Elektrotehnički fakultet Univerziteta u Beogradu
Katedra za računarsku tehniku i informatiku

Diplomski rad

Analiza i implementacija kriptografskih heš funkcija

Mentor:
Dr Milo Tomašević

Student:
VladimirJelić

Beograd, 2010.

Sadržaj

1.Uvod

Nagli razvoj informacionih tehnologija narocito na polju računarskih mreža, a time i sve veće primene interneta, omogućio je sve većem broju organizacija i pojedinaca da unaprede svoje poslovanje. Primena računara u svakodnevnom poslovanju je sve veća, a u prilog tome govori i činjenica da je u svetu registrovan sve veći broj korisnika interneta, na koji je takođe postavljen veliki broj Web sajtova. Ovakva ekspanzija primene računara u savremenom poslovanju uslovila je razvoj sve većeg broja specijalizovanih sistema za potrebe elektronskog poslovanja (e-bussines), iz koga su izvedeni sistemi u oblastima elektronske trgovine (e-commerce), elektronskog bankarstva (e-banking), elektronskog poslovanja u javnoj upravi (e-government) i dr. Banke su razvile sopstvene sisteme elektronskog plaćanja. U poslednje vreme u savremenom poslovanju sve je veća primena digitalnog potpisa.

Kod najvećeg broja korisnika interneta, zbog niskih cena široka je primena servisa elektronske razmene dokumenata (e-mail), a u poslednje vreme u porastu je upotreba telefona putem Internet mreže.

Spomenute aktivnosti kao savremeni vid poslovanja, doprinose povećanju aktivnosti na tržištu, razvoju novih poslova, povećanju produktivnosti i profitabilnosti, naučnom istraživanju i dr. Korišćenjem interneta pristup informacijama postaje neuporedivo brži u poređenju sa tradicionalnim formama razmene informacija.

Uprkos značajnim prednostima elektronskog poslovanja, sa otvaranjem lokalnih mreža prema Internetu rastu bezbednosni rizici i pretnje od zlonamernih napadača. Činjenica da milioni korisnika Interneta imaju pristup Web aplikacijama i da svi oni mogu, slučajno ili namerno ugroziti bezbednost podataka, doprinosi da bezbednost elektronskih transakcija u aplikacijama elektronskog poslovanja dobija sve veći značaj. Ključni zahtevi subjekata elektronskog poslovanja su: očuvanje privatnosti (poverljivosti) transakcija, provera autentičnosti subjekata i njihovih transakcija, provera integriteta transakcija i neporecivost razmene transakcija.

Za zadovoljenje istaknutih zahteva, pored fizičkih, tehničkih i organizacionih mera bezbednosti, u sklopu ukupne bezbednosne politike, postoji i specifična grupa tzv. kriptografskih mera zaštite elektronskih transakcija. Korišćenjem savremenih kriptografskih algoritama i sistema moguće je izgraditi kriptografske mehanizme kao što su: šifrovanje, dešifrovanje, digitalni potpis i digitalni sertifikat. Sa odgovarajućim kriptografskim protokolima ostvaruje se funkcionalnost navedenih mehanizama i zadovoljenje postavljenih bezbednosnih zahteva.

Cilj ovog rada je da se izvrši analiza i prikaže dizajn posebne grupe algoritama koji se nazivaju kriptografske heš funkcije. Ovo su algoritmi koji na ulazu prihvataju poruku

(npr.,elektronski dokument) proizvoljne dužine, a na izlazu daju kratak niz bitova. Njihova najvažnija uloga je u zaštiti autentičnosti podataka. Takođe, često se koriste u kombinadiji sa digitalnim potpisom ili kao gradivni blokovi drugih kriptografskih aplikacija kao što su aplikacije za zaštitu lozinki i aplikacije za generisanje pseudo-slučajnih nizova karaktera .

1.1. Osnovni koncepti

Heš funkcije (engl. Hash function) predstavljaju jedan od osnovnih elemenata kriptografije. Ove funkcije pretvaraju ulazni podatak proizvoljne dužine u izlazni podatak fiksne dužine – heš [2, 4, 9, 10]. Drugim rečima, heš funkcija vrši kompresiju ulaznog podatka proizvoljne dužine u izlazni podatak fiksne dužine. Kako postoji proizvoljan broj ulaznih podataka i ograničen broj izlaznih podataka, ova kompresija ne može da se koristi za čuvanje celokupnog sadržaja originalne poruke koja se nalazila na ulazu u heš funkciju.

Najprostiji primer heš funkcije je metod deljenja [1, 10]. Kod metoda deljenja bilo koji ulaz u heš funkciju možemo da interpretiramo kao celobrojni podatak (bilo koji podatak može da se prikaže kao niz bita, a ovaj niz bita može da se interpretira kao celobrojni podatak), tako da rezultat heš funkcije dobijamo kao ostatak pri deljenju tog celobrojnog ulaznog podatka nekim celim brojem n.

Heš funkcija može da se posmatra kao digitalni otisak prsta [2, 8, 9, 10]. Kao što se otisci prstiju koriste da jedinstveno identifikuju neku osobu, tako se i heš vrednosti koriste da identifikuju odgovarajuću poruku. Ovo znači da ako imamo poruku i neku heš vrednost, mi možemo da proverimo da li ta heš vrednost odgovara datoj poruci, ali iz same heš vrednosti ne možemo da saznamo ništa o sadržaju originalne poruke isto kao što na osnovu otisaka prstiju ne možemo ništa da zaključimo o osobi kojoj ti otisci pripadaju.

Kako heš funkcije predstavljaju preslikavanje većeg ulaznog skupa u manji izlazni skup, uvek postoji mogućnost da napač pronađe više ulaznih podataka koji daju istu heš vrednost kao neki drugi podatak i da na taj način izvrši falsifikovanje heš vrednosti tog podatka. Situacija kada više ulaznih podataka daje istu izlaznu heš vrednost naziva se *kolizija*. Kriterijumi po kojima se procenjuje sigurnost heš funkcija usko su vezani za trenutni nivo razvijenosti računarske tehnike. U nastavku rada kada budemo upotrebljavali izraze “nemoguće je naći“, “teško se izračunava“ ili “računski neizvodljivo“, podrazumevaćemo da se radi o izračunavanjima za koja bi bilo potrebno nekoliko miliona godina ukoliko bi se koristila procesorska snaga svih računara na svetu [2, 3].

Heš funkcije imaju veliki broj i kriptografskih i nekriptografskih primena. U ovom radu opisaćemo upotrebu heš funkcija u kriptografske svrhe. U kriptografiji heš funkcije se najčešće koriste za autentifikaciju i zaštitu integriteta poruka [2, 3]. Da bismo mogli da pričamo o primeni heš funkcija i da bismo izbegli mešanje pojmova, prvo moramo da

objasnimo šta znače pojmovi *poverljivost, autentifikacija, integritet podataka* i *neporecivost učestvovanja u komunikaciji* [2, 5, 11].

Poverljivost

Poverljivost (engl. confidentiality) je zaštita podataka od neovlašćenog pristupa ili otkrivanja. Ova usluga se tipično ostvaruje preko kontrole pristupa skladištu podataka u sprezi sa šifrovanjem i šifrovanjem podataka tokom prenosa. Radi jednostavnijeg razumevanja pojam poverljivost se može poistovetiti sa neprovidnom kovertom. Poruka je u koverti i nije vidljiva spolja. Naravno, gotovo svako može da otvori kovertu i pročita sadržaj poruke. Međutim šifrovanje je poput neverovatno jake koverte koja ne može da bude otvorena izuzev od strane autorizovanih osoba kojima je u stvari poslata ta koverta.

Autentifikacija

Autentifikacija (engl. Authentication) ili verodostojnost je proces verifikacije identiteta korisnika i/ili porekla podataka. Drugim rečima neka osoba (ili čak kompjuter) koji šalje ili prima podatke od nekog drugog entiteta (kompjuter ili osoba) treba da verifikuje da se radi o identitetu za koji se misli da jeste, odnosno, o entitetu za koji on sam tvrdi da je to baš on. U svakodnevnom životu verifikcija je česta pojava. Na primer kada pričamo telefon, osobu sa druge strane telefonske linije prepoznajemo po boji glasa.

Integritet

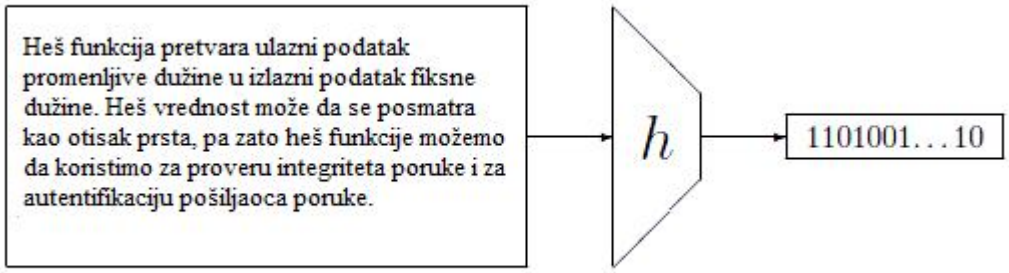
Provera integriteta (engl. integrity) podataka predstavlja zaštitu podataka od neovlašćene modifikacije ii zamene informacija. Radi jednostavnijeg razumevanja, pojam integriteta se može poistovetiti sa providnom kovertom. Naravno, poruka koja se nalazi unutra može da se pročita spolja što znači da tu nema nikakve poverljivosti. Međutim ta koverta je takva da jednoznačno ukazuje na činjenicu da li je neko prethodno manipulisao sa njom ili nije. Znači, onaj kome je ona adresirana, kada je primi može da pogleda u kovertu i da verifikuje da li je ona otvarana, pocepana ili da li je možda menjan sadržaj unutar koverte.

Neporecivost učestvovanja u komunikaciji

Neporecivost učestvovanja u komunikaciji (engl. Non-repudation) je dokaz da je poruka bila poslata ili primljena i u suštini predstavlja kombinaciju autentifikacije i zaštite integriteta poruke. Ova usluga nudi izuzetno visok stepen zaštite pošto se kao dokaz može dostaviti nekoj trećoj strani. Asimetrična kriptografija obezbeđuje mehanizam koji se naziva *digitalni potpis,* tako da je jedino pošiljalac mogao da bude taj koji je potpisao poruku. Prema tome, bilo ko drugi, uključujući tu i primaoca potpisane poruke, može da verifikuje digitalni potpis. Ovo ima jake implikacije kao što je slučaj u sudovima, jer potpisnik poruke ne može ničim da opovrgne potpisivanje poruke.

2. Definicije, klasifikacija i sigurnosni zahtevi

Kao što smo već rekli, kriptografske heš funkcije se koriste za sažimanje ulaznog podatka proizvoljne dužine u izlazni podatak fiksne dužine - heš. Ako ovakva funkcija zadovoljava dodatne uslove (koji će u nastavku ovog poglavlja biti detaljno objašnjeni), onda ona može da se koristi u kriptografskim aplikacijama kao što je zaštita autentičnosti poruke koja je poslata preko nesigurnog kanala. Glavna ideja je da heš funkcija izgeneriše otisak (jedinstven sažetak) poruke i da se onda vrši zaštita ovog otiska što je mnogo lakše od zaštite cele poruke. Na osnovu otiska primalac može da utvrdi da li je poruka stigla u originalnom ili izmenjenom obliku [8]. Na **slici 1** je ilustrovana upotreba heš funkcija.



Slika 1. Prikaz upotrebe heš funkcija. Ulazna poruka može da bude promenljive dužine, ali broj bajtova na izlazu je fiksan.

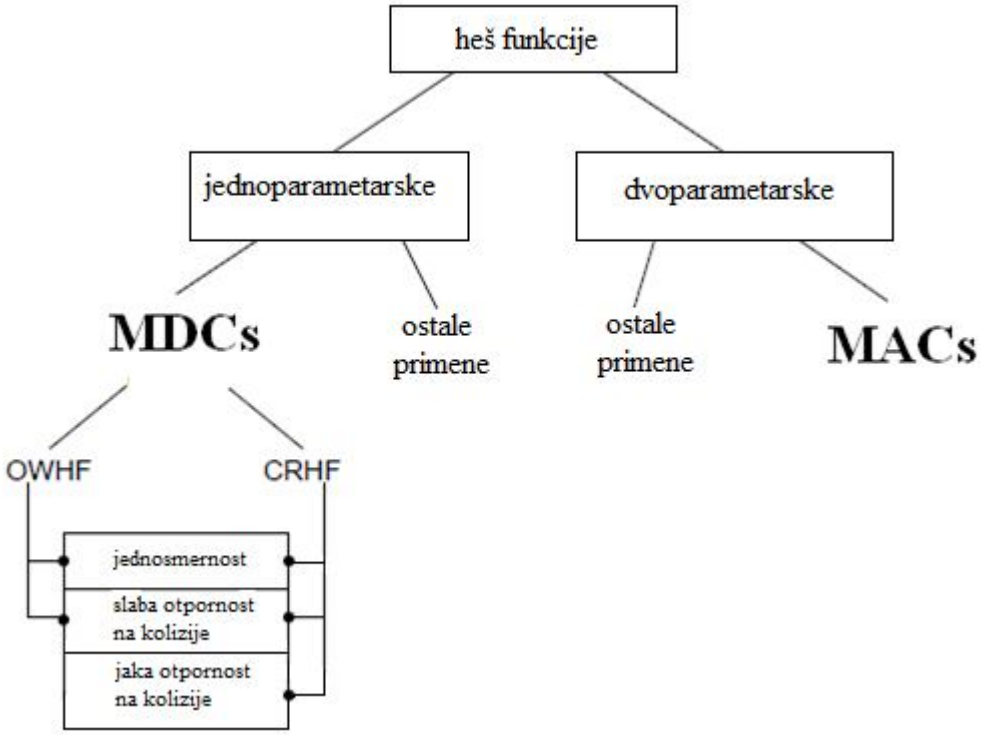
Definicija 1 [2, 3, 4]**:** *Heš funkcija* (u opštem smislu) je funkcija *h* koja poseduje bar sledeće dve osobine:

1. *osobinu kompresije* – h pretvara ulazni podatak *x* promenljive dužine u izlazni podatak *h(x)* fiksne dužine,
2. *efikasnost* – kada je dato *h* i ulazni podatak *x, h(x)* se lako računa.

Jedan od najjednostavnijih primera heš funkcije je računanje vrednosti operacije ekskluzivno ILI nad svim bajtovima poruke. Bez obzira na dužinu poruke, kao rezultat se uvek dobija jedan bajt [2].

Kako heš funkcije predstavljaju preslikavanje više-na-jedan, može da dođe do pojave kolizija ili "sudara". Kolizija (engl. collision) je pojava kada heš funkcija za dva (ili više) različitih ulaza daje istu izlaznu vrednost [1, 2]. Dakle postoji mogućnost da dve različite poruke rezultuju identičnim izlazom. Ova pojava predstavlja veliki problem ako se heš funkcija koristi u okviru mehanizma autentifikacije. Međutim, verovatnoća da dve slučajne poruke daju isti heš dužine *n* je 2^{-n}, što znači da se kod dobrih heš funkcija sa dovoljno dugačkom izlaznom vrednošću teško pronalaze dve poruke koje generišu istu heš vrednost.

Kod dobrih heš funkcija, promena jednog bita u ulaznom podatku dovodi do promene najmanje polovine bitova izlaza. U opštem slučaju, algoritam koji opisuje heš funkciju se ne skriva, a sigurnost funkcije zavisi od njene jednosmernosti [3].



Slika 2. Podela heš funkcija po funkcionalnosti

2.1 Klasifikacija heš funkcija

Heš funkcije se dele na jednoparametarske (ulazni argument je samo poruka) i dvoparametarske (ulazni argument su poruka i tajni ključ). U nastavku rada pojam "*kriptografska heš funkcija* " odnosiće se isključivo na jednoparametarske heš funkcije. Pored ove podele postoji i podela zasnovana na specifičnoj primeni pojedinih funkcija (**slika 2**). Po funkcionalnoj podeli heš funkcije se dele na [3, 4]:

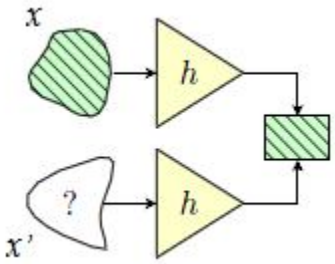
- **Mehanizme za uočavanje promena** (engl. Modification detection codes, MDC). Osnovna uloga ovih funkcija je da obezbede sažetak poruke kojim se poruka može jednoznačno identifikovati u daljoj obradi. MDC funkcije su potklasa jednoparametarskih heš funkcija. MDC funkcije se dele na:
    1. *jednosmerne heš funkcije* (engl. one-way hash functions, OWHF): kod ovih funkcija teško je naći ulazni podatak koji daje unapred specificiranu izlaznu heš vrednost

2. *heš funkcije otporne na kolizije* (engl. collision resistant functions, CRHF): kod ovih funkcija teško je naći bilo koja dva ulazna podatka koja daju istu izlaznu heš vrednost

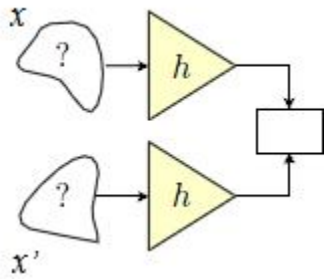
- **Mehanizme za autentifikaciju poruka** (engl. Message autentification codes, MAC). MAC su heš funkcije koje na osnovu dva funkcionalno nezavisna ulaza (poruka i tajni ključ) proizvode heš. Sistem je projektovan tako da bude skoro nemoguće dobiti originalni heš bez poznavanja tajnog ključa. Ova klasa funkcija je potklasa dvoparametarskih heš funkcija.

Da bismo omogućili dalje razmatranje heš funkcija kao i uvođenje novih definicija potrebno je da uvedemo sledeća tri svojstva koja se odnose na jednoparametarske heš funkciju *h* sa ulaznim podacima *x* i *x'* , i sa izlaznim heš vrednostima *y* i *y'* [4] :

1. **jednosmernost** (engl. one-way, pre-image resistance) – za zadato y praktično je nemoguće naci *x* takvo da je *h(x)=y*
2. **slaba otpornost na kolizije** (engl. 2nd pre-image resistance, weak collisin resistance) **–** za zadato *x* nemoguće je naći *x'* takvo da je *h(x')=h(x)* (**slika 3**)
3. **jaka otpornost na kolizije** (engl. collision resistance, strong collision resistance) **–** nemoguće je naći *x* i *x', x≠x'* takve da je *h(x')=h(x)* (**slika 4**)



Slika 3. Slaba otpornost na kolizije



Slika 4. Jaka otpornost na kolizije

Jednosmerna heš funkcija

Jednosmerne heš funkcije su jedan od najvažnijih pojmova u kriptografiji. Iako same nisu protokoli, jednosmerne heš funkcije predstavljaju osnovne gradivne delove mnogih protokola koji se koriste u kriptografiji. Osobina ovih funkcija je da se za poznatu izlaznu heš vrednost teško pronalazi ulazna poruka koja daje tu heš vrednost.

