KRIPTOGRAFSKE HASH FUNKCIJE

Činčurak, Dorian

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

Kriptografske hash funkcije

Završni rad

Dorian Činčurak

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 26.08.2020.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Dorian Činčurak
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3904, 21.10.2019.
OIB studenta:	64567054786
Mentor:	Izv. prof. dr. sc. Ivica Lukić
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Kriptografske hash funkcije
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	26.08.2020.
Datum potvrde ocjene Odbora:	09.09.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 16.09.2020.

Ime i prezime studenta:	Dorian Činčurak
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3904, 21.10.2019.
Turnitin podudaranje [%]:	22

Ovom izjavom izjavljujem da je rad pod nazivom**: Kriptografske hash funkcije**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

U današnje se vrijeme sve više i više radnji, koje zahtijevaju visoku razinu sigurnosti, tajnosti i integriteta, odvijaju preko internetskog medija. Neke od tih radnji su pohrana lozinki, internet bankarstvo, osiguravanje integriteta podataka, razmjena kriptovaluta i sl. Radi ostvarivanja ovih zahtjeva, nastala je potreba za primjenom kriptografije i njezinih rješenja u svakodnevnom životu. Kao jedno od rješenja, često se u kriptografiji koriste hash funkcije. Hash funkcija se dugo koristi u računalnoj znanosti i matematička je ili druga funkcija koja pretvara određeni niz podataka u niz fiksne veličine, no kako bi zadovoljila zahtjevima kriptografije mora imati i neka druga svojstva koje ćemo detaljnije opisati u radu. Postoje brojne hash funkcije koje se koriste u kriptografiji, no često se zna dogoditi da, zbog napretka tehnologije, neke od tih funkcija postanu nesigurne za korištenje. Iako postoje mnogobrojne hash funkcije, njihov je stalan razvoj nužno potreban. U prvom djelu rada objasnit ćemo što su to kriptografske hash funkcije kao i strukture podataka koje se koriste u njima, te ćemo dati neke primjere tih algoritama. U drugom dijelu rada obrađivati ćemo razne primjene hash funkcija kao što su kriptovalute i digitalni potpisi dokumenata. Na posljetku, primjenom stečenih znanja prikazati će se primjer jednostavne kriptovalute.

1.1. Zadatak završnog rada

Cilj je ovog rada detaljno objasniti pojam hash funkcija i značajke koje bi kriptografska hash funkcija trebala imati. Također, fokusirati ćemo se i na njihovu ulogu u različitim primjenama. Zbog sigurnosnog značaja objasnit ćemo i enkripciju javnim i privatnim ključem, te na kraju rada izraditi kod jednostavne kriptovalute.

2. HASH FUNKCIJE

Hash funkcije za ulaz uzimaju poruku, *string* ili podatak proizvoljne duljine te kao izlaz generiraju izlazni *string* ili podatak kojeg nazivamo hash-code, hash-vrijednost, hash-rezultat, sažetak poruka ili jednostavno hash. Preciznije, takve funkcije *h* nizovima znakova proizvoljne konačne duljine pridružuje nizove znakova fiksne duljine od *n* bita. Za domenu *D* i kodomenu *R* definiramo preslikavanje $h : D \rightarrow R$, gdje je $|D| > |R|$ što znači da s manjim brojem znakova trebamo reprezentirati veći broj znakova. Zbog ovog svojstva neizbježno je dolaženje do kolizija (eng. *collision*) o kojima ćemo kasnije raspraviti u radu.

Definicija 2.1. *Hash funkcija h je funkcija koja zadovoljava iduća dva svojstva (govoreći u najširem smislu):*

1. sažimanje - h ulazu x proizvoljne konačne duljine se pridružuje izlaz h(x) fiksne duljine n.
2. jednostavnost izračuna - lako je za izračunati funkciju h(x) za zadani h i ulaz x.

Osnovna ideja hash-funkcija je da ta vrijednost služi kao kompaktna reprezentativna slika ulazne vrijednosti i da se ne može dobiti pomoću neke druge ulazne vrijednosti [1, str.321-322].

U ovom ćemo poglavlju objasniti koje uvjete funkcija treba zadovoljavati kako bi se smatrala kriptografskom hash funkcijom i vezom među tim uvjetima, odrediti ćemo glavnu podjelu kriptografskih hash funkcija, objasniti hash tablice i pokazivače, te proći kroz neke značajnije algoritme koji se koriste za izračun sažetka.

2.1. Kriptografske hash funkcije

Kriptografske hash funkcije, za razliku od uobičajenih hash funkcija, zahtijevaju dodatne uvjete kako bi se smatrale sigurnima za upotrebu. No, prvo moramo objasniti pojam jednosmjerne funkcije i pojam kolizija prije nego što definiramo te uvjete.

Za funkciju $f : X \rightarrow Y$ kažemo da je jednosmjerna ukoliko ju je lako za izračunati, a njenu je inverznu funkciju f^{-1} teško za izračunati. Zašto nam je ovo važno? Vjerojatno najčešći primjer u praksi za korištenje ovih funkcija je kod pohranjivanja lozinki (šifri). U programskim sustavima, umjesto da se pohranjuje tablica sa lozinkama, za svaku lozinku *p* u tablicu se može pohraniti vrijednost *f(p)*. Ovom metodom lagano je provjeriti ispravnost unesene lozinke, ali jako je teško doći do originalnih šifri gledajući tako pohranjenu tablicu [2, str.16].

Također moramo objasniti pojam kolizija (eng. *collisions*). Kolizija se pojavljuje kada se dva po sebi različita podatka (dokumenti, binarni zapis, mrežni certifikati, itd.) hashiraju u istu vrijednost. U praksi se kolizije nikada ne bi trebale pojavljivati kako bi se ta funkcija smatrala sigurnom. Ako dani algoritam daje istu hash vrijednost za dva različita podatka, tada algoritam nije siguran i može se probiti (*crack*) [3,4].

Sada kada smo objasnili neke osnovne pojmove možemo definirati ostala specifična svojstva koja određuju kriptografske hash funkcije. U nastavku ćemo definirati ta svojstva za hash funkciju h uz ulaze x, x' i izlaze y, y'.

1. jednosmjernost (eng. *preimage resistance*) - za unaprijed određene izlaze računski je nemoguće pronaći ulaz koji prolaskom kroz funkciju daje taj izlaz, tj. pronaći originalni x' takav da vrijedi $h(x') = y$, za neki y kojemu odgovarajući ulaz nije poznat.

2. slaba otpornost na koliziju (jednoznačnost, eng. *second-preimage resistance*) - nemoguće je računski pronaći neki drugi ulaz koji ima isti izlaz kao određeni ulaz, tj. za neki x pronaći drugi originalni x' tako da vrijedi $x \neq x'$ i $h(x) = h(x')$

3. jaka otpornost na koliziju - nemoguće je računski pronaći bilo koja dva različita ulaza x i x' koji nakon prolaska kroz funkciju daju isti izlaz, tj. takvi da vrijedi $h(x) = h(x')$ (važno je napomenuti kako je izbor ulaza slobodan zbog sličnosti sa slabom otpornošću na koliziju)

Poznavajući ova svojstva možemo definirati jednosmjernu hash funkciju i kriptografsku hash funkciju.

Definicija 2.2. Jednosmjerna hash funkcija (eng. *one-way hash function, OWHF*) je hash funkcija h prema definiciji 2.1 sa dodatnim svojstvima jednosmjernosti i slabe otpornosti na koliziju.

Definicija 2.3. Hash funckija otporna na koliziju (eng. *collision resistant hash function, CRHF*) je hash funkcija h prema definiciji 2.1 sa dodatnim svojstvima slabe otpornosti na koliziju i jake otpornosti na koliziju.

Prema definicijama 2.2. i 2.3. možemo definirati kriptografsku hash funkciju.

Definicija 2.4. Kriptografska hash funkcija je hash funkcija h prema definiciji 2.1 ukoliko je jednosmjerna ili otporna na koliziju [1, str.325, 5].

2.2. Veze među svojstvima

U ovom dijelu poglavlja objasnit ćemo povezanost određenih svojstava kriptografske hash funkcije.

Teorem 2.5. Hash funkcija sa svojstvom jake otpornosti na koliziju je ujedno i funkcija slabo otporna na koliziju.

Dokaz: Recimo da je funkcija *h* jako otporna na koliziju, ali ne i slabo otporna na koliziju. U slučaju da *h* nije slabo otporna na koliziju, tada za zadani ulaz *x* možemo pronaći *x′* tako da vrijedi *x=/x′* i *h(x) = h(x′)* što znači da smo dobili par različitih ulaza (*x,x′*) koji se hashiraju i istu vrijednost, što je kontradiktorno svojstvu jake otpornosti na koliziju.

U praksi *CRHF* gotovo svaki put ima svojstvo jednosmjernosti, iako se ne spominje u definiciji.

Teorem 2.6. Svojstvo jake otpornosti na koliziju ne garantira jednosmjernost.

Dokaz: Neka je *g* funkcija s jakom otpornosti na koliziju s *n*-bitnim izlazima, a *h* (*n* +1)-bitna hash funkcija definirana na sljedeći način:

$$h(x) = \begin{cases} 1||\ x, & ako\ je\ x\ duljine\ bita\ n \\ 0||g(x), & inače \end{cases}$$

Operator || predstavlja ulančavanje, tj. konkatenaciju. Funkcija *h* je jako otporna na koliziju, no ona nije jednosmjerna. Kao primjer, funkcija identiteta na ulaze fiksne duljine je u isto vrijeme jako i slabo otporna na koliziju, ali nije jednosmjerna.

Za jednosmjerne funkcije se mogu postaviti i neki dodatni zahtjevi kao što su:

- međusobna nepovezanost (eng. *non-correlation*) ulaznih i izlaznih podataka – između ulaznih i izlaznih bitova ne smije postojati korelacija.
- otpornost na blisku koliziju (eng. *near-collision resistance*) – mora biti teško pronaći bilo koje dvije različite vrijednosti *x* i *x′* za koje vrijedi da se za njih izračunati sažetci *h(x)* i *h(x′)* razlikuju u malo bitova.
- lokalna jednosmjernost (eng. *local one-wayness*) - mora biti teško pronaći bilo koji podniz ulaznog niza, kao i sam ulazni niz, a čak i kad je poznat dio ulaznog niza, mora biti teško pronaći ostatak [1, str.331, 6, 7].

2.3. Pojednostavljena klasifikacija hash funkcija

Hash funkcije najopćenitije možemo podijeliti u dvije skupine, funkcije bez ključa i funkcije sa ključem. Funkcije bez ključa specificiraju jedan ulazni parametar tj. poruku, dok funkcije sa ključem specificiraju dva ulazna parametra, a to su poruka i tajni ključ.

