Razvoj kripotografskih hash funkcija bez ključa

Šormaz, Danilo

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:866940

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Danilo Šormaz

Razvoj kriptografskih hash funkcija bez ključa

Diplomski rad

Osijek, 2019.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Danilo Šormaz

Razvoj kriptografskih hash funkcija bez ključa

Diplomski rad

Mentor: izv.prof.dr. sc. Ivan Matić

Osijek, 2019.

Sadržaj

Uvod

U ovom diplomskom radu bavit ćemo se razvojem kriptografskih hash funkcija bez ključa. Idealna hash funkcija je opisana s idućih 5 svojstava:

1. Deterministička je, što znači da ista poruka uvijek rezultira s istom hash vrijednosti.
2. Izračun funkcije se brzo izvršava.
3. Računski je neizvedivo pronaći ulaznu poruku iz unaprijed zadane hash vrijednosti.
4. Male promijene unutar ulazne poruke rezultiraju s potpuno drugačijom hash vrijednosti.
5. Računski je neizvedivo pronaći dvije različite poruke s istom hash vrijednosti.

Kriptografske hash funkcije pronalaze široku primjenu u polju sigurnosti. Preciznije, primjenjuju se u kreiranju digitalnog potpisa, autentikaciji, provjeri ispravnosti podataka, odnosno provjeri integriteta podataka.
U prvom dijelu rada definirat ćemo hash funkcije, opisati njihova svojstva, te veze među njima. U nastavku ćemo postaviti problem rođendanskog paradoksa koji će nam pomoći pri definiranju rođendanskog napada. Pomoću rođendanskog napada postavit ćemo donju granicu za duljinu hash vrijednosti.
U drugom dijelu opisat ćemo Merkle-Dåmgard konstrukciju na kojoj se zasniva daljnji razvoj hash funkcija. Definirat ćemo kompresijsku funkciju koja predstavlja vrlo bitan dio konstrukcije. Nadalje, raspisat ćemo Merkle-Dåmgard algoritam te ga primijeniti na ilustrativnom primjeru. Navest ćemo vezu između kompresijske funkcije sa svojstvom jake otpornosti i njene odgovarajuće hash funkcije. U trećem poglavlju navest ćemo i opisati MD4 kriptografsku hash funkciju. Raspisat ćemo MD4 algoritam koji ćemo primijeniti na primjeru hashiranja poruke "a". Ukratko ćemo opisati trenutno stanje MD4 funkcije sa sigurnosnog stajališta. Nadalje, u četvrtom poglavlju je opisana MD5 funkcija koja je ujedno i nasljednica MD4 funkcije. Navest ćemo razlike između te dvije funkcije, opisati algoritam te njegovu primjenu na primjeru hashiranja poruke "a". U petom poglavlju opisat ćemo SHA-1 funkciju, razlog njenog nastanka, te ćemo raspisati SHA-1 algoritam. Algoritam ćemo primijeniti na primjeru hashiranja poruke "abc". Na kraju ćemo spomenuti trenutno sigurnosno stanje SHA-1 funkcije. Na kraju diplomskog rada se nalaze implementacije obrađenih funkcija u programskom jeziku Python 3.7.2.

1 Hash funkcije

Hash funkcija je funkcija koja za ulaz uzima string, poruku ili podatak proizvoljne duljine te generira izlazni string ili podatak fiksne duljine. Vrijednost koju hash funkcija generira nazivamo hash vrijednost, hash code ili sažetak (engl. digest).

Definicija 1.1 (Hash funkcija)**.** *Neka je Σ_{in} ulazni alfabet, a Σ_{out} izlazni alfabet. Svaka funkcija $h : \Sigma_{in}^* \to \Sigma_{out}^n$ koja se može izračunati efikasno se naziva hash funkcija. Hash funkcija generira hash vrijednost duljine n.*

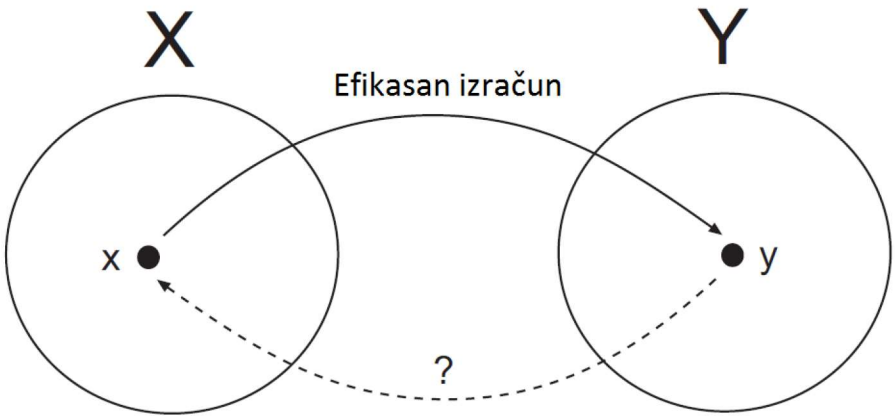
U prethodnoj definiciji domena hash funkcije je jezik Σ_{in}^*, tj. domena je zadana kao skup svih stringova nad alfabetom Σ. Navedeni stringovi, u teoriji, mogu biti beskonačne duljine. Međutim, u praksi se iz tehničkih razloga određuje maksimalna dopuštena duljina stringova. U tom slučaju se hash funkcija zapisuje u obliku:

$$h : \Sigma_{in}^{n_{max}} \to \Sigma_{out}^n$$

Primjetimo kako dva zadana alfabeta Σ_{in} i Σ_{out} mogu biti, i najčešće i jesu, jednaka. U tom slučaju, možemo ih označavati s Σ.
U kriptografiji Σ predstavlja binarni alfabet (npr. $\Sigma = \{0, 1\}$), a n je najčešće duljine 128 ili 160 bitova. U tom slučaju hash funkcija h generira binarni string od 128 ili 160 bitova.

1.1 Kriptografske hash funkcije

Najprije ćemo, za potrebe definiranja kriptografske hash funkcije, uvesti pojam jednosmjerne funkcije (engl. one-way function). Naime, jednosmjerna funkcija predstavlja vrlo bitan pojam u suvremenoj kriptografiji. Govoreći općenito, funkcija $f : X \to Y$ je jednosmjerna ukoliko ju je lako izračunati, a njen je inverz f^{-1} teško izračunati. Ukoliko je f^{-1} lako izračunati, uz poznavanje nekog dodatnog podatka, tada funkciju f nazivamo osobna jednosmjerna funkcija.



Slika 1: Jednosmjerna funkcija

Definicija 1.2 (Jednosmjerna funkcija)**.** *Funkcija $f : X \to Y$ je jednosmjerna funkcija ukoliko se $f(x)$ može izračunati efikasno $\forall x \in X$, ali $f^{-1}(y)$ se ne može izračunati efikasno za slučajno izabran $y \in Y$.*

Preciznije, možemo reći da možda postoji mogućnost izračunavanja inverza funkcije, ali entitet koji vrši računanje ne zna na koji način to učiniti. Obrazložimo ukratko što bi značilo "efikasno izračunavanje". Možemo se složiti da izračunavanje koje se izvršava u nekom linearnom ili kvadratnom vremenu jeste efikasno. Stoga, pod efikasnim izračunavanjem smatramo da za neki problem postoji algoritam koji ga rješava u polinomnom vremenu.

Primjer jednosmjerne funkcije u svakodnevnom životu bi bio telefonski imenik. Funkciju koja određenom imenu dodjeljuje telefonski broj je lako za izračunati, budući su imena sortirana abecednim poretkom. Međutim, teško je izračunati inverz te funkcije, budući da telefonski brojevi nisu sortirani numerički. Nadalje, neke fizičke procese također možemo opisati jednosmjernom funkcijom, Npr. ukoliko razbijemo staklenu bocu u komadiće, općenito je nemoguće sastaviti bocu u prvobitno stanje iz razbijenih komadića.

Kriptografske hash funkcije imaju specifična svojstva koja ih određuju, a koja ćemo navesti u nastavku.

- hash funkcija h je jednosmjerna (engl. preimage resistant) ukoliko je računski neizvedivo pronaći ulaznu riječ $x \in \Sigma_{in}^*$ tako da je $h(x) = y$, za zadanu izlaznu riječ $y \in \Sigma_{out}^n$.
- hash funkcija h je jednoznačna (engl. second-preimage resistant) ili slabo otporna na koliziju (engl. weak collision resistant) ukoliko je računski neizvedivo pronaći ulaznu riječ $x' \in \Sigma_{in}^*$ takvu da je $x' \neq x$ i $h(x') = h(x)$ za zadanu riječ $x \in \Sigma_{in}^*$.
- hash funkcija h je općenito jednoznačna ili jako otporna na koliziju (engl. strong collision resistant) ukoliko je računski neizvedivo pronaći bilo koje dvije ulazne riječi $x, x' \in \Sigma_{in}^*$ tako da je $x' \neq x$ i $h(x') = h(x)$.

Bitno je napomenuti kako se u pojedinoj literaturi svojstvo otpornosti na koliziju poistovjećuje sa svojstvom kolizijske nepostojanosti (engl. collision free). Ovaj termin nije prikladan za uporabu u danom kontekstu, budući se kolizija ne može izbjeći prilikom uporabe hash funkcija. Stoga ćemo u nastavku koristiti samo termin otpornost na koliziju.

Pomoću prethodnih svojstava možemo definirati jednosmjernu (one-way) hash funkciju i hash funkciju otpornu na koliziju.

Definicija 1.3 (Jednosmjerna hash funkcija)**.** *Jednosmjerna hash funkcija je hash funkcija $h : \Sigma_{in}^* \to \Sigma_{out}^n$ koja je jednosmjerna (preimage resistant) i slabo otporna na koliziju (second-preimage resistant).*

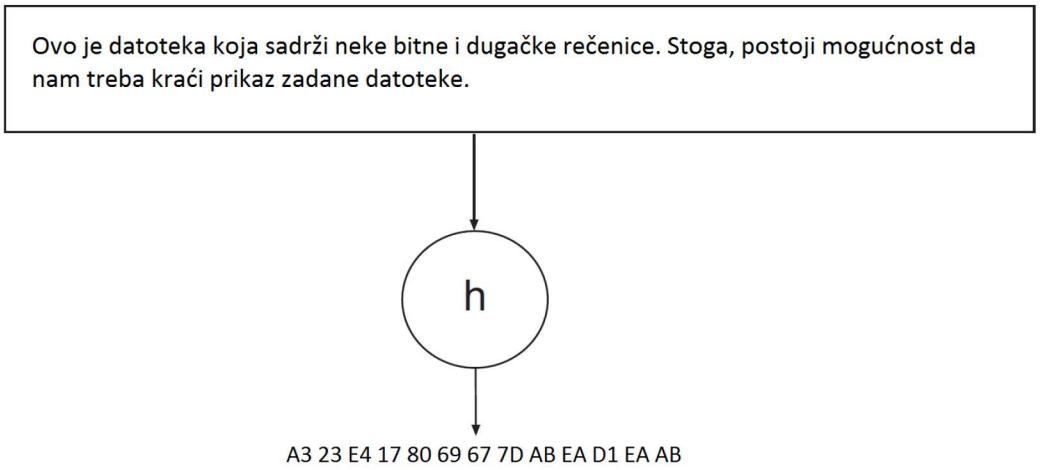
Definicija 1.4 (Hash funkcija otporna na koliziju)**.** *Hash funkcija otporna na koliziju je hash funkcija $h : \Sigma_{in}^* \to \Sigma_{out}^n$ koja je jednosmjerna (preimage resistant) i jako otporna na koliziju (collision resistant).*

Primjetimo da je hash funkcija otporna na koliziju također i jednosmjerna hash funkcija, no obrat ne mora vrijediti. Sada možemo definirati kriptografsku hash funkciju.

Definicija 1.5 (Kriptografska hash funkcija)**.** *Hash funkcija $h : \Sigma_{in}^{n_{max}} \to \Sigma_{out}^n$ je kriptografska hash funkcija ukoliko je jednosmjerna ili otporna na koliziju.*

Kriptografska hash funkcija hashira poruku proizvoljne duljine u binarni string fiksne duljine. Npr. Slika 2 prikazuje ASCII enkodiranu poruku "Ovo je datoteka koja sadrži neke bitne

i dugačke rečenice. Stoga, postoji mogućnost da nam treba kraći prikaz zadane datoteke." koja je hashirana hash funkcijom h u heksadecimalni zapis A3 23 E4 17 80 69 67 7D AB EA D1 EA AB.



Slika 2: Kriptografska hash funkcija

Rezultirajuća vrijednost se naziva otisak (engl. fingerprint) koji je karakterističan za poruku i jednoznačno ju određuje. Svojstvo otpornosti na koliziju omogućuje računsku neizedivost pronalaska neke druge poruke koja se hashira u isti otisak.

Osim prethodno navedenih svojstava, navest ćemo još neka svojstva karakteristična za hash funkcije.

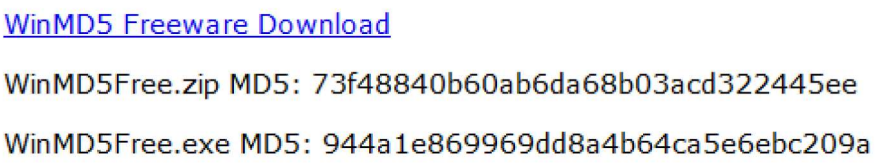
- hash funkcija h je nekorelirana ukoliko između ulaznih i izlaznih bitova ne postoji korelacija.
- hash funkcija h je generalizirano otporna na koliziju ukoliko je računski neizvedivo pronaći dvije ulazne riječi x i x', $x \neq x'$ t.d. su $h(x)$ i $h(x')$ slični u nekom određenom smislu (npr. podudaraju se u pojedinim bitovima).
- hash funkcija h je oslabljeno otporna na koliziju ukoliko je računski neizvedivo pronaći dvije ulazne riječi x i x', $x \neq x'$ i $h(x) = h(x')$ tako da su x i x' slični u nekom određenom smislu (npr. podudaraju se u pojedinim bitovima).

Zbog ranije navedenih svojstava, kriptografske hash funkcije svoju primjenu mogu pronalaziti prilikom pohrane korisničkih lozinki u bazu podataka, te verificiranje istih. Naime, nakon što korisnik kreira korisnički račun, njegova lozinka se hashira, te se odgovarajuća hash vrijednost pohranjuje u bazu podataka. Kada korisnik pokuša da izvrši prijavu, hash vrijednost unesene lozinke se uspoređuje s hash vrijednosti stvarne lozinke u bazi. Ukoliko se hash vrijednosti podudaraju prijava je uspješna. Na taj način je izvršena verifikacija lozinke. Ukoliko neovlaštena osoba pristupi bazi podataka korisnikova lozinka je sigurna, budući da je hashirana i da je pronalazak inverza računski neizvediv.

Također, hash funkciju tj. njenu hash vrijednost možemo koristiti za verificiranje preuzetih datoteka. To bi značilo da pomoću hash vrijednosti provjeravamo da li je preuzeta datoteka

identična originalnoj datoteci, odnosno provjeravamo integritet datoteke. Npr. na web stranici s koje preuzimamo određenu datoteku, nalazi se originalna hash vrijednost te datoteke. Nakon preuzimanja primijenimo odgovarajuću hash funkciju na zadanu datoteku, te ukoliko je rezultirajuća hash vrijednost jednaka originalnoj, to znači da je datoteka nepromijenjena.