Definicija 2 [4, 11]**:** *Jednosmerna heš funkcija* je funkcija *h* koja poseduje sledeće osobine:

1. *osobinu kompresije – h* pretvara ulazni podatak *x* promenljive dužine u izlazni podatak *h(x)* fiksne dužine,

2. *efikasnost* – kada je dato *h* i ulazni podatak *x, h(x)* se lako računa
3. *jednosmernost*
4. *slabu otpornost na kolizije*

Heš funkcija otporna na kolizije

Glavna karakteristika heš funkcija otpornih na otkaze je teško (računski neizvodljivo) pronalaženje bilo koja dva ulazna podatka koja daju istu izlaznu heš vrednost

Definicija 3 [4, 11]: *Heš funkcija otporna na kolizije* je funkcija *h* koja poseduje sledeća osobine :

1. *osobinu kompresije* – *h* pretvara ulazni podatak *x* promenljive dužine u izlazni podatak *h(x)* fiksne dužine,
2. *efikasnost* – kada je dato *h* i ulazni podatak *x, h(x)* se lako računa
3. *slabu otpornost na kolizije*
4. *jaku otpornost na kolizije*

Mehanizmi za autentifikaciju poruka (MAC)

Ove funkcije na svom ulazu primaju dva parametra, poruku i tajni ključ. Osnovna ideja upotrebe tajnog ključa je da protivnik bez poznavanja ovog tajnog ključa ne može za posmatranu poruku da falsifikuje heš vrednost koja se dobija na izlazu funkcije, čak ni kada mu je prethodno poznat veliki broj poruka i njihovih odgovarajućih heš vrednosti.

Definicija 4 [4, 8]: *Mehanizam za autentifikaciju podataka (MAC)* je funkcija *h* koja poseduje sledeća osobine :

1. *osobinu kompresije* – *h* pretvara ulazni podatak *x* promenljive dužine u izlazni podatak *h(x,K)* fiksne dužine, gde je *K* dodatni ulazni parametar (dužine k bitova) koji predstavlja tajni ključ
2. *efikasnost* – kada je dato *h* , ulazni podatak *x* i tajni ključ *K* *h(x,K)* se lako računa
3. ako je dato *x* i ako je nepoznat tajni ključ *K* računski je neizvodljivo da se pronađe *h(x, K);* čak i ako je poznat veliki skup parova $\{x_i , h(x_i , K)\}$ računski je neizvodljivo otkriti tajni ključ *K* ili pronaći *h(y,K)* za bilo koju poruku $y \neq x_i$ (za svako x_i)

Za MAC funkciju koja ne poseduje **osobinu 3** smatra se da je nesigurna i da je podlona napadima. Dakle, iz **osobine 3** zaključujemo da je za sigurnu MAC funkciju računski neizvodljivo pronaći tajni ključ, ali moramo da imamo na umu da sama nemogućnost pronalaženja tajnog ključa ne znači i sigurnost MAC funkcije, jer uvek postoji mogućnost da se na neki drugi način pronađe ulazni podatak koji odgovara specificiranoj heš vrednosti [4].

Jednosmerna funkcija

Zbog velike sličnosti sa jednosmernim heš funkcijama dodatno ćemo objasniti pojam "jednosmerna funkcija". Jednosmerne funkcije se relativno lako računaju, ali se veoma teško na osnovu izlazne vrednosti određuje ulazna vrednost. Ovo znači da, ako nam je poznata ulazna vrednost *x*, možemo lako da izračunamo *y=f(x),* ali za poznato y računski je neizvodljivo da se pronađe $x=f^{-1}(x)$. Jednosmerne funkcije ne mogu da se koriste za šifrovanje podataka. Poruka jednosmernim funkcijama je beskorisna zato što niko ne može da je dešifruje [2, 4].

Za razliku od jednosmernih heš funkcija, jednosmerne funkcije na ulazu primaju podatak fiksne dužine *m*, a na izlazu daju podatak fiksne dužine *n*. Takođe, jednosmerne funkcije, prema gore navedenom opisu, ne poseduju slabu otpornost na kolizije. U slučaju da jednosmerna funkcija ne vrši kompresiju (ulazni i izlazni podatak su iste dužine) slaba otpornost na kolizije je podrazumevana, jer u tom slučaju svakoj izlaznoj vrednosti odgovara tačno jedan ulazni podatak [4].

Jednosmerne funkcije ne mogu da se koriste za šifrovanje podataka. Poruka šifrovana jednosmernim funkcijama je beskorisna zato što niko ne može da je dešifruje [2].

Jednosmerna funkcija sa zamkom, tj. privatna jednosmerna funkcija (engl. Trapdoor one-way function), za koju važi da je lako izračunati u jednom smeru i da je teško izračunati u drugom smeru. Ali, ako znamo tajnu informaciju, možemo lako da je izračunamo i u suprotnom smeru. Dakle, jednosmerna funkcija sa zamkom je funkcija za koju važi [2]:

1. za dato x, f(x) se određuje relativno lako i efikasno
2. za dato y=f(x), $x=f^{-1}(x)$ se određuje relativno teško
3. za dato y=f(x) i tajnu informaciju z (zamka), $x=g(x=f^{-1}(x),z)$ određuje se relativno lako i efikasno.

Dobar primer za jednosmernu funkciju sa zamkom je rasklapanje ručnog časovnika. Rasklapanje časovnika u delove y=f(x) je jednostavan posao. Sklapanje časovnika iz delova, tj. određivanje $x=f^{-1}(x)$ je vremenski zahtevno i komplikovano, ali se može relativno brzo obaviti ko na raspolaganju imate uputstvo za sklapanje z.

Dodatna svojstva heš funkcija

Većina jednoparametarskih heš funkcija je dizajnirana tako da obezbedi integritet podataka. Ove funkcije se uglavnom koriste kao mehanizmi za uočavanje promena, i kao takve dizajnirane su da poseduju jednosmernost i jaku ili slabu otpornost na kolizije. Zbog toga

što poseduju osobine jednosmernosti i nasumičnosti, ove funkcije mogu pored očuvanja integriteta, da se koriste i u sledeće svrhe:

1. *potvrda znanja (engl. confirmation of knowledge)* – proverava se kome podaci pripadaju a da se pritom ne otkrivaju sami podaci
2. *izvođenje ključeva (engl. key derivation)* – računa se sekvenca novih ključeva osnovu nekog od ranijih ključeva
3. *Pseudoslučajno generisanje brojeva (engl. pseudorandom number generation)* – računa se sekvenca brojeva u skladu sa odgovarajućim kriterijumima slučajnosti.

O ovim i o drugim primenama heš funkcija govoriće se više u nekom od narednih poglavlja.

Da bi jednosmerne heš funkcije mogle da se koriste u prethodno navedene svrhe, moraju da poseduju sledeća svojstva [4]:

1. *nekorelisanost (engl. non-correlation)* - ulaz i izlaz heš funkcije ne smeju biti korelisani. Sa ovim svojstvom je povezan ***lavinski efekat*** za koji važi da promena jednog bita na ulazu utiče na sve bite na izlazu
2. *otpornost na bliske kolizije (engl. near-collision resistance)* – teško se nalaze dva ulazna podatka *x* i *x'*,takva da se njihovi heševi *h(x)* i *h(x')* razlikuju u veoma malom broju bitova
3. *lokalna jednosmernost (engl. local one-wayness)* – kod funkcija koje poseduju ovo svojstvo, otkrivanje dela ulaznog podatka podjednako je teško kao i otkrivanje celog ulaznog podatka. Čak i kada je poznat deo ulaznog podatka ostatak se veoma teško određuje (npr. ako je ostalo nepoznato *n* bitova ulaznog podatka, da bi se oni izračunali potrebno je izvršiti 2^{n-1} operacija).

2.2. Klasifikacija napada na heš funkcije

Pod napadom na heš funkcije podrazumeva se upotreba algoritma koji je napravljen tako da naruši neko od sigurnosnih svojstava heš funkcija (jednosmernost, slaba i jaka otpornost na kolizije).

Prema količini informacija koje kriptoanalitičar poseduje o posmatranoj heš funkciji, napadi se dele na [8, 10]:

1. *opšte napade* (engl. generic attacks)
2. *napadi prečicom* (engl. shortcut attacks)

Opšti napadi ne zavise od algoritma koji je implementiran unutar heš funkcije. Ovi napadi tretiraju heš funkciju kao crnu kutiju. Prilikom implementacije ovih napada koriste se informacije kao što su dužina izlazne heš vrednosti ili pretpostavke o raspodeli tih izlaznih vrednosti [10].

Napadi prečicom su napadi koji zavise od samog algoritma heš funkcije. Ovi napadi koriste specifične detalje unutar strukture heš funkcije kako bi iskoristili slabosti u dizajnu heš funkcije. Ovi napadi bice detaljnije objašnjeni u **poglavlju 3**, u kome su objašnjeni metodi za konstruisanje heš funkcija [10].

Takođe, podela napada na heš funkcije može da se i prema kriterijum računske složenosti u odnosu na trenutni limit računarske tehnologije [10]. Prema ovom kriterijum postoje dve vrste napada: praktični i teoretski. Praktični napad je napad koji zahteva računarske resurse koji su trenutno distupni. Teoretski napadi su napadi koji ne mogu da se implementiraju pomoću trenutno raspoložive računarske tehnologije. Granica između praktičnih i teoretskih napada ne može precizno da se odredi i menja se sa razvojem računarske tehnike. Danas se smatra, da algoritmi koji zahtevaju 2^{80} ili više operacija ne mogu da budu implementirani pomoću dostupne tehnologije. Smatra se da će ova granica od 2^{80} operacija da važi još značajan broj godina.

2.2.1 Opšti napadi

Kao što smo već rekli, opšti napadi su oni napadi za koje nije potrebno bilo kakvo znanje o samoj strukturi heš funkcije. U ovom delu rada razmatraćemo samo one napade koji mogu da budu primenjeni na bilo koju heš funkciju.

Napad grubom silom

Napad grubom silom (engl. Brute-force attack) [3, 4, 8 10] je vrsta napada, kod koga napadač isprobava redom sve moguće kombinacije ulaznog podatka, sve dok ne dobije poklapanje sa posmatranim hešom. S obzirom na broj različitih rezulteta heš funkcije (ako funkcija daje heš dužine npr. 160 bita, broj mogućih rezultata te heš funkcije je 2^{160}) , dolazimo do zaključka da je napad čistom silom teško izvodljiv i da zahteva mnogo vremena. Na primer: na procesoru Athlon XP 1800+, isprobavanje svih kombinacija poruke dugačke 8 bita (poruka može da sadrži mala slova, velika slova i brojeve) traje oko šest dana.

U opštem slučaju, ako je dužina heš vrednosti n bita i ako napadač može da izvede 2^r računskih operacija, tada je verovatnoća da za slučajno izabrani ulazni podatak dobije odgovarajuću izlaznu heš vrednost, jednaka 2^{r-n}. Ovaj podatak je veoma važan i trebalo bi da se uzme u obzir prilikom određivanja dužine izlaza heš funkcije, jer za suviše kratku izlaznu vrednost, verovatnoća da ovaj napad uspe je veoma velika. U slučaju kada imamo samo jedan pokušaj za napad ($r = 0$), verovatnoća da dođe do uspeha je 2^{-n} i ovakav napad se naziva **slučajni napad.**

Rođendanski napad

Rođendanski napad (engl. Birthday paradox attack) [3, 4, 8 10] se zasniva na rođendanskom paradoksu koji kaže da će u grupi od ***r=23*** slučajno izabranih ljudi, bar dvoje imati rođendan istog dana sa verovatnoćom većom od 50%, što ćemo i da dokažemo. Kako godina ima 365 dana, verovatnoća q da u grupi od r osoba svi imaju rođendan različitog dana data je izrazom:

$$q = \frac{365 \cdot 364 \cdot \ldots \cdot (365 - r + 1)}{365^r} = \prod_{i=0}^{r-1}(1 - \frac{i}{365})$$

Odavde sledi da je verovatnoća p da bar voje ljudi u grupi imaju rođendan istog dana data izrazom: $p = (1 - q)$. Za $r = 23$ sledi da je verovatnoća $p \approx 0.507$. Treba primetiti da sa povećanjem r verovatnoća p vrlo brzo raste, tako da za $r = 46$ verovatnoća isnosi $p \approx 0.948$.

U rođendanskom napadu napadač na heš funkciju bira skup od r slučajnih ulaznih podataka sa očekivanjem da će bar dve poruke dati istu heš vrednost, što bi značilo da je pronađena kolizija. Ako izlazna heš vrednost ima dužinu od n bita, tada je ukupan broj mogućih heš vrednosti na izlazu funkcije jednak 2^n. Zamenjujući u prethodnim jednačinama 365 sa 2^n, dobijamo da je verovatnoća da kolizija bude pronađena data izrazom:

$$p = 1 - \prod_{i=1}^{r-1} \frac{2^n - i}{2^n} = 1 - \prod_{i=1}^{r-1} \left(1 - \frac{i}{2^n}\right) \approx 1 - \prod_{i=1}^{r-1} e^{-i/2^n} \approx 1 - e^{-\frac{r^2}{2^{n+1}}}$$

Iz ove jednačine sledi da, ako skup ulaznih podataka ima veličinu $r = \sqrt{2} \cdot 2^{n/2}$, verovatnoća pronalaženja kolizije ima vrednost $p = 0.63 = 63\%$. Takođe može da se zaključi da ako sigurno želimo da pronađemo koliziju, skup ulaznih podataka mora da ima veličinu $r = \sqrt{(\pi/2)} \cdot 2^{n/2}$, tj. moramo da izvršimo $\sqrt{(\pi/2)} \cdot 2^{n/2}$ izračunavanja heš vrednosti. Odavde zaključujemo da vremenska složenost rođendanskog napada iznosi $O(2^{n/2})$. Glavni problem ovog napada je potreba za velikom količinom memorije. Kako se pri svakom novom izračunavanju heš vrednosti vrši poređenje sa prethodno izračunatim heš vrednostima, zaključujemo da nam je potreban prostor za čuvanje O(r) poruka, što predstavlja veliki memorijski zahtev.

3. Dizajn kriptografskih heš funkcija

Da bi heš funkcija zadovoljila određene parametre koji se odnose na sigurnost i performanse, potrebno je da se prilikom dizajniranja te heš funkcije obrati posebna pažnja na odabir metode za konstruisanje heš funkcija. Prema načinu na koji su konstruisane,većina heš funkcija spada u grupu ***iterativnih heš funkcija.*** U zavisnosti od načina na koji vrše kompresiju podataka *iterativne heš funkcije* mogu da budu: [4, 8, 11]:

1. *Heš funkcije zasnovane na simetričnim blokovskim algoritmima* (engl. Hash Functions Based on Block Ciphers)
2. *Heš funkcije zasnovane na modularnoj aritmetici* (engl. Hash Functions Using Modular Arithmetic)
3. *Namenske heš funkcije* (engl. Dedicated Hash Functions)

Takođe, pri dizajniranju heš funkcije mora se voditi računa o veličini izlazne heš vrednosti, jer je dužina heš vrednosti jedan od glavnih faktora od kojih zavisi sigurnost heš funkcije.

3.1 Dužina izlazne heš vrednosti

U prethodnom poglavlju smo pokazali da je vremenska složenost rođendanskog napada reda $O(2^{n/2})$ operacija, gde je *n* dužina proizvedene heš vrednosti. Ovo je podatak koji je od presudne važnosti pri dizajniranju heš funkcije, jer od odabrane dužine heš vrednosti zavisi otpornost heš funkcije na pronalaženje kolizija [8].

Postavlja se pitanje kolika je potrebno da bude dužina proizvedene heš vrednosti, da bi u praksi svi napadi na heš funkciju bili neizvodljivi. 1995 godine, P. Van Oorschot i M.Wiener su napravili analizu rođendanskog napada na funkciju MD5 koja proizvodi heš vrednost dužine 128 bita (oko 2^{64} operacija je potrebno da bi se pronašla kolizija) [8]. U ovoj analizi su došli do zaključka da bi mašini koja vredi 10 miliona dolara bio potreban 21 dan da pronađe koliziju. Uzimajući u obzir Murov zakon koji kaže da se računarska snaga udvostručuje svakih 18 meseci, dolazimo do zaključka da dužina heš vrednosti od 128 bita više nije dovoljna da garantuje otpornost na kolizije. Kako smo u prethodnom poglavlju rekli da je granica između teoretskih i praktičnih napada 2^{80} operacija, dolazimo do zaključka da heš funkcije otporne na kolizije treba da imaju dužinu izlazne heš vrednosti od najmanje 160 bita. Da bi se obezbedila dugoročna sigurnost, preporučuje se da se biraju heš funkcije sa što većom dužinom proizvedenog heša.

Ukoliko pretpostavljamo da dužina generisane heš vrednosti nije dovoljna, a imamo poverenja u datu funkciju, uporebom sledećeg algoritma možemo da uvećamo dužinu dobijene heš vrednosti [3, 5]:

1. Generiše se heš vrednost poruke koristeći neku od heš funkcija
2. Na kraj heš vrednosti, dobijene u prethodnom koraku, nadoveže se originalna poruka

3. Generiše se heš vrednost podatka dobijenog u koraku 2
4. Kreira se duži heš koji sadrži heš vrednost dobijenu u koraku 1 na koju je nadovezana heš vrednost dobijena u koraku 3
5. Ponavljaju se koraci od 2 do 4 nadovezujući tako dobijene heš vrednosti, sve dok se ne dobije heš željene dužine

Postoje sumnje u ovaj algoritam, mada nikada nije dokazana ni njegova sigurnost ni njegova nesigurnost [5].

3.2 Opšti model iterativnih heš funkcija

Nije lako projektovati funkciju koja prihvata podatak proizvoljnre dužine, a na izlazu daje podatak fiksne dužine. Kako heš funkcija mora da obrađuje podatke proizvoljne dužine, ideja je da se ulazni podatak podeli na blokove fiksne veličine koji se dalje obrađuju jedan po jedan. Na svaki od ovih blokova se primenjuje funkcija kompresije koja na ulazu prihvata podatak fiksne dužine [6].