Za stvarnu upotrebu u praksi, hash funkcije je neophodno klasificirati na temelju svojstva koje pružaju i zahtjevu određenih aplikacija koje ih primjenjuju. Od velikog broja daljnjih podjela, fokusirat ćemo se na dva tipa hash funkcija:

1. Kodovi za otkrivanje preinake (eng. *modification detection codes, MDC*) - koriste se u svrhu očuvanja integriteta podataka prema zahtjevima za specifičnu primjenu. *MDC*-ovi su podklasa hash funkcija bez ključa te se oni sami mogu dalje klasificirati u:
   i. Jednosmjerne hash funkcije opisane u definiciji 2.2.
   ii. Hash funkcije otporne na koliziju opisane u definiciji 2.3.
2. Kodovi za autentifikaciju poruke (eng. *message authentication codes, MAC*) - svrha im je osigurati autentičnost i integritet poruke. *MAC*-ovi su podklasa hash funkcija sa ključem jer se sastoje od dva parametra: poruke i tajnog ključa.

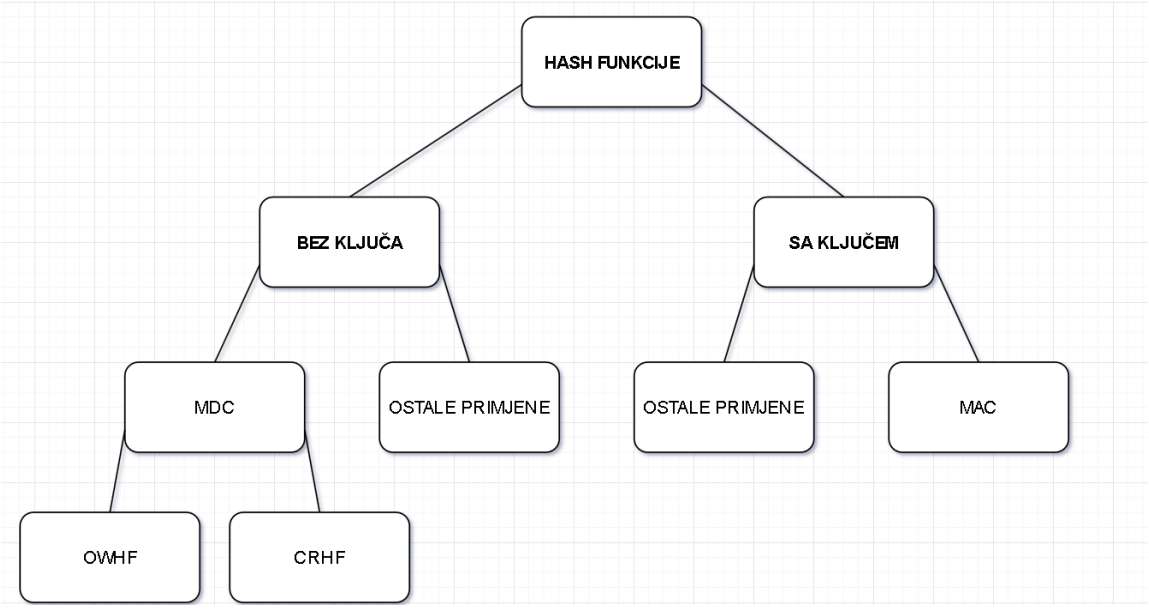
Poznavajući ovo, proširiti ćemo našu definiciju hash funkcija na kodove za autentifikaciju poruke.

Definicija 2.7. *MAC* algoritam je porodica funkcija h_k, sa parametrom koji predstavlja tajni ključ k, sa sljedećim svojstvima:

1. jednostavnost izračuna (eng. *ease of computation*) - Lako je za izračunati vrijednost $h_k(x)$ uz poznatu funkciju h_k sa danom vrijednosti k i ulazom x. Ova se vrijednost naziva *MAC* vrijednost ili jednostavnije *MAC*.
2. kompresija (eng. *compression*) - h_k pridružuje ulazu x proizvoljne konačne duljine izlaz $h_k(x)$ fiksne duljine n.
3. računska otpornost (eng. *computation-resistance*)- daje 0 ili više *MAC* parova $(x_i, h_k(x_i))$, računski nije moguće izračunati bilo koji *MAC* par $(x, h_k(x))$ za bilo koji novi ulaz $x \neq x_i$.

Računska se otpornost treba zadržati, u protivnom, *MAC* algoritam može postati predmet krivotvorine. Računska otpornost implicira svojstvo neizvedivosti otkrivanja ključa, tj. računski je nemoguće otkriti k.

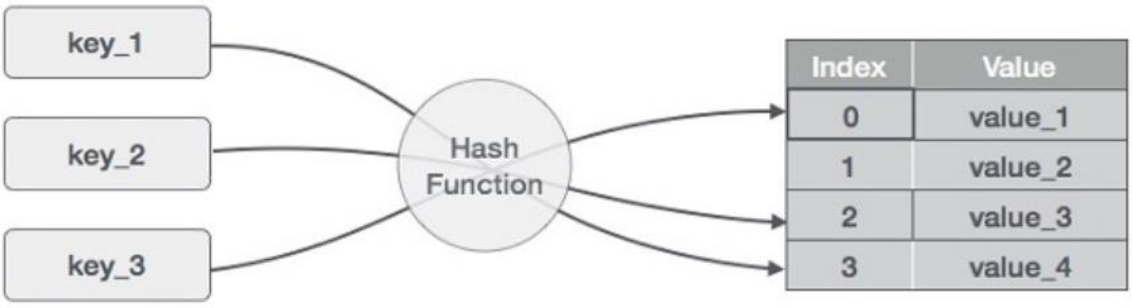
Na slici 2.1. prikazana je pojednostavljena klasifikacija hash funkcija.



***Slika 2.1**. Pojednostavljena klasifikacija hash funkcija [1]*

2.4. Hash tablice

Hash tablica je podatkovna struktura koja rabi hash funkciju za učinkovito preslikavanje određenih ključeva u njima pridružene vrijednosti. U hash tablici, podaci se pohranjuju u podatkovnu strukturu polja, gdje svaka vrijednost podatka ima svoju jedinstvenu vrijednost indeksa. Prednosti ovakve strukture podataka su brzina dodavanja, brisanja i pretrage podataka, bez obzira na veličinu podataka (polja). Princip rada hash tablice dana je slikom 2.2. [8, 9, str.82].



***Slika 2.2.** Princip rada hash tablice [8]*

Kako bi bolje objasnili rad hash tablice, u nastavku ćemo ovog poglavlja dati primjer.

Recimo da nam je zadan zadatak da učenike nekog razreda posložimo u hash tablicu. Radi jednostavnosti, koristiti ćemo se poljem veličine $n = 10$. Za našu hash funkciju h koristit ćemo operaciju modulo od naše veličine polja (*mod n*), kako bi smo izračunali indekse prema danome ključu. Podaci koji se često stavljaju u tablicu su parovi ključa i vrijednosti. Za ključ k koristit

ćemo ime učenika, a kao vrijednost može se koristiti prezime učenika, imena roditelja, uspjeh iz pojedinih predmeta, itd. U praksi se često sve vrijednosti grupiraju u objekt neke klase, tako da vrijednost može biti i cijeli objekt, no trenutno se nećemo fokusirati na vrijednosti. Algoritam će naše hash funkcije izgledati ovako: kako za ulaz (ključ) primamo ime (*string*), prvo ćemo rastaviti ime na slova (*charove*), te svakom slovu dodijeliti njegovu ASCII vrijednost, zatim ćemo te vrijednosti zbrojiti, te odrediti *mod n* od te vrijednosti. Ta vrijednost će predstavljati indeks na kojemu ćemo postaviti naš podatak. Ponavljati ćemo sve dok ne unesemo sve učenike u hash tablicu. U nastavku ćemo primijeniti takvu funkciju za nekoliko učenika:

- Mia, h(Mia) = M + i + a = 77 + 105 + 97 = 279 *mod* 10 = 9
- Ana, h(Ana) = A + n + a = 65 + 110 + 97 = 272 *mod* 10 = 2
- Iva, h(Iva) = I + v + a = 73 + 118 + 97 = 288 *mod* 10 = 8
- Leo, h(Leo) = L + e + o = 76 + 101 + 111 = 288 *mod* 10 = 8
- Rea, h(Rea) = R + e + a = 82 + 101 + 97 = 280 *mod* 10 = 0
- Teo, h(Teo) = T + e + o = 84 + 101 + 111 = 296 *mod* 10 = 6
- Ema, h(Ema) = E + m + a = 69 + 109 + 97 = 275 *mod* 10 = 5
- Lea, h(Lea) = L + e + a = 76 + 101 + 97 = 274 *mod* 10 = 4
- Una, h(Una) = U + n + a = 85 + 110 + 97 = 292 *mod* 10 = 2
- Tin, h(Tin) = T + i + n = 84 + 105 + 110 = 299 *mod* 10 = 9 [10]

Na primjeru Ive i Lea kao dva ulaza različite vrijednosti, vidimo kako se funkcija preslikava u istu vrijednost, odnosno prouzročena je kolizija. Intuitivno znamo kako ne možemo na isti indeks postaviti dva različita podatka bez da narušavamo integritet jednog od njih. Postoji više načina kako rukovati sa nastalim kolizijama (eng. *collision handling*), no u ovom ćemo radu proučit dvoje od njih.

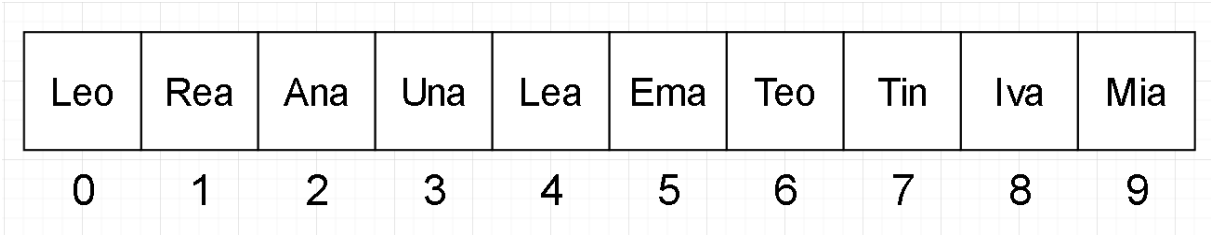
1. *Linearno ispitivanje* (eng. *linear probing*) - u slučaju kolizije tražimo sljedeće slobodno mjesto u polju sve dok ne naiđemo na prazno mjesto u polju. Ovakav se način rukovanja kolizijom naziva otvoreno adresiranje (eng. *open addressing*).

Vrijednost će sa ključem k u daljnjem obrazloženju biti zamijenjeno samo vrijednosti ključa k. Za naš primjer, linearno bi ispitivanje išlo sljedećim redom:

1) Mia se sprema na deseto mjesto u polju.
2) Ana se sprema na treće mjesto u polju.
3) Iva se sprema na deveto mjesto u polju.

4) Leo se ne može spremiti u polje jer je mjesto na tom indeksu već okupirano.
5) Linearno se pretražuje iduće slobodno mjesto. Kako je pretraživanje stiglo do kraja polja i slobodno se mjesto nije uspjelo naći, počinje se pretraživati od početka polja.
6) Leo se sprema na prvo mjesto u polju.
7) Rea se ne može spremiti u polje jer je mjesto na tom indeksu već okupirano.
8) Linearno se pretražuje iduće slobodno mjesto. Rea se sprema na drugo mjesto u polju.
9) Teo se sprema na sedmo mjesto u polju.
10) Ema se sprema na šesto mjesto u polju.
11) Lea se sprema na peto mjesto u polju.
12) Una se ne može spremiti u polje jer je mjesto na tom indeksu već okupirano.
13) Linearno se pretražuje iduće slobodno mjesto. Una se sprema na četvrto mjesto u polju.
14) Tin se ne može spremiti u polje jer je mjesto na tom indeksu već okupirano.
15) Linearno se pretražuje iduće slobodno mjesto. Kako je pretraživanje stiglo do kraja polja i slobodno se mjesto nije uspjelo naći, počinje se pretraživati od početka polja.
16) Linearno se pretražuje iduće slobodno mjesto. Tin se sprema na osmo mjesto u polju.