Primjer 1.1. *Pokažimo da je hash vrijednost preuzetog software-a WinMD5 [4] jednaka originalnoj hash vrijednosti, koja je dostupna na web stranici software-a.*



Slika 3: Originalna hash vrijednost [4]

Nakon što preuzmemo WinMD5Free.zip datoteke, pomoću naredbe "CertUtil -hashfile putanjaDoDatoteke [HashAlgoritam]" u Windows PowerShell-u možemo provjeriti integritet datoteke, što je prikazano na slici 4. *Koristit ćemo MD5 funkciju koju ćemo kasnije i preciznije objasniti.*

```
PS C:\Users\dsormaz\Downloads> CertUtil -hashfile .\winmd5free.zip MD5
MD5 hash of .\winmd5free.zip:
73f48840b60ab6da68b03acd322445ee
CertUtil: -hashfile command completed successfully.
```

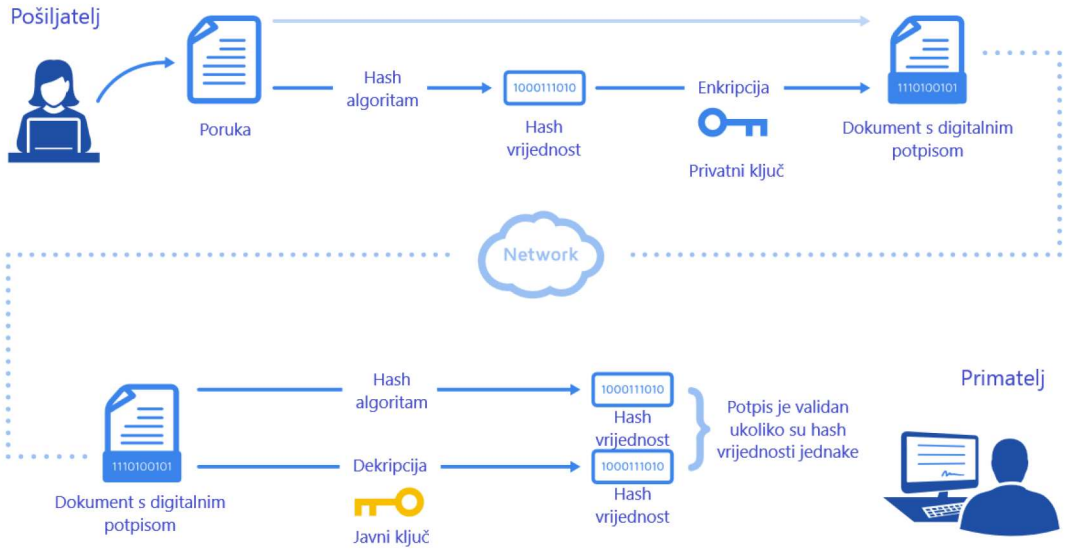
Slika 4: MD5 hash vrijednost preuzete datoteke

Iz slika 3 *i* 4 *vidimo da se hash vrijednosti podudaraju, što znači da je datoteka originalna i da na njoj nisu izvršene modifikacije.*

Osim u prethodnim slučajevima, kriptografska hash funkcija se upotrebljava i prilikom kreiranja digitalnog potpisa. Digitalni potpis omogućuje:

1. Provjeru integriteta - provjeravamo da li je primljena poruka identična poslanoj poruci
2. Autentikaciju - provjeravamo da li je poruka koju smo primili zaista poslana od strane odgovarajućeg pošiljatelja
3. Neporecivost - osoba koja je poslala poruku ne može poreći slanje, budući da samo ona ima pristup svom privatnom ključu kojim je poruka potpisana. Na taj način znamo da digitalni potpis nije falsifikovan od strane treće osobe.

Slika 5 prikazuje primjer uporabe digitalnog potpisa i hash funkcije u njemu.



Slika 5: Proces kreiranja digitalnog potpisa

Pošiljatelj najprije hashira poruku, te nakon toga odgovarajuću hash vrijednost enkriptira privatnim ključem te tako nastaje digitalni potpis. Poruku s digitalnim potpisom šalje primatelju, koji s odgovarajućim javnim ključem dekriptira digitalni potpis te dobija hash vrijednost. Nakon toga vrši hashiranje poruke, te uspoređuje hash vrijednost iz digitalnog potpisa i hash vrijednost primljene poruke. Potpis je validan ukoliko su hash vrijednosti identične.

1.2 Veze među svojstvima

U nastavku ćemo objasniti povezanost određenih svojstava kriptografske hash funkcije.
Hash funkcija sa svojstvom jake otpornosti na koliziju je ujedno i jednoznačna hash funkcija, tj. hash funkcija slabo otporna na koliziju. Pretpostavimo da je funkcija h jako otporna na koliziju, ali ne i slabo. Budući da funkcija h nije slabo otporna na koliziju, za zadani ulaz $x \in \Sigma_{in}^*$ možemo pronaći $x' \in \Sigma_{in}^*, x \neq x'$, tako da je $h(x) = h(x')$ što je u suprotnosti sa svojstvom jake otpornosti na koliziju, tj. dobili smo kontradikciju s polaznom pretpostavkom. Obrat ne vrijedi. Naime, ukoliko hash funkcija zadovoljava svojstvo slabe otpornosti na koliziju, ne mora značiti da će zadovoljavati svojstvo jake otpornosti na koliziju.

Sljedeća je veza između svojstava jake otpornosti na koliziju i jednosmjernosti. Hash funkcija sa svojstvom jake otpornosti na koliziju ne garantira jednosmjernost. Npr. neka je g hash funkcija sa svojstvom jake otpornosti na koliziju s n-bitnim izlazima, a h $(n+1)$-bitna hash funkcija definirana na sljedeći način:

$$h(x) = \begin{cases} 1 \;||\; x, & |x| = n \\ 0 \;||\; g(x), & \text{inače} \end{cases}$$

Operator $||$ predstavlja ulančavanje, tj. konkatenaciju. Ako $h(x)$ počinje s 1 tada je nemoguće da dođe do kolizije, budući će se za svaka dva različita ulaza, izlazi razlikovati u dijelu koji se konkatenira s jedinicom. Ukoliko $h(x)$ počinje s 0 tada pronalaženje kolizije

podrazumijeva pronalaženje kolizije za funkciju g, što je računski neizvedivo, budući je g jako otporna na koliziju. Prema tome, funkcija h je jako otporna na koliziju. Međutim, h nije jednosmjerna. Naime, za sve vrijednosti $h(x)$ koje počinju s 1, lako je pronaći inverz (samo uklonimo vodeću jedinicu). Stoga, h je hash funkcija koja je otporna na koliziju, ali nije jednosmjerna.

Ranije smo rekli da je Σ u kriptografiji najčešće definiran kao skup $\{0,1\}$. Tada je domena hash funkcije $\{0,1\}^*$, a kodomena $\{0,1\}^n$. Postavlja se pitanje koliko velik treba biti parametar n. Donju granicu za n možemo dobiti iz tzv. rođendanskog napada. U nastavku ćemo prezentirati rođendanski paradoks i rođendanski napad.

1.3 Rođendanski paradoks

U teoriji vjerojatnosti, rođendanski paradoks još poznat i kao rođendanski problem, a bavi se problemom određivanja vjerojatnosti da u skupu od n slučajno izabranih ljudi postoje barem dvije osobe koje slave rođendan istog dana. Iz Dirichletovog principa zaključujemo da navedena vjerojatnost postiže vrijednost 1 ukoliko je broj ljudi u grupi 366, a broj dana u godini 365. Zbog jednostavnosti postavljanja problema i izračuna u nastavku ćemo pretpostaviti da je broj dana u godini 365. Naime, pažnju će nam privući činjenica da je dovoljna grupa od 70 ljudi kako bi se postigla vjerojatnost 0.99. Za vjerojatnost vrijednosti 0.5 dovoljna je grupa od 23 čovjeka. To znači da je vjerojatnost 50% da u grupi od 23 čovjeka barem dvije osobe slave rođendan istog dana. Pretpostavimo da je dan rođenja osobe jednako vjerojatan za svaki dan u godini koja se sastoji od 365 dana. Cilj nam je pronaći $P(A)$, tj. vjerojatnost da barem dvije osobe iz grupe slave rođendan istog dana. Zbog jednostavnosti računat ćemo vjerojatnost suprotnog događaja $P(A^c)$, odnosno vjerojatnost da nitko iz grupe ljudi ne slavi rođendan istog dana. Budući da su A i A^c disjunktni događaji, vrijedi $P(A) = 1 - P(A^c)$. U nastavku ćemo objasniti paradoks na primjeru grupe od 23 osobe. Primijetimo da je događaj da sve osobe iz grupe slave rođendan različitog dana jednak događaju da osoba 2 ne slavi rođendan kad i osoba 1, i da osoba 3 ne slavi rođendan istog dana kao osoba 1 ili osoba 2, i tako sve do osobe 23 koja ne slavi rođendan istog dana kao bilo koja osoba od 1 do 22. Prema tome, možemo definirati 23 događaja. Definirajmo ih redom i to tako da svaki događaj odgovara jednoj osobi koja ne dijeli rođendan niti s jednom osobom koja je prethodno analizirana. Za prvi događaj nemamo niti jednu osobu koja je ranije analizirana, stoga sa 100% sigurnosti tvrdimo da ta osoba niti s jednom osobom, koja je ranije analizirana, ne dijeli rođendan. Vjerojatnost tog događaja iznosi $P(1) = \frac{365}{365} = 1$. U drugom događaju postoji samo jedna osoba koja je ranije analizirana, te će vjerojatnost da druga osoba s njom ne dijeli rođendan iznositi $P(2) = \frac{364}{365}$, budući da smo iz skupa svih dana u godini isključili jedan dan, koji predstavlja rođendan prve osobe. Nadalje, vjerojatnost da treća osoba ne dijeli rođendan s prve dvije je jednaka $P(3) = \frac{363}{365}$. Dalje nastavljamo sve do osobe 23, kojoj dodjeljujemo vjerojatnost da ta osoba nema rođendan istog dana kao prethodne osobe, tj. $P(23) = \frac{365-22}{365} = \frac{343}{365}$. Konačno, zbog principa uvjetne vjerojatnosti vrijedi da je $P(A^c)$ jednak umnošku svih prethodnih nezavisnih događaja.

$$
\begin{aligned}
P(A^c) &= \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{343}{365} \\
&= \frac{1}{365^{23}} \cdot \left(365 \cdot 364 \cdots 343\right) \\
&= \frac{1}{365^{23}} \cdot \frac{365!}{(365-23)!} \\
P(A^c) &\approx 0.492703
\end{aligned}
$$

Stoga, vrijedi $P(A) = 1 - P(A^c) = 1 - 0.492703 = 0.507297$ (50.7297%). Zaključujemo da je vjerojatnost da barem dvije osobe iz grupe od 23 osobe slave rođendan istog dana malo veća od 50%. U nastavku ćemo problem generalizirati na N osoba. $P(N)$ predstavlja vjerojatnost da barem dvije osobe iz grupe od N ljudi slave rođendan istog dana. $P'(N)$ će predstavljati vjerojatnost da sve osobe slave rođendan različitog dana. Prema Dirichletovu principu $P'(N)$ je vrijednosti 0 za $N > 365$. Za $N \leq 365$ vrijedi:

$$
P'(N) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{N-1}{365}\right)
$$

$$
= \frac{365 \cdot 364 \cdots (365 - N + 1)}{365^N}
$$

$$
= \frac{1}{365^N} \cdot \frac{365!}{(365-N)!}
$$

Događaj da barem dvije osobe iz grupe ljudi od N osoba dijeli rođendan je suprotan događaju da sve osobe slave rođendan istog dana. Stoga vrijedi: $P(N) = 1 - P'(N)$.
Razvoj eksponencijalne funkcije u Taylorov red

$$
e^x = 1 + x + \frac{x^2}{2} + \ldots
$$

omogućuje aproksimaciju prvog reda za e^x, $x << 1$

$$
e^x \approx 1 + x.
$$

Kako bi mogli primijeniti ovu aproksimaciju na naš problem za $P'(N)$, zadajmo

$$
x = \frac{-a}{365}.
$$

Tada vrijedi,

$$
e^{\frac{-a}{365}} \approx 1 - \frac{a}{365}.
$$

Nadalje, a možemo zamijeniti s nenegativnim cijelim brojevima koji odgovaraju raspisu za $P'(N)$. Za $a = 1$ vrijedi

$$
e^{\frac{-1}{365}} \approx 1 - \frac{1}{365},
$$

za $a = 2$

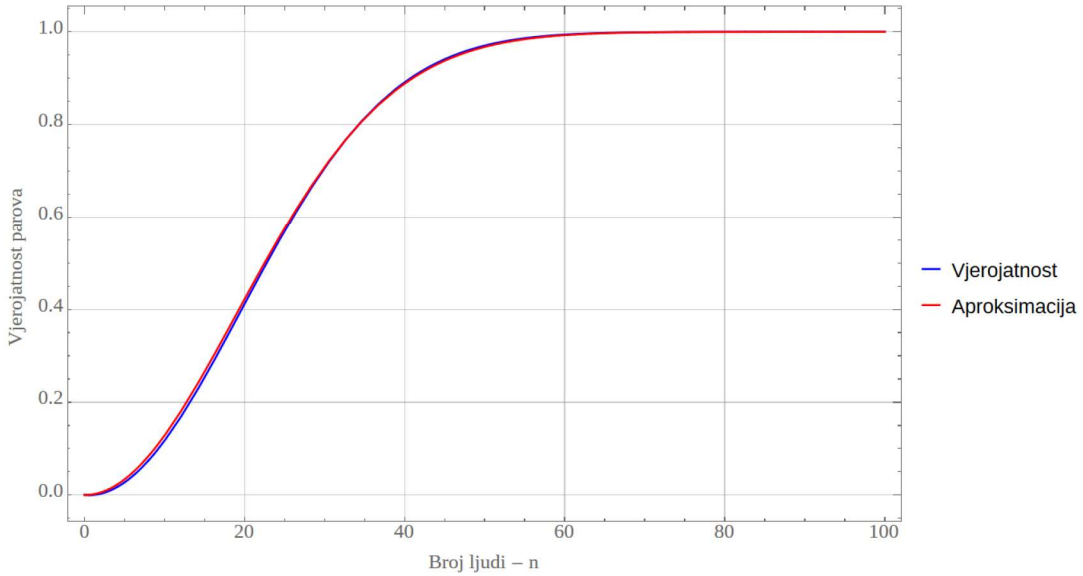
$$
e^{\frac{-2}{365}} \approx 1 - \frac{2}{365}
$$

itd.

Sada možemo aproksimirati vrijednost $P'(N)$.

$$\begin{aligned} P'(N) &\approx 1 \cdot e^{\frac{-1}{365}} \cdot e^{\frac{-2}{365}} \cdot \ldots \cdot e^{-\frac{N-1}{365}} \\ &= e^{-\frac{1+2+\ldots+N-1}{365}} \\ &= e^{-\frac{\frac{N(N-1)}{2}}{365}} \\ &= e^{-\frac{N(N-1)}{730}} \end{aligned}$$

Stoga, vrijedi $P(N) = 1 - P'(N) \approx 1 - e^{\frac{N(N-1)}{730}}$. Grublja aproksimacija bi bila $P(N) \approx 1 - e^{\frac{-N^2}{730}}$. Iz slike 6 vidimo da je to i dalje dovoljno dobra aproksimacija. Sada možemo poopćiti problem, tj. da iz skupa od N elemenata biramo k elemenata, pri čemu svaki put odabrani element vraćamo nazad u skup. Pitanje koje se postavlja je kolika je vjerojatnost da odaberemo bar jedan element dva ili više puta?