Ulazni podatak X se prvo dopunjava odgovarajućim sadržajem sve dok njegova dužina ne postane celobrojni umnožak dužine jednog bloka. Ovaj dodatak često sadrži dužinu originalnog ulaznog podatka, što ćemo kasnije detaljnije da objasnimo. Zatim, podatak se deli u t blokova od X_1 do X_t. Rezultujuća heš vrednost se dobija na sledeći način:

$H_0=IV$
$H_i=f(X_i , H_{i-1}),\quad i=1,2,\ldots t$
$h(X)=g(H_t)$

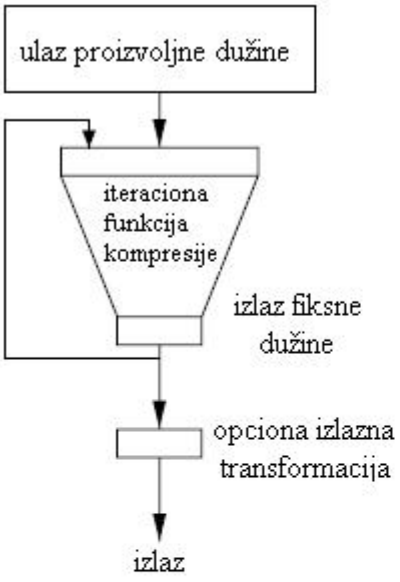
IV je skraćenica za *inicijalnu vrednost*, H_i je ulančavajuća promenljiva (engl. chaining variable), funkcija f je funkcija kompresije, funkcija g predstavlja izlaznu transformaciju. Izlazna transformacija g se često izostavlja prilikom konstrukcije heš funkcije, tj. izlazna transformacija predstavlja funkciju identiteta ($g(H_t) = H_t$), pa u tom slučaju izlazna heš vrednost ima istu dužinu kao ulančavajuće promenljive [8]. Na **slici 5** je prikazan način rada iterativnih heš funkcija.

Dva elementa u ovom algoritmu imaju poseban uticaj na sigurnost heš funkcije: način dopunjavanja ulaznog podatka i izbor inicijalne vrednosti IV.

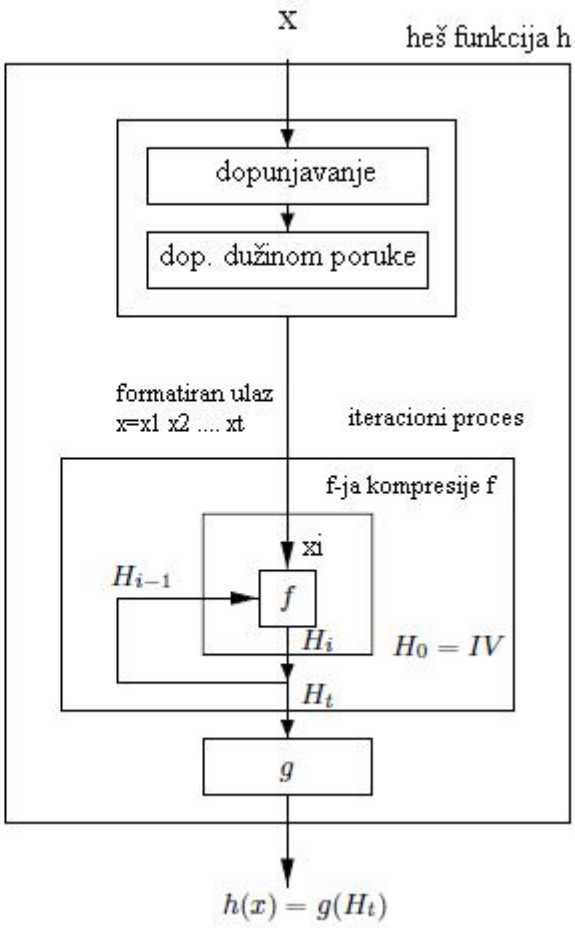
Važno je da način dopunjavanja bude nedvosmislen, tj. ne smeju da postoje dve poruke koje mogu da budu dopunjene na isti način. Zbog ovoga, prilikom samog dopunjavanja potrebno je na poruku nadovezati i dužinu originalne nedopunjene poruke, jer se na taj način prevazilazi sigurnosni problem da poruke različitih dužina mogu imati istu heš vrednost. Ova tehnika dopunjavanja se ponekad naziva i ***Merkle-Damgard ojačavanje*** **(**engl. Merkle-Damgard strengthening). [5].

Inicijalna vrednost IV treba da bude definisana kao deo specifikacije heš funkcije [6].

a) uopšteni prikaz



b) detaljniji prikaz



Slika 5. Prikaz realizacije iterativnih heš funkcija

Merkleov meta-metod heširanja

Ovaj algoritam je osmislio R. Merkle. Merkle-ov algoritam je iterativni metod za konstrukciju heš funkcija, koji koristi funkciju kompresije f sa jakom otpornošću na kolizije da bi konstruisao heš funkciju h koja takođe, poseduje jaku otpornost na kolizije [5]. Merkle-ov algoritam se sastoji od sledećih koraka:

1. Pretpostavimo da imamo funkciju f koja preslikava ulaz dužine $n+r$ u izlaz dužine n. Cilj nam je da, koristeći funkciju f, napravimo heš funkciju h koja proizvodi heš duzine n
2. Ulazni podatak X dužine b se podeli u t blokova $X=X_1X_2...X_t$ dužine r, dopunjavajući poslednji blok X_t binarnim ciframa 0 ako je to potrebno

3. Definiše se dodatni blok x_{t+1} koji sadrži binarnu predstavu broja b (pretpostavlja se da je $b<2^r$)
4. Izlazni heš dužine n bita izračunat za ulazni podatak X dat je izrazom $h(x)=H_{t+1}=f(H_t \| x_{t+1})$ i izračunava se pomoću sledećih izraza:
   $H_0=0^n$; gde je 0^n niz od n binarnih cifri
   $H_i=f(H_{i-1} \| x_{i+1})$

Dokaz da se ovim algoritmom dobija heš funkcija h koja poseduje jaku otpornost na kolizije, se nalazi u tome da funkcija kompresije f poseduje jaku otpornost na kolizije. Kad bi se u funkciji h desila kolizija, to bu značilo i da se u funkciji f , na nekom nivou i desila kolizija, što je u suprotnosti sa pretpostavkom da funkcija f poseduje jaku otpornost na kolizije.

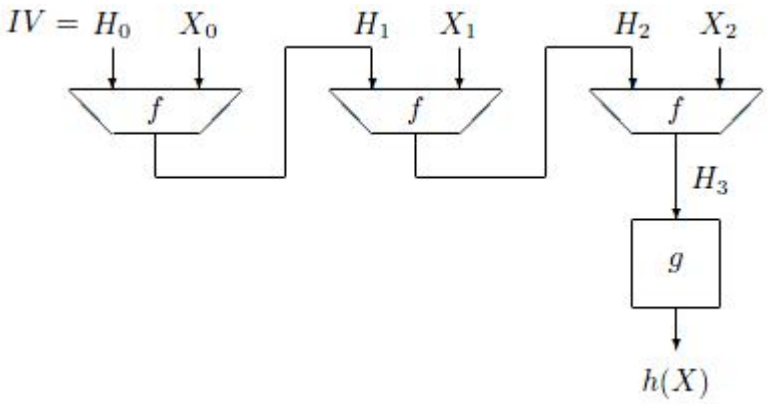
Dopunjavanje poruke blokom x_{t+1} koji sadrži originalnu dužinu poruke x naziva se ***Merkle-Damgard ojačavanje*** (engl. Merkle-Damgard strengthening). Ovo ojačanje je bitno zbog toga što povećava otpornost na kolizije rezultujuće heš funkcije h, tako što sprečava da dve poruke različitih dužina mogu imati istu heš vrednost [4, 5].

Definicija 5 [4]: Merkle-Damgard ojačavanje se koristi neposredno pre samog heširanja ulaznog podatka. Na ulazni podatak $X=X_1X_2...X_t$ se dodaje blok X_{t+1} koji sadrži binarnu reprezentaciju dužine b ulaznog podatka X. Prilikom binarne reprezentacije dužine b koristi se desno poravnanje. Merkle-Damgard ojačanje se koristi kada je ispunjen uslov: $b<2^r$.

3.2.1 Sigurnost iterativnih heš funkcija

Kao što smo već rekli na sigurnost iterativnih heš funkcija poseban uticaj imaju izbor inicijalne vrednosti IV i izbor pravila za dopunjavanje.

Posmatrajmo primer iterativne funkcije sa **slike 6.** Ako u ovom primeru napadač zameni IV sa H_1 i potom izbriše prvi blok ulaznog podatka X_0 , na ulazu u heš funkciju dobija novi ulazni podatak $X = X_1 \| X_2$ i početna ulančavajuća promenljiva ima vrednost H_1. Posmatrajući sliku lako se zaključuje da se nakon ovih promena dobija ista izlazna heš vrednost kao i za originalni ulazni podatak, što znači da je napadač pronašao koliziju. Da bismo sprečili ovu situaciju, potrebno je da inicijalna vrednost IV ulančavajuće promenljive H_0 bude nepromenljiva. Tačna inicijalna vrednost najčešće se definiše u samoj specifikaciji algoritma [8].



Slika 6. Primer iterativne heš funkcije

Druga važna stvar za sigurnost heš funkcija je upotreba pravila za dopunjavanje koje u dopunu uključuje i dužinu originalnog ulaznog parametra. Upotreba ovakvog pravila za dopunjavanje sprečava napade zasnovane na ***nepokretnoj tački.*** Napadi zasnovani na *nepokretnnnoj tački* se izvode tako što se u originalni ulazni podatak umeću dodatni blokovi čime se proizvode kolizije. Ovaj napad će biti detaljno objašnjen nešto kasnije. Nakon što smo objasnili najvažnije faktore koji utiču na bezbednost iterativnih heš funkcija, možemo da definišemo vezu između otpornosti na kolizije heš funkcije *h* i funkcije za kompresiju *f*.

Teorema 1. (Merkle - Damgard) [8]: Ako je *IV* fiksno i ako procedura za dopunjavanje uzima u obzir dužinu ulaznog podatka, onda je heš funkcija *h* otporna na kolizije ako i samo ako je funkcija *f* otporna na kolizije.

Dokaz [8] ove teoreme se zasniva na činjenici da ako postoji kolizija za heš funkciju *h*, to znači da je u nekoj od iteracija došlo do kolizije za funkciju kompresije *f*. Odavde zaključujemo da uspešan napad na funkciju kompresije *f* predstavlja i uspešan napad na kompletnu heš funkciju *h*. U napade na heš funkciju spadaju [6]:

1. *napad zasnovan na nepokretnoj tački* (engl. Fixed Point Attack)
2. *napad "susret u sredini"* (engl. Meet-in-the-Midle-Attack) [3]
3. *napad zasnovan na promeni bloka* (engl. Correcting-Block Attack)

Napad zasnovan na nepokretnoj tački

U matematici *nepokretna tačka* funkcije je tačka koju funkcija preslikava u samu sebe. Jednostavnije rečeno, *x* je nepokretna tačka funkcije *f* ako i samo ako je *f(x) = x*. Nepokretna tačka za funkciju kompresije je par (H_{i-1} , X_i), za koji važi *f* (H_{i-1} , X_i) = H_{i-1}. Ako je ulančavajuća promenljiva data sa H_{i-1} , napadač može da umetne u ulazni podatak , na mestu gde se javlja ulančavajuća promenljiva sa vrednošću H_{i-1} , proizvoljan broj blokova X_i i da pritom izlazna heš vrednost ostane nepromenjena. Ovaj napad može da se

izvede samo u slučaju kada u proceduru za dopunjavanje nije uključena dužina originalnog ulaznog podatka heš funkcije. Kada procedura za dopunjavanje generiše dopunu koja sadrži dužinu ulaznog podatka, umetanjem novih blokova u ulazni podatak menja se njegova dužina , a samim tim i vrednost dopune što dovodi do promene izlazne heš vrednosti.

Napad "susret u sredini"

Ovaj napad je varijanta rođendanskog napada. Za razliku od rođendanskog napada gde se traže bilo koja dva ulaza koji daju isti heš (*jaka otpornost na kolizije*), u ovom napadu se za tačno specificirani ulaz X traži X' takvo da važi $h(X) = h(X')$**.** Dakle, ovim napadom se napada svojstvo "*slaba otpornost na kolizije*" [8]. Ovaj napad se zasniva na poređenju ulančavajućih promenljivih koje se generišu nakon svake iteracije heš funkcije.

Pretpostavimo da nam je poznato X . U tom slučaju inicijalna vrednost IV i izlazna heš vrednost $h(x)$ su fiksni. Kriptoanalitičar određuje tačku napada negde u lancu između neka dva bloka ulaznog podatka X'. Potom kriptoanalitičar generiše r_1 varijacija prvog dela poruke X' i r_2 varijacija drugog dela poruke X'. Sada krećemo sa računanjem heš vrednosti za poruku X' , *gde je* $H_0 = IV$. Istovremeno idemo unazad kroz lanac ulančavajućih promenljivih počev od heš vrednosti $h(X') = h(X)$, sa ciljem da se ova dva izračunavanja sretnu u unapred odabranoj tački sa istom vrednošću ulančavajuće promenljive. Verovatnoća da se ovo dogodi data je izrazom $p = 1 - exp(-r_1 \cdot r_2/2^n)$. Vremenska složenost ovog napade je $O(2^{n/2})$, gde je n dužina izlazne heš vvrednosti. [6] .

Da bi ovaj napad mogao da bude izvodljiv kriptoanalitičar mora da ima mogućnost da ide unazad kroz lanac ulančavajućih promenljivih. Ovo znači da funkcija kompresije f mora da bude takva da se za dato H_{i+1} može pronaći par (H_i , X_i) takav da važi $f(H_i , X_i) = H_{i+1}$ [8].

Napad zasnovan na promeni bloka

Posmatrajmo heš funkciju h. Pretpostavimo da nam je dat ulazni podatak X i da napadač pokušava da pronađe X' *takvo* da važi $h(X') = h(x)$ (napad na svojstvo "slaba otpornost na kolizije"). U ovom napadu napadač bira blok X_i poruke X i menja ga blokom X'_i tako da za funkciju kompresije f važi $f(H_i , X_i) = f(H_i , X'_i)$. Na ovaj način se dobija poruka X' koja ima istu heš vrednost kao poruka X [6, 8].

Ako je veličina ulančavajuće promenljive c bitova i ako je veličina svakog od blokova poruke b bitova, tada broj blokova X_i koji zadovoljavaju uslov $f(H_i , X_i) = f(H_i , X'_i)$ iznosi $2^b/2^c = 2^{b-c}$. Ovaj broj predstavlja mali deo od ukupnog broja blokova, i kod idealne iterativne heš funkcije potrebno je 2^c operacija da bi se pronašao blok X'_i. Ovaj napad je moguć samo ako je funkcija kompresije takva da za H_i i H_{i+1} može da se nađe X'_i tako da važi $f(H_i , X'_i) = H_{i+1}$ [8].

3.3 Heš funkcije zasnovane na simetričnim blokovskim algoritmima

Dizajniranje sigurne heš funkcije nije lak zadatak. Jedno od mogućih rešenja je da se koristi neka od već postojećih kriptografskih primitiva, kao kao što su simetrični blokovski algoritmi. Prednost ovog pristupa je što mogu da se koriste gotovi simetrični blokovski algoritmi čija je bezbednost ispitana i za koje postoje već gotove implementacije. Nedostatak ovakvih algoritama su nešto lošije performanse, kao i to što korišćeni blokovski algoritmi mogu da pokažu neke slabosti koje ne bi pokazali kada se koriste samo za šifrovanje podataka [8].

U nastavku teksta operaciju šifrovanja označavaćemo sa $Y = E_K(X)$. Ovde X predstavlja otvoreni tekst, Y je šifrat, a K je ključ. Osobine simetričnog blokovskog algoritma za šifrovanje su sledeće:

- isti ključ koristi i za šifrovanje i za dešifrovanje
- algoritam obrađuje jedan po jedan blok otvorenog teksta koristeći ključ
- jedan blok otvorenog teksta se pomoću istog ključa uvek prevodi u isti šifrat
- veličine ulaznog i izlaznog bloka su iste

Simetrični blokovski algoritmi se sastoje od dva para algoritama, jednog za šifrovanje (E), i drugog za dešifrovanje (E^{-1}). Oba algoritma prihvataju dva ulazna parametra: ulazni blok veličine b bitova i ključ veličine k bitova. Šifrovanje i dešifrovanje su inverzne operacije tako da važi $E_K^{-1}(E_K(X))=X$, za poruku X i ključ K. Najpoznatiji simetrični blokovski algoritmi su: *DES (engl. Data encryption Standard)* i *AES (engl. Advanced Encryption Standard)* [8].

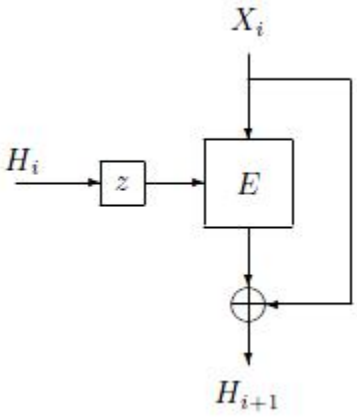
Na osnovu veličine izlazne heš vrednosti, heš funkcije zasnovane na simetričnim blokovskim algoritmima se dele u dve osnovne grupe: heš funkcije koje proizvode heš čija je dužina jednaka veličini bloka šifrata i heš funkcije koje proizvode heš čija je dužina dva puta veća od veličine bloka šifrata.

Heš funkcije sa izlaznom heš vrednošću koja je iste veličine kao i blok šifrata

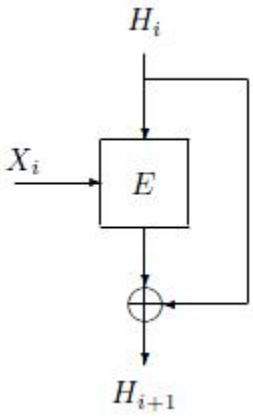
Pri konstrukciji ovih funkcija najčešće se koriste **Matyas-Meyer-Oseas** i **Davies-Meyer** metod (**slike 7 i 8).** Oba ova metoda predstavljaju iterativne algoritme. U svakoj od iteracija, trenutna vrednost ulančavajuće promenljive H_i i blok X_i koji se trenutno obrađuje, predstavljaju ključ i otvoreni tekst (ili obrnuto) za algoritam šifrovanja E. U prvoj rundi, kada ne postoji izračunata prethodna heš vrednost, koristi se inicijalna vrednost H_0 [8]. Operacija ekskluzivno ILI koja se nalazi na izlazu algoritma E ima veoma veliku važnost. Za poznato H_{i+1}, bez ove operacije bilo bi lako (primenom funkcije za dešifrovanje E^{-1}) pronaći par (H_i , X_i) takav da važi $f(H_i , X_i) = H_{i+1}$, gde je f funkcija kommpresije u čiji

sastav ulaze algoritam šifrovanja *E* i operacija ekskluzivno ILI . Matematički ovi metodi mogu da se opiše na sledeći način:

- *Matyas-Meyer-Oseas* metod: $H_{i+1} = f(H_i , X_i) = E_{Z(H_i)}(X_i) \oplus X_i$
- *Davies-Meyer* metod: $H_{i+1} = f(H_i , X_i) = E_{Hi}(H_i) \oplus H_i$

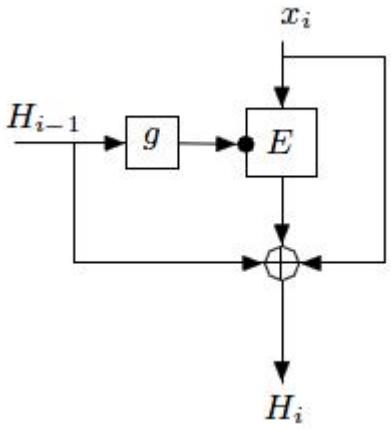


Slika 8. Matyas-Meyer-Oseas metod



Slika 9. Davies-Meyer metod

Kod oba ova algoritma izlazna heš vrednost ima istu dužina kao izlaz koji se dobija na izlazu algoritma za šifrovanje *E*. Kod *Davies-Meyer*-ovog metoda ulančavajuća promenljiva H_i, izlazna heš vrednost, ključ i veličina bloka šifrata imaju istu veličinu ($c = n = k = b$). Za razliku od *Davies-Meyer*-ovog metoda, *Matyas-Meyer-Oseas* metod koristi algoritam za šifrovanje kod koga je veličina ključa *k* različita od veličine bloka *b*. Zato se kod *Matyas-Meyer-Oseas* metoda, na ulazu u algoritam šifrovanja nalazi funkcija *z* koja ulančavajuću promenljivu H_i koja ima istu veličinu kao i blok šifrata ($c = n = b \neq k$), pretvara u ključ odgovarajuće veličine k [8].