Nakon našeg algoritma, hash tablica će poprimiti vrijednosti ključeva prema slici 2.3.



Slika 2.3. *Hash tablica nakon linearnog ispitivanja*

Želimo li pretražiti hash tablicu sa slike 2.3. i pronaći vrijednost sa ključem Iva, trebamo implementirati mogućnost pretrage, tako da se za zadani ključ *k* računa njegov hash *h(k),* te se prema toj vrijednosti traži podatak na izračunatom indeksu. Za primjer ključa Iva funkcija *h*(Iva) vraća vrijednost osam i prema tome se dohvaća vrijednost iz hash tablice sa indeksom osam (deveto mjesto). Vidimo kako je brzina ovakvog pretraživanja *O*(1). No, moramo razmotriti i najgori scenarij. Za primjer ključa Tin, pretraživanje kreće od zadnjeg mjesta u polju (*h*(Tin) = 9), te prolazi kroz gotovo cijelo polje kako bi pronašao zadani ključ. U ovom bi slučaju brzina pretraživanja bila približno *O*(*n*). Ova metoda može biti efikasna ako stupanj opterećenja (ukupan broj zauzetih mjesta podijeljen sa ukupnim brojem mjesta *n*, eng. *load factor*) nije prevelik. Tada se pretraživanje podataka sa poznatom vrijednosti ključa događa u konstantnom vremenu *O(1).*

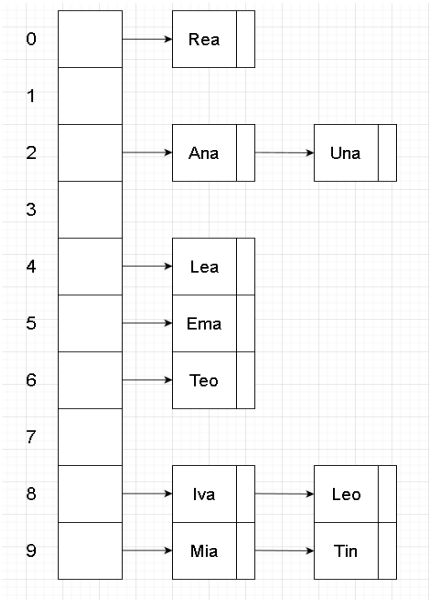
Ostale metode otvorenog adresiranja mogu biti kvadratno ispitivanje, duplo hashiranje, *Hopscotch* hashiranje i dr.

2. *Ulančavanje pomoću povezanih lista* (eng. *chaining using linked lists*) - ključ se uvijek pohranjuje na mjesto u koje je hashirano, a kolizije se rješavaju pomoću zasebne strukture podataka tj. povezane liste. Ovakav se način rukovanja kolizijom naziva zatvoreno adresiranje (eng. *closed addressing*).

U nastavku ćemo proći kroz isti primjer, ovog puta primjenjujući ulančavanje pomoću povezanih lista.

1) Pokazivač povezane liste sa desetog mjesta u polju pokazuje na prvi čvor povezane liste Mia.
2) Pokazivač povezane liste sa trećeg mjesta u polju pokazuje na prvi čvor povezane liste Ana.
3) Pokazivač povezane liste sa devetog mjesta u polju pokazuje na prvi čvor povezane liste Iva.
4) Povezana lista Leo ne može biti prvi čvor u listi na devetom mjestu jer je prvi čvor već zauzet. Traži se iduće slobodno mjesto u listi.
5) Pokazivač povezane liste Iva pokazuje na drugi čvor povezane liste Leo.
6) Pokazivač povezane liste sa prvog mjesta u polju pokazuje na prvi čvor povezane liste Rea.
7) Pokazivač povezane liste sa sedmog mjesta u polju pokazuje na prvi čvor povezane liste Teo.
8) Pokazivač povezane liste sa šestog mjesta u polju pokazuje na prvi čvor povezane liste Ema.
9) Pokazivač povezane liste sa petog mjesta u polju pokazuje na prvi čvor povezane liste Lea.
10) Povezana lista Una ne može biti prvi čvor u listi na četvrtom mjestu jer je prvi čvor već zauzet. Traži se iduće slobodno mjesto u listi.
11) Pokazivač povezane liste Ana pokazuje na drugi čvor povezane liste Una.
12) Povezana lista Tin ne može biti prvi čvor u listi na desetom mjestu jer je prvi čvor već zauzet. Traži se iduće slobodno mjesto u listi.
13) Pokazivač povezane liste Mia pokazuje na drugi čvor povezane liste Tin.

Nakon našeg algoritma, hash tablica će poprimiti vrijednosti ključeva prema slici 2.4.



Slika 2.4 *Hash tablica nakon ulančavanja pomoću povezanih lista*

Želimo li implementirati mogućnost pretrage nad ovakvom hash tablicom, vrijeme će pronalaska u najboljem slučaju biti konstantna vrijednost O(1). Npr. za ključ sa vrijednosti Una, izračunata je hash vrijednost *h*(Una) = 2, te se na trećem mjestu u polju pretražuje vrijednost tog ključa. Pretraživanje se nastavlja povezanom listom sve dok se ne nađe tražena vrijednost. Najgori bi nam slučaj bio kada bi nam se sve vrijednosti ključa hashirale u isti indeks. Ako nam se dogodi takav slučaj, tada bi nam vrijeme pretraživanja trajalo onoliko koliko imamo elemenata u listi, tj. O(n). Generalno su metode zatvorenog indeksiranja brže za pretraživanje, ali moramo uzeti u obzir da tada koristimo više resursa (više memorije) na implementaciju i održavanje dodatnih struktura podataka kao što su povezane liste.

Ostale metode zatvorenog adresiranja su ulančavanje pomoću dinamičkih polja i samoodržavajuća binarno pretraživačka stabla (eng. *self-balancing binary search trees*). Glavne zadaće hash funkcija za hash tablice su minimiziranje kolizija i osiguravanje ujednačene distribucije hash vrijednosti. Funkcija mora biti jednostavna za računanje i potrebna je mogućnost rukovanja bilo kakvim kolizijama. Pošto se hash tablice uglavnom koriste za minimiziranje vremena potrebnog za pronalaženje nekog podatka prema njegovoj hash vrijednosti, onda je za dohvaćanje podatka, tj. za izračun hasha, važno da bude izrazito brz [11].

2.5. Algoritmi za izračun sažetka bez ključa

Algoritmi za izračun sažetka nisu ništa drugo nego hash funkcije. Prema definiciji 2.1., osnovno svojstvo svih efikasnih algoritama za izračunavanje sažetka je zahtjev prema kojemu se, za dva

različita sažetka izračunata istim algoritmom, moraju i ulazni podaci iz kojih su sažeci izračunati, biti različiti. To znači da algoritmi za izračunavanje sažetka moraju biti deterministički. Nasuprot prethodnom svojstvu, drugo svojstvo određuje da algoritmi za izračunavanje sažetka ne moraju biti nužno injektivni, tj. jednakost dva sažetka ne mora garantirati da su ulazni podaci iz kojih su oni nastali također jednaki. Ako se izračuna sažetak za jednu vrijednost i nakon toga se promijeni samo jedan bit u toj vrijednosti, novi bi sažetak trebao biti potpuno različit od prethodnog. Tipičan algoritam za izračunavanje sažetka može prihvatiti bilo koju veličinu ulaznog podatka, a kao izlaz obično daje niz bitova fiksne dužine. [6]

U nastavku ćemo poglavlja opisati neke od danas najčešćih i najvažnijih algoritama.

2.5.1. *MD4*

MD4 je hash funkcija koja za izlaz daje poruku od 128-bita. Naziv *MD* dolazi od "*Message Digest*", a broj 4 u nazivu označava broj serije MD funkcija. Za „slamanje" originalne *MD4* funkcije, tj. za pronalaženje poruka sa istom vrijednosti, potrebno je oko 2^{64} operacija, a za pronalazak poruke koja unaprijed daje određenu vrijednost potrebno je 2^{128} operacija. Zbog porasta računalne snage, *MD4* funkcija više nije sigurna za korištenje iz razloga što se kolizije mogu pronaći u 2^{20} pokušaja računanja funkcije kompresije. Iako nije preporučljiv za korištenje, u ovom ćemo ga radu opisati zbog njegove kriptoanalitičke važnosti i korištenja u svrhu usporedbe i opisivanja drugih funkcija u toj familiji.

Kako bi ispisali sljedeći algoritam, u tablici 2.1. prikazana je notacija kojom ćemo se koristiti za *MD4* porodicu algoritama.

Tablica 2.1. Notacija za *MD4* familiju algoritama

Notacija	Značenje
u, v, w	varijable 32-bitne vrijednosti
0x67452301	heksadecimalni 32-bitni cijeli broj (najmanje značajan *byte*: 01)
+	zbrajanje modulo 2^{32}
$\bar{u}$	bitovni komplement
$u \hookleftarrow s$	rezultat rotacije *u* lijevo za *s* pozicija
uv	Bitovno I (eng. *AND*)
$u \vee v$	Bitovno ILI (eng. *OR*)
$u \oplus v$	Bitovno NILI (eng. *XOR*)

$f(u, v, w)$	$uv \vee \bar{u}w$
$g(u, v, w)$	$uv \vee uw \vee vw$
$h(u, v, w)$	$u \oplus v \oplus w$
$(X_{1,\ldots},X_j) \leftarrow (Y_{1,\ldots}, Y_j)$	simultano dodjeljivanje $(X_i \leftarrow Y_i)$, gdje je $(Y_{1,} \ldots , Y_j)$ procjenjen prije bilo kakvog dodjeljivanja

Algoritam 2.1. *MD4* hash funkcija

ULAZ: niz x proizvoljne duljine $b \geq 0$

IZLAZ: 128-bitna vrijednost od niza x

1. *Određivanje konstanti* - Potrebno je odrediti četiri 32-bitne inicijalne ulančane vrijednosti (eng. *IV's*): h_1=0x67452301, h_2=0xefcdab89, h_3=0x98badcfe, h_4=0x10325476.

Odrediti aditivne 32-bitne konstante:

$y[j] = 0,\ 0 \leq j \leq 15$;

$y[j]$ = 0x5a827999, $16 \leq j \leq 31$; (konstanta = $\sqrt{2}$)

$y[j]$ = 0x6ed9eba1, $32 \leq j \leq 47$; (konstanta = $\sqrt{3}$)

Odrediti red pristupanja izvornim riječima:

$z[0 \ldots 15]$ = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],

$z[16 \ldots 31]$ = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15],

$z[32 \ldots 47]$ = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15].

Odrediti broj pozicija bitova za lijevu rotaciju:

$s[0 \ldots 15]$ = [3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19],

$s[16 \ldots 31]$ = [3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13],

$s[32 \ldots 47]$ = [3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15].

