82          #define IV5 0x211A1477
83          #define IV6 0xE6AD2065
84          #define IV7 0x77B5FA86
85          #define IV8 0xC75442F2
86          #define IV9 0xFB9D35CF
87
88          USHORT P2[32] = {11, 18, 20, 12, 14, 30, 7, 6, 23,
89                  19, 28, 0, 3, 24, 4, 21, 27, 26, 1, 9, 2, 15, 31,
90                  8, 13, 29, 17, 16, 10, 25, 5, 22};
91  #endif
92
```

```
93
94  #define ROTATE_LEFT(X, n) (((X) << (n)) | ((X) >> (32-(n))))
95  #define F(A,B,C,D) ((A) ^ (B) ^ (C) ^ ~(D))
96  #define G(A,B,C,D) (((B) & (D)) ^ ((A) & (C)) ^ ((A) & (D)))
97  #define H(A,B,C,D) (((A) & (B)) ^ ~(B) ^ (D))
98  #define I(A,B,C,D) (((B) & (C)) | (~(D) & (A)) ^ (C))
99
100 #define J(A,B,C,D,E) (((A) | (B)) ^ (C) ^ ~((D) & (E)))
101 #define R0 3
102 #define R1 11
103 #define R2 17
104 #define R3 5
105
106 UINT K[24] = { ROUNDKEYS };
107
108 USHORT P1[16] = {2,6,4,8,12,10,14,
109                  1,15,3,7,11,9,13,0,5};
110
111 struct pvlh_CTX {
112         UINT a[2];
113         UINT b[2];
114         UINT c[2];
115         UINT d[2];
116         UINT e[2];
117         UINT w[64];
118         ULONG size;
119         USHORT number;
120 };
121
122 void init_context(struct pvlh_CTX *context, USHORT number)
123 {
124         memset(context->w, '\0', 64*sizeof(UINT));
125
126         context->a[0] = IV0;
127         context->a[1] = IV1;
128         context->b[0] = IV2;
129         context->b[1] = IV3;
130         context->c[0] = IV4;
131         context->c[1] = IV5;
132         context->d[0] = IV6;
133         context->d[1] = IV7;
134         context->e[0] = IV8;
135         context->e[1] = IV9;
136
137         context->number = number;
138 }
139
140 void extend_context(struct pvlh_CTX *context)
141 {
142         UINT i, d, j;
143         UINT *w;
```

```
144
145         w = context->w;
146
147         for (i = 0; i <  192; i++)
148         {
149                 d = i % 16;
150                 for (j=0; j < (i / 16) ; j++) d = P1[d];
151                 w[i / 4 + 16] ^= ROTATE_LEFT(w[d], 3);
152         }
153 }
154
155 UINT permutation(UINT m)
156 {
157         UINT mp = 0;
158         UINT i;
159
160         for (i = 0 ; i < SIZEINTBIT ; i++)
161         {
162                 mp |= (((m >> i) & 0x01) << (P2[i]));
163         }
164
165         return mp;
166 }
167
168 void crypt_context(struct pvlh_CTX *context)
169 {
170         UINT i, k;
171         UINT a, b, c, d, e;
172         UINT *w;
173         UINT temp, temp2, rotate;
174         USHORT n;
175
176         a = context->a[0];
177         b = context->b[0];
178         c = context->c[0];
179         d = context->d[0];
180         e = context->e[0];
181         w = *(&context->w);
182         n = context->number*8;
183
184         for (i = 0; i < 64; i++)
185         {
186                 if (i < 16)
187                 {
188                         rotate = R0;
189                         k = K[n+0];
190                         temp = F(a, b, c, d);
191                 }
192                 if ((i >= 16) && (i < 32))
193                 {
194                         rotate = R1;
```