Slika 10. *Miyaguchi-Preneel metod*

Pored ova dva opisana metoda koji su veoma slični, postoji i treći metod za konstrukciju heš funkcija baziran na simetričnim blokovskim algoritmima koji se naziva ***Miyaguchi-Preneel metod*** (**slika 10**) **.** Funkcija kompresije *f* koju koristi ovaj algoritam je data izrazom: $H_{i+1} = f(H_i , X_i) = E_{Hi}(H_i) \oplus X_i \oplus H_i$ [4].

Kada se *DES* koristi kao algoritam za šifrovanje, *Davies-Meyer* metod obrađuje ulazni podatak u blokovima veličine 56 bita, dok *Matyas-Meyer-Oseas* i *Miyaguchi-Preneel* metod to rade u blokovima veličine 64 bita. U slučaju kada se koristi *AES*, za *Matyas-Meyer-Oseas* i *Miyaguchi-Preneel* metode dobijamo izlazni heš od 128 bita. [4].

Heš funkcije sa izlaznom heš vrednošću koja je dva puta veći od veličine bloka šifrata

Metode koje smo prethodno opisali generišu heš funkcije koje na izlazu proizvode heš vrednost koja je suviše kratka (npr. 128 bita), posebno kda se posmatra otpornost na kolizije. Jedno od rešenja za ovaj problem je da se konstruišu heš funkcije koje proizvode heš rezultat koji je dva puta duži od veličine blok šifrata [8]. Ovo znači da bismo upotrebom *DES* algoritma dobili 128-bitnu heš funkciju, a upotrebom *AES* algoritma 256-bitnu funkciju.

Najpoznatije funkcije ove vrste su *MDC-2* i *MDC-4* [8]. MDC-2 funkcija poseduje funkciju kompresije *f* koja se sastoji od dva paralelna izračunavanja funkcije kompresije iz *Matyas-Meyer-Oseas* algoritma. Označimo sa C^L i C^R levu i desnu polovinu promenljive *C* dužine *b* bita. Funkcija kompresije može da se prikaže izrazom

$$H_{i+1} \| \tilde{H}_{i+1} = f(H_i \| \tilde{H}_i, X_i) ,$$

pri čemu ova funkcija kompresije sadrži sledeća izračunavanja:

$$\begin{aligned} C_{i+1} &= E_{z(H_i)}(X_i) \oplus X_i , \\ \tilde{C}_{i+1} &= E_{\bar{z}(\tilde{H}_i)}(X_i) \oplus X_i , \\ H_{i+1} &= C^L_{i+1} \| \tilde{C}^R_{i+1} , \\ \tilde{H}_{i+1} &= \tilde{C}^L_{i+1} \| C^R_{i+1} . \end{aligned}$$

Ako bi veza između leve idesne polovine bila ukinuta, imali bismo dva lanca izračunavanja koji između sebe uopšte nisu povezani, što pruža mogućnost za odvojene napade na svaki od lanaca. Takođe treba primetiti da ulančavajuće promenljive kao i izlazna heš vrednost imaju dužinu duplo veću od dužine bloka šifrata [8].

Funkcija kompresije za *MDC-4* se sastoji od dva sekvencijalna izvršavanja funkcije za kompresiju algoritma *MDC-2*., gde se ključevi i otvoreni tekst sa drugo izvršavanje dobija kao izlazne vrednosti prvog izvršavanja.

3.2.1 Sigurnost heš funkcija zasnovanih na simetričnim blokovskim algoritmima

Za ovu vrstu heš funkcija slabost u korišćenom blokovskom algoritmu može da predstavlja priliku za napadača da izvede uspešan napad na tu heš funkciju. Diferencijalna kriptoanaliza je vrsta napada na simetrične blokovske algoritme u kome se posmatraju dva bloka sa specifičnim razlikama i analizira se evolucija te razlike pri prolasku kroz algoritam [2]. Upravo diferencijalna kriptoanaliza predstavlja najveću opasnost za ovako konstruisane heš funkcije. Odavde zaključujemo da je prilikom konstruisanja heš funkcije najbolje izabrati neki simetrični blokovski algoritam koji je prošao detaljnu kriptoanalizu i za koji se smatra da je siguran. Ipak, treba imati na umu da neke slabosti algoritma koje nemaju značaj prilikom upotrebe algoritma u svrhe šifrovnja, mogu da imaju veoma veliki uticaj na sigurnost konstruisane heš funkcije [8].

3.4 Heš funkcije zasnovane na modularnoj aritmetici

Heš funkcije mogu da koriste i modularnu aritmetiku kao osnovni gradivni element za funkciju kompresije. Prednost funkcija zasnovanih na modularnoj aritmetici se nalazi u tome što se u skladu sa sigurnosnim uslovima lako može izabrati dužina modula po kome se vrši deljenje. Najveći nedostatak ovako konstruisanih funkcija je brzina izvršavanja. Ove funkcije su značajno sporije čak i od heš funkcija zasnovanih na simetričnim blokovskim algoritmima [4, 8].

Najpoznatije funkcije ove vrste su ***MASH-1*** i ***MASH-2.*** *Mash-1* je funkcija kod koje se takođe, ulazni podatak deli na blokove X_i koji zajedno sa ulančavajućim promenljivima H_i predstavljaju ulaz u funkciju kompresije koja je zasnovana na modularnoj aritmetici. Funkcija kompresije za *MASH-1* je data izrazom $H_i = (((X_i \oplus H_{i-1}) \vee A)^2 \bmod N) \oplus H_{i-1}$. Parametar A ima vrednost 0xf00...00, što znači da su četiri bita najveće važnosti u bloku X_i postavljena na vrednost 1111. Ako parametar N ima dužinu n bita, to znači da je operacijom deljenja po modulu rezultat kvadratiranja skraćen na n bita [6].

MASH-2 predstavlja poboljšanu i sigurniju varijantu funkcije *MASH-1*.

3.5 Namenske heš funkcije

Namenske heš funkcije su funkcije koje su dizajnirane sa unapred tačno određenom vrstom primene. Ovo znači da ove funkcije nisu unapred ograničene na korišćenje već postolećih komponenti, već mogu da se dizajniraju od samog početka pri tome vodeći računa o performansama. U grupu namenskih heš funkcija spadaju funkcije zasnovane na ***MD4*** heš algoritmu, o čemu će detaljno biti reči u **poglavlju 5**. *MD4* funkcija je dizajnirana za upotrebu na 32-bitnim platformama, ali zbog sigurnosnih nedostataka koji su vrlo brzo

pronađeni pristupilo se konstruisanju nove funkcije koja je dobila ime ***MD5.*** U grupu funkcija koje su nastale od MD4 spadaju i funkcije ***RIPEMD*** i ***SHA.***

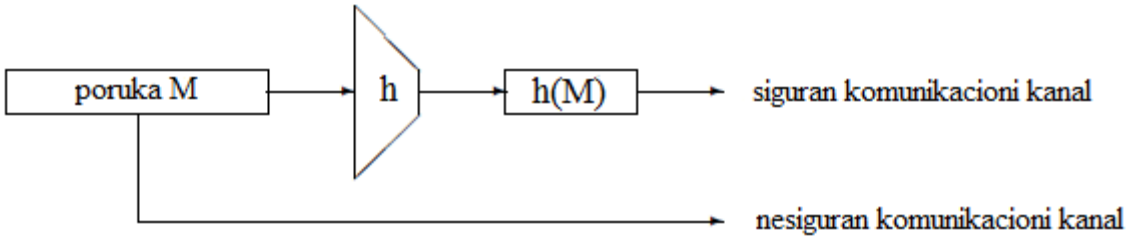
Takođe, u nameske heš funkcije spadaju funkcije kao što je ***Tiger*** koja je dizajnirana za upotrebu na 32-bitnim platformama ili funkcija ***Whirpool*** koja je konstruisana koristeći *Miyaguchi-Preneel* metod. Poseban primer predstavlja funkcija ***PANAMA*** koja je dizajnirana tako da može da se koristi i za heširanje i za šifrovanje [4, 6].

4. Primena kriptografskih heš funkcija

Kriptografske heš funkcije predstavljaju moćan alat kojim mogu da se ostvare mnogi sigurnosni ciljevi. Neke od oblasti u kojima se primenjuju kriptografske heš funkcije su: *autentifikacija poruka, digitalno potpisivanje, vremenske oznake, čuvanje lozinki*.

4.1 Autentifikacija poruka (engl. Message authentication)

Posmatrajmo situaciju kada se poruka šalje preko nesigurnog kanala. Kod ovakvog slanja postavlja se pitanje autentičnosti primljene poruke. Zbog ovoga mora da se napravi mehanizam koji će moći da utvrdi da li je poruka menjana na svom putu do odredišta. Osnovna ideja ovakvog mehanizma je da se zaštita vrši nad kratkim otiskom poruke, a ne nad čitavom porukom [4, 8, 10].

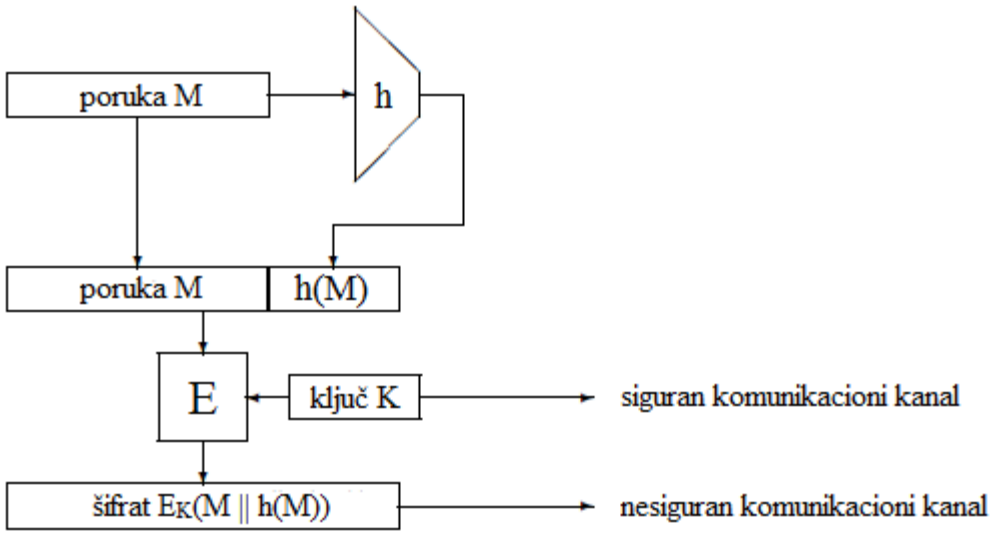


Slika 11. Autentifikacija poruke pomoću heš funkcije *h*

Slika 11 prikazuje jedan od načina na koji može da se izvrši autentifikacija poruke uz pomoć heš funkcije. Potrebno je da postoji siguran heš kanal, u kome poruka ne može da bude promenjena od strane napadač. Siguran kanal se koristi da na siguran način prenese heš vrednost *h(M)* dobijenu heširanjem poruke *M*. Kao siguran komunikacioni kanal može da se koristi telefonska linija jer je veličina heš vrednosti dovoljno mala da može da se prenese rečima. Najvažnija osobina sigurnog kanala je to što primalac zna ko mu je poslao poruku preko ovog kanala. Primer sigurnog kanala je telefonska veza gde autentifikacija može de se izvrši prepoznavanjem glasa. Pre samog slanja poruke pošiljalac izračunava heš vrednost poruke i tako dobijeni heš šalje preko sigurnog kanala, dok se slanje same poruke vrši preko kanala za koji se smatra da nije siguran. Napadač može da izmeni poruku pre nego što ona stigne do primaoca. Po prijemu, primalac izračunava heš vrednost primljene poruke i poredi je sa heš vrednošću koju je dobio preko sigurnog kanala. Primalac prihvata poruku samo ako su ove dve heš vrednosti identične. U slučaju da su ove dva heša različita primalac zaključuje da je autentičnost poruke narušena i da je poruka menjana na putu do odredišta. Da bi izvršio uspešan napad, napadač mora da pronađe poruku *M'* koja ima istu heš vrednost kao originalna poruka *M*, što bi trebalo da bude neizvodljivo ako se u ovom sistemu koristi jednosmerna heš funkcija ili funkcija otporna na kolizije [8, 10].

Autentifikacija poruke kombinovana sa šifrovanjem

Upotreba šifrovanja nije dovoljna da bi sačuvala integritet poruke, jer promenom bilo kog bita šifrata menja se i odgovarajući bit otvorenog teksta. Kod simetričnih blokovskih algoritama, promena jednog bita šifrata utiče na veliki deo otvorenog teksta i to na način koji ne može da se predvidi. Zštita integriteta takođe zavisi i od redundantnosti informacija u otvorenom tekstu. Ako u otvorenom tekstu ne postoji redundantnost, dešifrovanjem šifrata, bez obzira da li je modifikovan ili ne, dobija se otvoreni tekst koji ima smisla. Da bi se u otvoreni tekst dodala redundantnost mogu da se koriste kriptografske heš funkcije.



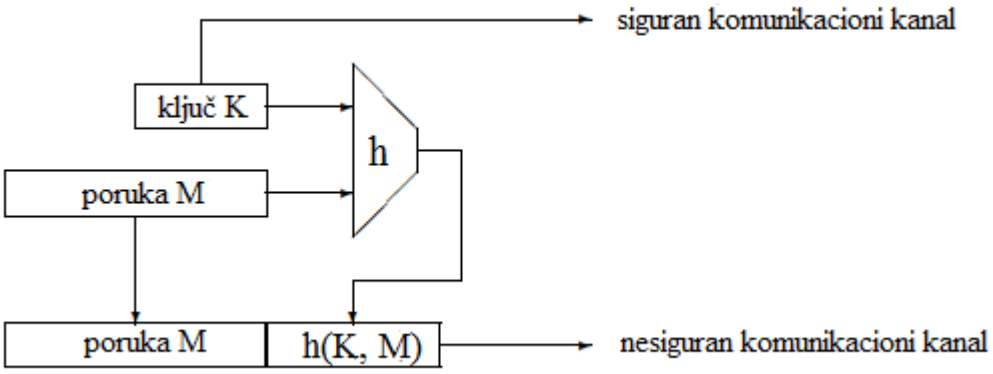
Slika 12. Autentifikacija poruke kombinovana sa šifrovanjem

Na **slici 12** prikazan je mehanizam prema kome pošiljalac koristi heš funkciju *h* da izračuna heš vrednost poruke *M* i onda tako dobijen heš dodaje na kraj originalne poruke generišući novu poruku *M* || *h(M)*. Potom se izvrši šifrovanje $E_K(M \| h(M))$ i ovako dobijeni šifrat se šalje preko nesigurnog kanala. Takođe, preko sigurnog kanala se šalje tajni ključ *K*. Na prijemnoj strani se vrši dešifrovanje i razdvajanje poslate heš vrednosti od originalne poruke *M*. Nakon ovoga ponovo se računa heš vrednost primljene poruke *M*, i akoje ova heš vrednost ista kao poslata heš vrednost dolazimo do zaključaka da je primljena poruka autentična [8]. Za napadača koji ne zna tajni kluč, nemoguće je da izmeni šifrat tako da izmenjen otvoreni tekst i dalje daje isti heš kao i originalna poruka *M* [8].

Da bi se realizovao mehanizam sa **slike 12** potrebno je da postoji siguran komunikacioni kanal. Kako se preko ovog kanala šalje tajni ključ potrebno je da se, pored autentičnosti, obezbedi i tajnost ovog kanala.

Autentifikacija poruke pomoću MAC algoritama

Kod ove vrste autentifikacije, pošiljalac koristi MAC algoritam sa tajnim ključem *K* da bi izvršio kompresiju poruke *M*. Rezultat kompresije je MAC vrednost *h(K, M)*. Ova vrednost se dodaje na kraj poruke *M* čime se dobija nova poruka *M* || *h(K, M)* koja se šalje preko nesigurnog komunikacionog kanala (**slika 13**). Istovremeno, preko sigurnog kanala šalje se tajni ključ *K*. Primalac vrši razdvajanje primljene poruke na poslatu MAC vrednost i na originalnu poruku *M*. Zatim se vrši ponovno računanje MAC vrednosti poruke *M* i ako je ova vrednost ista kao poslata MAC vrednost poruka se prihvata kao autentična.



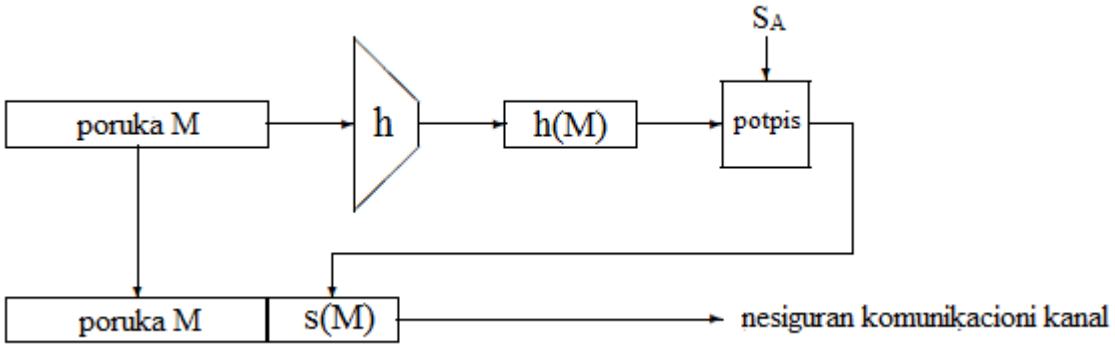
Slika 13. Autentifikacija poruke pomoću MAC algoritma

Da bi se ovaj mehanizam slanja poruke realizovao potrebno jee da postoji siguran komunikacioni kanal preko koga se šalje tajni ključ. Ovaj kanal mora da sačuva i tajnost i autentičnost poslatog ključa. Za napaača koji ne poznaje ključ *K*, nemoguće je da izmeni poruku *M* a da joj MAC vrednost i dalje ostane ista [8].