2. *Predprocesiranje* - duljina od x mora biti višekratnik od 512. To ćemo napraviti dopunom x-a na sljedeći način: dodajemo jedan 1-bit, zatim dodajemo $r-1$ (≥ 0) 0−bita za najmanji r, takav da krajnja duljina bude za 64 manja od višekratnika od 512. Na kraju dodajemo 64-bitni rezultat od b *mod* 2^{64}, kao dvije 32-bitne riječi s najmanjom značajnom riječi prvom. Recimo da je m broj 512-bitnih blokova u rezultirajućem *stringu* ($b + r + 64 = 512m = 32 \cdot 16m$). Formatirani se ulaz sastoji od 16m 32-bitnih riječi: $x_0x_1 \ldots x_{16m-1}$. Zatim treba inicijalizirati $(H_1, H_2, H_3, H_4) \leftarrow (h_1, h_2, h_3, h_4)$.

3. *Procesiranje* - potrebno je kopirati svaki *i*-ti blok od 16 32-bitnih riječi u međuspremnik na sljedeći način: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$, za svaki i od 0 do *m*-1. Nakon toga ih obrađujemo kao tri runde od po 16 koraka prije ažuriranja ulančanih varijabli prema sljedećem:

(Određivanje radnih varijabli) $(A, B, C, D) \leftarrow (H_1, H_2, H_3, H_4)$.

(Runda 1) Za svaki *j* od 0 do 15 potrebno je napraviti:

$t \leftarrow (A + f(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j], B, C)$

(Runda 2) Za svaki *j* od 16 do 31 potrebno je napraviti:

$t \leftarrow (A + g(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j], B, C)$

(Runda 3) Za svaki *j* od 32 do 47 potrebno je napraviti:

$t \leftarrow (A + h(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j], B, C)$

(ažuriranje ulančanih vrijednosti) $(H_1, H_2, H_3, H_4) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$.

4. *Završetak algoritma* - konačna vrijednost sažetka je konkatenacija: $H_1 || H_2 || H_3 || H_4$ (s prvim i zadnjim *byte*-om kao *byte*-ovima niskog i visokog reda od H_1 prema H_4).

2.5.2. *MD5*

MD5 hash funkcija je dizajnirana kao poboljšana verzija *MD4*. Smišljena je i prije nego što su pronađene kolizije u prijašnjoj inačici. *MD5* je popularan algoritam i ima široku primjenu u praksi, no poznato je da ima i svoje slabosti. Iako na samoj *MD5* funkciji nisu pronađene kolizijije, one su pronađene na *MD5* funkciji kompresije. *MD5* se često koristi za provjeru integriteta datoteka pomoću sažetka datoteke – prilikom primitka datoteke izračunava se njen *MD5* sažetak i uspoređuje se sa sažetkom koji je dobiven uz datoteku. U slučaju da su identični, datoteka nije promijenjena. Druga najčešća primjena *MD5* algoritma je za enkripciju kod pohranjivanja zaporki, no postoji velik broj *MD5* reverznih baza podataka kojima se tako zaštićene zaporke mogu dekriptirati.

Kako bi od *MD4* dobili *MD5*, napravljene su sljedeće promjene:

1. dodana je četvrta runda od 16 koraka i dodana je funkcija te runde
2. funkcija runde 2 se zamijenila drugom funkcijom
3. u rundi 2 i 3 je izmijenjen red pristupa za riječi
4. izmjenjen je iznos pomaka (kako bi se pomak razlikovao u različitim rundama)

5. koriste se jedinstvene aditivne konstante u svakom od 4 puta po 16 koraka, baziranih na cjelobrojnom dijelu od $2^{32} \cdot \sin(j)$ u koraku *j* (trebajući sveukupno 256 *byte*-ova memorije za skladištenje)
6. dodan je izlaza iz prethodnog koraka u svaki od 64 koraka.

2.5.3. *SHA-1*

SHA-1 (eng. *Secure Hash Algorithm*), baziran na *MD4*, razvijen je od strane Nacionalnog instituta za standrade i tehnologiju (*NIST*) za određene svrhe američke vlade. Za razliku od 128-bitnog hasha, 160-bitna hash vrijednost *SHA-1* algoritma pruža veću sigurnost od probijanja *brute-force* napadom. Iako je *SHA-1* algoritam također dokazano probijen, za ostvarenje njegovog proboja su korištene kompleksne analitičke funkcije. Unatoč tom proboju sigurnosti, upitna je upotreba *SHA-1* algoritma u budućim primjenama.

Glavne razlike *SHA-1* naspram *MD4* algoritma su sljedeće:

1. Duljina hash rezultata je 160-bita, a koristi se pet (u *MD4* ih je četiri) 32-bitnih ulančanih varijabli.
2. Funkcija kompresije sastoji se od četiri runde (u *MD4* ih je tri), a koriste se funkcije iz *MD4* *f*, *g* i *h* i to: *f* u prvoj, *g* u trećoj, a *h* u drugoj i četvrtoj rundi. Svaka runda ima 20 (u *MD4* ih ima 16)
3. Gledajući funkciju kompresije, svaka 16-riječni blok je proširen na 80-riječni blok, procesom u kojem svaka od zadnjih 64 riječi od 80 riječi je *XOR* od 4 riječi iz ranije pozicije u proširenom bloku. Te riječi su tada ulaz jedna-riječ-jedan-korak u 80 koraka.
4. Ključni je korak promijenjen: koristi se jedna rotacija koja je konstantna i iznosi 5-bita. U svaki je korak rezultata dodana i peta radna varijabla. Poruci se iz proširene blok poruke pristupa sekvencijalno, a radna varijabla *C* je nadogradnja od radne varijable *B* rotirane lijevo za 30 bita.
5. *SHA-1* koristi četiri aditivne konstante različite od 0(u MD4 su korištene tri, od kojih je za dvije vrijedilo da su različite od nula)[2, str. 344-349].

2.5.4. Primjeri

U tablici 2.2. dani su primjeri hash vrijednosti za navedene hash funkcije bez ključa.

Tablica 2.2. Primjeri hash-vrijednosti za različite hash funkcije

Hash funkcija	*String*	Hash vrijednost
MD4	""	"31d6cfe0d16ae931b73c59d7e0c089c0"
	"a"	"bde52cb31de33e46245e05fbdbd6fb24"
	"Primjer A"	"6bea7b7ad2c2a3b4c6934fd154c1a501"
	"Primjer B"	"b0df30f0c3cd8300ec6fe09d84ba2f49"
	"FERIT"	"68e4ddf0d802d9324e5df689dbe15cc4"
MD5	""	"d41d8cd98f00b204e9800998ecf8427e"
	"a"	"0cc175b9c0f1b6a831c399e269772661"
	"Primjer A"	"2bea8b264d5522e4ddcba852160e6b40"
	"Primjer B"	"95e413d953f820f84d281f172d700255"
	"FERIT"	"507fae3d098131ff21cc37e61c9580f3"
SHA-1	""	"da39a3ee5e6b4b0d3255bfef95601890afd80709"
	"a"	"86f7e437faa5a7fce15d1ddcb9eaeaea377667b8"
	"Primjer A"	"1092161b2c9fe0dbca8aa5ecfb51fd3a182942a6"
	"Primjer B"	"af4473c1351c9c6e85befe3bffaf15ad09ce91c8"
	"FERIT"	"ac68e7f74488ad738b01fdef710997dfd1c63884"

Iz tablice 2.2. vidimo da i najmanja promjena u nizu znakova (*stringu*) dovodi do velike razlike u hash-vrijednosti osmotrenih funkcija.

Ostali algoritmi koji se koriste su:

- *RIPEMD* (eng. *RACE Integrity Primitives Evaluation Message Digest*) - bazira se na istim principima kao i *MD4*, a po sigurnosnim svojstvima je sličan *SHA-1* algoritmu. Postoje četiri varijante *RIPEMD* algoritma *RIPEMD-128, RIPEMD-160, RIPEMD-256* i *RIPEMD-320,* pri čemu brojke u nazivu odgovaraju dužinama sažetka koje se algoritmima dobivaju.
- *Snefru* algoritam - generira 128 ili 256 bitni sažetak.
- *Tiger* algoritam - predviđen za 64-bitne procesore, generira 128-bitni, 160-bitni ili 192-bitni sažetak.
- *Whirlpool* - generira 512-bitni sažetak, prihvaćen kao *ISO/IEC* 10118-3:2004 standard.
- *HAVAL* - generira 128-bitni, 160-bitni, 192-bitni, 224-bitni i 256-bitni sažetak.

2.6. Algoritmi za izračun sažetka sa ključem

Hash-funkcije s ključem čija je glavna svrha autentikacija poruka, nazivaju se kodovi autentičnosti poruke (*MAC*). Takvi se algoritimi upotrebljavaju za autentikaciju i provjeru integriteta poruka tako što se uz poruku šalje i dodatni podatak koji se naziva *MAC* sažetak koji se dobiva upotrebom *MAC* algoritma i tajnog ključa. Primatelj poruke može uz posjed identičnog tajnog ključa, upotrebom istog algoritma, provjeriti ukoliko *MAC* sažetak odgovara primljenoj poruci, te time može verificirati integritet i autentičnost poruke. *MAC* algoritmi imaju striktne sigurnosne zahtjeve koji specificiraju da takav algoritam mora biti siguran od krivotvorenja, tj. da nitko ne smije biti u stanju generirati važeći *MAC* sažetak za neku poruku *M* bez obzira na sadržaj poruke. *MAC* algoritmi imaju striktne sigurnosne zahtjeve koji specificiraju da takav algoritam mora biti siguran od krivotvorenja, tj. da nitko ne smije biti u stanju generirati važeći *MAC* sažetak za neku poruku *M* bez obzira na sadržaj poruke [6].

2.6.1. *CBC-MAC*

CBC-MAC je tehnika konstruiranja *MAC* algoritama baziranih na blok šiframa. Poruka je šifrirana sa *CBC* (eng. *Cipher Block Chaining*) algoritmom blok šifri u svrhu kreiranja lanca blokova takvih da svaki blok ovisi o valjanoj enkripciji prijašnjih blokova. Algoritam za CBC-MAC je dan u nastavku.