Slika 6: Vjerojatnost $1 - P'(N)$ i aproksimacija $1 - e^{\frac{-N^2}{730}}$ se dovoljno dobro podudaraju

Računamo vjerojatnost da se u k biranja ne ponovi niti jedan element. Analogno prethodnom primjeru, dobijemo vjerojatnost

$$\begin{aligned} P(\textit{nema ponavljanja u k biranja}) &= 1 \cdot \left(1 - \frac{1}{N}\right) \cdot \left(1 - \frac{2}{N}\right) \cdot \ldots \cdot \left(1 - \frac{(k-1)}{N}\right) \\ &= e^{-\frac{k(k-1)}{2N}} \\ P(\textit{ponavljanje u k biranja}) &\approx 1 - e^{-\frac{k(k-1)}{2N}} \\ P(\textit{ponavljanje u k biranja}) &\approx 1 - e^{-\frac{k^2}{2N}} \end{aligned}$$

Koliko elemenata k treba odabrati da bi vrijedilo $P(\textit{ponavljanje u k biranja}) = p$, za unaprijed određeni $p \in [0, 1)$:

$$
\begin{aligned}
p &\approx 1 - e^{-\frac{k^2}{2N}} \\
e^{-\frac{k^2}{2N}} &\approx 1 - p \\
e^{\frac{k^2}{2N}} &\approx \frac{1}{1-p} \\
\frac{k^2}{2N} &\approx \ln \frac{1}{1-p} \\
k &\approx \sqrt{2 \ln \frac{1}{1-p}} \cdot \sqrt{N}
\end{aligned}
$$

Rođendanski napad pronalazi koliziju hash funkcije u $O(2^{\frac{n}{2}})$ vremenu. Pokažimo da to vrijedi. Budući da su izlazi hash funkcije duljine n i sastoje se od binarnih elemenata, tada vrijedi da je kardinalnost skupa svih binarnih elemenata duljine n jednaka $N = 2^n$. Neka je potrebno odabrati k elemenata da bi se dogodila kolizija. Vrijedi,

$$
\begin{aligned}
k &\approx \sqrt{2 \ln \frac{1}{1-p}} \cdot \sqrt{N} \\
N &= 2^n \\
O(k) &= O\left(\sqrt{2 \ln \frac{1}{1-p}} \cdot \sqrt{N}\right)
\end{aligned}
$$

Za $p \in [0, 1)$ prvi dio izraza je konstanta i označit ćemo ga s c. Tada vrijedi da je

$$
O(c\sqrt{N}) = O(\sqrt{N}) = O(\sqrt{2^n}) = O(2^{\frac{n}{2}}).
$$

Prema rođendanskom napadu, ukoliko je pretraživanje prostora od 2^{64} elemenata računski neizvedivo, tada je funkcija, čiji se izlazi sastoje od $2 \cdot 64 = 128$ bitova, otporna na rođendanski napad. Da bi se osigurali od rođendanskog napada, izlaz hash funkcije mora biti dovoljno velik.

2 Merkle - Dåmgard konstrukcija

Većina današnjih kriptografskih hash funkcija je bazirana na konstrukciji koju su postavili Ralph C. Merkle i Ivan B. Dåmgard u kasnim 80-ima. U nastavku ćemo definirati kompresijsku funkciju, budući da se direktno koristi u konstrukciji hash funkcije.

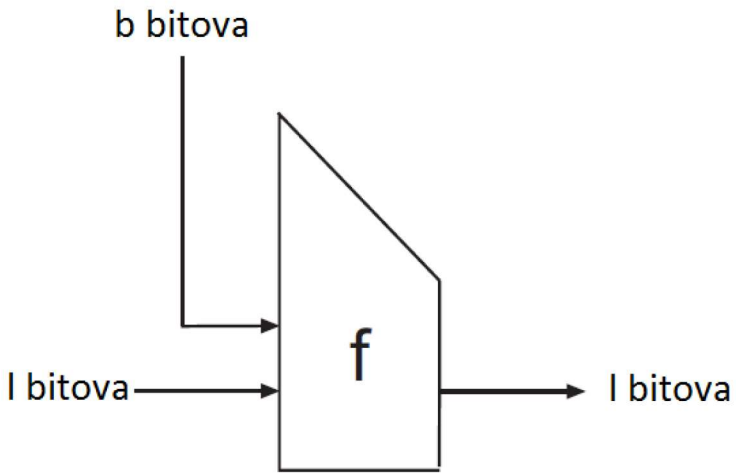
Definicija 2.1 (Kompresijska funkcija)**.** *Neka je Σ_{in} ulazni alfabet,a Σ_{out} izlazni alfabet. Funkcija $f : \Sigma_{in}^m \to \Sigma_{out}^n, m, n \in \mathbb{N}, m > n$, koja se može izračunati efikasno, naziva se kompresijska funkcija.*

Primijetimo da je definicija kompresijske funkcije vrlo slična definiciji hash funkcije. Razlika je u tome što kompresijska funkcija ima ulaze fiksne duljine. Prema Merkle - Dåmgard konstrukciji, iterativna hash funkcija h se računa ponavljajućom primjenom kompresijske funkcije $f : \Sigma^m \to \Sigma^n,\ m, n \in \mathbb{N},\ m > n$ sa svojstvom otpornosti na koliziju. Sa x ćemo označavati ulaznu poruku hash funkcije. Kompresijska funkcija se primjenjuje uzastopno na elemente $x_1, x_2, \ldots, x_n$ koji predstavljaju blokove poruke x.

Kao što je prikazano na slici 7, kompresijska funkcija f za ulaz uzima dva argumenta:

1. b-bitovnu blok poruku
2. l-bitovnu vrijednost ulančavanja (označavat ćemo ju s $H_i, i = 0, 1, \ldots, n$)

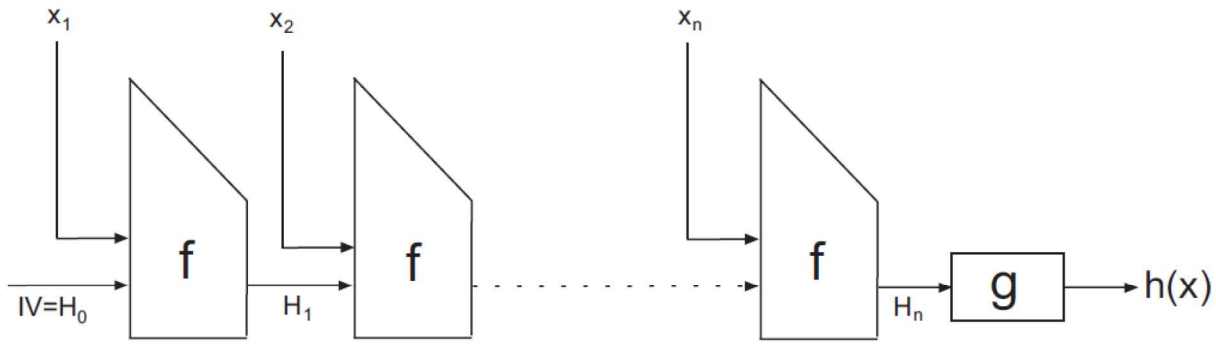
Izlaz kompresijske funkcije koristi se kao nova $l-$bitovna vrijednost ulančavanja, što ujedno predstavlja jedan od ulaza za iduću iteraciju kompresijske funkcije. Prema prethodno uvedenoj notaciji, vrijedi $m = b + l,\ n = l$.



Slika 7: Kompresijska funkcija f

Sada se iterativna hash funkcija može konstruirati kao što je prikazano na slici 8, gdje f predstavlja kompresijsku funkciju, a g predstavlja izlaznu funkciju koja može biti identiteta. Funkcija g se naziva finalizacijska funkcija i ona može imati ulogu sažimanja posljednjeg rezultata u izlaz manje duljine. Također, dodaje efekat lavine, tj. svojstvo da male promjene u ulazu funkcije dovode do velikih promjena u izlazu. Finalizacijska funkcija je često bazirana na kompresijskoj funkciji.

Budući da kompresijska funkcija za ulaz prima poruku fiksne duljine, tj. $b-$bitovne blokove poruke, a naša hash funkcija prima ulaze proizvoljne duljine, morat ćemo modificirati originalnu poruku.
Poruka x bi se modificirala na način da se nadopunjuje do duljine višekratnika broja b te se nakon toga dijeli na niz od n $b-$bitovnih blokova $x_1, x_2, \ldots, x_n$. Kompresijska funkcija f se tada iterativno primjenjuje, pri čemu su ulazi pri prvoj iteraciji početna, odnosno, inicijalna vrijednost $IV = H_0$ i prvi blok poruke, x_1. Nakon svake $i-$te iteracije, nove ulazne vrijednosti će biti H_i i idući blok poruke x_{i+1}, $i = 1, 2, \ldots, n-1$. Inicijalna vrijednost se zadaje ovisno o algoritmu ili implementaciji, a najčešće ima vrijednost 0^l. Nakon što se kompresijska funkcija primjeni i na posljednji blok poruke x_n, konačna vrijednost ulančavanja H_n predstavlja ulazni parametar za funkciju g. Izlaz ove funkcije je izlaz iterativne hash funkcije h za ulaznu poruku x.



Slika 8: Iterativna hash funkcija h

Stoga, iterativna hash funkcija h se za određenu poruku $x = x_1, x_2, \ldots, x_n$ može rekurzivno izračunati prema sljedećem skupu jednadžbi

$$H_0 = IV$$
$$H_i = f(H_{i-1}, x_i), i = 1, \ldots, n$$
$$h(x) = g(H_n).$$

Kao što smo ranije spomenuli, ulazna poruka x se mora proširiti do duljine višekratnika broja b, koji predstavlja duljinu jednog bloka poruke x. Jedna od mogućnosti proširenja je nadopuniti x s nulama, pri čemu se nule dodaju na kraj poruke x. Međutim, takvo proširenje može dovesti do dvosmislenosti poruke x. Npr. uslijed proširivanja poruke 110110 s nulama do 8−bitovne duljine dobijemo poruku 11011000. Međutim, sada je nepoznato koliko je posljednjih nula poruka imala prije proširenja, tj. nepoznato je stanje originalne poruke. Merkle je predložio metodu koja bi riješila ovaj problem. Metoda se sastoji od dodavanja duljine poruke x u bitovnom zapisu na kraju poruke. Proširenje poruke se sada izvršava tako da se na kraj poruke dodaje jedinica, varijabilan broj nula te duljina originalne poruke x u obliku binarnog zapisa. Treba napomenuti kako duljinu originalne poruke u bitovnom zapisu možemo dodati ili u bloku gdje se dodaju nule ili dodati novi blok x_{n+1} koji će sadržavati duljinu poruke. Još jedna metoda koja rješava problem proširenja poruke, zasniva se na dodavanju broja nadodanih nula, u obliku binarnog zapisa, na kraj poruke. Naime, nakon dodavanja nula u posljednjem bloku poruke, dodaje se novi blok poruke koji će sadržavati binarni zapis broja nadodanih nula.

Primjer 2.1. *Neka je* $x = 1011010110$ *ulazna poruka, te neka je zadana duljina bloka* $b = 4$. *Primjenom prethodnih metoda izvršimo nadopunjavanje originalne poruke.*

Najprije podijelimo ulaznu poruku na blokove od b bitova, pri čemu dobijemo 1011 0101 10.

1. *Metoda nadopunjavanja nulama: 1011 0101 1000*
2. *Metoda nadopunjavanja duljinom poruke: 1011 0101 1010 1010*
3. *Metoda nadopunjavanja brojem dodanih nula: 1011 0101 1000 0010*

Merkle i Dåmgard su pokazali da je u njihovoj konstrukciji pronalaženje kolizije za funkciju h teško barem kao pronalaženje kolizije za odgovarajuću kompresijsku funkciju. To također znači da ako je f kompresijska funkcija otporna na koliziju i ukoliko je h iterativna hash funkcija koja prilikom konstrukcije koristi funkciju f, tada je h kriptografska hash funkcija otporna na koliziju. Iterativna hash funkcija nasljeđuje svojstvo otpornosti na koliziju od svoje odgovarajuće kompresijske funkcije.
U nastavku ćemo koristiti kompresijsku funkciju $f : \{0,1\}^{b+l+1} \to \{0,1\}^l$, zbog konstrukcije algoritma.

Algoritam 1 Merkle - Dåmgard algoritam

procedure MERKLE - DÅMGARD(x)
  $f \leftarrow$ *kompresijska funkcija*
  $b \leftarrow$ *duljina bloka poruke*
  $N \leftarrow |x|$
  $n \leftarrow \lceil \frac{N}{b} \rceil$
  $d \leftarrow n \cdot b - N$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    $y_i \leftarrow x_i$ ▷ x_i predstavlja i-ti blok poruke x
  end for
  $y_n \leftarrow x_n || 0^d$
  $y_{n+1} \leftarrow$ *broj dodanih nula u* $n -$ *tom bloku*
  $H_0 = IV \leftarrow 0^{l+1}$
  $z_1 \leftarrow H_0 || y_1$
  $H_1 \leftarrow f(z_1)$
  for $i \leftarrow 1$ **to** n **do**
    $z_{i+1} \leftarrow H_i || 1 || y_{i+1}$
    $H_{i+1} \leftarrow f(z_{i+1})$
  end for
  $h(x) \leftarrow g(H_{n+1})$
  return $h(x)$
end procedure

Teorem 2.1. *Neka je* $f : \Sigma^m \to \Sigma^l$, $m = b + l + 1$, $b \geq 1$ *kompresijska funkcija sa svojstvom jake otpornosti. Tada je funkcija h, konstruirana Merkle - Dåmgard algoritmom, kriptografska hash funkcija sa svojstvom otpornosti na koliziju.*

Dokaz. Pokazat ćemo da ukoliko hash funkcija nije otporna na koliziju, da onda ni odgovarajuća kompresijska funkcija nije otporna na koliziju. Pretpostavimo da možemo pronaći $x \neq x'$ takve da je $h(x) = h(x')$. U nastavku ćemo pokazati da možemo pronaći koliziju kompresijske funkcije u polinomijalnom vremenu. Neka su n_1 i n_2 duljine poruka x i x', redom. Označimo

$$y(x) = y_1||y_2||\dots||y_{n_1+1}$$

i

$$y(x') = y'_1||y'_2||\dots||y'_{n_2+1}$$

gdje su x i x' prošireni s d i d' nula, redom. Označimo s $H_1, H_2, \dots, H_{n_1+1}$ i $H'_1, H'_2, \dots, H'_{n_2+1}$ ulančane vrijednosti koje se izračunavaju tijekom izvršavanja algoritma.
Razlikujemo dva slučaja, u ovisnosti o tome da li vrijedi $|x| \equiv |x'| \pmod b$.

Slučaj 1: $|x| \not\equiv |x'| \pmod b$

Pokažimo da je $d \neq d'$.

$$d = \left\lceil \frac{|x|}{b} \right\rceil \cdot b - |x|$$
$$d' = \left\lceil \frac{|x'|}{b} \right\rceil \cdot b - |x'|$$

$$|x| = \left\lceil \frac{|x|}{b} \right\rceil \cdot b - d$$
$$|x'| = \left\lceil \frac{|x'|}{b} \right\rceil \cdot b - d'$$

Iz pretpostavke vrijedi:

$$\left\lceil \frac{|x|}{b} \right\rceil \cdot b - d \not\equiv \left\lceil \frac{|x'|}{b} \right\rceil \cdot b - d' \pmod b$$
$$d \not\equiv d' \pmod b,$$

iz čega slijedi da je $d \neq d'$.
Također, vrijedi da je $y_{n_1+1} \neq y'_{n_2+1}$, budući da posljednji blok sadrži binarnu reprezentaciju broja d odnosno d', koji se međusobno razlikuju.
Sada imamo,

$$\begin{aligned} f(H_{n_1}||1||y_{n_1+1}) &= H_{n_1+1} \\ &= h(x) \\ &= h(x') \\ &= H'_{n_2+1} \\ &= f(H'_{n_2}||1||y'_{n_2+1}) \end{aligned}$$

Pronašli smo koliziju kompresijske funkcije, budući da za $y_{n_1+1} \neq y'_{n_2+1}$ vrijedi $f(H_{n_1}||1||y_{n_1+1}) = f(H'_{n_2}||1||y'_{n_2+1})$

Slučaj 2: $|x| \equiv |x'| \pmod b$

Ovaj slučaj možemo razdvojiti na dva podslučaja.

Slučaj 2a: $|x| = |x'|$

Budući da je $|x| = |x'|$, tada vrijedi da je $d = d'$, iz čega dalje slijedi jednakost $y_{n_1+1} = y'_{n_2+1}$. Duljine poruka su jednake,te zbog toga vrijedi $n_1 = n_2 = n$. Tada je,

$$\begin{aligned} f(H_n||1||y_{n+1}) &= H_{n+1} \\ &= h(x) \\ &= h(x') \\ &= H'_{n+1} \\ &= f(H'_n||1||y'_{n+1}) \end{aligned}$$

Ukoliko je $H_n \neq H'_n$, tada smo pronašli koliziju za kompresijsku funkciju jer smo za dva različita ulaza pronašli isti izlaz. Zato pretpostavimo da je $H_n = H'_n$. Tada imamo

$$f(H_{n-1}||1||y_n) = H_n = H'_n = f(H'_{n-1}||1||y'_n)$$

Sada, ili smo pronašli koliziju za kompresijsku funkciju, ili vrijedi $H_{n-1} = H'_{n-1}$ i $y_n = y'_n$. Pretpostavimo da nismo pronašli koliziju, te se nastavljamo vraćati unatrag dok konačno ne dobijemo

$$f(0^{l+1}||y_1) = H_1 = H'_1 = f(0^{l+1}||y'_1)$$

Ukoliko je $y_1 \neq y'_1$, tada smo pronašli koliziju za kompresijsku funkciju. Nadalje, pretpostavimo da je $y_1 = y'_1$. Ali tada vrijedi $y_i = y'_i, 1 \leq i \leq n+1$, budući da smo pretpostavili da nismo pronašli koliziju te smo došli do prve iteracije. Tada je $y(x) = y(x')$. Međutim to bi značilo da je $x = x'$, budući je funkcija y po svojoj definiciji injekcija. Polazna pretpostavka je bila da je $x \neq x'$, te smo stoga dobili kontradikciju.