Razlika između ovog mehanizma za slanje poruke i mehanizma koji koristi heš funkciju je ta što ovaj mehanizam vreši slanje tajnog ključa, a ne slanje otiska poruke, pa je jako bitno da se za ovaj mehanizam obezbedi siguran komunikacioni kanal koji poseduje osobinu tajnosti.

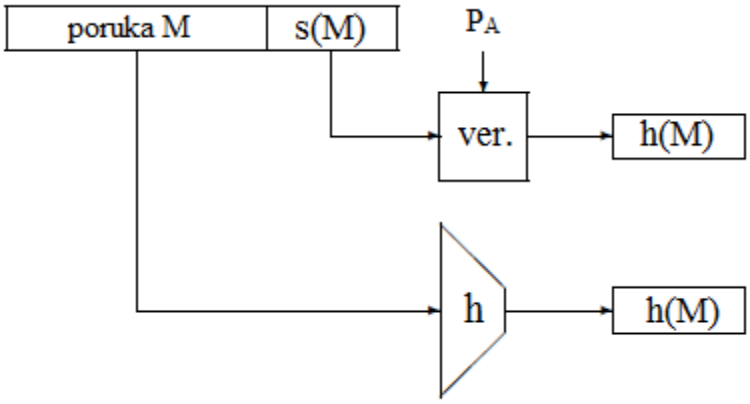
4.2 Primena u mehanizmu za digitalno potpisivanje

Digitalni potpis se koristi da bi se obezbedila autentičnost poruke i da bi se sprečila mogućnost da pošiljalac poruke u nekom kasnije trenutku negira da je baš on poslao tu poruku. Digitalni potpis se zasniva na kriptografiji sa javnim ključem. U kriptosistemima sa javnim ključem postoji javni ključ koji je poznat svim korisnicima tog sistema. Pored javnog ključa, svaki korisnik poseduje svoj tajni ključ koji je samo njemu poznat [2].



Slika 14. Mehanizam za digitalno potpisivanje u kombinaciji sa heš funkcijom

Pretpostavimo da imamo korisnika *A* koji poseduje par ključeva (S_A, P_A) , gde je S_A njegov tajni ključ, a P_A javni ključ. Prema algoritmu sa **slike 14.,** korisnik *A* prvo računa heš vrednost poruke *M*. Zatim se nad dobijenim hešom vrši digitalno potpisivanje pomoću tajnog ključa S_A i na taj način se dobija digitalni potpis *s(M)*. Digitalni potpis se dodaje na kraj poruke čime se dobija nova poruka $M \parallel s(M)$, koja se zatim šalje preko nesigurnog komunikacionog kanala [8].



Slika 15. Verifikacija digitalnog potpisa

Iz primljene informacije primalac vrši izdvajanje poruke i digitalnog potpisa. Zatim se nad tim digitalnim potpisom vrši verifikacija pomoću javnog ključa P_A. Kao rezultat verifikacije, dobija se ono što bi trebalo da bude heš vrednost poruke *M* u slučaju da prilikom prenosa nije došlo do nekog od napada. Ovako dobijena heš vrednost poredi se sa heš vrednošću primljene poruke, i ako su iste zaključuje se da je potpis originalan i da je poruka autentična.

Cilj upotrebe heš funkcija u digitalnom potpisu je da se smanji količina informacija koje se šalju preko kanala. Ukoliko poruku dužine 2 MB šifrujete nekim asimetričnim algoritmom (ovo su algoritmi koji imaju privatni i javni ključ.), dobićete šifrat dužine 2 MB. To znači da ćete nekom poslati duplo veću količinu podataka. Da bi se potpis sveo na razumnu dužinu, a da pritom ne izgubi svoj integritet, pošiljalac računa heš poruke koji je mnogo manji od same poruke, potpisuje ga svojim privatnim ključem i šalje svoju originalnu poruku i potpisani heš primaocu. Upotrebom heš funkcija se poboljšavaju i performanse digitalnog potpisivanja. Kako su asimetrični algoritmi veoma spori potpisivanjem heš vrednosti umesto čitave poruke znatno se smanjuje vreme koje je potrebno za potpisivanje [3].

Prilikom napada, napadač može da poruku *M* zameni porukom *M'*, ali sve dok mu je nepoznat tajni ključ kojim je potpisan heš poruke, on ne može da promeni potpis *s(M)*. Jedino što može da pokuša je da pronađe poruku *M'* koja ima istu heš vrednost kao originalna poruka *M*, ali to je računski neizvodljivo ako se koristi bezbedna heš funkcija.

4.3 Primena u mehanizmu za generisanje digitalnih vremenskih oznaka

Digitalne vremenske oznake (engl. Digital Timestamp) predstavljaju oznovni način za realizaciju *vremenske autentifikacije*. Preciznije, vremenske oznake predstavljaju dokaz da je dokument postojao pre vremenskog trenutka koji je zabeležen u vremenskoj oznaci. Vremenske oznake se najčešće koriste za zaštitu intelektualne svojine, na primer u situaciji kada neka firma želi da dokaže integritet svojih poslovnih podatak kako bi pokazala da nisu menjani retroaktivno.

Osnovno rešenje za generisanje vremenskih oznaka podrazumeva postojanje poverljivog ***centra za vremenske oznake*** (engl. Time Stamping Authority (TSA)). Klijent *C* koji želi vremensku oznaku za dokument *X*, prvo koristi heš funkciju h_1 da bi zračunao heš vrednost dokumenta. Potom se šalje zahtev *centru za vremenske oznake*. Ovaj zahtev sadrži identitet klijenta ID_c i heš vrednost $h_1(X)$. *Centar za vremenske oznake* izvrši dopunjavanje ovog zahteva tako što na kraj zahteva doda serijski broj i trenutno vreme. Ovako dopunjeni zahtev se digitalno potpisuje i na taj način se dobija vremenska oznaka koja se vraća klijentu: $vremenska\_oznaka = sign_{TSA}(ID_C , h_1(X), n, t)$. Oznake *n* i *t* predstavljaju serijski broj i vreme. Kanal između klijenta i *centra za sertifikate* mora da obezbedi integritet i autentičnost poruke, ali ne mora da obezedi i tajnost.

Ako klijent želi da izvrši validaciju vremenske oznake, on prvo mora da verifikuje potpis uz pomoć javnog ključa koji pripada *centru za sertifikate*. Pored ovoga, potrebno je proveriti da li heš vrednost koja se nalazi u okviru vremenske oznake odgovara dokumentu

kome je posmatrana vremenska oznaka dodeljena. Iz ovoga možemo da zaključimo da korišćena heš funkcija mora da poseduje *jaku otpornost na kolizije* [8].

4.4 Čuvanje lozinki na disku sistema

Jedan od glavnih problema sigurnosti je autentičnost korisnika. Zbog toga, korisnik pre korišćenja sistema mora da se identifikuje, odnosno da se predstavi. Nakon autentifikacije, sistem korisniku daje prava da koristi samo one resurse za čije korišćenje je ovlašćen.

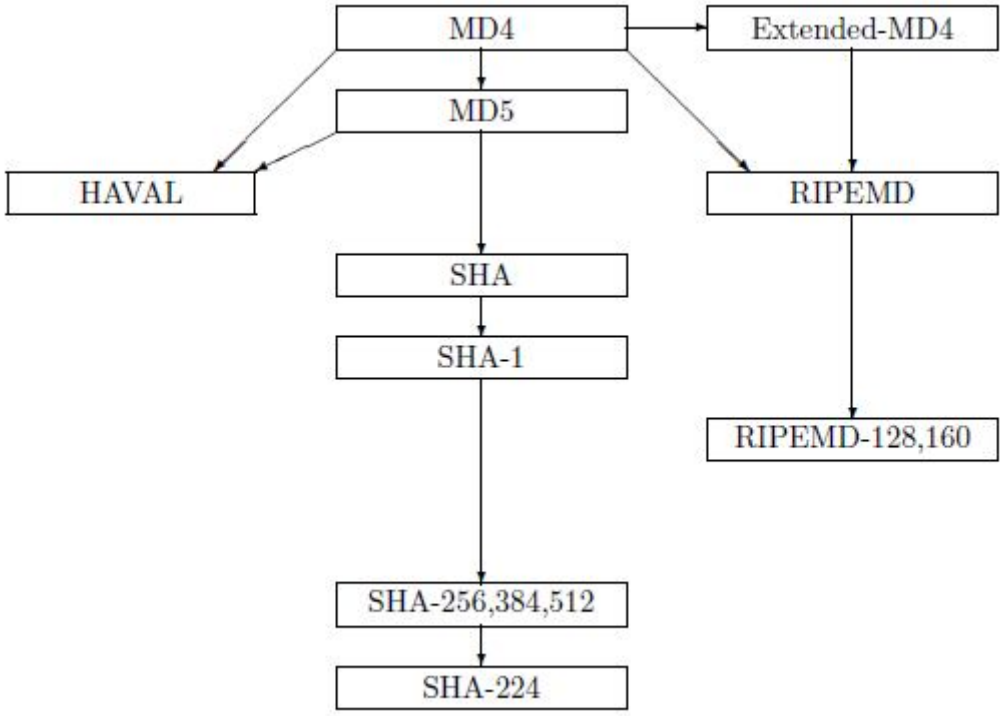
Najčešće autentifikacija se obavlja navođenjem poverljivih informacija, kao što je lozinka. Kod ove vrste autentifikacije, korisnik se, najpre, predstavi sistemu, odnosno navede svoje korisničko ime, a sistem zatim zahteva da korisnik navede svoju lozinku. Ako uneta vrednost lozinke odgovara vrednosti koja se nalazi na sistemu +, operativni sistem smatra da je korisnik prošao autentifikaciju. Problem predstavlja čuvanje lozinki na disku rašunarskog sistema. Ukoliko lozinke nisu dobro zaštićene, uljez može da dođe do svih lozinki koje se šuvaju na disku,uključujući i lozinke administratora. Da bi se ovo izbeglo informacije o lozinkama se obrađuju jednosmernim heš funkcijama. Prilikom prvog prijavljivanja na sistem, korisnik smišlja lozinku. Smeštanje lozinke na disk obavlja se na sledeći način:

- korisnik unosi novu lozinku l_a
- operativni sistem računa heš vrednost unete lozinke: $h_a=h(l_a)$
- dobijena vrednost se smešta na disk u odgovarajuću tabelu koju čine uređeni parovi (*korisničko ime , heš vrednost*)

Svaki sledeći put kada želi da se prijavi, korisnik navodi korisničko ime i lozinku. Sistem računa heš unete lozinke $h_b=h(l_b)$, u tabeli traži heš koji odgovara navedenom korisnočkom imenu i upoređuje ga sa dobijenom vrednošću. Ako je $h_a=h_b$ korisnik je autentifikovan [3].

5. Heš funkcije iz MD4 familije

Algoritmi iz MD4 familije su algoritmi koji imaju najveću praktičnu primenu. U osnovi ove familije algoritama se nalazi MD4 algoritam koji je 1990. godine kreirao R. Rivest. Ubrzo nakon kreiranja, ovaj algoritam se pokazao nedovoljno sigurnim što je dovelo do kreiranja novih algoritama u kojima su otklonjeni sigurnosni propusti koji su se nalazili u algoritmu MD4. Na **slici 16** je prikazan tok razvoja algoritama koji su zasnovani na MD4 algoritmu [8].



Slika 16. Istorija razvoja algoritama iz MD4 familije

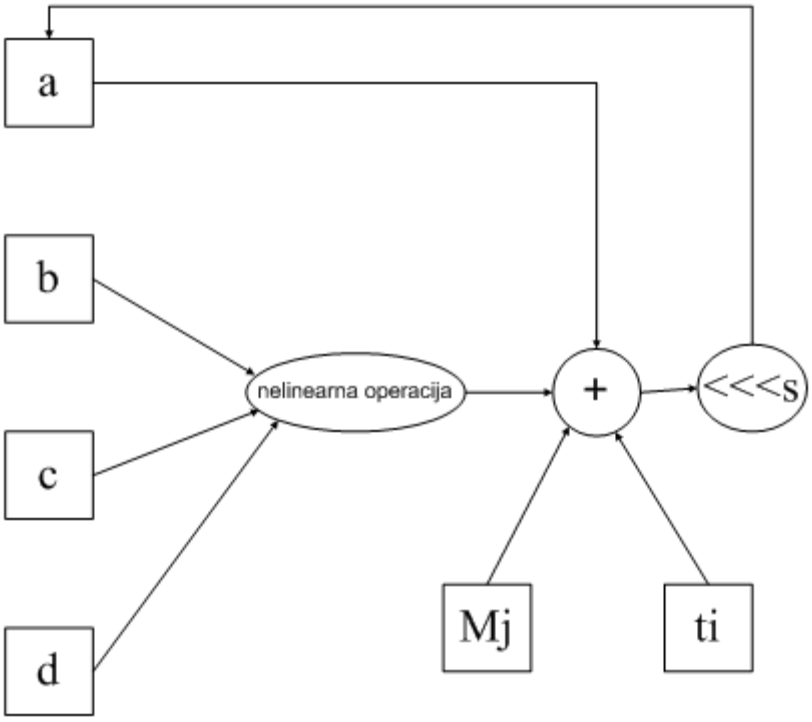
Jedni od najčešće korišćenih algoritmi iz ove familije su MD5, SHA-1 i RIPEMD-160. Zbog velike praktične primene detaljno ćemo analizirati ova tri algoritma zajedno sa MD4 algoritmom koji predstavlja temelj čitave familije. Pored analize, autor ovog rada je napravio i implementaciju ova četiri algoritama: MD4, MD5, SHA-1 i RIPEMD-160. Na ovoj implementaciji su izvršena merenja performansi algoritama. Rezultati merenja i poređenje performansi algoritama biće prikazana u **poglavlju 6.**

5.1 MD4 algoritam

MD4 (engl. **Message Digest**) je heš algoritam koji je 1990. godine dizajnirao Rivest za RSA Dat Security Inc. Izlazna vrednost ovog heš algoritma ima dužinu 128 bitova. MD4 je dizajniran za izvršavanje na 32-bitnim mašinama. MD4 se može besplatno koristi, tj. za njegovo korišćenje nije potrebna licenca. Ovaj algoritam je direktno uticao na dizajn algoritama kao što su: MD5, SHA-1 i RIPEMD-160 [3, 8].

Slabosti ovog algoritma prvi su uočili Den Boer i Bosselaers. Oni su u svom radu, koji je objavljen 1991. godine, opisali napad na na ovaj algoritam sa nedostatkom prve i poslednje runde. Zatim je, 1996. godine, Hans Dobbertin pokazao kako kolizija za kompletan MD4 može da se odredi za manje od minut vremena na prosečnom personalnom računaru.

MD4 obrađuje ulazni podatak u blokovima od 512 bita, tako što ga, prvo dopuni nizom 1000…000 do dužine ($n \times 512 - 64$) bita, pa zatim doda još 64 bita koji predstavljaju dužinu originalne poruke. Ukupna dužina dopunjene poruke je $n \times 512$ bita [4].



Slika 17. Jedna MD4 nelinearna operacija.

Nakon dopunjavanja vrši se inicijalizacija četiri 32-bitne promenljive A, B, C i D čija se vrednost dodatno upisuje u četiri pomoćne promenljive a, b, c i d nad kojima će se vršiti izračunavanja:

- A = a = 0x67452301
- B = b = 0xEFCDAB89
- C = c = 0x98BADCFE
- D = d = 0x10325476

Glavna petlja sadrži onoliko iteracija koliko dopunjeni ulazni podatak ima 512- bitnih blokova. Svaka iteracija se sastoji od tri runde u kojima se obrađuje jedan 512-bitni blok koji je podeljen na 16 32-bitnih podblokova. U svakoj rundi se izvršava 16 nelinearnih operacija, po jedna nad svakim od 16 podblokova (**slika 17**). M_j je j-ti 32-bitni podblok 512-bitnog bloka poruke. Konstanta t_i u prvoj rundi ima vrednost 0, u drugoj rundi ima vrednost prvih 32 bita kvadratnog korena od 2, a u trećoj rundi ima vrednost prvih 32 bita kvadratnog korena od 3. Operacija $<<<s$ predstavlja kružno pomeranje ulevo za s mesta. Operacija $\oplus$ predstavlja sabiranje po modulu 2^{32}. Prilikom deljenja 512-bitnih blokova u 32-bitne podblokove koristi se ***little-endian*** notacija, tj. viši bajt se smešta na višu adresu a niži bajt na nižu adresu.