Algoritam 2.2. CBC-MAC algoritam

ULAZ: poruka x, specifikacija blok-šifre E i tajni *MAC* ključ k za E
IZLAZ: n-bitovni *MAC* za x (n je blok-veličine E)

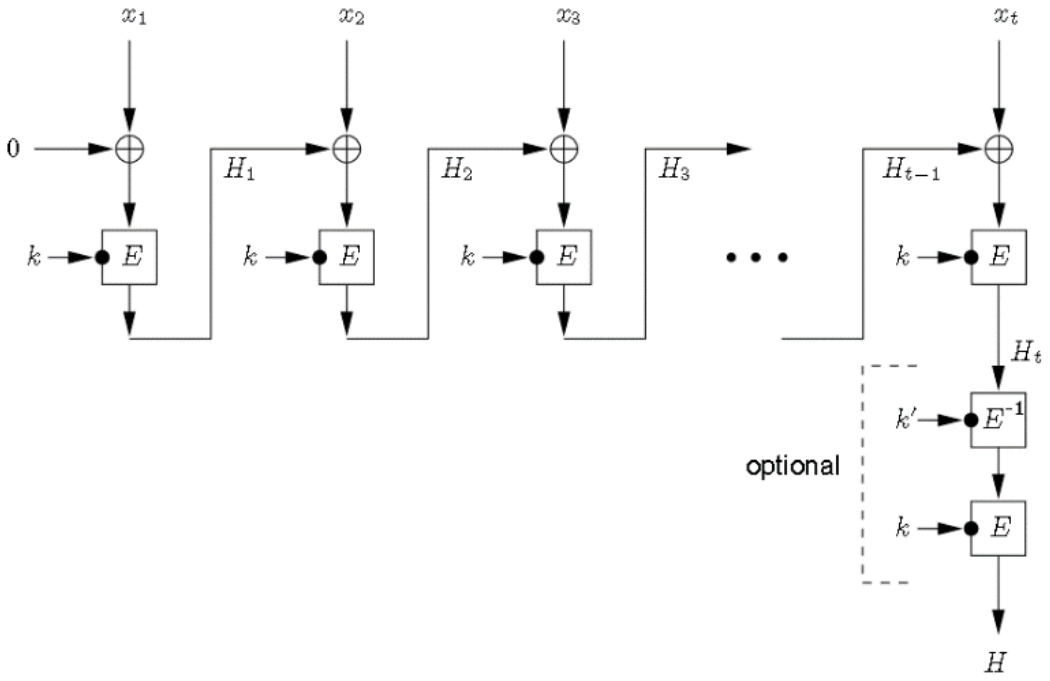
1. *Dopunjavanje duljine i podjela na blokove* - dopuniti x ako je potrebno, a zatim podijeliti x u n-bitne blokove označene s $x_1, \ldots, x_t$.

2. *CBC procesiranje* - Neka E_k označava šifriranje koristeći blok-šifru E s ključem k.
Izračunati blok H_t prema sljedećem: $H_1 \leftarrow E_k(x_1)$; $H_i \leftarrow E_k(H_{i-1} \oplus x_i)$, $2 \leq i \leq t$
(Ovo predstavlja standardno blok-šifriranje, $IV = 0$)

3. *Proizvoljna obrada za povećanje snage MAC-a* - koristeći drugi tajni ključ $k' \neq k$, proizvoljno izračunati: $H'_t \leftarrow E_k^{-1}(H_t)$, $H_t \leftarrow E_k(H'_t)$

4. *Završetak algoritma - MAC* je n-bitni blok H_t.

Na slici 2.5. dana je shema *CBC-MAC* algoritma [1].



Slika 2.5. *Shema CBC-MAC algoritma [1, str. 353]*

3. ENKRIPCIJA JAVNIM KLJUČEM

Enkripcija se javnim ključem (eng. *public-key encryption*) također naziva i asimetrična enkripcija. Kod sustava koji koriste enkripciju javnog ključa svaki entitet *A* ima svoj javni ključ *e* i svoj privatni ključ *d*. U sigurnosnim sustavima računski je nemoguće dobiti *d* pomoću *e*. Javni ključ definira enkripcijsku transformaciju E_e, dok privatni ključ definira pripadajuću dekripcijksu transformaciju D_d. Bilo koji entitet *B* koji želi poslati poruku *m* prema *A* dobiva autentičnu kopiju javnog ključa *e* entiteta *A*, koristi enkripcijsku transformaciju za dobivanje kriptiranog teksta $c = E_e(m)$, te prenosi kriptiranu poruku *c* prema *A*. Za dekripciju teksta c, *A* primjenjuje dekripcijsku transformaciju za dobivanje originalne poruke $m = D_d(c)$.

Javni ključ ne bi trebao biti čuvan u tajnosti, štoviše može biti široko dostupan. Samo je njegova autentičnost potrebna za jamstvo da je *A* uistinu jedina stranka koja zna odgovarajući privatni ključ. Prvenstvena prednost ovakvih sustava je u pružanju jednostavnosti u distribuciji javnih ključeva za razliku od sigurne distribucije tajnih ključeva koji su potrebni u simetričnim sustavima.

Glavni je cilj enkripcije javnim ključem pružanje privatnosti i povjerljivosti. Kako je enkripcijska transformacija entiteta *A* javna, enkripcija javnim ključem ne pruža autentikaciju podrijetla niti integritet podataka. Takvi se zahtjevi mogu pružiti koristeći dodatne tehnike kao što su *MAC* i digitalni potpisi.

Algoritmi enkripcije javnim ključem su tipično dosta sporiji od algoritama simetrične enkripcije. Zbog tog razloga, enkripcija javnim ključem se najčešće u praksi koristi za prijenos ključeva korištenih za enkripciju gomile podataka simetričnim algoritmima. Također se koriste za enkripciju objekata malih podataka kao što su brojevi kreditnih kartica ili PIN-ovi. [1, str. 283]

U nastavku ćemo poglavlja objasniti digitalne potpise dokumenata i kriptovalute te njihovu povezanost sa hash funkcijama.

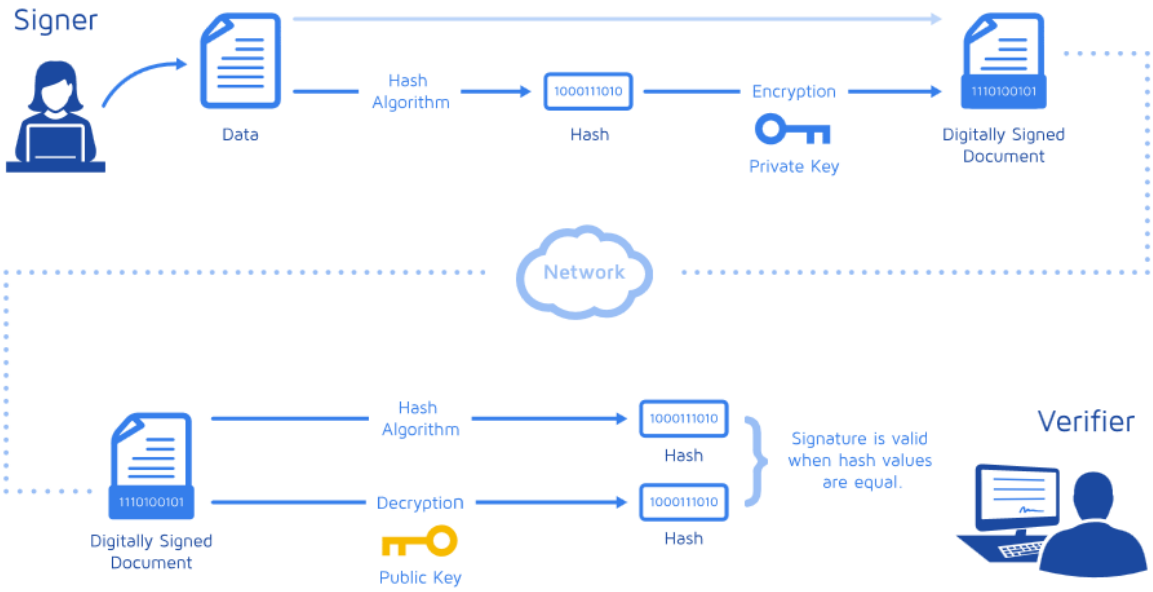
3.1. Digitalni potpisi dokumenata

Digitalni potpis poruke predstavlja broj ovisan o nekoj tajni poznatoj jedino potpisniku poruke i dodatno može ovisiti o sadržaju potpisane poruke. Potpisi moraju biti dokazivani, tj. ako se povede rasprava o tome koja je stranka potpisala dokument, treća nepristrana stranka mora moći riješiti spor bez zahtijevanja tajne informacije (privatnog ključa) potpisnika [1].

Sigurnosni zahtjevi kojima digitalni potpis treba ugoditi su:

- dokaz podrijetla (eng. *Proof of Origin*)- primatelj poruke mora biti siguran u podrijetlo poruke.
- integritet poruke (eng. *Message Integrity*) - primatelj mora biti siguran da poruka nije prijevarno izmijenjena od kako je poslana.
- neporecivost (eng. *Non-Repudiation*) - niti jedna stranka uključena u transakciju ne smije u nekom kasnijem vremenu zanijekati uključenost u tu transakciju [12].

Proces kreiranja digitalnog potpisa prikazan je slikom 3.1.



Slika 3.1. *Primjer uporabe digitalnog potpisa [13]*

Na primjeru ćemo sa slike 3.1. objasniti proces kreiranja i uporabe digitalnog potpisa. Pošiljatelj (potpisnik, eng. *Signer*) hashira svoju poruku nekim od algoritama za izračun sažetka, npr. *MD5*. Zatim tu vrijednost enkriptira svojim privatnim ključem, te tako nastaje digitalni potpis. Potpisana poruka se kroz mrežu šalje do primatelja (verifikatora, eng. *Verifier*). Primatelj tada pošiljateljevim javnim ključem dekriptira digitalni potpis, te dobiva hash vrijednost. Zatim se vrši hashiranje poruke istim algoritmom korištenim od strane pošiljatelja. Nakon toga se uspoređuju vrijednosti digitalnog potpisa i dobivene poruke. Potpis je važeći ako su te vrijednosti identične.

Digitalni potpisi imaju mnogo primjena u informacijskoj sigurnosti uključujući autentikaciju, integritet podataka i neporecivost. Jedna od najvažnijih primjena digitalnih potpisa je ovjera javnih ključeva u velikim mrežnim sustavima.

3.2. Kriptovalute

Prema definiciji, kriptovaluta je digitalna valuta koja koristi enkripcijske tehnike za reguliranje generiranja jedinice valute, te provjerava prijenos novčanih sredstava neovisno o radu centralne banke.

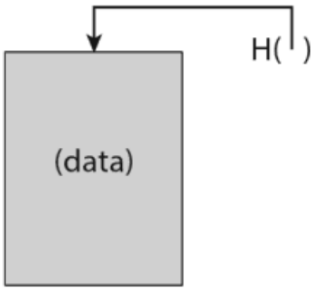
Sve valute trebaju imati neki način kontrole opskrbe i trebaju primjenjivati različita sigurnosna svojstva koja bi sprječavala prevaru. Kod fizičkih valuta, organizacije poput centralnih banaka kontroliraju opskrbu novca i dodaju protu-krivotvorne značajke fizičkim valutama. Takve sigurnosne značajke otežavaju napadačima mogućnost krivotvorine, ali nije u potpunosti nemoguće krivotvoriti takvu valutu. Naposljetku, agencije za provođenje zakona su potrebne kako bi spriječile ljude u kršenju pravila sustava.

Kriptovalute također trebaju imati sigurnosne mjere koje bi spriječile ljude od petljanja u stanje sustava i od okolišanja (eng. *equivocating*, davanje međusobno nedosljednih izjava drugim ljudima). Primjera radi, ako Alice uvjeri Boba da mu je platila digitalnim novcem, tada ne bi smjela moći uvjeriti Carol da je njoj platila istim tim novcem. Za razliku od fizičkih valuta, sigurnosna pravila kriptovaluta moraju se provesti isključivo tehnološki bez oslanjanja na središnju vlast.