Slučaj 2b: $|x| \neq |x'|$

Možemo pretpostaviti da $|x| < |x'|$, pa je tada $n_2 > n_1$. Ovaj slučaj je sličan slučaju 2a. Pretpostavimo da ne pronalazimo koliziju za kompresijsku funkciju. Vraćajući se unatrag, kao u slučaju 2a, možemo eventualno doći do situacije gdje je

$$f(0^{l+1}||y_1) = H_1 = H_{n_2-n_1+1} = f(H'_{n_2-n_1}||1||y_{n_2-n_1+1})'$$

Međutim, $l+1$-vi bit ulaza $0^{l+1}||y_1$ je 0, a $l+1$-vi bit drugog ulaza $H'_{n_2-n_1}||1||y_{n_2-n_1+1}$ je 1. Stoga, za dva različita ulaza smo pronašli identičan izlaz, tj. pronašli smo koliziju kompresijske funkcije.

Pokrili smo sve slučajeve i time pokazali željenu tvrdnju.

□

Primjer 2.2. *Primjenom Merkle-Dåmgard algoritma 1 izračunajmo hash vrijednost za ulaznu poruku "mathos", za zadane vrijednosti $b = 7$ i $l = 8$.*
Zapišimo string "mathos" u binarnom obliku, pomoću ASCII tablice[7].

Char	Hex	Bin
m	6d	01101101
a	61	01100001
t	74	01110100
h	68	01101000
o	6f	01101111
s	73	01110011

Prema tome ulazna poruka x u bitovnom zapisu glasi:

$$
\begin{aligned}
x &= 011011010110000101110100011010000110111101110011 \\
x_1 &= 0110110 \\
x_2 &= 1011000 \\
&\vdots \\
x_7 &= 110011
\end{aligned}
$$

Zadajmo funkciju f

$$
\begin{aligned}
f(z) &= (z \wedge (z \hookleftarrow 3)) \vee z \pmod{2^8} \\
N &= |x| = 48 \\
n &= \left\lceil \frac{N}{b} \right\rceil = \left\lceil \frac{48}{7} \right\rceil = 7 \\
d &= n \cdot b - N = 7 \cdot 7 - 48 = 1 \\
y_1 &= x_1 = 0110110 \\
&\vdots \\
y_6 &= x_6 = 0111101 \\
y_7 &= x_7 || 0^d = x_7 || 0 \\
y_{n+1} &= y_8 = 0000001 \\
H_0 &= IV = 0^{l+1} = 000000000
\end{aligned}
$$

Izračunajmo konstante N, n, d:

$$
\begin{aligned}
N &= |x| = 48 \\
n &= \left\lceil \frac{N}{b} \right\rceil = \left\lceil \frac{48}{7} \right\rceil = 7 \\
d &= n \cdot b - N = 7 \cdot 7 - 48 = 1
\end{aligned}
$$

Postavimo vrijednost za $y_1, \ldots, y_{n+1}$*:*

$$\begin{aligned}
y_1 &= x_1 = 0110110 \\
&\vdots \\
y_6 &= x_6 = 0111101 \\
y_7 &= x_7||0^d = x_7||0 \\
y_{n+1} &= y_8 = 0000001
\end{aligned}$$

Sada, zadajmo inicijalnu vrijednost H_0*:*

$$H_0 = IV = 0^{l+1} = 0^9 = 000000000$$

Nadalje, pratimo algoritam 1

$$\begin{aligned}
z_1 &= H_0||y_1 = 0000000000110110 \\
H_1 &= f(z_1) = 00110110 \\
z_2 &= H_1||1||y_2 = 0011011011011000 \\
H_2 &= f(z_2) = 01111100 \\
z_3 &= H_2||1||y_3 = 0111110010101110 \\
H_3 &= f(z_3) = 10101110 \\
z_4 &= H_3||1||y_4 = 1010111011000110 \\
H_4 &= f(z_4) = 11000110 \\
z_5 &= H_4||1||y_5 = 1100011011000011 \\
H_5 &= f(z_5) = 11000011 \\
z_6 &= H_5||1||y_6 = 1100001110111101 \\
H_6 &= f(z_6) = 10111101 \\
z_7 &= H_6||1||y_7 = 1011110111100110 \\
H_7 &= f(z_7) = 11100110 \\
z_7 &= H_6||1||y_7 = 1011110111100110 \\
H_7 &= f(z_7) = 11100110 \\
z_8 &= H_7||1||y_8 = 1110011010000001 \\
H_8 &= f(z_8) = 10000001
\end{aligned}$$

Neka je funkcija g *identiteta. Tada vrijedi:*

$$h(x) = g(H_{n+1}) = g(H_8) = 10000001$$

Tj. u heksadecimalnom zapisu

$$h(x) = 81$$

3 MD4

MD4 je kriptografska hash funkcija dizajnirana 1990. godine od strane Ronalda Rivesta. Naziv MD dolazi od "Message Digest", a broj 4 u nazivu označava broj u seriji "Message Digest" hash funkcija. Funkcija reprezentira Merkle-Dåmgard konstrukciju koja hashira poruku podijeljenu u 512 - bitovne blokove ($b = 512$) u hash vrijednost duljine 128 bitova ($l = 128$). MD4 je dizajnirana tako da se može efikasno implementirati na 32-bitnim procesorima, te koristi "little-endian" arhitekturu. Korištenje ove vrste arhitekture podrazumijeva da npr. riječi od 4 byte-a $a_1a_2a_3a_4$ reprezentira sljedeći izraz:

$$a_4 2^{24} + a_3 2^{16} + a_2 2^8 + a_1.$$

Kod "big-endian" arhitekture istu riječ od 4 byte-a $a_1a_2a_3a_4$ reprezentira izraz

$$a_1 2^{24} + a_2 2^{16} + a_3 2^8 + a_4.$$

Algoritam 2 MD4

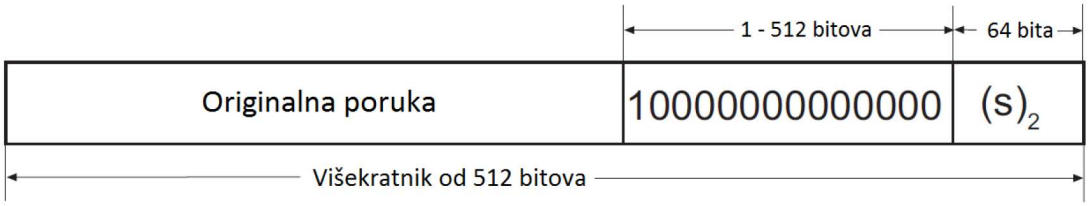
$m = m_0 m_1 \dots m_{s-1}$
Konstruirati $M[0]M[1]\dots M[N-1]$
$A \leftarrow 0x67452301$
$B \leftarrow 0xefcdab89$
$C \leftarrow 0x98badcfe$
$D \leftarrow 0x10325476$
$c_1 \leftarrow 0x5a827999$
$c_2 \leftarrow 0x6ed9eba1$
for $i \leftarrow 0$ **to** $N/16 - 1$ **do**
    for $j \leftarrow 0$ **to** 15 **do**
        $X[j] = M[i \cdot 16 + j]$
    end for
    $A' \leftarrow A$
    $B' \leftarrow B$
    $C' \leftarrow C$
    $D' \leftarrow D$
    Runda 1
    Runda 2
    Runda 3
    $A \leftarrow A + A'$
    $B \leftarrow B + B'$
    $C \leftarrow C + C'$
    $D \leftarrow D + D'$
end for
return $h(m) = A||B||C||D$

Neka je $m = m_0 m_1 \dots m_{s-1}$ s-bitovna poruka, proizvoljne duljine. Iduća 4 koraka su potrebna kako bi se hashirala poruka m uz pomoć MD4 funkcije.

Korak 1 - Dodavanja bitova
Poruka m se proširuje tako da njena bitovna duljina bude kongruentna 448 modulo 512. Proširenoj poruci nedostaje još 64 bita kako bi njezina duljina bila višekratnik broja 512.

Nadopunjavanje poruke se uvijek izvršava, i u slučaju da je duljina početne poruke kongruentna 448 modulo 512. U takvom scenariju je potrebno dodati 512 bitova. Nadopunjavanje potrebnim bitovima se izvršava na idući način: na kraj poruke m se dodaje jedan bit 1, te nakon toga dovoljno 0 bitova kako bi se zadovoljilo prethodno svojstvo kongruencije. Operacija nadopunjavanja je invertibilna, pa prema tome različiti ulazi imaju različite izlaze. To ne bi vrijedilo da smo primjenjivali metodu proširivanja s nulama, bez dodatne jedinice.

Korak 2 - Dodavanja duljine
Na rezultat nakon koraka 1 dodajemo 64-bitnu reprezentaciju s-a, koji predstavlja duljinu originalne poruke. Navedena 64-bita se dodaju kao dvije 32-bitovne riječi pri čemu se prvo dodaje low-order riječ. U slučaju da je $s > 2^{64}$, tada se koristi samo "low-order" 64 bita od s. Nakon ovog proširenja rezultirajuća poruka je duljine višekratnika broja 512. Proširenu poruku sada dijelimo na 32 - bitovne riječi, pri čemu je broj riječi višekratnik broja 16. S $M[0], M[1], \ldots, M[N-1]$ označimo 32 - bitovne riječi, gdje je N broj riječi.



Slika 9: Struktura poruke nakon predprocesiranja tj. prva dva koraka

Korak 3 - Inicijalizacija MD buffera
Buffer (spremnik) od 4 riječi (A, B, C, D) se koristi u svrhu izračunavanja sažetka poruke. Svaki navedeni buffer je 32 - bitovni. Prema Merkle - Dåmgard konstrukciji, buffer (A, B, C, D) odgovara ulančanoj vrijednosti. Bufferi se inicijaliziraju u heksadecimalnom obliku, pravilom low-order byte-ova:

$$A \leftarrow 0x67452301$$
$$B \leftarrow 0xefcdab89$$
$$C \leftarrow 0x98badcfe$$
$$D \leftarrow 0x10325476$$

Korak 4 - Procesuiranje poruke u blokovima od 16 riječi
U nastavku su navedene operacije koje se koriste u MD4 konstrukciji, gdje X i Y označavaju ulazne riječi.

$X \wedge Y$	Bitovno AND
$X \vee Y$	Bitovno OR
$X \oplus Y$	Bitovno XOR
$\neg X$	Bitovno NOT
$X + Y$	Zbrajanje modulo 2^{32}
$X \hookleftarrow s$	Rotacija u lijevo od X za s pozicija ($0 \leq s \leq 31$)

U ovom koraku definiramo tri pomoćne funkcije f, g, h. Svaka od funkcija za ulaz uzima tri 32-bitovne riječi, dok izlaz predstavlja jednu 32-bitovnu riječ. Funkcije su definirane na idući način:

$$f(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$
$$g(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$
$$h(X, Y, Z) = X \oplus Y \oplus Z$$

Primijetimo da su sve ove operacije vrlo brze i da je jedina aritmetička operacija operacija zbrajanja modulo 2^{32}. Sve operacije koje se izvršavaju na 32-bitovnim riječima su modulo 2^{32}.
Za svaku bitovnu poziciju funkcija f se ponaša kao uvijet: if X then Y else Z. Funkcija g se za svaku bitovnu poziciju ponaša tako da ukoliko su od X, Y, Z bar dvije vrijednosti 1, tada g ima 1 na toj poziciji. Funkcija h je bitovna XOR funkcija, tj. h zbraja sve ulazne riječi modulo 2. MD4 koristi dvije magične konstante u tzv. rundama 2 i 3. Konstanta runde broj 2 je $\lfloor 2^{30}\sqrt{2} \rfloor$, a runde 3 $\lfloor 2^{30}\sqrt{3} \rfloor$. Njihove vrijednosti u heksadecimalnom zapisu su:

$$c_1 = \lfloor 2^{30}\sqrt{2} \rfloor \leftarrow 0x5a827999$$
$$c_2 = \lfloor 2^{30}\sqrt{3} \rfloor \leftarrow 0x6ed9eba1$$

Nakon definiranja funkcija i konstanti, sada se proširena poruka M iterativno procesuira. U svakoj iteraciji se po 16 riječi iz M sprema u polje X. Glavni dio algoritma se sastoji od tri runde hashiranja. Svaka runda se sastoji iz jedne operacije nad svakoj od 16 riječi iz odgovarajućeg polja X. Operacije koje se izvršavaju u tri runde rezultiraju novim vrijednostima za buffere A, B, C, D. Nakon toga se bufferi ažuriraju na način da se vrši zbrajanje modulo 2^{32}, pri čemu se zbrajaju početne vrijednosti, koje su ranije spremljene u zasebne varijable, i nove vrijednosti. Sve tri runde korištene u MD4 konstrukciji se razlikuju. Na kraju, 128-bitovna hash vrijednost se dobije konkatenacijom buffera A, B, C, D.

Runda 1	
1.	$A \leftarrow (A + f(B,C,D) + X[0]) \hookleftarrow 3$
2.	$D \leftarrow (D + f(A,B,C) + X[1]) \hookleftarrow 7$
3.	$C \leftarrow (C + f(D,A,B) + X[2]) \hookleftarrow 11$
4.	$B \leftarrow (B + f(C,D,A) + X[3]) \hookleftarrow 19$
5.	$A \leftarrow (A + f(B,C,D) + X[4]) \hookleftarrow 3$
6.	$D \leftarrow (D + f(A,B,C) + X[5]) \hookleftarrow 7$
7.	$C \leftarrow (C + f(D,A,B) + X[6]) \hookleftarrow 11$
8.	$B \leftarrow (B + f(C,D,A) + X[7]) \hookleftarrow 19$
9.	$A \leftarrow (A + f(B,C,D) + X[8]) \hookleftarrow 3$
10.	$D \leftarrow (D + f(A,B,C) + X[9]) \hookleftarrow 7$
11.	$C \leftarrow (C + f(D,A,B) + X[10]) \hookleftarrow 11$
12.	$B \leftarrow (B + f(C,D,A) + X[11]) \hookleftarrow 19$
13.	$A \leftarrow (A + f(B,C,D) + X[12]) \hookleftarrow 3$
14.	$D \leftarrow (D + f(A,B,C) + X[13]) \hookleftarrow 7$
15.	$C \leftarrow (C + f(D,A,B) + X[14]) \hookleftarrow 11$
16.	$B \leftarrow (B + f(C,D,A) + X[15]) \hookleftarrow 19$

Tablica 1: Runda 1 MD4 algoritma

Runda 2	
1.	$A \leftarrow (A + g(B,C,D) + X[0] + c_1) \hookleftarrow 3$
2.	$D \leftarrow (D + g(A,B,C) + X[4] + c_1) \hookleftarrow 5$
3.	$C \leftarrow (C + g(D,A,B) + X[8] + c_1) \hookleftarrow 9$
4.	$B \leftarrow (B + g(C,D,A) + X[12] + c_1) \hookleftarrow 13$
5.	$A \leftarrow (A + g(B,C,D) + X[1] + c_1) \hookleftarrow 3$
6.	$D \leftarrow (D + g(A,B,C) + X[5] + c_1) \hookleftarrow 5$
7.	$C \leftarrow (C + g(D,A,B) + X[9] + c_1) \hookleftarrow 9$
8.	$B \leftarrow (B + g(C,D,A) + X[13] + c_1) \hookleftarrow 13$
9.	$A \leftarrow (A + g(B,C,D) + X[2] + c_1) \hookleftarrow 3$
10.	$D \leftarrow (D + g(A,B,C) + X[6] + c_1) \hookleftarrow 5$
11.	$C \leftarrow (C + g(D,A,B) + X[10] + c_1) \hookleftarrow 9$
12.	$B \leftarrow (B + g(C,D,A) + X[14] + c_1) \hookleftarrow 13$
13.	$A \leftarrow (A + g(B,C,D) + X[3] + c_1) \hookleftarrow 3$
14.	$D \leftarrow (D + g(A,B,C) + X[7] + c_1) \hookleftarrow 5$
15.	$C \leftarrow (C + g(D,A,B) + X[11] + c_1) \hookleftarrow 9$
16.	$B \leftarrow (B + g(C,D,A) + X[15] + c_1) \hookleftarrow 13$