Definisaćemo tri funkcije koje se izvršavaju na nivou bitova. Ove funkcije se koriste prilikom izvršavanja, gore pomenutih, nelinearnih operacija. To su funkcije F, G i H:

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

Nelinearne operacije u jednoj iteraciji algoritma se izvode prema sledećem rasporedu:

1. runda

a = (a + F(b,c,d) + M[0]) <<< 3
d = (d + F(a,b,c) + M[1]) <<< 7
c = (c + F(d,a,b) + M[2]) <<< 11
b = (b + F(c,d,a) + M[3]) <<< 19
a = (a + F(b,c,d) + M[4]) <<< 3
d = (d + F(a,b,c) + M[5]) <<< 7
c = (c + F(d,a,b) + M[6]) <<< 11
b = (b + F(c,d,a) + M[7]) <<< 19
a = (a + F(b,c,d) + M[8]) <<< 3
d = (d + F(a,b,c) + M[9]) <<< 7
c = (c + F(d,a,b) + M[10]) <<< 11
b = (b + F(c,d,a) + M[11]) <<< 19
a = (a + F(b,c,d) + M[12]) <<< 3

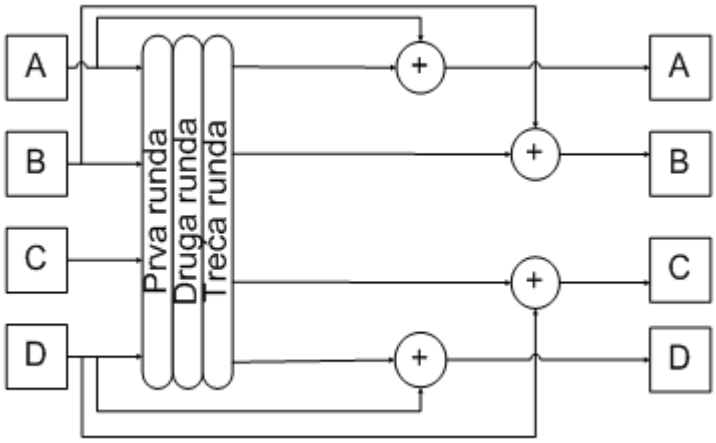
d = (d + F(a,b,c) + M[13]) <<< 7
c = (c + F(d,a,b) + M[14]) <<< 11
b = (b + F(c,d,a) + M[15]) <<< 19

2. runda

a = (a + G(b,c,d) + M[0] + 5A827999) <<< 3
d = (d + G(a,b,c) + M[1] + 5A827999) <<< 5
c = (c + G(d,a,b) + M[2] + 5A827999) <<< 9
b = (b + G(c,d,a) + M[3] + 5A827999) <<< 13
a = (a + G(b,c,d) + M[4] + 5A827999) <<< 3
d = (d + G(a,b,c) + M[5] + 5A827999) <<< 5
c = (c + G(d,a,b) + M[6] + 5A827999) <<< 9
b = (b + G(c,d,a) + M[7] + 5A827999) <<< 13
a = (a + G(b,c,d) + M[8] + 5A827999) <<< 3
d = (d + G(a,b,c) + M[9] + 5A827999) <<< 5
c = (c + G(d,a,b) + M[10] + 5A827999) <<< 9
b = (b + G(c,d,a) + M[11] + 5A827999) <<< 13
a = (a + G(b,c,d) + M[12] + 5A827999) <<< 3
d = (d + G(a,b,c) + M[13] + 5A827999) <<< 5
c = (c + G(d,a,b) + M[14] + 5A827999) <<< 9
b = (b + G(c,d,a) + M[15] + 5A827999) <<< 13

3.runda

a = (a + H(b,c,d) + M[0] + 6ED9EBA1) <<< 3
d = (d + H(a,b,c) + M[1] + 6ED9EBA1) <<< 9
c = (c + H(d,a,b) + M[2] + 6ED9EBA1) <<<11
b = (b + H(c,d,a) + M[3] + 6ED9EBA1) <<< 15
a = (a + H(b,c,d) + M[4] + 6ED9EBA1) <<< 3
d = (d + H(a,b,c) + M[5] + 6ED9EBA1) <<< 9
c = (c + H(d,a,b) + M[6] + 6ED9EBA1) <<< 11
b = (b + H(c,d,a) + M[7] + 6ED9EBA1) <<< 15
a = (a + H(b,c,d) + M[8] + 6ED9EBA1) <<< 3
d = (d + H(a,b,c) + M[9] + 6ED9EBA1) <<< 9
c = (c + H(d,a,b) + M[10] + 6ED9EBA1) <<< 11
b = (b + H(c,d,a) + M[11] + 6ED9EBA1) <<< 15
a = (a + H(b,c,d) + M[12] + 6ED9EBA1) <<< 3
d = (d + H(a,b,c) + M[13] + 6ED9EBA1) <<< 9
c = (c + H(d,a,b) + M[14] + 6ED9EBA1) <<< 11
b = (b + H(c,d,a) + M[15] + 6ED9EBA1) <<< 15



Slika 18. Jedna MD4 iteracija

Na kraju svake iteracije (**slika 18**) promenljive *a,b,c* i *d* se dodaju na promenljive A, B, C i D:

- A = A + a
- B = B + b
- C = C + c
- D = D + d

Krajnji rezultat algoritma se dobija nadovezivanjem promenljivih A, B, C i D koristeći notaciju ***little-endian.***

5.2 MD5 algoritam

Kada su se pojavile sumnje u sigurnost MD4 algoritma Ronald Rivest je 1991. godine dizajnirao algoritam MD5 koji je predstavljao zamenu za algoritam MD4. MD5 algoritam može besplatno da se koristi (nije potrebna licenca). Mane ovog algoritma prvi su uočili Den Boer i Bosselaers. Oni su 1993. Godine objasnili način kako mogu da se izazovu kolizije u algoritmu MD5. Godine 1996. Dobertin je takođe objasnio kako može da se dođe do kolizija u ovom algoritmu. Iako ovo nije bio pravi napad na algoritam, od tada se MD5 smatra nesigurnim algoritmom i preporučuje se upotreba drugih algoritama kao što su SHA1 i RIPEMD160 [3].

MD5 algoritam je veoma sličan algoritmu MD4 i predstavlja njegovu nadogradnju. Isto kao i MD4, MD5 obrađuje tekst u 512-bitnim blokovima, koji su podeljeni u 16 32-bitnih podblokova. Izlaz algoritma je 128-bitni heš, tj. četiri 32-bitna bloka. Inicijalna obrada obuhvata dopunjavanje poruke, tako što se poruka nastavlja nizom 1000…0000 do dužine

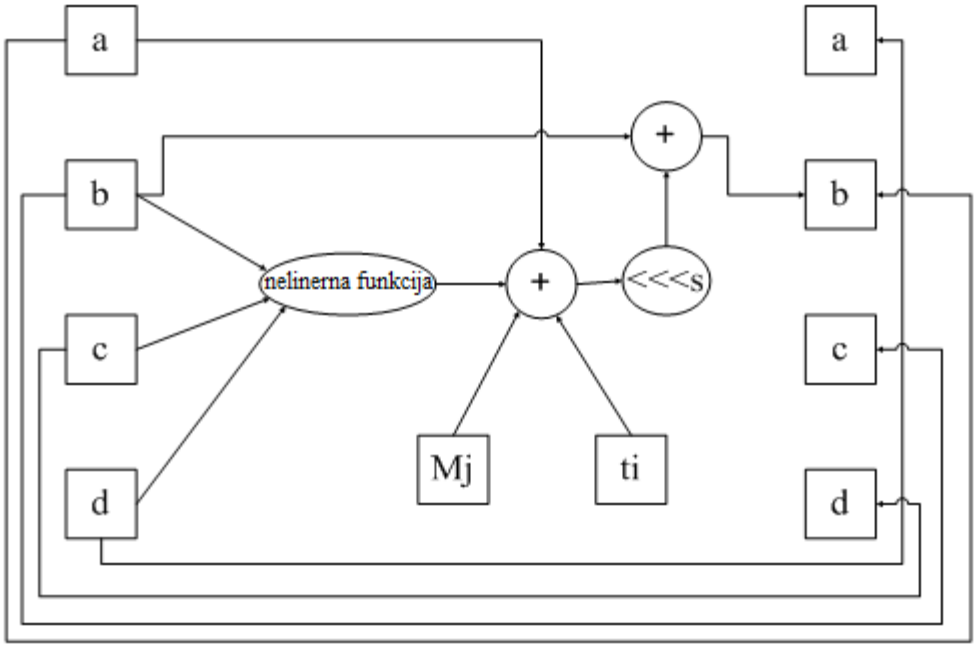
n*×*512-64 bita. Na kraju se dodaju 64 bita koji predstavljaju dužinu poruke. Ukupna dužina dopunjene poruke je $n \times 512$ bita [3]. Nakon dopunjavanja inicijalizuju se četiri promenljive A, B, C i D čija se vrednost dodatno upisuje u četiri promenljive a, b, c i d nad kojima će se izvršavati nelinearne operacije:

- A = a = 0x67452301
- B = b = 0xEFCDAB89
- C = c = 0x98BADCFE
- D = d = 0x10325476

Glavna petlja se ponavlja onoliko puta koliko dopunjena poruka ima ima 512-bitnih blokova. Svaka iteracija sastoji se od četiri runde. U svakoj rundi se izvršava 16 nelinearnih operacija (u svakoj rundi različita operacija) nad tri od četiri promenljive a, b, c i d. Jedna nelinearna MD5 operacija je prikazana na **slici 19**. M_j je *j*-ti podblok 512-bitnog bloka poruke. Konstanta t_i ima vrednost prvih 32 bita binarne vrednosti izraza *abs(sin(i+1)); 0≤i≤63;* pri čemu je *i* izraženo u radijanima [3, 4]. Definisaćemo i četiri funkcije koje se izvršavaju na nivou bitova:

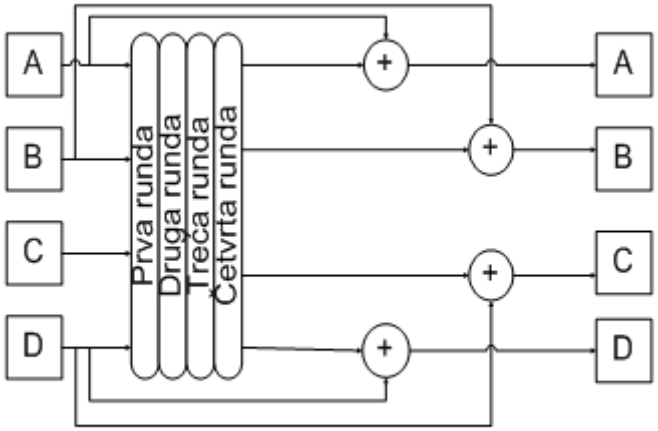
- F(X,Y,Z) = $(X \wedge Y) \vee ((\neg X) \wedge Z)$
- G(X,Y,Z) = $(X \wedge Y) \vee (Y \wedge (\neg Z))$
- H(X,Y,Z) = $X \oplus Y \oplus Z$
- I(X,Y,Z) = $Y \oplus (X \vee (\neg Z))$

Ove funkcije se koriste prilikom izračunavanja, već pomenutih, nelinearnih operacija.



Slika 19. Jedna nelinearna MD5 operacija

Nakon jedne iteracije (**slika 20**) promanljive a, b, c i d se dodaju na A, B, C i D respektivno, i algoritam nastavlja rad sa sledećim blokom podataka. Izlaz algoritma je konkatenacija promenljivih A, B, C i D u ***little-endian*** notaciji.



Slika 20. Jedna MD5 iteracija

Algoritam MD5 je opisan sledećim pseudokodom [12]:

```
//sve promenljive su predstavljene kao neoznačeni celi brojevi i //sva izračunavanja se vrše po modulu 2^32
var int[64] r, k

//r nam govori za koliko se mesta vrši kružno pomeranje
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,
7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,
5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,
4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,
6, 10, 15, 21}

//Konstante :
for i from 0 to 63
    t[i] := int(abs(sin(i + 1)) × (2 pow 32))

//Initialize variables:
var int A := 0x67452301
var int B := 0xEFCDAB89
var int C := 0x98BADCFE
var int D := 0x10325476

//Inicijalna obrada:
```

```
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit length of unpadded message as 64-bit little-endian

integer to message

//Deljenje poruke u 512-bitne blokove:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit little-endian words M[i],
0 ≤ i ≤ 15

    //Inicijalizacija pomoćnih promenljivih:
    var int a := A
    var int b := B
    var int c := C
    var int d := D

    //Glavna petlja:
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then  //1. runda
            f := (b and c) or ((not b) and d) //F(b,c,d)
            g := i
        else if 16 ≤ i ≤ 31   //2.runda
            f := (d and b) or ((not d) and c)  //G(b,c,d)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47    //3.runda
            f := b xor c xor d     //H(b,c,d)
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63      //4.runda
            f := c xor (b or (not d))  //I(b,c,d)
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
        //Nelinearna operacija:
        b := b + leftrotate((a + f + t[i] + M[g]) , r[i])
        a := temp
    A := A + a
    B := B + b
    C := C + c
    D := D + d

//Računanje heša(engl. digest) u little-endian notaciji
var int digest := h0 append h1 append h2 append h3
```

Poboljšanja MD5 u odnosu na MD4 algoritam su sledeća:

- MD5 ima jednu rundu više po iteraciji
- u svakom koraku se koristi jedinstvena aditivna konstanta t_i
- umesto funkcije G(X,Y,Z) = $(X \wedge Y) \vee (X \wedge Y) \vee (Y \wedge Z)$, radi smanjenja simetričnosti koristi se funkcija G(X,Y,Z) = $(X \wedge Y) \vee (Y \wedge (\neg Z))$
- efekat lavine je ubrzan korišćenjem rezultata iz prethodnih koraka
- redosled pristupanja podblokovima u drugoj i trećoj rundi je izmenjen kako bi se izbegle sličnosti

5.3 SHA-1 algoritam

Američki institut za standarde i tehnologiju (NIST) i nacionalna agencija za bezbednost (NSA) sastavili su **SHA** (engl.**Secure Hash Algorithm**) familiju algoritama. U SHA familiju algoritama spadaju SHA-0, SHA-1 i SHA-2 algoritmi. Od svih SHA algoritama najviše se koristi SHA-1, koji je ugrađen u veliki broj sigurnosnih aplikacija i protokola. SHA-1 daje heš vrednost dužine 160 bita. Godine 2005. Uočeni su sigurnosni nedostaci SHA-1 algoritma, tj. utvrđeno je da postoje određene matematičke slabosti koje bi se mogle iskoristiti u napadu. Iako još nije sproveden nijedan napad preporučuje se prelazak na neki sigurniji algoritam. Kako je za SHA-0 već pronađen način za dobijanje kolizije, i kako je SHA-2 veoma sličan algoritmu SHA-1 u toku je razvoj novog algoritma SHA-3 [2, 3].

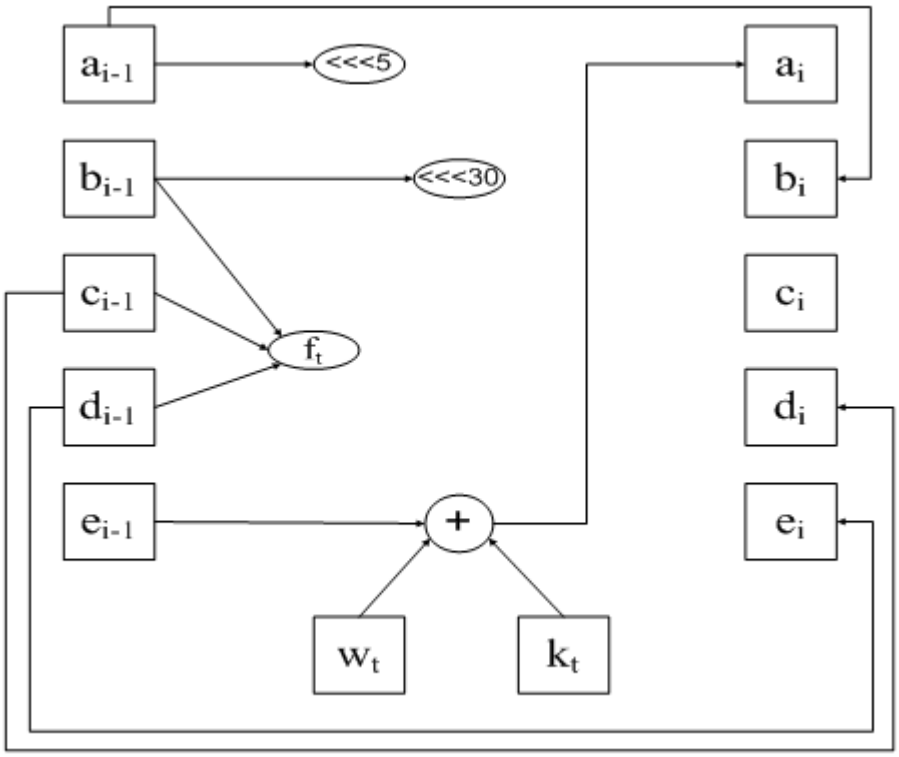
SHA-1 je zasnovan na istoj ideji kao i MD4 algoritam. Za razliku od MD4 algoritma SHA-1 koristi **big-endian** notaciju (viši bajt se čuva na nižoj adresi, a niži bajt se čuva na višoj adresi). SHA-1 algoritam daje heš dužine 160 bita. Inicijalna obrada ista je kao kod MD4 i MD5 algoritama: poruka se nastavlja nizom 1000…000do dužine **n×512-64** bita. Na kraju se dodaje 64 bita koji predstavljaju dužinu poruke. Ukupna dužina poruke je **n×512** bita. Nakon dopunjavanja poruke vrši se inicijalizacija pet 32-bitnih promenljivih A, B, C, D i E (jedna promenljiva više u odnosu na MD5 zato što SHA-1 proizvodi heš od 160 bita). Ista inicijalizacija se vrši i nad pomoćnim promenljivima a, b, c, d i e:

- A = a = 0x67452301
- B = b = 0xEFCDAB89
- C = c = 0x98BADCFE
- D = d = 0x10325476
- E = e = 0xC3D2E1F0

Definisaćemo SHA skup nelinearnih operacija (vrednost *t* označava redni broj operacije u rundi):

- f_t(X,Y,Z) = $(X \wedge Y) \vee ((\neg X) \wedge Z)$, za t ∈ [0, 19]
- f_t(X,Y,Z) = $X \oplus Y \oplus Z$, za t ∈ [20, 39]

- $f_t(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$, za $t \in [40, 59]$
- $f_t(X,Y,Z) = X \oplus Y \oplus Z$, za $t \in [60, 79]$



Slika 21. Jedna iteracija SHA-1 algoritma

Definisaćemo još i vrednosti konstante k_t:

- $k_t = 2^{1/2} \cdot 2^{32} / 4 = 0x5A827999$, za $t \in [0, 19]$
- $k_t = 3^{1/2} \cdot 2^{32} / 4 = 0x6ED9EBA1$, za $t \in [20, 39]$
- $k_t = 5^{1/2} \cdot 2^{32} / 4 = 0x8F1BBCDC$, za $t \in [40, 59]$
- $k_t = 10^{1/2} \cdot 2^{32} / 4 = 0xCA62C1D6$, za $t \in [60, 79]$

Glavna petlja se ponavlja onoliko puta koliko podatak sadrži 512-bitnih blokova. Svaka iteracija ima četiri runde. U svakoj rundi se obavlja 20 operacija nad tri od pet promenljivih *a, b, c, d* i *e*, a zatim se radi pomeranje i sabiranje slično kao kod MD5 algoritma.