Kao što i samo ime govori, kriptovalute se jako oslanjaju na kriptografiju i kriptografske tehnike. U ovom smo radu već spomenuli neke od glavnih temelja za izgradnju kriptovalute, a to su hash funkcije i digitalni potpisi. U nastavku ćemo potpoglavlja objasniti dodatne strukture podataka korištene za izradu kriptovalute, te izabrane algoritme.

3.2.1. Hash pokazivači

Hash pokazivači su pokazivači koji pokazuju na mjesto pohrane nekog podatka zajedno sa kriptografskim hashem tog podatka. Regularni pokazivači nam daju mogućnost povrata podatka, dok nam pokazivači omogućavaju provjeru informacije kako bi znali je li podatak izmijenjen. Prikaz hash pokazivača dan je slikom 3.2.



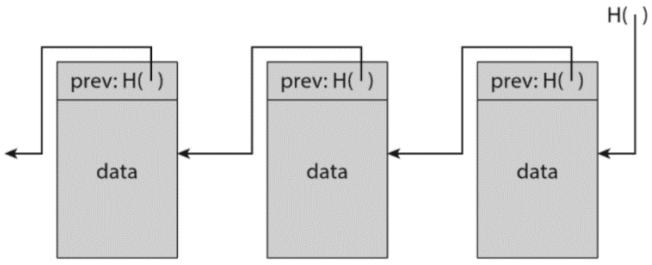
Slika 3.2. *Prikaz hash pokazivača*

Hash pokazivače možemo koristiti za izradu raznih struktura podataka. Intuitivno možemo uzeti slične strukture koje koriste pokazivače, kao npr. povezane liste ili binarna stabla, i implementirati ih sa hash pokazivačima umjesto sa regularnim pokazivačima.

3.2.2. Lanac blokova

Slika 3.3. prikazuje povezane popise koristeći hash pokazivače. Ovakvu strukturu podataka nazivamo lancem blokova (eng. *block-chain*). U regularnoj povezanoj listi, svaki blok sadrži podatke i pokazivač na sljedeći ili prethodni blok (ovisi o implementaciji). Kod lanca će blokova, pokazivač na prethodni blok biti zamijenjen sa hash pokazivačem. Svaki će nam blok tada govoriti gdje se vrijednost prijašnjeg bloka nalazi, te će još dodatno sadržavati sažetak od te vrijednosti, što nam omogućava provjeru kako bi znali je li vrijednost bila promijenjena.

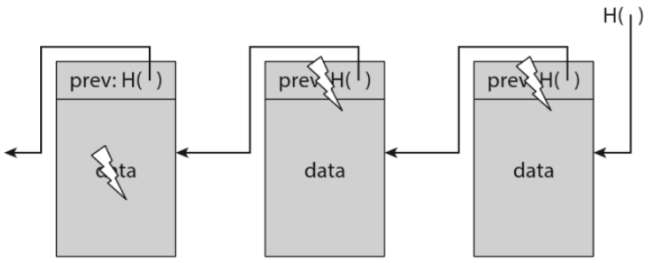
Primjena ovakve strukture podataka je zapisnik sa mogućnošću evidencije zlonamjerne izmjene (eng. *tamper-evident log*) - recimo da želimo izgraditi zapisnik koji sprema podatke i koji omogućava dodavanje podataka na kraj zapisnika, ali ako netko izmjeni podatke koji se pojavljuju prijašnje u zapisniku, tada ćemo opaziti tu izmjenu.



Slika 3.3. *Prikaz blockchaina*

Prikaz izmjene usred *blockchaina* dana je na slici 3.4. Ako se podaci u hash bloku *k* promjene, tada će se promijeniti i hash vrijednost tih podataka. Kako je hash vrijednost promijenjena, hash

vrijednost u bloku *k* + 1 (koji sadrži hash vrijednost bloka *k*) neće odgovarati hash vrijednosti bloka *k*. Tako ćemo uočiti nedosljednost u bloku *k* i hash pokazivaču bloka *k* + 1. Napadač može pokušati prikriti ovu izmjenu tako što će promijeniti hash idućeg bloka. Ova strategija se može koristiti sve dok se ne dođe do glave popisa. Sve dok smo pohranili hash pokazivač na glavu popisa, mjesto koje napadač ne može promijeniti, napadač neće moći promijeniti bilo koji blok bez da ga se ne opazi. Kriptovalute najčešće rabe *block-chain* kao svoju strukturu.



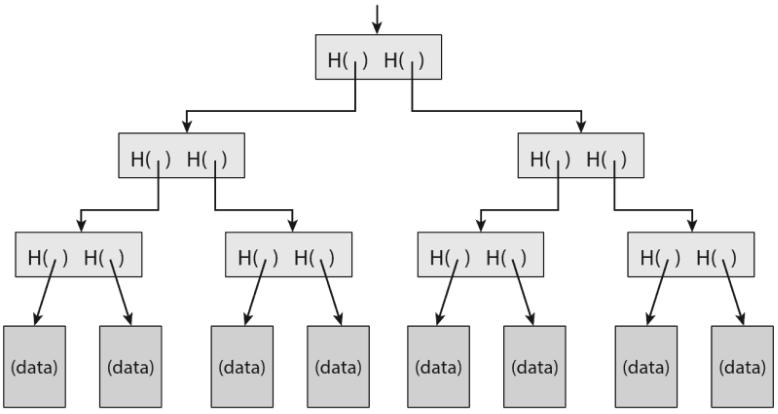
Slika 3.4. *Prikaz izmjene podataka u block-chainu i utjecaj na block-chain*

3.2.3. *Merkle* stabla

Binarna stabla koja koriste hash pokazivače nazivaju se *Merkle* stabla. Recimo da imamo neke blokove podataka. Ovi blokovi nam predstavljaju listove stabla. Grupirajmo te blokove podataka u parove po dva i zatim za svaki par izrađujemo strukturu podataka koja ima dva hash pokazivača, po jedan za svaki od blokova. Ova struktura podataka nam predstavlja sljedeću razinu stabla. Nastavljamo time što te strukture dalje grupiramo po dva i za svaki par izrađujemo strukturu podataka koja sadrži hash vrijednosti svake strukture. Nastavljamo ovim postupkom sve dok ne dođemo do jednog bloka tj. do korijena stabla.

Slično kao i kod *block-chaina*, ako napadač pokuša promijeniti neki blok podataka u stablu, ta promjena će prouzročiti nedosljednost u hash pokazivaču više razine stabla. Čak iako napadač odluči promijeniti blokove više razine stabla, naposlijetku će doći do vrha stabla gdje neće moći izmijeniti hash pokazivač. Time će bilo koji pokušaj izmjene bloka podataka biti uočen [14, str. 10-13].

Prikaz *Merkle* stabla dan je slikom 3.5.



Slika 3.5. *Prikaz Merkle stabla*

3.3. *Blockchain* algoritmi

U ovom dijelu poglavlja usredotočiti ćemo se na najpoznatije algoritme korištene u tehnologiji kriptovalute, a to su: dokaz o radu (eng. *Proof-of-Work, PoW*), dokaz o ulogu (eng. *Proof-of-Stake, PoS*) i dokaz o važnosti (eng. *Proof-of-Importance, PoI*).

Kako bi smo objasnili algoritme koji se koriste u kriptovalutama prvo moramo objasniti termin rudarenja. Naime, rudarenje kriptovaluta (nadalje u tekstu rudarenje) je proces u kojemu se transakcije različitih formi kriptovaluta potvrđuju i dodaju u glavnu knjigu lanca blokova. Svaki put kada se izvrši transakcija kriptovaluta, rudar kriptovaluta je zadužen za osiguravanje autentičnosti informacija i ažuriranje lanca blokova sa transakcijom. [15]

3.3.1. *Proof-of-Work*

PoW je rudarski proces koji zahtjeva od korisnika instalaciju rudarske opreme ili jakih računala kako bi rješavale kompleksne matematičke probleme u svrhu izrade novih virtualnih jedinica kriptovalute. Kompleksni matematički problemi su relativno teški, ali su jednostavni za provjeru od strane mreže. Jednom kada se nekolicina kalkulacija riješi za različite transakcije, provjerene se transakcije u konačnici spremaju u novi blok na javnoj raspodijeljenoj knjizi. Rudar provjerava legalnost transakcije te se stvaraju nove jedinice valute. Prvi rudar koji riješi matematički problem biva nagrađen sa novonastalom jedinicom kriptovalute pruženu od strane protokola koji se koristi. Nagrada tada potiče privlačenje novih rudara u mrežu te se time povećava računalna snaga u mreži. S povećanjem računalne snage u mreži povećava se i prosječan broj kalkulacija potrebnih za stvaranje novih jedinica valute što na posljetku otežava stupanj rudarenja čineći proces težim i skupljim za pojedinog rudara.

PoW pruža platformu gdje se odluke za značajne promjene implementacije *Blockchaina* donose zajednički. Većina glasova je od razvijača, rudara i ostalih bitnih članova zajednice.

Neke loše strane *PoW*-a su gubitak računalne snage i električne energije u generiranju slučajnih pretpostavki te je moguće da se, preplavljivanjem tržišta kriptovalutama, vrijednost kriptovaluta počne obezvrijediti.

Neki od funkcija koji se koriste pri implementaciji *PoW*-a su:

- oslabljeni *Fiat–Shamir* potpis
- parcijalna hash inverzija
- *Mbound*
- *Hokkaido*
- *Cuckoo* ciklus
- funkcije bazirane na *Merkle* stablu

3.3.2. Proof-of-Stake

Imajući na umu da je rudarenje postalo sporije i skuplje tijekom vremena, nastala je *PoS* metoda. *PoS* metoda se kapitalizira na nedostatku valute tj. fokusira se vise na svojstvo korisnika umjesto na rješenje problema. Drugim riječima, *PoS* je sustav koji zahtjeva od korisnika da pokaže vlasništvo određenog broja kriptovaluta u *Blockchaina* kako bi kreirao nove blokove. U ovom sustavu umjesto dobitka jedinica kriptovalute kao nagrade, korisnici najčešće dobivaju naknadu za transakcije.

Kod *PoS*-a, što je više jedinica kriptovalute u novčaniku ili što se više transakcija odvilo u novčaniku to korisnik ima više šanse za dobitak. Kako bi se izbjeglo da jedna osoba ima kontrolu nad valutom, proces je raspodijeljen kroz mrežu. Ovaj proces ovisi o veličini mreže i također o broju ljudi koji zalažu novac. Što više ljudi ulaže novac, to manje nagrada dobivaju i obratno.

Cijeli proces je jeftin zbog malih računa za struje te nema potrebe za skupom računalnom opremom. Proces je također jednostavan jer ne zahtjeva rješavanje složenih jednadžbi.

PoS krši princip kriptovalute koji kaže da ne smije postojati centralna vlast jer omogućava onima koji imaju više novca da diktiraju promjene implementirane na platformi. Onima koji imaju više je garantirano da će više i dobiti, povećavajući razmak između onih koji imaju i onih koji nemaju tj. oni koji nemaju neće imati šansu da dobiju, a oni koji imaju će imati još više.