Tablica 2: Runda 2 MD4 algoritma

Runda 3	
1.	$A \leftarrow (A + h(B, C, D) + X[0] + c_2) \hookleftarrow 3$
2.	$D \leftarrow (D + h(A, B, C) + X[8] + c_2) \hookleftarrow 9$
3.	$C \leftarrow (C + h(D, A, B) + X[4] + c_2) \hookleftarrow 11$
4.	$B \leftarrow (B + h(C, D, A) + X[12] + c_2) \hookleftarrow 15$
5.	$A \leftarrow (A + h(B, C, D) + X[2] + c_2) \hookleftarrow 3$
6.	$D \leftarrow (D + h(A, B, C) + X[10] + c_2) \hookleftarrow 9$
7.	$C \leftarrow (C + h(D, A, B) + X[6] + c_2) \hookleftarrow 11$
8.	$B \leftarrow (B + h(C, D, A) + X[14] + c_2) \hookleftarrow 15$
9.	$A \leftarrow (A + h(B, C, D) + X[1] + c_2) \hookleftarrow 3$
10.	$D \leftarrow (D + h(A, B, C) + X[9] + c_2) \hookleftarrow 9$
11.	$C \leftarrow (C + h(D, A, B) + X[5] + c_2) \hookleftarrow 11$
12.	$B \leftarrow (B + h(C, D, A) + X[13] + c_2) \hookleftarrow 15$
13.	$A \leftarrow (A + h(B, C, D) + X[3] + c_2) \hookleftarrow 3$
14.	$D \leftarrow (D + h(A, B, C) + X[11] + c_2) \hookleftarrow 9$
15.	$C \leftarrow (C + h(D, A, B) + X[7] + c_2) \hookleftarrow 11$
16.	$B \leftarrow (B + h(C, D, A) + X[15] + c_2) \hookleftarrow 15$

Tablica 3: Runda 3 MD4 algoritma

Primjer 3.1. *Hashirajmo poruku "a" pomoću MD4 kriptografske hash funkcije prateći prethodno navedene korake i algoritam. Obratimo pažnju na little-endian arhitekturu!*

Najprije zapišimo string "a" u binarnom obliku. Iz ASCII tablice vrijedi:

Char	*Dec*	*Hex*	*Oct*
a	97	61	141

Sada pretvorimo npr. heksadecimalnu vrijednost za "a" u odgovarajuću binarnu vrijednost. Heksadecimalno 61 *je u binarnom zapisu jednako* 01100001*. Prema tome, dobili smo binarni zapis stringa "a", odnosno* $m = 01100001$*. Duljina binarnog zapisa ulazne poruke je* $s = 8$*. Idući korak je nadopuniti poruku* m *do duljine koja će biti kongruentna* 448 *modulo* 512*. To činimo na način da dodajemo na kraj poruke jedan bit* 1*, te nakon toga nule, sve dok ne zadovoljimo uvjet kongruencije.*

$$01100001||1||0^{(448-s-1)} = 01100001||1||0^{(448-9)}$$

Nakon proširenja do navedene kongruencije, ostala su nam još slobodna 64*-bita da bi dobili duljinu koja je višekratnik broja* 512*. Navedena* 64*- bita će reprezentirati duljinu originalne poruke u binarnom zapisu. Vrijedi:*

$$s_2 = 1000$$

odnosno,

$$s_2 = 0^{60}||1000$$

Sada, podijelimo s_2 *na dvije* 32*-bitovne riječi. Dvije* 32*-bitovne riječi nadodajemo na već produljenu poruku pri čemu prvo dodajemo low-order riječ. Zbog jednostavnosti, objasnit ćemo proces proširenja na heksadecimalnom zapisu.*

$$s_2 = 0^{60}||1000$$
$$s_{16} = 0x8$$

Dvije 32*-bitovne riječi u heksadecimalnom obliku:* $0x00000000$ *i* $0x00000008$*. Nakon dodavanja duljine na proširenu poruku dobijemo:*

$$0x61800000(0)^{104}0000000080000000$$

Primijenimo još little-endian arhitekturu na prvi dio poruke, prije dodavanja duljine originalne poruke.

$$0x80610000(0)^{104}0000000080000000$$

Podijelimo produljenu poruku na 32*-bitovne riječi.Time dobijemo* $N = 16$ 32*-bitovnih riječi. Vrijedi:*

$$\begin{aligned} M[0] &= 0x80610000 \\ M[1] &= 0x00000000 \\ M[2] &= 0x00000000 \\ &\vdots \\ M[13] &= 0x00000000 \\ M[14] &= 0x00000008 \\ M[15] &= 0x00000000 \end{aligned}$$

Sada definirajmo buffere i konstante za drugu i treću rundu, pazeći na little-endian svojstvo.

$$\begin{aligned} A &\leftarrow 0x67452301 \\ B &\leftarrow 0xefcdab89 \\ C &\leftarrow 0x98badcfe \\ D &\leftarrow 0x10325476 \\ c_1 &\leftarrow 0x5a827999 \\ c_2 &\leftarrow 0x6ed9eba1 \end{aligned}$$

Funkcije f, g, h *i runde* $1, 2, 3$ *su definirane kao u opisu algoritma 2. U nastavku ćemo procesuirati blokove od po* 16 *riječi. U našem slučaju, budući da je* $N = 16$*, prva for petlja će se izvršiti samo jednom. Također, u ovom primjeru, vrijedi da je* $X[i] = M[i], i = 0, \ldots, 15$*. Nadalje, spremimo* A, B, C, D *u privremene varijable* A', B', C', D'*, redom. Bitno je napomenuti kako operaciju zbrajanja izvršavamo modulo* 2^{32}*.*

Runda 1*:*

1. *riječ*

$$A = (A + f(B, C, D) + X[0]) \hookleftarrow 3$$
$$f(B, C, D) = 10011000101110101101110011111110$$
$$A = 01100111010001010010001100000001$$
$$X[0] = 00000000000000001000000001100001$$
$$A = 00000000000001000000001100000000$$

2. *riječ*

$$D = (D + f(A, B, C) + X[1]) \hookleftarrow 7$$
$$f(A, B, C) = 10011000101111101101111111111110$$
$$D = 00010000001100100101010001110110$$
$$X[1] = 00000000000000000000000000000000$$
$$D = 01111000100110100011101001010100$$

3. *riječ*

$$C = (C + f(D, A, B) + X[2]) \hookleftarrow 11$$
$$f(D, A, B) = 10000111010001011000001110001001$$
$$C = 10011000101110101101110011111110$$
$$X[2] = 00000000000000000000000000000000$$
$$C = 00000011000001000011100100000000$$

$$\vdots$$

15. *riječ*

$$C = (C + f(D, A, B) + X[14]) \hookleftarrow 11$$
$$f(D, A, B) = 00111111011011011100111111101111$$
$$C = 11011101100101000100001111010000$$
$$X[14] = 00000000000000000000000000001000$$
$$C = 00010000100111100011100011101000$$

16. *riječ*

$$B = (B + f(C, D, A) + X[15]) \hookleftarrow 19$$
$$f(C, D, A) = 10100111111000111111111010011111$$
$$B = 00111011001011011110010111100001$$
$$X[15] = 00000000000000000000000000000000$$
$$B = 00100100000001110001100010001111$$

Runda 2*:*

1. *riječ*

$$A = (A + g(B, C, D) + X[0] + c_1) \hookleftarrow 3$$
$$g(B, C, D) = 00100100100001100011100010001110$$
$$A = 10100111011000011100111011111111$$
$$X[0] = 00000000000000001000000001100001$$
$$A = 00110011010110000000110000111001$$

$$\vdots$$

16. *riječ*

$$B = (B + g(C, D, A) + X[15] + c_1) \hookleftarrow 13$$
$$g(C, D, A) = 00111111000001110110100100000101$$
$$B = 11011010111100011011001011110110$$
$$X[15] = 00000000000000000000000000000000$$
$$B = 01110010101100101000111010001111$$

Runda 3*:*

1. *riječ*

$$A = (A + h(B, C, D) + X[0] + c_2) \hookleftarrow 3$$
$$h(B, C, D) = 11101110101111101100011000010000$$
$$A = 01011101110001110101110001100101$$
$$X[0] = 00000000000000001000000001100001$$
$$A = 11011011000001000111001110111101$$

$$\vdots$$

16. *riječ*

$$B = (B + h(C, D, A) + X[15] + c_2) \hookleftarrow 15$$
$$h(C, D, A) = 00111101011001001100000111111111$$
$$B = 11000010111010011111111101000010$$
$$X[15] = 00000000000000000000000000000000$$
$$B = 01010110011100010011011110010100$$

Nakon izvršavanja treće runde, računamo finalne vrijednosti za buffere (A, B, C, D)*, na*

način da zbrojimo početne i posljednje vrijednosti određenog buffera:

$$A = A + A' = 0xb32ce5bd$$
$$B = B + B' = 0x463ee31d$$
$$C = C + C' = 0xfb055e24$$
$$D = D + D' = 0x24fbd6db$$

Konačnu hash vrijednost dobijemo konkateniranjem riječi A, B, C, D, pri čemu pazimo na little-endian arhitekturu.

$$\begin{aligned} h(\text{"}a\text{"}) &= A||B||C||D \\ &= 0xbde52cb31de33e46245e05fbdbd6fb24 \end{aligned}$$

Rezultat se može provjeriti u testnoj tablici:

m	MD4(m)
""	$0x31d6cfe0d16ae931b73c59d7e0c089c0$
"a"	$0xbde52cb31de33e46245e05fbdbd6fb24$
"abc"	$0xa448017aaf21d8525fc10ae87aa6729d$
"$message\ digest$"	$0xd9130a8164549fe818874806e1c7014b$
"$abcdefghijklmnopqrstuvwxyz$"	$0xd79e1c308aa5bbcdeea8ed63df412da9$
"$ABCDEFGHIJKLMNOPQRSTUVWX YZabcdefghijklmnopqrstuvwxyz0123456789$"	$0x043f8582f241db351ce627e153e7f0e4$
"12345678901234567890123456789012345678901234567890123456789012345678901234567890"	$0xe33b4ddc9c38f2199c3e7b164fcc0536$

Tablica 4: Testni primjer [5]

```
if __name__=="__main__":

    md4 = MD4()
    md4.add(b'a')
    d = md4.finish()
    print(codecs.encode(d, "hex"))
```

Programski code 1: Testiranje navedenog primjera u implementiranom python algoritmu

MD4 zadovoljava efekat lavine, budući da najmanja promjena u ulazu rezultira potpuno drugačijom hash vrijednošću. To možemo vidjeti na idućem primjeru:

$$\begin{aligned} &MD4(\text{"The quick brown fox jumps over the lazy dog"}) \\ &= 0x1bee69a46ba811185c194762abaeae90 \end{aligned}$$

$$\begin{aligned} &MD4(\text{"The quick brown fox jumps over the lazy cog"}) \\ &= 0xb86e130ce7028da59e672d56ad0113df \end{aligned}$$

Sigurnost MD4 algoritma je bila ugrožena u par navrata, što je dovelo do potrebe razvijanja novih hash funkcija. Prvi potpuni napad kolizijom na MD4 je objavljen 1995. od strane Hansa Dobbertina. Implementacija njegovog napada na računalu pronalazi koliziju za MD4 u nekoliko sekundi. Nakon toga je objavljeno još nekoliko napada. Kao odgovor na pronalazak slabosti MD4 funkcije, Ronald Rivest je dizajnirao MD5 funkciju.

4 MD5

MD5 je ojačana verzija MD4 hash funkcije. Nastala je 1991. godine kao odgovor na pronalazak slabosti MD4 funkcije. Razlike između MD4 i MD5 nisu velike. Jedna od bitnijih razlika je ta što MD5 ima 4 runde umjesto 3. Dodavanjem četvrte runde poboljšana je sigurnost funkcije. Međutim, u usporedbi s MD4, performanse su smanjene. MD5 funkcija je, strukturno gledano, vrlo slična MD4, budući da je iz nje i nastala. Proces nadopunjavanja poruke m tj. koraci 1 i 2 su identični. Korak 3 koji predstavlja inicijalizaciju buffera je jednak. MD5 funkcija također ima pomoćne funkcije. Funkcije f i h su definirane kao i ranije, dok je funkcija g predefinirana i zamijenjena asimetričnom funkcijom:

$$g(X, Y, Z) = ((X \wedge Z) \vee (Y \wedge (\neg Z)))$$

Dodatno, uvedena je još jedna pomoćna funkcija i :

$$i(X, Y, Z) = Y \oplus (X \vee (\neg Z))$$

Nadalje, MD5 funkcija koristi polje konstanti T koje se sastoji od 64 elementa definiranih na idući način:

$$T[i] = \lfloor 2^{32} \cdot |\sin(i)| \rfloor, \ 1 \leq i \leq 64,$$

gdje se i uzima u radijanima. Budući da $|sin(i)|$ poprima vrijednosti između 0 i 1, te da se te vrijednosti dodatno množe s 2^{32} i uzima pod konačne vrijednosti, svaki element polja T je cijeli broj koji se može reprezentirati 32-bitovnim zapisom. U svakom koraku runde se sada dodaje rezultat prethodnog koraka, npr:

$$A \leftarrow B + (A + f(B, C, D) + X[0] + T[1]) \hookleftarrow 7$$

Razlike između MD4 i MD5:

1. Dodana je četvrta runda.
2. Svaki korak runde ima jedinstvenu konstantu $T[i]$.
3. Funkcija g je predefinirana.
4. U svakom koraku runde se vrši dodavanje rezultata prethodnog koraka, što omogućuje brže postizanje efekta lavine.
5. Redoslijed pristupanja riječima produljene poruke je izmijenjen u rundama 2 i 3.
6. Broj rotacije u lijevo je također promijenjen za svaku rundu.