```
195                     k = K[n+1];
196                     temp = G(a, b, c, d);
197             }
198             if ((i >= 32) && (i < 48))
199             {
200                     rotate = R2;
201                     k = K[n+2];
202                     temp = H(a, b, c, d);
203             }
204             if ((i >= 48))
205             {
206                     rotate = R3;
207                     k = K[n+3];
208                     temp = I(a, b, c, d);
209             }
210
211             temp = ROTATE_LEFT((temp + w[i] + k + e), rotate);
212             temp2 = b + d;
213             e = d;
214             d = c + w[i] + temp;
215             c = temp2;
216             b = a;
217             a = temp;
218     }
219     context->a[0] += a;
220     context->b[0] += b;
221     context->c[0] += c;
222     context->d[0] += d;
223     context->e[0] += e;
224
225     /*
226      * second circle, with permutation
227      */
228
229     a = context->a[1];
230     b = context->b[1];
231     c = context->c[1];
232     d = context->d[1];
233     e = context->e[1];
234     w = *(&context->w);
235
236     for (i = 0; i < 64; i++)
237     {
238             if (i < 16)
239             {
240                     rotate = R0;
241                     k = K[n+4];
242                     temp = F(a, b, c, d);
243             }
244             if ((i >= 16) && (i < 32))
245             {
```

```
246                         rotate = R1;
247                         k = K[n+5];
248                         temp = G(a, b, c, d);
249                 }
250                 if ((i >= 32) && (i < 48))
251                 {
252                         rotate = R2;
253                         k = K[n+6];
254                         temp = H(a, b, c, d);
255                 }
256                 if ((i >= 48))
257                 {
258                         rotate = R3;
259                         k = K[n+7];
260                         temp = I(a, b, c, d);
261                 }
262
263                 temp = ROTATE_LEFT((temp + permutation(w[i]) + k + e),
264                         rotate);
265                 temp2 = b + d;
266                 e = d;
267                 d = c + w[i] + temp;
268                 c = temp2;
269                 b = a;
270                 a = temp;
271         }
272
273         context->a[1] += a;
274         context->b[1] += b;
275         context->c[1] += c;
276         context->d[1] += d;
277         context->e[1] += e;
278 }
279
280 void prepare_context(UCHAR *plain, struct pvlh_CTX **context,
281         USHORT type)
282 {
283         UINT i = 0, j = 0, k = 0;
284         ULONG l = strlen(plain);
285         ULONG s = 0;
286         UINT temp;
287         UCHAR *p;
288
289         s = ((l + (8*SIZEINTBYTE*2) +
290                 (2*SIZEINTBYTE))/(SIZEINTBIT*2) + 1) * (SIZEINTBIT*2);
291
292         if ((p = malloc(s)) == NULL)
293         {
294                 printf("[-] Out of memory.\n");
295                 exit(-1);
296         }
```

```
297         memset(p, 0, s);
298
299         /*
300          * There is no 128bit long integer in the ansi C, so we drop
301          * the 2^128bit long messages just to 2^64bit long like in
302          * standard 32bit mode. This problem is solvable with
303          * bignum/gmp or a better implementation
304          */
305
306 #if _WIN64 || __amd64__
307         p[0] = l & 0xFFFFFFFFFFFFFFFF;
308         strncpy(p + (8*SIZEINTBYTE*2), plain, l);
309         p[l+(8*SIZEINTBYTE*2)] = 0x80;
310         p[s-(SIZEINTBYTE)] = l & 0xFFFFFFFFFFFFFFFF;
311 #else
312         p[0] = l >> SIZEINTBIT;
313         p[SIZEINTBYTE] = l & 0xFFFFFFFF;
314         strncpy(p + (8*SIZEINTBYTE*2), plain, l);
315         p[l+(8*SIZEINTBYTE*2)] = 0x80;
316         p[s-(2*SIZEINTBYTE)] = l >> SIZEINTBIT;
317         p[s-SIZEINTBYTE] = l & 0xFFFFFFFF;
318 #endif
319
320         for (j = 0 ; j < (s/(SIZEINTBIT*2)); j++)
321         {
322                 i = 0;
323                 k = 0;
324                 if ((type == 1) && (j % 2)) k = 1;
325                 if ((type == 2) && ((j % 3) == 1)) k = 1;
326                 if ((type == 2) && ((j % 3) == 2)) k = 2;
327                 memset(context[k]->w, 0, 64*sizeof(UINT));
328                 while (i < (SIZEINTBIT*2))
329                 {
330                         temp = p[i+(j*(SIZEINTBIT*2))];
331                         context[k]->w[i / SIZEINTBYTE] |= temp <<
332                                 (8*((SIZEINTBYTE-1)-i%(SIZEINTBYTE)));
333                         i++;
334                 }
335                 extend_context(context[k]);
336                 crypt_context(context[k]);
337         }
338 }
339
340
341 void finalize_context(struct pvlh_CTX **pvlh, USHORT type)
342 {
343         if (type == 2)
344         {
345                 printf(PVLH, J(pvlh[1]->a[1], pvlh[0]->a[0],
346                         pvlh[2]->a[0], pvlh[0]->a[1], pvlh[1]->c[1]),
347                         J(pvlh[1]->b[1], pvlh[0]->b[0], pvlh[2]->b[0],
```