Neki od *Blockchainova* koji koriste PoS su:

- *Ethereum* (*Casper* ažuriranje)
- *Peercoin*
- *Nxt*

3.3.3. Proof-of-Importance

PoI je algoritam koji razmatra cjelokupnu aktivnost korisnika u mreži. Što je korisnik aktivniji to on dobiva više nagrada. Svakom korisniku se upisuju postignuća i što je veće postignuće korisnika to on dobiva više. Ovaj algoritam je osmišljen tako da nagrađuje vjerne korisnike *Blockchaina* i tako potiče veću uporabu platforme. Algoritam se primarno oslanja na aktivnosti praćenih od strane svakog korisnika, a ne toliko na prebačenu količinu unutar transakcije.

Na ovakvoj platformi oni koji imaju više neće dobivati više jer količina novca nije jedini faktor koji se uzima u obzir kod mjerenja reputacije korisnika.

Glavni problem ovakve metode je korištenje lažnih transakcija koje nagrađuju korisnike koji šalju transakcije naprijed-nazad kako bi prevarili algoritam. [16]

4. PRAKTIČNI PRIMJER JEDNOSTAVNE KRIPTOVALUTE U C++

U ovom ćemo dijelu poglavlja primijeniti stečena znanja iz prijašnjih poglavlja te konstruirati jednostavnu kriptovalutu u C++ programskome jeziku. Ispis koda dan je u prilogu.

Prilikom konstruiranja jednostavne kriptovalute, potrebno je definirati i implementirati transakcije koje se odvijaju na mreži te ih pohraniti u blokove. Takvi će se blokovi na kraju spajati na lanac blokova koji će biti vidljiv cijeloj mreži. Također, potrebno je implementirati *Blockchain* funkciju kako bi rudari mogli ocijeniti valjanost transakcije i za to biti nagrađeni.

Kako bi se mogli služiti hash algoritmima bez da sami pišemo te algoritme od nule, koristit ćemo se *CryptoPP* bibliotekom [17].

Za definiranje transakcije prvo moramo odrediti koji će nam podaci biti potrebni kako bi mogli tu transakciju jednoznačno odrediti. Definirat ćemo strukturu koja sadrži ključeve pošiljatelja i primatelja, vrijeme odrađene transakcije te količinu prebačene jedinice valute. [Prilog, 12-18].

Potrebno nam je zatim definirati klasu *blok* koja će sadržavati podatke transakcije. Dodatno, blok će također sadržavati svoju hash vrijednost i hash vrijednost prijašnjeg bloka. Za dodavanje bloka na lanac blokova, svaki blok mora sadržavati indeks koji predstavlja poziciju u tom lancu. Kako bi se hash vrijednost mogla izračunati, potrebno je implementirati funkciju koja će tu vrijednost generirati. Radi mogućnosti rudarenja implementirati ćemo i jednostavni *Proof-of-Work* algoritam. Za potrebe tog algoritma klasa će sadržavati cjelobrojnu varijablu *nonce* koja predstavlja broj pokušaja izvedbe procesa rudarenja. [Prilog, 19-26].

Generiranje hash vrijednosti će se provoditi na sljedeći način:

1) *String S1* će predstavljati riječ koja je rezultat povezivanja podataka transakcije () bloka pretvorenih u *stringove*. Drugim riječima, *S1* je jednak zbroju *stringova* količine prijenosa jedinica valute, ključevima pošiljatelja i primatelja te vremenu sklopljene transakcije.
2) *String* S2 će biti jednak hashu prijašnjeg bloka.
3) Rezultirajući *string S3* će biti rezultat zbrajanja dvaju *stringova S1* i *S2*.
4) Pomoću *CryptoPP* biblioteke, *string* S3 provodimo kroz neku kriptografsku hash funkciju po izboru. U našem slučaju koristiti ćemo se SHA1 algoritmom.
5) String *digest* nam predstavlja generiranu hash vrijednost datog bloka dobivenog iz prethodnog koraka [Prilog, 67-79]

Radi provjere ispravnosti hasha, uvest ćemo još jednu funkciju. Funkcija uspoređuje hash vrijednost bloka i novu izračunatu vrijednost hasha sa trenutnim podacima bloka. Ako je tijekom vremena napadač promijenio neke od podataka u bloku, tada će funkcija vratiti vrijednost *FALSE* jer se hash vrijednosti neće podudarati [Prilog 89-93].

Trenutno imamo sve što je potrebno kako bi generirali neki blok. Sada je te generirane blokove potrebno spojiti u lanac blokova. Kreiramo novu klasu *Blockchain* koja će sadržavati javnu listu generiranih blokova. Za tu listu je potrebno implementirati funkciju dodavanja blokova u listu [Prilog 95-109].

Radi potrebe rudarenja, klasa će sadržavati privatnu cjelobrojnu varijablu *difficulty* koja predstavlja težinu *Proof-of-Work* zadatka [Prilog, 98].

Pri kreiranju lanca blokova valja izraditi iskonski blok koji nema pokazivač na prethodni blok. Blok se može kreirati sa proizvoljnim parametrima [Prilog, 116-125].

Trenutno je naš lanac spreman za dodavanje novih blokova na kraj lanca (liste). Ovakvo dodavanje blokova nam ne osigurava nikakvu verifikaciju transakcije pa ćemo implementirati jednostavni *Proof-of-Work* matematički zadatak. Pri generiranju hash vrijednosti bloka, moramo u algoritam stringu *S1* dodati i cjelobrojnu varijablu *nonce* koja predstavlja broj pokušaja rješavanja našeg *Proof-of-Work* zadatka.

Naš će *Proof-of-Work* matematički zadatak glasiti ovako:

1) U *string zeros* dodati onoliko nula koliko to nalaže težina zadatka (cjelobrojna varijabla koja se nalazi u klasi *Blockchain*).
2) Izračunati hash vrijednost datog bloka i usporediti dio tog hasha sa *stringom zeros* u ovisnosti o veličini *stringa zeros*, tj. usporediti prvih nekoliko znakova u oba *stringa*.
3) Ako je rezultat u koraku 2) netočan (daje *FALSE*), tada treba cjelobrojnu varijablu *nonce* povećati za 1. Ponoviti korak 2).
4) Ako je rezultat u koraku 2) točan (daje TRUE), tada se smatra da je zadatak uspješno riješen. Blok se dodaje na lanac blokova [Prilog, 39-57].

Kako bi osigurali sigurnost i integritet lanca blokova, implementiramo funkciju koja će vratiti FALSE u slučaju da je netko pokušao izmijeniti ijedan od blokova. Funkcija će ponovno izračunavati hash za svaki pojedini blok i uspoređivati sa onom hash vrijednosti koju je imao kada

je uveden u lanac. Također će se hash vrijednosti svakog bloka uspoređivati sa prijašnjom hash vrijednosti bloka koji je ispred njega u lancu [Prilog, 138-158].

Radi primjera ćemo dodati funkciju koja će nam dohvaćati adresu zadnjeg bloka u lancu blokova kako bi mogli izmijeniti zadnji blok i uočiti nepravilnosti u lancu. U praksi se ova funkcija ne bi trebala primjenjivati jer se time narušava sigurnost *Blockchaina* [Prilog, 127-130].

Naš je lanac spreman za korištenje. U glavnom dijelu programa kreiramo *Blockchain* kojega ćemo nazvati *Hardkooin*. Na taj lanac blokova dodajemo nekolicinu blokova sa proizvoljnim podacima. Program prije svakog dodavanja bloka ispisuje pokušaje rudarenja te naposlijetku, kada je rudarenje uspješno izvršeno, prikazuje se poruka o uspješnosti. Blok se tada dodaje na lanac blokova. Radi lakše vizualizacije, pri izvršavanju programa nakon svih dodavanja, ispisuju se podaci za svaki blok u *Blockchainu*. Uvedemo li malicioznog korisnika koji će izmijeniti zadnji blok u *Blockchainu*, tada će funkcija koja provjerava ispravnost *Blockchaina* vratiti *FALSE* [Prilog, 159-209].

Ovim kodom je uspješno kreirana jednostavna kriptovaluta sa *Proof-of-Work* algoritmom.

5. ZAKLJUČAK

S porastom zahtjeva za sigurnošću i integritetom podataka prenesenih putem interneta, sve je značajnija uloga kriptografskih hash funkcija u sigurnosnim sustavima. Takve funkcije imaju široku primjenu u matematici i računarstvu. Pomoću njih se vrši šifriranje podataka, tj. funkcija kao ulaz uzima podatak proizvoljne duljine i kao izlaz stvara vrijednost fiksne duljine. U radu su objašnjeni osnovni pojmovi vezani za kriptografske hash funkcije kao i strukture podataka koje se koriste u tim funkcijama. Detaljno su objašnjeni i neki od najčešćih algoritama za izračun sažetka bez ključa. Objašnjena je enkripcija javnim i privatnim te je velika pozornost dana primjeni hash funkcija u potpisima dokumenata, a pogotovo kriptovalutama.

U završnom djelu rada objašnjen je postupak izrade jednostavne kriptovalute, odnosno jednostavnog *blockchaina*, čiji je kod ispisan u programskom jeziku C++. Kako bi se povisila razina sigurnosti kriptovalute iz primjera potrebno je izmijeniti složenost Proof-of-Work algoritma. Kriptovaluta se također može poboljšati implementirajući novčanike za svakog korisnika. Potrebno je tada rudarima osigurati nagrade za uspješno riješene matematičke probleme. Kako bi ova kriptovaluta bila funkcionalna i imala svoju vrijednost, potrebno ju je pustiti u opticaj na nekoj od platformi kriptovauluta kao npr. na Etherium platformi.

Iako se hash funckije imaju široku primjenu, razvojem računala i računalne snage rast će broj napada na te funckije kao i smanjenje vremena proboja takvih funkcija. Kako bi hash funkcije bile korak ispred napadača, potrebno ih je konstantno unaprjeđivati i prilagođavati za razne primjene.

LITERATURA

[1] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, „Handbook of Applied Cryptography“, R. L. Rivest, Massachusetts Institute of Technology, 1996.

[2] S. Goldwasser, M. Bellare, „Lecture Notes on Cryptography“, Cambridge, Massachusetts, 2008.

[3] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, C. Baisse, „Announcing the first SHA1 collision“, Google Security Blog, 2017. dostupno na: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html [8. 8. 2020.]

[4] Infosec, „Cryptography Fundamentals, Part 3 – Hashing“, dostupno na: https://resources.infosecinstitute.com/cryptography-fundamentals-part-3-hashing/#gref [8. 8. 2020.]

[5] E. Martin, „Differential Cryptanalysis of MD5“, KTH Computer Science and Communication, Stockholm, Švedska, 2009.

[6] Hrvatska akademska i istraživačka mreža CARNet, „Algoritmi za izračunavanje sažetka“, 2006., dostupno na: https://www.cert.hr/wp-content/uploads/2006/08/CCERT-PUBDOC-2006-08-166.pdf [8. 8. 2020.]

[7] S. Devadas, „Design and Analysis of Algorithms“, MIT OpenCourseWare, 2016. dostupno na: https://www.youtube.com/watch?v=KqqOXndnvic [8. 8. 2020.]

[8] tutorialspoint, „Data Structure and Algorithms - Hash Table“, dostupno na: https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm [8. 8. 2020.]