MD5 zadovoljava efekat lavine, što možemo vidjeti na primjeru:

$$MD5(\text{"The quick brown fox jumps over the lazy dog"}) \\ = 0x9e107d9d372bb6826bd81d3542a419d6$$

$$MD5(\text{"The quick brown fox jumps over the lazy cog"}) \\ = 0x1055d3e698d289f2af8663725127bd4b$$

Algoritam 3 MD5 [6]

$m = m_0 m_1 \dots m_{s-1}$
Konstruirati $M[0]M[1] \dots M[N-1]$
$A \leftarrow 0x67452301$
$B \leftarrow 0xefcdab89$
$C \leftarrow 0x98badcfe$
$D \leftarrow 0x10325476$
for $i \leftarrow 0$ **to** $N/16 - 1$ **do**
    for $j \leftarrow 0$ **to** 15 **do**
        $X[j] = M[i \cdot 16 + j]$
    end for
    $A' \leftarrow A$
    $B' \leftarrow B$
    $C' \leftarrow C$
    $D' \leftarrow D$
    Runda 1
    Runda 2
    Runda 3
    Runda 4
    $A \leftarrow A + A'$
    $B \leftarrow B + B'$
    $C \leftarrow C + C'$
    $D \leftarrow D + D'$
end for
return $h(m) = A||B||C||D$

Primjer 4.1. *Hashirajmo poruku "a" pomoću MD5 kriptografske hash funkcije prateći prethodno opisanu funkciju i algoritam.*

Prvi dio proširivanja poruke je identičan kao u primjeru 3.1. Nastavimo od dijela gdje definiramo riječi $M[0], M[1], \dots, M[N-1]$.

$$\begin{aligned} M[0] &= 0x80610000 \\ M[1] &= 0x00000000 \\ M[2] &= 0x00000000 \\ &\vdots \\ M[13] &= 0x00000000 \\ M[14] &= 0x00000008 \\ M[15] &= 0x00000000 \end{aligned}$$

Sada definirajmo buffere i konstante:

$$A \leftarrow 0x67452301$$
$$B \leftarrow 0xefcdab89$$
$$C \leftarrow 0x98badcfe$$
$$D \leftarrow 0x10325476$$

$T[1] = 0xd76aa478$	$T[17] = 0xf61e2562$	$T[33] = 0xfffa3942$	$T[49] = 0xf4292244$
$T[2] = 0xe8c7b756$	$T[18] = 0xc040b340$	$T[34] = 0x8771f681$	$T[50] = 0x432aff97$
$T[3] = 0x242070db$	$T[19] = 0x265e5a51$	$T[35] = 0x6d9d6122$	$T[51] = 0xab9423a7$
$T[4] = 0xc1bdceee$	$T[20] = 0xe9b6c7aa$	$T[36] = 0xfde5380c$	$T[52] = 0xfc93a039$
$T[5] = 0xf57c0faf$	$T[21] = 0xd62f105d$	$T[37] = 0xa4beea44$	$T[53] = 0x655b59c3$
$T[6] = 0x4787c62a$	$T[22] = 0x2441453$	$T[38] = 0x4bdecfa9$	$T[54] = 0x8f0ccc92$
$T[7] = 0xa8304613$	$T[23] = 0xd8a1e681$	$T[39] = 0xf6bb4b60$	$T[55] = 0xffeff47d$
$T[8] = 0xfd469501$	$T[24] = 0xe7d3fbc8$	$T[40] = 0xbebfbc70$	$T[56] = 0x85845dd1$
$T[9] = 0x698098d8$	$T[25] = 0x21e1cde6$	$T[41] = 0x289b7ec6$	$T[57] = 0x6fa87e4f$
$T[10] = 0x8b44f7af$	$T[26] = 0xc33707d6$	$T[42] = 0xeaa127fa$	$T[58] = 0xfe2ce6e0$
$T[11] = 0xffff5bb1$	$T[27] = 0xf4d50d87$	$T[43] = 0xd4ef3085$	$T[59] = 0xa3014314$
$T[12] = 0x895cd7be$	$T[28] = 0x455a14ed$	$T[44] = 0x4881d05$	$T[60] = 0x4e0811a1$
$T[13] = 0x6b901122$	$T[29] = 0xa9e3e905$	$T[45] = 0xd9d4d039$	$T[61] = 0xf7537e82$
$T[14] = 0xfd987193$	$T[30] = 0xfcefa3f8$	$T[46] = 0xe6db99e5$	$T[62] = 0xbd3af235$
$T[15] = 0xa679438e$	$T[31] = 0x676f02d9$	$T[47] = 0x1fa27cf8$	$T[63] = 0x2ad7d2bb$
$T[16] = 0x49b40821$	$T[32] = 0x8d2a4c8a$	$T[48] = 0xc4ac5665$	$T[64] = 0xeb86d391$

Tablica 5: Konstante $T[i]$ MD5 algoritma

Funkcije f, g, h, i i runde $1, 2, 3, 4$ su definirane kao u algoritmu 3. U nastavku ćemo procesuirati blokove od po 16 riječi. Pratimo algoritam:

Runda 1:

1. *riječ*

$$A = B + (A + f(B, C, D) + X[0] + T[1]) \hookleftarrow 7$$
$$f(B, C, D) = 10011000101110101101110011111110$$
$$A = 01100111010001010010001100000001$$
$$X[0] = 00000000000000001000000001100001$$
$$A = 10100101011000000001011111110100$$

$$\vdots$$

Runda 2*:*

1. *riječ*

$$A = B + (A + g(B, C, D) + X[1] + T[17]) \hookleftarrow 5$$
$$g(B, C, D) = 11110011000111100011110001111010$$
$$A = 01000100001001000100110011111000$$
$$X[1] = 00000000000000000000000000000000$$
$$A = 10011100001101000001011101100111$$

$$\vdots$$

Runda 3*:*

1. *riječ*

$$A = B + (A + h(B, C, D) + X[5] + T[33]) \hookleftarrow 4$$
$$h(B, C, D) = 01100100111000110100011010010101$$
$$A = 10010011100001000010111010011000$$
$$X[5] = 00000000000000000000000000000000$$
$$A = 10111101011000000111110100011110$$

$$\vdots$$

Runda 4*:*

1. *riječ*

$$A = A \leftarrow B + (A + i(B, C, D) + X[0] + T[49]) \hookleftarrow 6$$
$$i(B, C, D) = 11100010000010100000101100010011$$
$$A = 00001010110001010000111000011000$$
$$X[5] = 00000000000000001000000001100001$$
$$A = 11001010101110001111111001000010$$

$$\vdots$$

Nakon izvršavanja četvrte runde, računamo finalne vrijednosti za buffere (A, B, C, D)*:*

$$A = A + A' = 0xb975c10c$$
$$B = B + B' = 0xa8b6f1c0$$
$$C = C + C' = 0xe299c331$$
$$D = D + D' = 0x61267769$$

Konačnu hash vrijednost dobijemo konkateniranjem riječi A, B, C, D pri čemu pazimo na little-endian arhitekturu.

$$\begin{aligned} h(\text{"}a\text{"}) &= A||B||C||D \\ &= 0x0cc175b9c0f1b6a831c399e269772661 \end{aligned}$$

Rezultat se može provjeriti u testnoj tablici:

m	MD5(m)
""	$0xd41d8cd98f00b204e9800998ecf8427e$
"a"	$0x0cc175b9c0f1b6a831c399e269772661$
"abc"	$0x900150983cd24fb0d6963f7d28e17f72$
"$message\ digest$"	$0xf96b697d7cb7938d525a2f31aaf161d0$
"$abcdefghijklmnopqrstuvwxyz$"	$0xc3fcd3d76192e4007dfb496cca67e13b$
"$ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789$"	$0xd174ab98d277d9f5a5611c2c9f419d9f$
"12345678901234567890123456789012345678901234567890123456789012345678901234567890"	$0x57edf4a22be3c955ac49da2e2107b67a$

Tablica 6: Testni primjer [6]

```
if __name__=="__main__":

    md5 = MD5()
    md5.add(b'a')
    d = md5.finish()
    print(codecs.encode(d, "hex"))
```

Programski code 2: Testiranje primjera 4.1 u implementiranom python algoritmu

Runda 1	
1.	$A \leftarrow B + (A + f(B,C,D) + X[0] + T[1]) \hookleftarrow 7$
2.	$D \leftarrow A + (D + f(A,B,C) + X[1] + T[2]) \hookleftarrow 12$
3.	$C \leftarrow D + (C + f(D,A,B) + X[2] + T[3]) \hookleftarrow 17$
4.	$B \leftarrow C + (B + f(C,D,A) + X[3] + T[4]) \hookleftarrow 22$
5.	$A \leftarrow B + (A + f(B,C,D) + X[4] + T[5]) \hookleftarrow 7$
6.	$D \leftarrow A + (D + f(A,B,C) + X[5] + T[6]) \hookleftarrow 12$
7.	$C \leftarrow D + (C + f(D,A,B) + X[6] + T[7]) \hookleftarrow 17$
8.	$B \leftarrow C + (B + f(C,D,A) + X[7] + T[8]) \hookleftarrow 22$
9.	$A \leftarrow B + (A + f(B,C,D) + X[8] + T[9]) \hookleftarrow 7$
10.	$D \leftarrow A + (D + f(A,B,C) + X[9] + T[10]) \hookleftarrow 12$
11.	$C \leftarrow D + (C + f(D,A,B) + X[10] + T[11]) \hookleftarrow 17$
12.	$B \leftarrow C + (B + f(C,D,A) + X[11] + T[12]) \hookleftarrow 22$
13.	$A \leftarrow B + (A + f(B,C,D) + X[12] + T[13]) \hookleftarrow 7$
14.	$D \leftarrow A + (D + f(A,B,C) + X[13] + T[14]) \hookleftarrow 12$
15.	$C \leftarrow D + (C + f(D,A,B) + X[14] + T[15]) \hookleftarrow 17$
16.	$B \leftarrow C + (B + f(C,D,A) + X[15] + T[16]) \hookleftarrow 22$

Tablica 7: Runda 1 MD5 algoritma

Runda 2	
1.	$A \leftarrow B + (A + g(B, C, D) + X[1] + T[17]) \hookleftarrow 5$
2.	$D \leftarrow A + (D + g(A, B, C) + X[6] + T[18]) \hookleftarrow 9$
3.	$C \leftarrow D + (C + g(D, A, B) + X[11] + T[19]) \hookleftarrow 14$
4.	$B \leftarrow C + (B + g(C, D, A) + X[0] + T[20]) \hookleftarrow 20$
5.	$A \leftarrow B + (A + g(B, C, D) + X[5] + T[21]) \hookleftarrow 5$
6.	$D \leftarrow A + (D + g(A, B, C) + X[10] + T[22]) \hookleftarrow 9$
7.	$C \leftarrow D + (C + g(D, A, B) + X[15] + T[23]) \hookleftarrow 14$
8.	$B \leftarrow C + (B + g(C, D, A) + X[4] + T[24]) \hookleftarrow 20$
9.	$A \leftarrow B + (A + g(B, C, D) + X[9] + T[25]) \hookleftarrow 5$
10.	$D \leftarrow A + (D + g(A, B, C) + X[14] + T[26]) \hookleftarrow 9$
11.	$C \leftarrow D + (C + g(D, A, B) + X[3] + T[27]) \hookleftarrow 14$
12.	$B \leftarrow C + (B + g(C, D, A) + X[8] + T[28]) \hookleftarrow 20$
13.	$A \leftarrow B + (A + g(B, C, D) + X[13] + T[29]) \hookleftarrow 5$
14.	$D \leftarrow A + (D + g(A, B, C) + X[2] + T[30]) \hookleftarrow 9$
15.	$C \leftarrow D + (C + g(D, A, B) + X[7] + T[31]) \hookleftarrow 14$
16.	$B \leftarrow C + (B + g(C, D, A) + X[12] + T[32]) \hookleftarrow 20$

Tablica 8: Runda 2 MD5 algoritma

Runda 3	
1.	$A \leftarrow B + (A + h(B, C, D) + X[5] + T[33]) \hookleftarrow 4$
2.	$D \leftarrow A + (D + h(A, B, C) + X[8] + T[34]) \hookleftarrow 11$
3.	$C \leftarrow D + (C + h(D, A, B) + X[11] + T[35]) \hookleftarrow 16$
4.	$B \leftarrow C + (B + h(C, D, A) + X[14] + T[36]) \hookleftarrow 23$
5.	$A \leftarrow B + (A + h(B, C, D) + X[1] + T[37]) \hookleftarrow 4$
6.	$D \leftarrow A + (D + h(A, B, C) + X[4] + T[38]) \hookleftarrow 11$
7.	$C \leftarrow D + (C + h(D, A, B) + X[7] + T[39]) \hookleftarrow 16$
8.	$B \leftarrow C + (B + h(C, D, A) + X[10] + T[40]) \hookleftarrow 23$
9.	$A \leftarrow B + (A + h(B, C, D) + X[13] + T[41]) \hookleftarrow 4$
10.	$D \leftarrow A + (D + h(A, B, C) + X[0] + T[42]) \hookleftarrow 11$
11.	$C \leftarrow D + (C + h(D, A, B) + X[3] + T[43]) \hookleftarrow 16$
12.	$B \leftarrow C + (B + h(C, D, A) + X[6] + T[44]) \hookleftarrow 23$
13.	$A \leftarrow B + (A + h(B, C, D) + X[9] + T[45]) \hookleftarrow 4$
14.	$D \leftarrow A + (D + h(A, B, C) + X[12] + T[46]) \hookleftarrow 11$
15.	$C \leftarrow D + (C + h(D, A, B) + X[15] + T[47]) \hookleftarrow 16$
16.	$B \leftarrow C + (B + h(C, D, A) + X[2] + T[48]) \hookleftarrow 23$

Tablica 9: Runda 3 MD5 algoritma

Runda 4	
1.	$A \leftarrow B + (A + i(B, C, D) + X[0] + T[49]) \hookleftarrow 6$
2.	$D \leftarrow A + (D + i(A, B, C) + X[7] + T[50]) \hookleftarrow 10$
3.	$C \leftarrow D + (C + i(D, A, B) + X[14] + T[51]) \hookleftarrow 15$
4.	$B \leftarrow C + (B + i(C, D, A) + X[5] + T[52]) \hookleftarrow 21$
5.	$A \leftarrow B + (A + i(B, C, D) + X[12] + T[53]) \hookleftarrow 6$
6.	$D \leftarrow A + (D + i(A, B, C) + X[3] + T[54]) \hookleftarrow 10$
7.	$C \leftarrow D + (C + i(D, A, B) + X[10] + T[55]) \hookleftarrow 15$
8.	$B \leftarrow C + (B + i(C, D, A) + X[1] + T[56]) \hookleftarrow 21$
9.	$A \leftarrow B + (A + i(B, C, D) + X[8] + T[57]) \hookleftarrow 6$
10.	$D \leftarrow A + (D + i(A, B, C) + X[15] + T[58]) \hookleftarrow 10$
11.	$C \leftarrow D + (C + i(D, A, B) + X[6] + T[59]) \hookleftarrow 15$
12.	$B \leftarrow C + (B + i(C, D, A) + X[13] + T[60]) \hookleftarrow 21$
13.	$A \leftarrow B + (A + i(B, C, D) + X[4] + T[61]) \hookleftarrow 6$
14.	$D \leftarrow A + (D + i(A, B, C) + X[11] + T[62]) \hookleftarrow 10$
15.	$C \leftarrow D + (C + i(D, A, B) + X[2] + T[63]) \hookleftarrow 15$
16.	$B \leftarrow C + (B + i(C, D, A) + X[9] + T[64]) \hookleftarrow 21$

Tablica 10: Runda 4 MD5 algoritma

Bert den Boer i Antoon Bosselaers su 1993. godine pronašli pseudo-koliziju kompresijske funkcije MD5 [1]. Prvu potpunu koliziju kompresijske funkcije pronašao je Hans Dobbertin 1996. godine. Sigurnost MD5 je narušena te je zbog toga došlo do potrebe za razvojem novih hash funkcija.

5 SHA-1

SHA-1 (engl. Secure Hash Algorithm 1) algoritam je razvijen 1995. godine od strane američke vladine agencije NSA (engl. National Security Agency). On je drugi u nizu SHA algoritama, budući da je nastao iz tzv. SHA-0. SHA-0 algoritam je razvijen 1993. godine, ali zbog svojih slabosti je ubrzo povučen i zamijenjen sa SHA-1. SHA-0 i SHA-1 su praktički identični, a jedina razlika je dodavanje rotacije u lijevo za 1 kod definiranja rasporeda poruka. SHA-1 je konceptualno i strukturno sličan MD4 i MD5 hash funkcijama. Dvije najbitnije razlike su te da je SHA-1 dizajniran tako da se optimalno izvršava na big-endian računalnim sustavima, i da funkcija koristi 5 buffera umjesto 4. Zbog dodatnog, petog buffera, hash vrijednost SHA-1 funkcije je duljine 160 bitova. Proširenjem hash vrijednosti za 32 bita, postignuta je veća sigurnost.

SHA-1 hash funkcija koristi niz funkcija $f_0, f_1, \ldots, f_{79}$, koje su definirane na idući način:

$$f_t(X,Y,Z) = \begin{cases} (X \wedge Y) \oplus (\neg X \wedge Z), & 0 \leq t \leq 19 \\ X \oplus Y \oplus Z, & 20 \leq t \leq 39 \\ (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z), & 40 \leq t \leq 59 \\ X \oplus Y \oplus Z, & 60 \leq t \leq 79 \end{cases}$$

Dodatno, SHA-1 funkcija koristi niz konstanti $K_0, K_1, \ldots, K_{79}$ definiranih kao 32-bitovne riječi. U nastavku su definirane konstante:

$$K_t = \begin{cases} \lfloor 2^{30}\sqrt{2} \rfloor = 0x5a827999, & 0 \leq t \leq 19 \\ \lfloor 2^{30}\sqrt{3} \rfloor = 0x6ed9eba1, & 20 \leq t \leq 39 \\ \lfloor 2^{30}\sqrt{5} \rfloor = 0x8f1bbcdc, & 40 \leq t \leq 59 \\ \lfloor 2^{30}\sqrt{10} \rfloor = 0xca62c1d6, & 60 \leq t \leq 79 \end{cases}$$

Primijetimo da prve dvije konstante odgovaraju konstantama c_1 i c_2 iz MD4 konstrukcije. Predprocesni dio nadopunjavanja poruke je jednak kao kod MD4 i MD5. Osim funkcija f_t i konstanti K_t, definiran je i raspored poruka W koji sadrži 80 32-bitovnih riječi:

$$W_t = \begin{cases} M[t], & 0 \leq t \leq 15 \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \hookleftarrow 1, & 16 \leq t \leq 79 \end{cases}$$

Nakon predprocesnog dijela poruka se iterativno hashira korištenjem algoritma 4. Najprije se inicijaliziraju bufferi A, B, C, D, E, a nakon toga se svaki blok poruke $M[0], M[1], M[2], \ldots, M[N-1]$ iterativno procesuira, pri čemu se rezultat svake iteracije koristi u idućoj iteraciji. Konačno, rezultat, odnosno hash vrijednost, se dobije konkatenacijom buffera A, B, C, D, E.