Na početku algoritma 512-bitni blok se prvo podeli na skup od 16 32-bitnih reči (M_0 do M_{15}), a zatim se ovaj skup proširi u skup od 80 32-bitnih reči (W_0 do W_{79}) na sledeći način:

- $W_t = M_t$, za $t \in [0, 15]$
- $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1$, za $t \in [16, 79]$

Nakon jedne iteracije (obrađen blok od 512 bita), promanljive *a*, *b*, *c*, *d* i *e* se dodaju na *A*, *B*, *C*, *D* i *E* respektivno, i algoritam nastavlja rad sa sledećim blokom podataka. Izlaz algoritma je 160-bitna konkatenacija promenljivih *A*, *B*, *C*, *D* i *E* (u **big-endian** notaciji). Način rada SHA-1 algoritma prikazan je sledećim pseudokodom [13]:

```
// Sve promenljive su neoznačeni celobrojni brojevi i sva
//izračunavanja se vrše po modulu 2^32 .
// Sve konstante su zapisane big-endian notacijom.

//Inicijalizacija:
A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0x10325476
E = 0xC3D2E1F0

//Dopunjavanje:
append the bit '1' to the message
append k bits '0', where k is the minimum number ≥ 0 such
that the resulting message
    length (in bits) is congruent to 448 (mod 512)
append length of message (before pre-processing), in bits, as
64-bit big-endian integer

//Deljenje poruke u 512-bitne blokove i u 32-bitne
podblokove:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0
≤ i ≤ 15

    //Proširivanje 16 32-bitnih reči u 80 32-bitnih reči:
    for i from 16 to 79
  w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-
16])leftrotate 1

    //Inicijalizacija pomoćnih promenljivih:
    a = A
    b = B
    c = C
    d = D
    e = E

    //Glavna petlja:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then       //1. runda
```

```
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39          //2. runda
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59          //3. runda
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79          //4. runda
            f = b xor c xor d
            k = 0xCA62C1D6

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


    A = A + a
    B = B + b
    C = C + c
    D = D + d
    E = E + e

//Računanje izlazne heš vrednosti (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4
```

5.4 RIPEMD-160 algoritam

RIPEMD (engl. RACE Integrity Primitives Evaluation Message Digest) je klasa kriptografskih heš funkcija kreiranih u Evropi. Ove funkcije su kreirali *Hans Dobbertin, Antoon Bosselaers i Bart Preneel.* Prve algoritmi iz ove klase su kreirane 1996. godine. Ovi algoritmi su kreirani na osnovu MD4 algoritma i veoma su slični algoritmima iz SHA klase, kako po načinu rada tako i po brzini izvršavanja. Ova klasu algoritama čine algoritmi RIPEMD, RIPEMD-128, RIPEMD-160, RIPEMD-256, i RIPEMD-320 (broj u imenu označava dužinu heš vrednosti koju generišu). RIPEMD je prvi algoritam koji je kreiran iz ove klase algoritama i davao je heš dužine 128 bita. Ovaj algoritam se više ne koristi zato što je pronađen način da se kod ovog algoritma izazovu kolizije. Najčešće korišćeni algoritam iz ove klase je RIPEMD-160. U RIPEMD-160 algoritmu još uvek nisu pronađeni nedostaci i pretpostavlja se da će biti siguran narednih 10 godina. RIPEMD-160 je nepatentiran i slobodan algoritam [3].

Svaka iteracija RIPEMD-160 algoritma ima pet rundi. Ovaj algoritam koristi **little- endian** notaciju. Osnovna razlika između RIPEMD-160 i algoritama iz familije MDx je u tome što RIPEMD-160 vrši obradu ulaznog podatka u dve linije koje se paralelno izvršavaju. Ovo izgleda kao da su startovana dva MD5 algoritma koji rade paralelno. Inicijalna obrada je ista kao kod svih prethodno obrađenih algoritama. Poruka se nastavlja nizom 1000…0000 do dužine **n×512-64** bita, i zatim se dodaju još 64 bita koji predstavljaju dužinu originalne poruke. Nakon ovoga, dopunjena poruka se deli u *t* 16-bitnih blokova koje ćemo označavati sa $X_i[j]$ gde je $0 \le i \le t-1$ i $0 \le i \le 15$.

Nakon dopujavanja, vrši se inicijalizacija pet promenljivih *A, B, C, D* i *E,* kao i pomoćnih promenljivih a,b,c,d,e,a',b',c',d' i e' nad kojima se u rundama glavne petlje izvršavaju izračunavanja:

- A = a = a' = 0x67452301
- B = b = b' = 0xEFCDAB89
- C = c = c' = 0x98BADCFE
- D = d = d' = 0x10325476
- E = e = e' = 0xC3D2E1F0

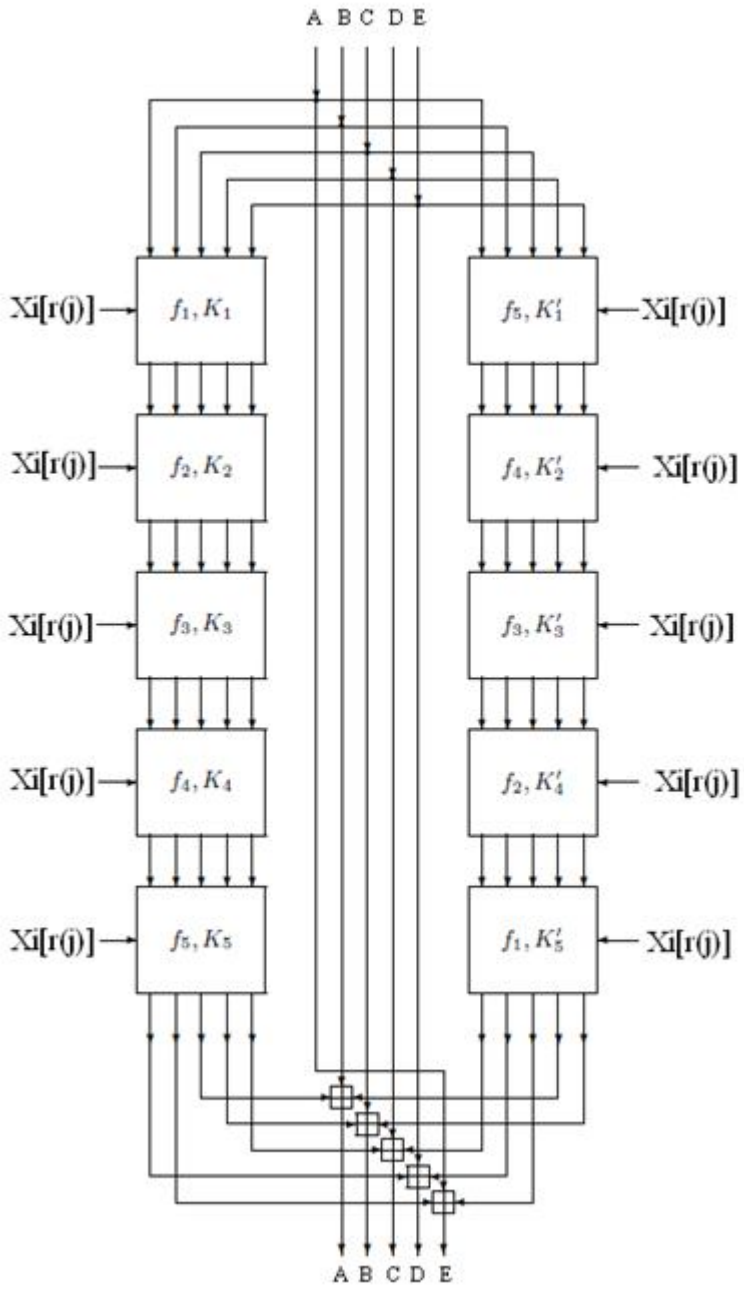
Glavna petlja algoritma se izvršava sve dok postoje 512-blokovi ulaznog podatka Svaka iteracija se sastoji od pet rundi. U svakoj rundi izvršava se 16 nelinearnih operacija (u svakoj rundi različita operacija). Izlaz algoritma se dobija konkatenacijiom promenljivih *A, B, C, D* i *E* u **little-endian** notaciji [4].

U daljem tekstu promenljiva *j* će označavati broj koraka unutar jedne iteracije. Definisaćemo pet funkcija koje se izvršavaju bit po bit i koje se koriste u odgovarajućim rundama:

- $f_1 = f(j,X,Y,Z) = X \oplus Y \oplus Z$, za $0 \le j \le 15$ (1. runda)
- $f_2 = f(j,X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$, za $16 \le j \le 31$ (2. runda)
- $f_3 = f(j,X,Y,Z) = (X \vee (\neg Y)) \oplus Z$, za $32 \le j \le 47$ (3. runda)
- $f_4 = f(j,X,Y,Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$, za $48 \le j \le 63$ (4. runda)
- $f_5 = f(j,X,Y,Z) = X \oplus (Y \vee (\neg Z))$, za $64 \le j \le 79$ (5. runda)

Vrednosti aditivnih konstanti koje se koriste za izračunavanja u prvoj liniji algoritma su sledeće:

- K(j) = 0x00000000, za $0 \le j \le 15$
- K(j) = 0x5A827999, za $16 \le j \le 31$, $[2^{30} \cdot \sqrt{2}]$
- K(j) = 0x6ED9EBA1, za $32 \le j \le 47$, $[2^{30} \cdot \sqrt{3}]$
- K(j) = 0x8F1BBCDC, za $48 \le j \le 63$, $[2^{30} \cdot \sqrt{5}]$
- K(j) = 0xA953FD4E, za $64 \le j \le 79$, $[2^{30} \cdot \sqrt{7}]$



Slika 22. Prikaz RIPEMD-160 algoritma

Vrednosti aditivnih konstanti koje se koriste za izračunavanja u drugoj liniji algoritma su sledeće:

- $K'(j) = 0x\ 50A28BE6, \quad za \quad 0 \le j \le 15, \quad [2^{30} \cdot \sqrt[3]{2}]$

- $K'(j) = 0x5C4DD124, \quad za \quad 16 \le j \le 31 , \quad [2^{30} \cdot \sqrt[3]{3}]$
- $K'(j) = 0x6D703EF3, \quad za \quad 32 \le j \le 47 , \quad [2^{30} \cdot \sqrt[3]{5}]$
- $K'(j) = 0x7A6D76E9, \quad za \quad 48 \le j \le 63 , \quad [2^{30} \cdot \sqrt[3]{7}]$
- $K'(j) = 0x00000000, \quad za \quad 64 \le j \le 79$

Prilikom obrade jednog 512-bitnog bloka on se deli na 16 32-bitnih podblokova. Redosled kojim se ovi podblokovi uzimaju na obradu u prvoj liniji algoritma je sledeći:

- $r(j) = j ; \ (0 \le j \le 15)$
- $r(16..31) = 7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8$
- $r(32..47) = 3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12$
- $r(48..63) = 1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2$
- $r(64..79) = 4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13$

Redosled kojim se 32-bitni podblokovi uzimaju na obradu u drugoj liniji algoritma je sledeći:

- $r'(0..15) = 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12$
- $r'(16..31) = 6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2$
- $r'(32..47) = 15, 5, 1, 3, 7, 14\ 6, 9, 11, 8, 12, 2, 10, 0, 4, 13$
- $r'(48..63) = 8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14$
- $r'(64..79) = 12, 15, 10, 4, 1, 5, 8, 7\ ,6, 2, 13, 14, 0, 3, 9, 11$

Broj mesta za koji se vrši kružno pomeranje ulevo u prvoj liniji algoritma u zavisnosti od koraka u kome se nalazimo je sledeći:

- $s(0..15) = 11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8$
- $s(16..31) = 7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11\ 7, 13, 12$
- $s(32..47) = 11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5$
- $s(48..63) = 11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12$
- $s(64..79) = 9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6$

Broj mesta za koji se vrši kružno pomeranje ulevo u drugoj liniji algoritma u zavisnosti od koraka u kome se nalazimo je sledeći:

- $s'(0..15) = 8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6$
- $s'(16..31) = 9, 13, 15, 7, 12, 8\ ,9, 11, 7, 7, 12, 7, 6, 15, 13, 11$
- $s'(32..47) = 9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5$
- $s'(48..63) = 15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15\ 8$
- $s'(64..79) = 8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11$

Našin rada algoritma nad ulaznim podatkom koji se sastoji od *t* blokova opisan je **slikom 22**. i sledećim pseudokodom:

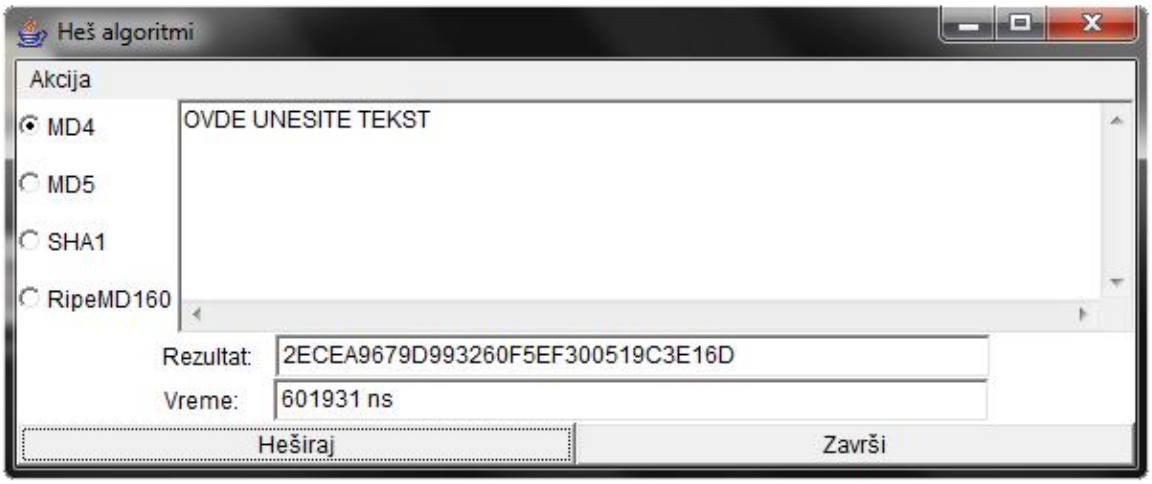
```
for i := 0 to t-1 {

    a := A; a := B; c := C; d = D; e = E;
    a' := A; b' := B; c' := C; d' = D; e' = E;

    for  j := 0 to 79 {

        T := rol_s(j)  ( a + f(j,b,c,d) + X_i[r(j)] + K(j)) + e ;
        a := e; e := d; d := rol_10(c); c := b; b := T;

        T := rol_s'(j)  ( a' + f(79-j,b',c',d') + X_i[r'(j)] + K'(j)) + e' ;
        a' := e'; e' := d'; d' := rol_10(c'); c' := b'; b' := T;

     }

    T := B + c = d';   B := C + d + e';   C := D + e + a';
    D := E +a + b';    E := A + b + c';   A := T;

}
```

U navedenom algoritmu operacija sabiranja se vrši po modulu 2^{32}. Operacija $\boldsymbol{rol_x}$ vrši kružno pomeranje ulevo za ***x*** mesta,.

6. Merenja i poređenja performansi kriptografskih heš funkcija MD4, MD5, SHA-1 i RIPEMD-160

U ovom poglavlju biće prikazani rezultati merenja koji sudobijeni korišćenjem aplikacije koju je autor ovog rada napravio i u kojoj je implementirao četiri heš algoritma: *MD4, MD5, SHA-1* i *RIPEMD-160*. Aplikacija je napravljena u programskom jeziku Java, pri čemu je korišćena ista tehnika implementacije ta sva četiri algoritma. Sva merenja su izvršena na procesoru *Intel Pentium T4400 (2,2 GHz)* i na operativnom sistemu W*indows 7 Ultimate*.

Korisnik može da koristi aplikaciju ili upotrebom grafičkog interfejsa ili pozivanjem komandi kroz komandnu liniju. Kada koristi grafički interfejs, korisnik u odgovarajuće tekst polje unosi tekst poruke koju želi da heširа i pritiskom na odgovarajuće radio dugme bira heš funkciju koju želi da koristi. Nakon toga, pritiskom na dugme *Heširaj* vrši se pokretanje izabrane heš funkcije. Dobijena heš vrednost se upisuje u tekst polje *Rezultat*, a vreme koje je bilo potrebno za izvršavanje heš funkcije se upisuje u tekst polje *Vreme*. Izgled grafičkog interfejsa aplikacije je prikazan na **slici 23.** Kako je ovaj deo aplikacije namenjen heširanju poruka manje veličine vreme se prikazuje u nano sekundama (ns) radi postizanja što veće preciznosti merenja.



Slika 23. Izgled grafičkog interfejsa aplikacije za heširanje

Kada se aplikacija poziva preko komandne linije, korisnik mora da pozove funkciju *main* iz klase *HashCmd.java* i da joj kao argumente prosledi oznaku za heš funkciju koju želi da koristi i putanju do fajla koji se heširа. Oznake koje se koriste za heš funkcije su sledeće: *md4*, *md5*, *sha1* i *ripemd160*. Na primer, prilikom heširanja fajla *C:\Test.txt* koristeći

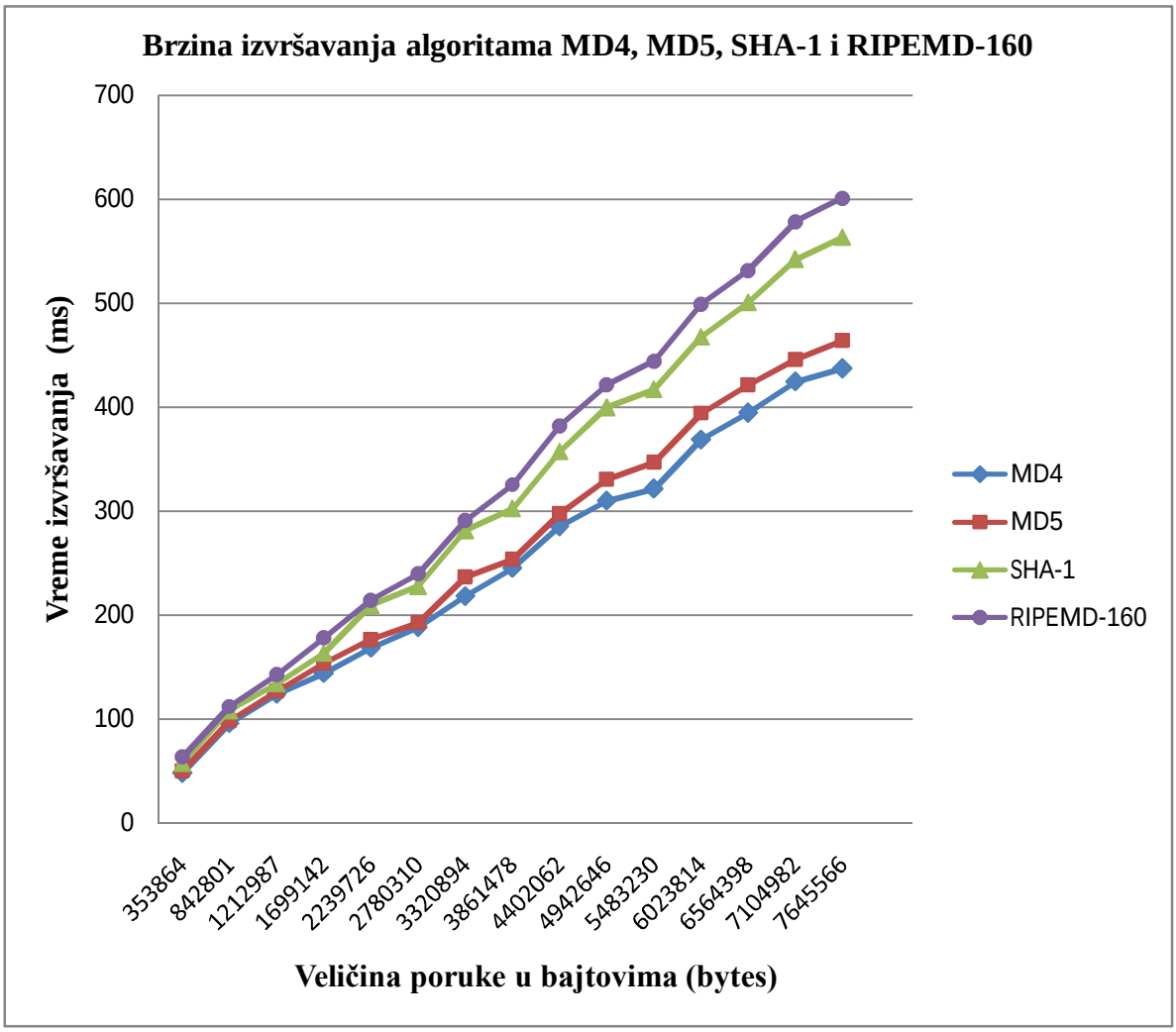
algoritam *MD4* mormo da pozovome komandu: *% java HashCmd md4 C:\Test.txt.* Nakon pozivanja ove komande ispisuje se rezultat heširanja i vreme izvršavanja heš algoritma. Deo aplikacije koji se poziva kroz komandnu liniju je namenjen heširanju fajlova neograničene veličine, pa je odabrano da se vreme potrebno za heširanje fajla prikazuje u mili sekundama (ms).