[9] K. Mehlhorn, P. Sanders, „Algorithms and Data Structures: The Basic Toolbox“, Springer, Karlsruhe, Saarbrücken, 2008.

[10] ASCII tablica, dostupno na: http://www.asciitable.com/

[11] programming[guide], „Hash Tables, Open vs Closed Addressing“, dostupno na https://programming.guide/hash-tables-open-vs-closed-addressing.html [8. 8. 2020.]

[12] M. J. Ganley, „Computers & Security“, Elsevier Science Ltd, Hook, Hamsphire, 1994.

[13] DocuSign, „Understanding Digital Signatures“, dostupno na: https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq [8. 8. 2020.]

[14] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder, „Bitcoin and Cryptocurrency Technologies: A Comperhensive Introduction“, Princeton University Press, Princeton, New Jersey, 2016.

[15] Binance Academy, „What is Cryptocurrency Mining?“, dostupno na: https://academy.binance.com/blockchain/what-is-cryptocurrency-mining [8. 8. 2020.]

[16] CryptoDigest, „Blockchain Basics – PoW vs. PoS vs. PoI“, dostupno na: https://cryptodigestnews.com/blockchain-basics-pow-vs-pos-vs-poi-10a9b7c67d51 [8. 8. 2020.]

[17] CryptoPP, dostupno na: https://www.cryptopp.com/ [8. 8. 2020.]

SAŽETAK

Završni je rad fokusiran na kriptografske hash funkcije. U svrhu shvaćanja ovog pojma, u radu su objašnjena osnovna svojstva hash funkcija i njihova klasifikacija. Dodatno, opisane su najčešće strukture podataka koje se koriste u hash funckijama kao i neki od najčešćih algoritama za izračun sažetka. Velika je pozornost dana primjeni kriptografskih hash funkcija od kojih su najznačajniji digitalni potpisi dokumenata i kriptovalute. Detaljno je objašnjen pojam lanca blokova kako bi mogli kreirati jednostavnu kriptovalutu. Za izradu jednostavne kriptovalute potrebno je definirati transakcije koje se odvijaju na našoj mreži, te odrediti koje će parametre sadržavati. Neki su od tih parametara: javni ključevi pošiljatelja i primatelja, količinu predane jedinice kriptovalute i vrijeme ugovorene transakcije. Transakcije zatim moramo pohraniti u blokove koji se prije dodavanja u lanac blokova moraju potvrditi procesom rudarenja. Svakom se bloku mora moći izračunati hash vrijednost pomoću izabranog hashing algoritma. Također, svaki blok mora sadržavati hash vrijednost prethodnog bloka u lancu blokova radi očuvanja sigurnosti i integriteta lanca. U radu je ovaj proces detaljnije objašnjen. Ispis je koda u programskom jeziku *C++* dan u prilogu.

Ključne riječi: digitalni potpisi dokumenata, integritet podataka, kriptografske hash funkcije, kriptovalute, sigurnost

ABSTRACT

Cryptographic hash functions

This undergraduate thesis is focused on cryptographic hash functions. In order to understand this term, basic properties and their classification are explained in this thesis. Additionally, the most commonly used data structures, which are used in hash functions, including some of the most used message digest algorithms are described. The application of cryptographic hash functions, some of which are digital document signatures and cryptocurrencies, are highly emphasized in this thesis. The term blockchain is explained in detail in order to create a simple cryptocurrency. In the interest of creating a simple cryptocurrency, it is necessary to define transactions that take place on our network and their parameters. Some of those parameters include senders and receivers public keys, the amount of delivered cryptocurrency unit, and the timestamp of contracted the transaction. Furthermore, transactions are stored in blocks, which must be verified in the mining process before inclusion in blockchain. There has to be a possibility to calculate a hash value for each block using the chosen hash algorithm. Moreover, each block must contain the hash value of the previous block in blockchain in order to preserve the security and integrity of a blockchain. This process is further explained in this thesis. Source code for this cryptocurrency is attached in this document.

Keywords: cryptocurrencies**,** cryptographic hash functions**,** data integrity**,** digital document signatures, security

PRILOG

ISPIS KODA KORIŠTENOG U POGLAVLJU 4.

```
#include "stdafx.h"
#include<string>
#include <iostream>
#include <ctime>
#include <vector>
#include<cryptlib.h>
#include<hex.h>
#include<sha.h>
#include<files.h>

using namespace std;
using namespace CryptoPP;

//Podaci transakcije
struct TransactionData {
double ammount;
string senderKey;
string receiverKey;
time_t timestamp;
};

//Klasa Blok
class Block {
private:
int index;
string blockHash;
string previousHash;
string generateHash();
int nonce = 0;

public:
//Konstruktor
Block(int passedIndex, TransactionData d, string prevHash);

//Dohvati originalni hash
string getHash();

//Dohvati prijašnji hash
string getPreviousHash();

//Transakcija
TransactionData data;

//Provjeri ispravnost hasha
```

```
bool isHashValid();

//Rudarenje
void mineBlock(int difficulty)
{
string zero = "";
for (int i = 0; i < difficulty; i++)
{
zero.append("0");
}
while (this->blockHash.substr(0, difficulty) != zero)
{
this->nonce++;
this->blockHash = this->generateHash();
cout << "Nonce: " << nonce << endl;
cout << "Looking for: " << zero << endl;
cout << "Current Hash: " << this->getHash() << endl;
cout << endl;
}
cout << "Block mined: " + this->getHash() << endl;
}
};

//Konstruktor sa parametrima
Block::Block(int passedIndex, TransactionData d, string prevHash)
{
index = passedIndex;
data = d;
previousHash = prevHash;
blockHash = generateHash();
}

//Privatne funckije
string Block::generateHash()
{
string hash1;
string hash2;
string hash3;
string digest;
SHA1 finalHash;

hash1 = to_string(data.ammount) + data.receiverKey +
   data.senderKey + to_string(data.timestamp + this->nonce);

hash2 = this->getPreviousHash();

hash3 = hash1 + hash2;
```

```
77. StringSource s(hash3, true, new HashFilter(finalHash, new
    HexEncoder(new StringSink(digest))));

78. return digest;
79. }

80. //Javne funkcije
81. string Block::getHash()
82. {
83. return blockHash;
84. }

85. string Block::getPreviousHash()
86. {
87. return previousHash;
88. }

89. bool Block::isHashValid()
90. {
91. //Ako su podaci transakcije izmijenjeni, tada funkcija vraća
    FALSE
92. return generateHash() == this->getHash();
93. }

94. //Klasa lanca blokova
95. class Blockchain {
96. private:
97. Block createGenesisBlock();
98. int difficulty = 1;

99. public:
100.     //Javni lanac
101.     vector<Block> chain;

102.     //Konstruktor
103.     Blockchain();

104.     //Javne funkcije
105.     void addBlock(TransactionData data);
106.     bool isChainValid();

107.     //Ne bi se trebalo implementirati, uzeto samo radi primjera
108.     Block *getLatestBlock();
109.     };

110.     //Kosntruktor lanca blokova
111.     Blockchain::Blockchain()
112.     {
113.     Block genesis = createGenesisBlock();
```

```
114.     chain.push_back(genesis);
115.     }

116.     Block Blockchain::createGenesisBlock()
117.     {
118.     TransactionData d;
119.     d.ammount = 0;
120.     d.receiverKey = "None";
121.     d.senderKey = "None";
122.     d.timestamp = time(0);

123.     Block genesis(0, d, "None");

124.     return genesis;
125.     }

126.     //implementirano samo radi primjera
127.     Block *Blockchain::getLatestBlock()
128.     {
129.     return &chain.back();
130.     }

131.     void Blockchain::addBlock(TransactionData d)
132.     {

133.     int index = (int)chain.size() - 1;



134.     Block newBlock(index, d, getLatestBlock()->getHash());

135.     newBlock.mineBlock(this->difficulty);
136.     chain.push_back(newBlock);
137.     }

138.     bool Blockchain::isChainValid()
139.     {
140.     int chainLen = (int)chain.size();
141.     for (int i = 1; i < chainLen; i++)
142.     {
143.     Block currentBlock = chain[i];
144.     if (!currentBlock.isHashValid())
145.     {
146.     return false;
147.     }
148.     if (chainLen > 1)
149.     {
150.     Block previousBlock = chain[i-1];
```

```
151.     if (currentBlock.getPreviousHash() !=
   previousBlock.getHash())
152.     {
153.     return false;
154.     }
155.     }
156.     }
157.     return true;
158.     }

159.     int main()
160.     {
161.     //Inicijalizacija lanca blokova
162.     Blockchain Hardkooin;

163.     //Podaci za prvi dodani blok
164.     TransactionData data1;
165.     time_t data1Time;
166.     data1.ammount = 1.5;
167.     data1.receiverKey = "Joe";
168.     data1.senderKey = "Sally";
169.     data1.timestamp = time(&data1Time);
170.     Hardkooin.addBlock(data1);

171.     //Podaci za drugi dodani blok
172.     TransactionData data2;
173.     time_t data2Time;
174.     data2.ammount = 10;
175.     data2.receiverKey = "Alice";
176.     data2.senderKey = "Bob";
177.     data2.timestamp = time(&data2Time);
178.     Hardkooin.addBlock(data2);

179.     for (int i = 0; i < Hardkooin.chain.size(); i++)
180.     {
181.     cout << "Amount: " << Hardkooin.chain[i].data.ammount <<
   endl;
182.     cout << "Sender: " << Hardkooin.chain[i].data.senderKey <<
   endl;
183.     cout << "Receiver: " << Hardkooin.chain[i].data.receiverKey
   << endl;
184.     cout << "Timestamp: " << Hardkooin.chain[i].data.timestamp
   << endl;
185.     cout << "Block Hash: " << Hardkooin.chain[i].getHash() <<
   endl;
186.     cout << "Previous Hash: " <<
   Hardkooin.chain[i].getPreviousHash() << endl;
187.     cout << "Is hash valid?" << Hardkooin.chain[i].isHashValid()
   << endl;
```

```
188.      cout << endl;
189.      }

190.      cout << "Lenght: " << Hardkooin.chain.size() << endl;

191.      //Maliciozni korisnik
192.      Block *hackBlock = Hardkooin.getLatestBlock();
193.      hackBlock->data.ammount = 100;
194.      hackBlock->data.receiverKey = "John";

195.      for (int i = 0; i < Hardkooin.chain.size(); i++)
196.      {
197.      cout << "Amount: " << Hardkooin.chain[i].data.ammount <<
   endl;
198.      cout << "Sender: " << Hardkooin.chain[i].data.senderKey <<
   endl;
199.      cout << "Receiver: " << Hardkooin.chain[i].data.receiverKey
   << endl;
200.      cout << "Timestamp: " << Hardkooin.chain[i].data.timestamp
   << endl;
201.      cout << "Block Hash: " << Hardkooin.chain[i].getHash() <<
   endl;
202.      cout << "Previous Hash: " <<
   Hardkooin.chain[i].getPreviousHash() << endl;
203.      cout << Hardkooin.chain[i].isHashValid() << endl;
204.      cout << "Is chain valid?" << endl;
205.      cout << Hardkooin.isChainValid() << endl;
206.      cout << endl;
207.      }

208.      cout << "Lenght: " << Hardkooin.chain.size() << endl;
209.      }
```