Algoritam 4 SHA-1 [2]

$m = m_0 m_1 \dots m_{s-1}$
Konstruirati $M[0]M[1]\dots M[N-1]$
$A \leftarrow 0x67452301$
$B \leftarrow 0xefcdab89$
$C \leftarrow 0x98badcfe$
$D \leftarrow 0x10325476$
$E \leftarrow 0xc3d2e1f0$
for $i \leftarrow 0$ **to** $N/16 - 1$ **do**
    Konstruirati raspored poruka W
    $A' \leftarrow A$
    $B' \leftarrow B$
    $C' \leftarrow C$
    $D' \leftarrow D$
    $E' \leftarrow E$
    for $t \leftarrow 0$ **to** 79 **do**
        $T \leftarrow (A \hookleftarrow 5) + f_t(B,C,D) + E + K_t + W_t$
        $E \leftarrow D$
        $D \leftarrow C$
        $C \leftarrow B \hookleftarrow 30$
        $B \leftarrow A$
        $A \leftarrow T$
    end for
    $A \leftarrow A + A'$
    $B \leftarrow B + B'$
    $C \leftarrow C + C'$
    $D \leftarrow D + D'$
    $E \leftarrow E + E'$
end for
return $h(m) = A||B||C||D||E$

SHA-1 zadovoljava efekat lavine, što možemo vidjeti na primjeru:

$$SHA1(\text{"The quick brown fox jumps over the lazy dog"}) = 0x2fd4e1c67a2d28fced849ee1bb76e7391b93eb12$$

$$SHA1(\text{"The quick brown fox jumps over the lazy cog"}) = 0xde9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3$$

Primjer 5.1. *Hashirajmo poruku "abc" pomoću SHA-1 kriptografske hash funkcije prateći prethodno opisanu funkciju i algoritam.*

Najprije zapišimo string "abc" u binarnom obliku. Iz ASCII tablice [7]vrijedi:

Char	*Dec*	*Hex*	*Oct*	*Bin*
a	97	61	141	01100001
b	98	62	142	01100010
c	99	63	143	01100011

Binarni zapis stringa "abc" je $m = 011000010110001001100011$. *Duljina binarnog zapisa ulazne poruke je* $s = 24$.
Idući korak je nadopuniti poruku m *do duljine koja će biti kongruentna* 448 *modulo* 512. *To činimo na način da dodajemo na kraj poruke jedan bit* 1, *te nakon toga nule, sve dok ne zadovoljimo uvjet kongruencije.*

$$011000010110001001100011||1||0^{(448-s-1)} = 011000010110001001100011||1||0^{(448-25)}$$

Nakon proširenja do navedene kongruencije, ostala su nam još slobodna 64*-bita da bi dobili duljinu koja je višekratnik broja* 512. *Navedena* 64*-bita će reprezentirati duljinu originalne poruke u binarnom zapisu. Vrijedi:*

$$s_2 = 11000$$

odnosno,

$$s_2 = 0^{59}||11000$$

Sada, podijelimo s_2 *na dvije* 32*-bitovne riječi. Dvije* 32*-bitovne riječi nadodajemo na već produljenu poruku u skladu sa svojstvom najveće značajnosti (engl. most significant). Zbog jednostavnosti, objasnit ćemo proces proširenja na heksadecimalnom zapisu.*

$$s_2 = 0^{59}||11000$$
$$s_{16} = 0x18$$

Dvije 32*-bitovne riječi u heksadecimalnom obliku:* $0x00000000$ *i* $0x00000018$. *Nakon dodavanja duljine na proširenu poruku dobijemo:*

$$0x616263800^{104}0000000000000018$$

Podijelimo produljenu poruku na 32*-bitovne riječi.Time dobijemo* $N = 16$ 32*-bitovnih riječi. Vrijedi:*

$$M[0] = 0x61626380$$
$$M[1] = 0x00000000$$
$$M[2] = 0x00000000$$
$$\vdots$$
$$M[13] = 0x00000000$$
$$M[14] = 0x00000000$$
$$M[15] = 0x00000018$$

Sada definirajmo buffere

$$A = 0x67452301$$
$$B = 0xefcdab89$$
$$C = 0x98badcfe$$
$$D = 0x10325476$$
$$E = 0xc3d2e1f0$$

koje spremamo u vrijednosti A', B', C', D', E', redom.

$W[0] = 0x61626380$	$W[20] = 0x00000000$	$W[40] = 0xd2e138c4$	$W[60] = 0x7ba06619$
$W[1] = 0x00000000$	$W[21] = 0x00000060$	$W[41] = 0x00000f00$	$W[61] = 0x6380aea2$
$W[2] = 0x00000000$	$W[22] = 0x0b131c03$	$W[42] = 0x3afb5079$	$W[62] = 0x0ae55269$
$W[3] = 0x00000000$	$W[23] = 0x00000030$	$W[43] = 0x898e04e5$	$W[63] = 0x627b49a1$
$W[4] = 0x00000000$	$W[24] = 0x85898ec1$	$W[44] = 0xe2ba3c2b$	$W[64] = 0x7cd45c9d$
$W[5] = 0x00000000$	$W[25] = 0x16263806$	$W[45] = 0x000060c0$	$W[65] = 0x000f0000$
$W[6] = 0x00000000$	$W[26] = 0x00000000$	$W[46] = 0x053a37cd$	$W[66] = 0xfb50753a$
$W[7] = 0x00000000$	$W[27] = 0x00000180$	$W[47] = 0x74458547$	$W[67] = 0xec6765e8$
$W[8] = 0x00000000$	$W[28] = 0x2c4c700c$	$W[48] = 0xda9415ed$	$W[68] = 0xba3c2be2$
$W[9] = 0x00000000$	$W[29] = 0x000000f0$	$W[49] = 0x26380a16$	$W[69] = 0x0060c000$
$W[10] = 0x00000000$	$W[30] = 0x93afb507$	$W[50] = 0x626383a1$	$W[70] = 0x3a37cd05$
$W[11] = 0x00000000$	$W[31] = 0x5898e048$	$W[51] = 0x4ebf54de$	$W[71] = 0x458546f4$
$W[12] = 0x00000000$	$W[32] = 0x8e9a9202$	$W[52] = 0x3835b44b$	$W[72] = 0xb8599dd6$
$W[13] = 0x00000000$	$W[33] = 0x00000600$	$W[53] = 0x0000f600$	$W[73] = 0x380a1a26$
$W[14] = 0x00000000$	$W[34] = 0xb131c0f0$	$W[54] = 0x1e84c7a3$	$W[74] = 0x01e02203$
$W[15] = 0x00000018$	$W[35] = 0x16263bc6$	$W[55] = 0x98e04d98$	$W[75] = 0xe7cc3456$
$W[16] = 0xc2c4c700$	$W[36] = 0x4ebed41e$	$W[56] = 0x651d16a0$	$W[76] = 0xe6e60b69$
$W[17] = 0x00000000$	$W[37] = 0x626380a1$	$W[57] = 0x62658ca1$	$W[77] = 0x00f60a00$
$W[18] = 0x00000030$	$W[38] = 0x16263806$	$W[58] = 0x458544d6$	$W[78] = 0x5795ef4f$
$W[19] = 0x85898e01$	$W[39] = 0x000018c0$	$W[59] = 0x44584cb7$	$W[79] = 0x822e0879$

Tablica 11: Raspored poruka W

Sada pristupamo koraku 15 *algoritma* 4 *, gdje prolazimo kroz for petlju i vršimo ažuriranje buffera.*
Prva iteracija for petlje:

$$
\begin{aligned}
f_0(B,C,D) &= 10011000101110101101110011111110 = 0x98badcfe \\
K_0 &= 01011010100000100111100110011001 = 0x5a827999 \\
W_0 &= 01100001011000100110001110000000 = 0x61626380 \\
T &= (A \hookleftarrow 5) + f_0(B,C,D) + E + K_0 + W_0 \\
&= 0xe8a4602c + 0x98badcfe + 0xc3d2e1f0 + 0x5a827999 + 0x61626380 \\
&= 0x116fc33 \\
E &= 0x10325476 \\
D &= 0x98badcfe \\
C &= 0x7bf36ae2 \\
B &= 0x67452301 \\
A &= 0x116fc33
\end{aligned}
$$

	A	B	C	D	E
$t = 0$	$0x116fc33$	$0x67452301$	$0x7bf36ae2$	$0x98badcfe$	$0x10325476$
$t = 1$	$0x8990536d$	$0x116fc33$	$0x59d148c0$	$0x7bf36ae2$	$0x98badcfe$
$t = 2$	$0xa1390f08$	$0x8990536d$	$0xc045bf0c$	$0x59d148c0$	$0x7bf36ae2$
$t = 3$	$0xcdd8e11b$	$0xa1390f08$	$0x626414db$	$0xc045bf0c$	$0x59d148c0$
$t = 4$	$0xcfd499de$	$0xcdd8e11b$	$0x284e43c2$	$0x626414db$	$0xc045bf0c$
$t = 5$	$0x3fc7ca40$	$0xcfd499de$	$0xf3763846$	$0x284e43c2$	$0x626414db$
$t = 6$	$0x993e30c1$	$0x3fc7ca40$	$0xb3f52677$	$0xf3763846$	$0x284e43c2$
$t = 7$	$0x9e8c07d4$	$0x993e30c1$	$0xfff1f290$	$0xb3f52677$	$0xf3763846$
$t = 8$	$0x4b6ae328$	$0x9e8c07d4$	$0x664f8c30$	$0xfff1f290$	$0xb3f52677$
$t = 9$	$0x8351f929$	$0x4b6ae328$	$0x27a301f5$	$0x664f8c30$	$0xfff1f290$
$t = 10$	$0xfbda9e89$	$0x8351f929$	$0x12dab8ca$	$0x27a301f5$	$0x664f8c30$
$t = 11$	$0x63188fe4$	$0xfbda9e89$	$0x60d47e4a$	$0x12dab8ca$	$0x27a301f5$
$t = 12$	$0x4607b664$	$0x63188fe4$	$0x7ef6a7a2$	$0x60d47e4a$	$0x12dab8ca$
$t = 13$	$0x9128f695$	$0x4607b664$	$0x18c623f9$	$0x7ef6a7a2$	$0x60d47e4a$
$t = 14$	$0x196bee77$	$0x9128f695$	$0x1181ed99$	$0x18c623f9$	$0x7ef6a7a2$
$t = 15$	$0x20bdd62f$	$0x196bee77$	$0x644a3da5$	$0x1181ed99$	$0x18c623f9$
$t = 16$	$0x4e925823$	$0x20bdd62f$	$0xc65afb9d$	$0x644a3da5$	$0x1181ed99$
$t = 17$	$0x82aa6728$	$0x4e925823$	$0xc82f758b$	$0xc65afb9d$	$0x644a3da5$
$t = 18$	$0xdc64901d$	$0x82aa6728$	$0xd3a49608$	$0xc82f758b$	$0xc65afb9d$
$t = 19$	$0xfd9e1d7d$	$0xdc64901d$	$0x20aa99ca$	$0xd3a49608$	$0xc82f758b$
$t = 20$	$0x1a37b0ca$	$0xfd9e1d7d$	$0x77192407$	$0x20aa99ca$	$0xd3a49608$
$t = 21$	$0x33a23bfc$	$0x1a37b0ca$	$0x7f67875f$	$0x77192407$	$0x20aa99ca$
$t = 22$	$0x21283486$	$0x33a23bfc$	$0x868dec32$	$0x7f67875f$	$0x77192407$
$t = 23$	$0xd541f12d$	$0x21283486$	$0xce88eff$	$0x868dec32$	$0x7f67875f$
$t = 24$	$0xc7567dc6$	$0xd541f12d$	$0x884a0d21$	$0xce88eff$	$0x868dec32$
$t = 25$	$0x48413ba4$	$0xc7567dc6$	$0x75507c4b$	$0x884a0d21$	$0xce88eff$
$t = 26$	$0xbe35fbd5$	$0x48413ba4$	$0xb1d59f71$	$0x75507c4b$	$0x884a0d21$
$t = 27$	$0x4aa84d97$	$0xbe35fbd5$	$0x12104ee9$	$0xb1d59f71$	$0x75507c4b$
$t = 28$	$0x8370b52e$	$0x4aa84d97$	$0x6f8d7ef5$	$0x12104ee9$	$0xb1d59f71$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$t = 65$	$0x101655f9$	$0x19dfa7ac$	$0x5ad87278$	$0xed9b0082$	$0x9523272c$
$t = 66$	$0xc3df2b4$	$0x101655f9$	$0x677e9eb$	$0x5ad87278$	$0xed9b0082$
$t = 67$	$0x78dd4d2b$	$0xc3df2b4$	$0x4405957e$	$0x677e9eb$	$0x5ad87278$
$t = 68$	$0x497093c0$	$0x78dd4d2b$	$0x30f7cad$	$0x4405957e$	$0x677e9eb$
$t = 69$	$0x3f2588c2$	$0x497093c0$	$0xde37534a$	$0x30f7cad$	$0x4405957e$
$t = 70$	$0xc199f8c7$	$0x3f2588c2$	$0x125c24f0$	$0xde37534a$	$0x30f7cad$
$t = 71$	$0x39859de7$	$0xc199f8c7$	$0x8fc96230$	$0x125c24f0$	$0xde37534a$
$t = 72$	$0xedb42de4$	$0x39859de7$	$0xf0667e31$	$0x8fc96230$	$0x125c24f0$
$t = 73$	$0x11793f6f$	$0xedb42de4$	$0xce616779$	$0xf0667e31$	$0x8fc96230$
$t = 74$	$0x5ee76897$	$0x11793f6f$	$0x3b6d0b79$	$0xce616779$	$0xf0667e31$
$t = 75$	$0x63f7dab7$	$0x5ee76897$	$0xc45e4fdb$	$0x3b6d0b79$	$0xce616779$
$t = 76$	$0xa079b7d9$	$0x63f7dab7$	$0xd7b9da25$	$0xc45e4fdb$	$0x3b6d0b79$
$t = 77$	$0x860d21cc$	$0xa079b7d9$	$0xd8fdf6ad$	$0xd7b9da25$	$0xc45e4fdb$
$t = 78$	$0x5738d5e1$	$0x860d21cc$	$0x681e6df6$	$0xd8fdf6ad$	$0xd7b9da25$
$t = 79$	$0x42541b35$	$0x5738d5e1$	$0x21834873$	$0x681e6df6$	$0xd8fdf6ad$

Tablica 12: Bufferi A, B, C, D, E

Nakon ažuriranja buffera slijedi konačno zbrajanje početnih i ažuriranih buffera te konkatenacija rezultata.

$$A = 0x42541b35, \quad A' = 0x67452301$$
$$B = 0x5738d5e1, \quad B' = 0xefcdab89$$
$$C = 0x21834873, \quad C' = 0x98badcfe$$
$$D = 0x681e6df6, \quad D' = 0x10325476$$
$$E = 0xd8fdf6ad, \quad E' = 0xc3d2e1f0$$

$$A = A + A' = 0xa9993e36$$
$$B = B + B' = 0x4706816a$$
$$C = C + C' = 0xba3e2571$$
$$D = D + D' = 0x7850c26c$$
$$E = E + E' = 0x9cd0d89d$$

$$h(m) = A||B||C||D||E = 0xa9993e364706816aba3e25717850c26c9cd0d89d$$

Rezultat se može provjeriti u testnoj tablici:

m	SHA-1(m)
"a"	$0x86f7e437faa5a7fce15d1ddcb9eaeaea377667b8$
"abc"	$0xa9993e364706816aba3e25717850c26c9cd0d89d$
"$abcdbcdecdefdefgefghfghighijhijk$ $ijkljklmklmnlmnomnopnopq$"	$0x84983e441c3bd26ebaae4aa1f95129e5e54670f1$

Tablica 13: Testni primjer [9]

```
if __name__ == '__main__':
    sha1 = SHA1()
    sha1.add(b'abc')
    d = sha1.finish()
    print(codecs.encode(d, "hex"))
```

Programski code 3: Testiranje primjera 5.1 u implementiranom python algoritmu

Kriptografkinja i računalna znanstvenica Wang [3] je 1997. godine predstavila prvi napad na SHA-0, te pokazala da se kolizija može pronaći uz kompleksnost 2^{58}, što je puno bolje od kompleksnosti za brute-force napad koji iznosi 2^{80}. Wang, Lisa Yin i Hongbo Yu su 2005. godine pokazali da se kolizije za SHA-1 mogu pronaći uz kompleksnost od 2^{69}. Ovo je bio prvi napad na svih 80 koraka, s kompleksnošću manjom od 2^{80}.
CWI Amsterdam i Google [10] su 2017. godine objavili da su uspješno izveli prvi napad kolizijom na SHA-1. Naime, pronašli su dvije različite pdf datoteke [11] koje su imale jednake SHA-1 hash vrijednosti ($0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a$). Ukupno je izvršeno $2^{63} = 9\ 223\ 372\ 036\ 854\ 775\ 808$ različitih SHA-1 izračuna. Za ovaj izračun je potrebno 6500 godina jednom CPU i 110 godina jednom GPU. Ovaj napad je 100 000 puta brži od brute-force napada, budući da bi brute-force napad zahtijevao 12 000 000 godina računanja s jednim GPU-om.
Budući da je pronađena kolizija, alternative koje bi zamijenile SHA-1 su bile SHA-256 i SHA-3.