```
348                                     pvlh[0]->b[1], pvlh[1]->d[1]),
349                             J(pvlh[1]->c[1], pvlh[0]->c[0], pvlh[2]->c[0],
350                                     pvlh[0]->c[1], pvlh[1]->e[1]),
351                             J(pvlh[1]->d[1], pvlh[0]->d[0], pvlh[2]->d[0],
352                                     pvlh[0]->d[1], pvlh[2]->a[1]),
353                             J(pvlh[1]->e[1], pvlh[0]->e[0], pvlh[2]->e[0],
354                                     pvlh[0]->e[1], pvlh[2]->b[1]));
355                     printf(PVLH, J(pvlh[2]->a[1], pvlh[1]->a[0],
356                             pvlh[0]->a[0], pvlh[1]->a[1], pvlh[2]->c[1]),
357                             J(pvlh[2]->b[1], pvlh[1]->b[0], pvlh[0]->b[0],
358                                     pvlh[1]->b[1], pvlh[2]->d[1]),
359                             J(pvlh[2]->c[1], pvlh[1]->c[0], pvlh[0]->c[0],
360                                     pvlh[1]->c[1], pvlh[2]->e[1]),
361                             J(pvlh[2]->d[1], pvlh[1]->d[0], pvlh[0]->d[0],
362                                     pvlh[1]->d[1], pvlh[0]->a[1]),
363                             J(pvlh[2]->e[1], pvlh[1]->e[0], pvlh[0]->e[0],
364                                     pvlh[1]->e[1], pvlh[0]->b[1]));
365                     printf(PVLH, J(pvlh[0]->a[1], pvlh[2]->a[0],
366                             pvlh[1]->a[0], pvlh[2]->a[1], pvlh[0]->c[1]),
367                             J(pvlh[0]->b[1], pvlh[2]->b[0], pvlh[1]->b[0],
368                                     pvlh[2]->b[1], pvlh[0]->d[1]),
369                             J(pvlh[0]->c[1], pvlh[2]->c[0], pvlh[1]->c[0],
370                                     pvlh[2]->c[1], pvlh[0]->e[1]),
371                             J(pvlh[0]->d[1], pvlh[2]->d[0], pvlh[1]->d[0],
372                                     pvlh[2]->d[1], pvlh[1]->a[1]),
373                             J(pvlh[0]->e[1], pvlh[2]->e[0], pvlh[1]->e[0],
374                                     pvlh[2]->e[1], pvlh[1]->b[1]));
375                     printf("\n");
376             }
377             else if (type == 1)
378             {
379                     printf(PVLH, J(pvlh[1]->a[1], pvlh[0]->a[0],
380                             pvlh[1]->b[0], pvlh[0]->a[1], pvlh[1]->c[1]),
381                             J(pvlh[1]->b[1], pvlh[0]->b[0], pvlh[1]->c[0],
382                                     pvlh[0]->b[1], pvlh[1]->d[1]),
383                             J(pvlh[1]->c[1], pvlh[0]->c[0], pvlh[1]->d[0],
384                                     pvlh[0]->c[1], pvlh[1]->e[1]),
385                             J(pvlh[1]->d[1], pvlh[0]->d[0], pvlh[1]->e[0],
386                                     pvlh[0]->d[1], pvlh[0]->a[1]),
387                             J(pvlh[1]->e[1], pvlh[0]->e[0], pvlh[0]->a[0],
388                                     pvlh[0]->e[1], pvlh[0]->b[1]));
389                     printf(PVLH, J(pvlh[0]->a[1], pvlh[1]->a[0],
390                             pvlh[0]->b[0], pvlh[1]->a[1], pvlh[0]->c[1]),
391                             J(pvlh[0]->b[1], pvlh[1]->b[0], pvlh[0]->c[0],
392                                     pvlh[1]->b[1], pvlh[0]->d[1]),
393                             J(pvlh[0]->c[1], pvlh[1]->c[0], pvlh[0]->d[0],
394                                     pvlh[1]->c[1], pvlh[0]->e[1]),
395                             J(pvlh[0]->d[1], pvlh[1]->d[0], pvlh[0]->e[0],
396                                     pvlh[1]->d[1], pvlh[1]->a[1]),
397                             J(pvlh[0]->e[1], pvlh[1]->e[0], pvlh[1]->a[0],
398                                     pvlh[1]->e[1], pvlh[1]->b[1]));
```