Dodatak

```
#Python 3.7.2
#MD4 hash funkcija
import codecs
import struct

#bitovna operacija rotacije u lijevo
def leftshift(x, n):
    return ((x << n)) | (x >> (32 - n))
#definiranje pomocne funkcije f
def f(x, y, z):
    return (x & y) | (~x & z)
#definirane pomocne funkcije g
def g(x, y, z):
    return (x & y) | (x & z) | (y & z)
#definiranje pomocne funkcije h
def h(x, y, z):
    return x ^ y ^ z

class MD4():
    def __init__(self, data=b''):
        self.remainder = data
        #def. A,B,C,D buffera
        self.buffer = [
                0x67452301,
                0xefcdab89,
                0x98badcfe,
                0x10325476
                ]
        #def. konstanti za runde 2 i 3
        self.c1 = 0x5a827999
        self.c2 = 0x6ed9eba1
        self.counter = 0

    def process_msg(self, msg):
        self.counter += 1

        #lista X, duljine 16, ciji su elementi 32-bitovne rijeci
        X = list( struct.unpack("<16I", msg))
        #pravimo radnu kopiju self.buffera
        buff = [b for b in self.buffer]

        # Runda 1
        shift = (3, 7, 11, 19)
        for j in range(16):
            i = (16-j)%4
            k = j
            buff[i] = leftshift( (buff[i] + f(buff[(i+1)%4], buff[(i+2)%4],
    buff[(i+3)%4]) + X[k]) %2**32, shift[j%4] ) % 2**32

        # Runda 2
        shift = (3, 5, 9, 13)
        for j in range(16):
            i = (16-j)%4
            k = 4*(j%4) + j//4
            buff[i] = leftshift( (buff[i] + g(buff[(i+1)%4], buff[(i+2)%4],
    buff[(i+3)%4]) + X[k] + self.c1) % 2**32, shift[j%4] ) % 2**32
```

```
55
56          # Runda 3
57          shift = (3, 9, 11, 15)
58          k = (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15)
59          for j in range(16):
60              i = (16-j)%4
61              buff[i] = leftshift( (buff[i] + h(buff[(i+1)%4], buff[(i+2)%4],
    buff[(i+3)%4]) + X[k[j]] + self.c2) % 2**32, shift[j%4] ) % 2**32
62
63          #finalno azuriranje buffera
64          self.buffer[0] = (buff[0] + self.buffer[0]) % 2**32
65          self.buffer[1] = (buff[1] + self.buffer[1]) % 2**32
66          self.buffer[2] = (buff[2] + self.buffer[2]) % 2**32
67          self.buffer[3] = (buff[3] + self.buffer[3]) % 2**32
68
69
70
71      def add(self, data):
72          message = self.remainder + data
73          r = len(message) % 64
74          if r != 0:
75              self.remainder = message[-r:]
76          else:
77              self.remainder = b''
78          for chunk in range(0, len(message)-r, 64):
79              self.process_msg( message[chunk:chunk+64] )
80
81
82      def finish(self):
83          s = len(self.remainder) + 64 * self.counter
84          M = b'\x80' + b'\x00' * ((55 - s) % 64) + struct.pack("<Q", s * 8)
85          self.add(M)
86          hash_value = struct.pack("<4I", *self.buffer)
87          return hash_value
```

Programski code 4: MD4 algoritam implementiran u Python programskom jeziku

```
#Python 3.7.2
#MD5 hash funkcija
import codecs
import struct
import math

#bitovna operacija rotacije u lijevo
def leftshift(x, n):
    return ((x<<n) | (x>>(32-n)))
#definiranje pomocne funkcije F
def F(x, y, z):
    return (x & y) | (~x & z)
#definirane pomocne funkcije G
def G(x, y, z):
    return (x & z) | (y & (~z))
#definiranje pomocne funkcije H
def H(x, y, z):
    return x ^ y ^ z
#definiranje pomocne funkcije I
def I(x,y,z):
    return y ^ (x | (~z))

class MD5():
    def __init__(self, data=b''):
        self.remainder = data
        #def. A,B,C,D buffera
        self.buffer = [
                0x67452301,
                0xefcdab89,
                0x98badcfe,
                0x10325476
                ]
        #kreiranje polja T pomocu funkcije sin
        self.T = [int(abs(math.sin(i+1)) * 2**32)  for i in range(64)]
        self.counter = 0

    def process_msg(self, msg):
        self.counter += 1
        round_count = 0;
        #lista X, duljine 16, ciji su elementi 32-bitovne rijeci
        X = list( struct.unpack("<16I", msg))
        #pravimo radnu kopiju self.buffera
        buff = [b for b in self.buffer]

        # Runda 1
        shift = (7,12,17,22)
        for j in range(16):
            i = (16-j)%4
            k = j
            buff[i] = (buff[(i+1)%4] + leftshift( (buff[i] + F(buff[(i+1)%4],
    buff[(i+2)%4], buff[(i+3)%4]) + X[k] + self.T[round_count + j])%2**32 ,
    shift[j%4] )) % 2**32
        round_count = round_count +16;

        # Runda 2
        shift = (5,9,14,20)
        for j in range(16):
            i = (16-j)%4
```

```
57              k = (5*j + 1)%16
58              buff[i] =(buff[(i+1)%4]+leftshift( (buff[i] + G(buff[(i+1)%4],
    buff[(i+2)%4], buff[(i+3)%4]) + X[k] + self.T[round_count + j]) % 2**32,
    shift[j%4] ) )% 2**32
59          round_count = round_count +16;
60
61          # Runda 3
62          shift = (4,11,16,23)
63          for j in range(16):
64              i = (16-j)%4
65              k=(3*j + 5)%16
66              buff[i] = (buff[(i+1)%4]+leftshift( (buff[i] + H(buff[(i+1)%4],
    buff[(i+2)%4], buff[(i+3)%4]) + X[k] + self.T[round_count + j]) % 2**32,
    shift[j%4] )) % 2**32
67          round_count = round_count +16;
68
69          # Runda 4
70          shift = (6,10,15,21)
71          for j in range(16):
72              i = (16-j)%4
73              k = (7*j)%16
74              buff[i] =(buff[(i+1)%4]+ leftshift( (buff[i] + I(buff[(i+1)%4],
    buff[(i+2)%4], buff[(i+3)%4]) + X[k] + self.T[round_count + j]) % 2**32,
    shift[j%4] )) % 2**32
75
76          #finalno azuriranje buffera
77          self.buffer[0] = (buff[0] + self.buffer[0]) % 2**32
78          self.buffer[1] = (buff[1] + self.buffer[1]) % 2**32
79          self.buffer[2] = (buff[2] + self.buffer[2]) % 2**32
80          self.buffer[3] = (buff[3] + self.buffer[3]) % 2**32
81
82
83
84      def add(self, data):
85          message = self.remainder + data
86          r = len(message) % 64
87          if r != 0:
88              self.remainder = message[-r:]
89          else:
90              self.remainder = b''
91          for chunk in range(0, len(message)-r, 64):
92              self.process_msg( message[chunk:chunk+64] )
93
94
95      def finish(self):
96          s = len(self.remainder) + 64 * self.counter
97          M = b'\x80' + b'\x00' * ((55 - s) % 64) + struct.pack("<Q", s * 8)
98          self.add(M)
99          hash_value = struct.pack("<4I", *self.buffer)
100         return hash_value
```

Programski code 5: MD5 algoritam implementiran u Python programskom jeziku

```
#Python 3.7.2
#SHA-1 hash funkcija
import codecs
import struct

def leftshift(x, n):
    return ((x << n) & 0xffffffff) | (x >> (32 - n))

class SHA1():
    def __init__(self, data=b""):
        self.remainder = data
        #def. A,B,C,D,E buffera
        self.buffer = [
                0x67452301,
                0xefcdab89,
                0x98badcfe,
                0x10325476,
                0xc3d2e1f0
                ]
        self.counter = 0

    def process_msg(self, msg):
        self.counter += 1

        K = [0x5a827999,0x6ed9eba1,0x8f1bbcdc,0xca62c1d6]
        w = list( struct.unpack(">16I", msg) + (None,) * (80-16) )

        for i in range(16, 80):
            w[i] = leftshift(w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16], 1)

        #pravimo radnu kopiju self.buffera
        buff = [b for b in self.buffer]

        for i in range(80):
            if i <= 19:
                f = (buff[1] & buff[2]) ^ (~buff[1] & buff[3])
                k = K[0]
            elif i <= 39:
                f = buff[1] ^ buff[2] ^ buff[3]
                k = K[1]
            elif i <= 59:
                f = (buff[1] & buff[2]) ^ (buff[1] & buff[3]) ^ (buff[2] &
    buff[3])
                k = K[2]
            else:
                f = buff[1] ^ buff[2] ^ buff[3]
                k = K[3]

            T = (leftshift(buff[0],5) + f + buff[4] + k + w[i]) % 2**32
            buff[4] = buff[3]
            buff[3] = buff[2]
            buff[2] = leftshift(buff[1], 30)
            buff[1] = buff[0]
            buff[0] = T

        #finalno azuriranje buffera
        self.buffer[0] = (self.buffer[0] + buff[0]) % 2**32
        self.buffer[1] = (self.buffer[1] + buff[1]) % 2**32
```

```
58          self.buffer[2] = (self.buffer[2] + buff[2]) % 2**32
59          self.buffer[3] = (self.buffer[3] + buff[3]) % 2**32
60          self.buffer[4] = (self.buffer[4] + buff[4]) % 2**32
61
62      def add(self, data):
63          message = self.remainder + data
64          r = len(message) % 64
65          if r != 0:
66              self.remainder = message[-r:]
67          else:
68              self.remainder = b""
69          for chunk in range(0, len(message)-r, 64):
70              self.process_msg( message[chunk:chunk+64] )
71
72
73      def finish(self):
74          s = len(self.remainder) + 64 * self.counter
75          M = b'\x80' + b'\x00' * ((55 - s) % 64) + struct.pack(">Q", s * 8)
76          self.add(M)
77          hash_value = struct.pack(">5I", *self.buffer)
78          return hash_value
```

Programski code 6: SHA-1 algoritam implementiran u Python programskom jeziku

Sažetak

U radu smo opisali funkcije koje zbog svojih svojstava zauzimaju vrlo značajno mjesto u kriptografiji, a to su hash funkcije. U prvom dijelu smo definirali jednosmjernu funkciju, budući da je jednosmjernost bitno svojstvo hash funkcija. Naveli smo svojstva kriptografske hash funkcije, kao i odnose među njima. Pomoću rođendanskog paradoksa odnosno rođendanskog napada, objasnili smo potrebu za definiranjem minimalne duljine hash vrijednosti. Nakon toga, detaljno je opisana Merkle-Dåmgard konstrukcija hash funkcije. Naveli smo primjer koji prati sve korake konstrukcije. Nadalje, opisali smo MD4 hash funkciju, definirali njene korake konstrukcije te raspisali algoritam. Kroz primjer je objašnjena MD4 funkcija i njene specifičnosti. Nakon MD4 funkcije, opisali smo njenu nasljednu MD5 funkciju. Na kraju smo opisali SHA-1 funkciju, njenu konstrukciju te smo naveli primjer hashiranja pomoću SHA-1 funkcije. Opisano je trenutno sigurnosno stanje SHA-1 funkcije i izvršavanje prvog napada kolizijom u praksi.

Ključne riječi

Hash funkcije, kriptografske hash funkcije, jednosmjerne funkcije, otpornost na koliziju, rođendanski paradoks, rođendanski napad, Merkle-Dåmgard konstrukcija, MD4, MD5, SHA-1

Summary

In this paper we described functions which takes very important place in cryptography because of their properties, and we call them hash functions. We defined one way function since one-way property is an essential feature of hash functions. We have specified the properties of hash functions as well as relationships between them. With the birthday paradox and the birthday attack we have explained the need to define the minimum length of hash values. After that, the Merkle-Dåmgard construction has been described in detail. We have mentioned an example that follows all constuction steps. Furthemore, we described the MD4 has function, indicated its construction steps, and wrote down algorithm. Through the example, MD4 function and its specifics are explained. After the MD4 function, we described its successor, MD5 function. Finally, we described the SHA-1 function, its construction, and we provided an example of the hashing using the SHA-1 function. The current state of the SHA-1 function and te first collision attack in practice are described.

Key words

Hash functions, cryptographic hash functions, one way functions, collision resistance, birthday paradox, birthday attack, Merkle-Dåmgard construction, MD4, MD5, SHA-1

Literatura

[1] Martin Ekerå, *Differential Cryptanalysis of MD5*, 2009. https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2009/rapporter09/ekera_martin_09008.pdf

[2] Rolf Oppliger, *Contemporary Cryptography*, Artech House, 2005.

[3] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Finding Collisions in the Full SHA-1*, https://www.iacr.org/archive/crypto2005/36210017/36210017.pdf

[4] http://www.winmd5.com/

[5] https://tools.ietf.org/html/rfc1320

[6] https://www.ietf.org/rfc/rfc1321

[7] http://www.asciitable.com/

[8] https://tools.ietf.org/html/rfc3174

[9] https://csrc.nist.gov

[10] https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

[11] https://shattered.io/

Životopis

Rođen sam 21.11.1994. godine u Vukovaru. Osnovnu školu Bobota sam završio u Boboti, nakon čega upisujem Gimnaziju Vukovar, prirodoslovno-matematički smjer. Gimnaziju završavam 2013. godine te tada upisujem Sveučilišni preddiplomski studij Matematike na Odjelu za matematiku u Osijeku. Studij završavam 2016. godine sa završnim radom na temu "Primjena graf algoritama za pronalaženje optimalne rute na mapama", pod mentorstvom izv. prof. dr. sc. Domagoja Matijevića, te stječem akademski naziv prvostupnika matematike. Iste godine upisujem diplomski studij na Odjelu za matematiku, smjer Matematika i računarstvo. Tijekom jeseni 2017. godine odrađujem stručnu praksu u tvrtki Gideon Brothers na poziciji front-end developera. U lipnju 2018. godine se zapošljavam u tvrtki Atos Convergence Creators d.o.o. na poziciji software inženjera. U ožujku 2019. godine prelazim u tvrtku Enea Software d.o.o. na poziciju software inženjera.