```
399                     printf("\n");
400             }
401             else if (type == 0)
402             {
403                     printf(PVLH, pvlh[0]->a[0]+pvlh[0]->a[1],
404                             pvlh[0]->b[0]+pvlh[0]->b[1],
405                             pvlh[0]->c[0]+pvlh[0]->c[1],
406                             pvlh[0]->d[0]+pvlh[0]->d[1],
407                             pvlh[0]->e[0]+pvlh[0]->e[1]);
408                     printf("\n");
409             }
410     }
411
412     void *pvlh(char *plain, USHORT type)
413     {
414             USHORT i;
415             struct pvlh_CTX **pvlh = (struct pvlh_CTX **)malloc(
416                     sizeof(struct pvlh_CTX) * 3);
417
418             pvlh[0] = NULL;
419             pvlh[1] = NULL;
420             pvlh[2] = NULL;
421
422             for (i = 0; i <= type; i++)
423             {
424                     if ((pvlh[i] = (struct pvlh_CTX *)malloc(
425                             sizeof(struct pvlh_CTX))) == NULL)
426                     {
427                             printf("[-] Out of memory, struct alloc\n");
428                             exit(-1);
429                     }
430                     init_context(pvlh[i], i);
431             }
432
433             prepare_context(plain, pvlh, type);
434             finalize_context(pvlh, type);
435
436             return NULL;
437     }
438
439     void usage(char *name)
440     {
441             printf("[-] Usage: %s text type\n\n"
442             "types can be:\t\t(x86)\t(x64)\tminimum length\n"
443             "\t\t1 for \t160bit\t320bit\t0byte\n"
444             "\t\t2 for\t320bit\t640bit\t47byte\n"
445             "\t\t3 for\t480bit\t960bit\t111byte\n", name);
446     }
447
448     int main(int argc, char **argv)
449     {
```

```
450         if (argc != 3)
451         {
452                 usage(argv[0]);
453                 exit(0);
454         }
455
456         if (!strncmp(argv[2], "1", 1)) pvlh(argv[1], 0);
457         if (!strncmp(argv[2], "2", 1))
458         {
459                 if (strlen(argv[1])>58) pvlh(argv[1], 1);
460                 else usage(argv[0]);
461         }
462         if (!strncmp(argv[2], "3", 1))
463         {
464                 if (strlen(argv[1])>123) pvlh(argv[1], 2);
465                 else usage(argv[0]);
466         }
467
468         return 0;
469 }
```