Za svaki od implementiranih heš algoritama merenja su vršena nad 15 poruka različite dužine. Za svaku od poruka rađeno je po 5, a potom se računala srednja vrednost ovih merenja i to sa ciljem da se dobije što tačnije vreme izvršavanja algoritama. U **tabeli 1** su prikazani dobijeni rezultati merenja. Ova tabela prikazuje vreme potrebno za heširanje poruke u zavisnosti od izabranog algoritma i dužine poruke. Dužina poruke je prikazana u bajtovima (bytes), a vreme u mili sekundama (ms).

Veličina poruke u bajtovima	Heš algoritmi			
	MD4	MD5	SHA-1	RIPEMD-160
353864	48,27 ms	50,09 ms	57,66 ms	63, 44 ms
842801	95,88 ms	98,38 ms	108,16 ms	112,02 ms
1212987	124 ms	126,76 ms	134,12 ms	142,95 ms
1699142	144,03 ms	153,87 ms	163,44 ms	178,08 ms
2239726	168,53 ms	176,31 ms	209,46 ms	214,30 ms
2780310	188,34 ms	192,84 ms	227,74 ms	239,67 ms
3320894	218,09 ms	236,69 ms	281,54 ms	291,02 ms
3861478	245,26 ms	253,85 ms	302,34 ms	325,49 ms
4402062	285,41 ms	297,50 ms	357,36 ms	382,05 ms
4942646	310,02 ms	330,58 ms	399,70 ms	421,46 ms
5483230	321,55 ms	346,72 ms	417,09 ms	444,22 ms
6023814	368,78 ms	394,20 ms	467,46 ms	499,02 ms
6564398	394,53 ms	421,50 ms	500,42 ms	531,49 ms
7104982	424,65 ms	445,63 ms	542,22 ms	578,41 ms
7645566	437,15 ms	464,01 ms	563,33 ms	601,04 ms

Tabela 1. Vreme izvršavanja heš algoritama u zavisnosti od veličine poruke

Na osnovu rezultata merenja napravljen je **grafik 1** koji prikazuje zavisnost vremena izvršavanja algoritama u zavisnosti od veličine poruke. Takođe, na osnovu dobijenih rezultata izračunate su i srednje brzine izvršavanja svakog od algoritama. Ove brzine su izražene u Mbit/s i prikazane su **tabeli 2.**



Grafik 1 Brzina izvršavanja implementiranih heš algoritama

Heš algoritmi	Brzina izvršavanja izražena u Mbit/s
MD4	118,20
MD5	111,85
SHA-1	94,28
RIPEMD-160	88,79

Tabela 2. Srednja brzina izvršavanja implementiranih algoritama izražena u Mbit/s

Posmatranjem **grafika 1** dolazimo do zaključka da vreme izvršavanja svakog od ovih algoritama, sa manjim odstupanjima, linearno zavisi od veličine ulazne poruke koja se obrađuje. Ova linearnost je posledica iterativne strukture implementiranih algoritama, koji poruku obrađuju tako što je dele na 512-bitne blokove. Sa povećanjem veličine poruke linearno se povećava broj blokova unutar te poruke, a samim tim se linearno povećava i vreme potrebno za izvršavanje algoritma. Do malih odstupanja u linearnosti dolazi zbog različitog broja iteracija koje se izvršavaju u funkciji koja vrši *MD ojačavanje,* jer različite poruke zahtevaju dopunjavanje sa različitim brojem bitova. Funkcija koja vrši *MD ojačavanje,* izvršava dopunjavanje ulazne poruke bajt po bajt sve dok njena dužina ne postane celobrojan umnožak broja 512.

Takođe može da se desi da poruke različite dužine, nakon dopunjavanja imaju istu dužinu, što je drugi razlog koji dovodi do odstupanja u linearnosti. Na primer, ako imamo dve poruke dužine 800 i 900 bita, nakon dopunjavanja obe imaju dužinu 1024 bita, pa je posle dopunjavanja, vreme izvršavanja posmatranog algoritma isto za obe poruke iako one inicijalno imaju različite dužine.

Treći razlog za odstupanje od linearnosti je uticaj Java virtualne mašine i operativnog sistema na rad aplikacije. Na primer, u toku izvršavanja aplikacije, operativni sistem može na kratko da joj oduzme procesor i dodeli ga nekom drugom procesu, što dovodi do uvećavanja vremena izvršavanja aplikacije. Sa obzirom na brzinu procesora koji je ovde korišćen, možemo da smatramo da je zanemarljiv uticaj Java virtualne mašine i operativnog sitema na rezultat merenja

Posmatrajući srednje brzine izvršavanja algoritama iz **tabele 2** vidimo da je algoritam *MD4* najbrži, a algoritam *RIPEMD-160* najsporiji. Algoritam *MD5* je 5,4% sporiji od algoritma *MD4*. *SHA-1* je 20,2% sporiji od *MD4* i 15,7% sporiji od *MD5* algoritma. Algoritam *RIPEMD-160* je 24,9% sporiji od *MD4*, od *MD5* je sporiji *20,6%* dok je od *SHA-1* sporiji za 5,8%. Svi ovi prikazani rezultati se u potpunosti slažu sa samom strukturom posmatranih algoritama. Algoritam *MD4* obrađuje blokove ulazne poruke u 3 runde od po 16 nelinearnih operacija, dok *MD5* blokove ulazne poruke obrađuje u četiri runde od po 16 nelinearnih operacija, pa je očekivano da se zbog jedne runde više algoritam *MD5* sporije izvršava od algoritma *MD4*. Algoritam *SHA-1* obrađuje blokove u 4 runde od po 20 nelinearnih operacija, a algoritam *RIPEMD-160* obrađuje blokove u dve paralelne linije izračunavanja od kojih svaka ima 5 rundi od po 16 nelinearnih operacija. Zbog ovakve strukture algoritmi *SHA-1* i *RIPEMD-160* su najsporiji, sa tim što *RIPEMD-160* ima ubedljivo najmanju brzinu.

7. Zaključak

Cilj ovog rada je da ukaže na pomalo skrivenu i u javnosti malo poznatu oblast kriptografije. Kriptografske heš funkcije igraju ključnu ulogu u očuvanju integriteta poruka, u autentifikaciji učesnika u komunikaciji i u ostalim sigurnosnim aplikacijama. Sa razvojem računara i računarskih mreža, kao i sa povećanjem načina za njihovu primenu, potreba za autentifikacijom i očuvanjem poruka postaje sve veća i veća. Kako veliki deo komunikacije između dva ili više lica treba da ostane tajna za ostale korisnike mreže,a da pritom kanali veze nisu sigurni od prisluškivanja i drugih napada, a resursi na kojima se smeštaju podaci su često deljeni, onda je značaj kriptografskih heš funkcija veoma veliki, posebno u oblastima kao što je ekonomija.

Jedna od namera ovog rada je da se pokaže način za implementaciju heš algoritama i njihovo prilagođavanje sopstvenim potrebama. Ovde treba istaći da je sopstvena implementacija sigurnosnih algoritama mnogo bolja opcija u pogledu sigurnosti, od primene gotovih tuđih implementacija. Kod tuđih implementacija, ako nemamo uvid u izvorni kod, ne možemo da budemo sigurni da u implementaciju nisu ugrađeni delovi koda koji kasnije mogu da se upotrebe za napad.

U ovom radu detaljno su opisani *MD4*, *MD5*, *SHA-1* i *RIPEMD-160* algoritmi i prikazana je njihova implementacija u programskom jeziku Java. Urađena su merenja i poređenja performansi ovih algoritama. Na osnovu sigurnosnih uslova koje ispunjavaju i na osnovu rezultata merenja preporučuje se upotreba algoritama *SHA-1* i *RIPEMD-160*, pri čemu treba imati na umu da *RIPEMD-160* ima nešto manju brzinu izvršavanja od algoritma *SHA-1*. Algoritmi *MD4* i *MD5* se ne smatraju sigurnima. Za algoritam *MD4*, odmah nakon njegovog kreiranja pronađeni su mnogi sigurnosni propusti i on nema nikakvu praktičnu vrednost u primeni za zaštitu podataka. Za algoritam *MD5* do sada nije izveden nijedan napad, ali postoje radovi u kojima se opisuje kako to može da se uradi. Takođe, algoritam *MD5* daje heš vrednost dužine 128 bitova što se smatra nedovoljnom dužinom, pa se ne preporučuje upotreba ovog algoritma.

Literatura

[1] Milo V. Tomašević: ***"Strukture podataka, drugo izdanje"***, Elektrotrehnički fakultet, Beograd 2000.

[2] D. Pleskonjić, B. Đorđević, N. Maček, Marko Carić: ***"Sigurnost računarskih mreža"***, Viša elektrotehnička škola, Beograd, 2006.

[3] D. Pleskonjić, B. Đorđević, N. Maček, Marko Carić: ***"Sigurnost računarskih mreža – priručnik za laboratorijske vežbe"***, Viša elektrotehnička škola, Beograd, 2006

[4] Menezes, P. van Oorschot, and S. Vanstone: ***"Handbook of Applied Cryptography"*** CRC Press, 1996.

[5] Bruce Schneier: ***"Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C"***, John Wiley & Sons, 1996

[6] Henk C. A. van Tilborg: ***"Encyclopedia of Cryptography and Security"***, Springer Science+Business Media, 2005

[7] Xuejia Lai, James L. Massey: ***"Hash Functions Based on Block Ciphers"*** Preuzeto sa: *http://www.isiweb.ee.ethz.ch/archive/massey_pub/pdf/BI322.pdf*

[8] Bart Van Rompay:**" Analysis and Design of Cryptographic Hash Functions, MAC Algorithyms and Block Ciphers"** PhD thesis, Katholieke Universiteit Leuven, June 2004. Preuzeto sa: *http://www.cosic.esat.kuleuven.be/publications/thesis-16.pdf*

[9] Magnus Daum: **"Cryptanalysis of Hash Functions of the MD4-Family"** PhD thesis, Ruhr-Universit¨at Bochum, May 2005. Preuzeto sa: *http://www.cits.rub.de/imperia/md/content/magnus/dissmd4.pdf*

[10] Krystian Matusiewicz: **"Analysis of Modern Dedicated Cryptographic Hash Functions"** PhD thesis, Macquarie University, Sydney, August 2007. Preuzeto sa: *http://web.science.mq.edu.au/groups/acac/researchstudent_completed/Krystian.pdf*

[11] Bart Preneel: **“Analysis and Design of Cryptographic Hash Functions”**
PhD thesis, February 2003.
Preuzeto sa:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.4996&rep=rep1&type=pdf

[12] MD5 pseudokod
Preuzeto sa Wikipedia.org: *http://en.wikipedia.org/wiki/MD5*

[13] SHA-1 pseudokod
Preuzeto sa Wikipedia.org: *http://en.wikipedia.org/wiki/SHA-1*

Dodatak – implementacija kriptografskih heš algoritama MD4, MD5, SHA-1 i RIPED-160 u programskom jeziku Java

Digest.java

```
public abstract class Digest {

   // Ime hash funkcije
   protected String name;

   // Velicina izlaza hash f-je u bajtovima
   protected int hashSize;

   //Velicina blokova, u bajtovima,  na koje se deli ulaz hash f-je
   protected int blockSize;

   //Trenutni broj obradjenih bajtova
   protected long count;

   //Bafer u kome se cuva blok koji se trenutno obradjuje
   protected byte[] buffer;

   // Konstruktor
   protected Digest(String name, int hashSize, int blockSize) {
      super();

      this.name = name;
      this.hashSize = hashSize;
      this.blockSize = blockSize;
      this.buffer = new byte[blockSize];

      resetContext();
   }


   public String name() {
      return name;
   }

   public int hashSize() {
      return hashSize;
   }

   public int blockSize() {
      return blockSize;
   }

   //Dopunjava bafer uzimajuci neobradjene delove teksta sa ulaza
   public void update(byte b) {

      int i = (int)(count % blockSize);
      count++;
      buffer[i] = b;
```

```
      if (i == (blockSize - 1)) {
         transform(buffer, 0);
      }
   }

   public void update(byte[] b) {

      update(b, 0, b.length);
   }

   public void update(byte[] b, int offset, int len) {

      int n = (int)(count % blockSize);
      count += len;
      int partLen = blockSize - n;
      int i = 0;

      if (len >= partLen) {
         System.arraycopy(b, offset, buffer, n, partLen);
         transform(buffer, 0);
         for (i = partLen; i + blockSize - 1 < len; i+= blockSize) {
            transform(b, offset + i);
         }
         n = 0;
      }

      if (i < len) {
         System.arraycopy(b, offset + i, buffer, n, len - i);
      }
   }

   public byte[] digest(){

      byte[] tail = padBuffer(); //Vrsi se dodavanje odredjenog sadrzaja
na kraj ulaznog teksta u zavisnosti od vrste algoritma
      update(tail, 0, tail.length); // Obrada poslednjeg bloka ulaznog
teksta
      byte[] result = getResult(); //Uzima rezultat koji se dobija kao
izlaz algoritma

      reset(); // resetuje algoritam da bi se omogucila njegova upotreba
nad novim ulazom

      return result;
   }

   public void reset() { // resetuje algoritam da bi se omogucila njegova
upotreba nad novim ulazom
        count = 0L;
      for (int i = 0; i < blockSize; ) {
         buffer[i++] = 0;
      }

      resetContext();
   }
```

```
   // Metode koje treba da se implementiraju u odgovarajucim podklasama

   public abstract Object clone();

  // public abstract boolean selfTest();

   //Vrsi nadopunjavanje ulaznog teksta odgovarajucim nizom bajtova
   protected abstract byte[] padBuffer();

   //Generise rezultat na osnovu sadrzaja odgovarajucih promenljivih koje
su karakteristicne za svaki algoritam
   protected abstract byte[] getResult();

   //Vrsi resetovanje konteksta algoritma tako da moze ponovo da se
koristi nad novim ulazom
   protected abstract void resetContext();

   //Vrsi transformaciju jednog bloka ulaznog teksta
   protected abstract void transform(byte[] in, int offset);
}
```

Gui.java

```
import java.awt.*;
import java.awt.event.*;

public class GUI extends Frame {

      private static Digest[] algorithm={
            new MD4(), new MD5(), new Sha160() , new RipeMD160()
      };

      private Digest f = algorithm[0];

      private TextArea entry = new TextArea("OVDE UNESITE TEKST");
      private Label result = new Label("Rezultat:");
      private TextField hash = new TextField("",50);
      private Label executionTimeLabel = new Label("Vreme:  ");
      private TextField executionTimeTextField = new TextField("",50);
      private Button calc= new Button("Heširaj");
      private Button term= new Button("Završi");

      private class RadioChange implements ItemListener{
            public void itemStateChanged(ItemEvent d){
                  String name = ((Checkbox)d.getSource()).getLabel();
                  int i;
                  for (i = 0; i < algorithm.length; i++)
                        if((algorithm[i].name()).equals(name))break;

                  f=algorithm[i];
            }
      }

      private void makeGUI () {
```

```
            add(entry, "Center");

            Panel panel =new Panel(new GridLayout(0,1));
            add(panel,"West");

            CheckboxGroup group = new CheckboxGroup();
            RadioChange rc = new RadioChange();
            for (int i = 0; i < algorithm.length; i++) {
                  Checkbox radio = new
Checkbox(algorithm[i].name(),group,i==0);
                  radio.addItemListener(rc);
                  panel.add(radio);
            }

            panel = new Panel(new GridLayout(3,0));
            add(panel, "South");
            Panel panel1 = new Panel(new GridBagLayout());
            Panel panel2 = new Panel(new GridLayout());
            Panel panel3 = new Panel(new GridBagLayout());
            panel.add(panel1);
            panel.add(panel3);
            panel.add(panel2);

            GridBagConstraints c = new GridBagConstraints();
            c.gridwidth = GridBagConstraints.RELATIVE;


            panel1.add(result,c);
            panel1.add(hash,c);

            panel3.add(executionTimeLabel,c);
            panel3.add(executionTimeTextField,c);


            calc.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent d) {
                        long executionBeginingTime = System.nanoTime();
                        f.update((entry.getText()).getBytes());
                        byte[]md=f.digest();
                        long executionEndingTime = System.nanoTime() -
executionBeginingTime;
                        hash.setText(Main.toString(md));

      executionTimeTextField.setText(Long.toString(executionEndingTime) +
" ns");
                  }
            });
            panel2.add(calc);

            term.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent d) {
                        dispose();
                  }
            });
            panel2.add(term);
```

```
	}


	private void addMenu(){

		MenuBar bar = new MenuBar(); setMenuBar(bar);
		Menu action = new Menu("Akcija"); bar.add(action);

		MenuItem item = new MenuItem("Heširaj", new
MenuShortcut('P'));
		item.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent d) {
				f.update((entry.getText()).getBytes());
				byte[]md=f.digest();
				hash.setText(Main.toString(md));
			}
		});
		action.add(item);

		item = new MenuItem("Završi", new MenuShortcut('T'));
		item.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent d) {
				dispose();
			}
		});
		action.add(item);
	}


	public GUI(){

		super("Heš algoritmi"); setBounds(100,100,600,250);

		makeGUI(); addMenu(); setVisible(true);

		addWindowListener(new WindowAdapter(){

			public void windowClosing(WindowEvent d){dispose();}
		});
	}

}
```

HashCmd.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class HashCmd {

	public static Digest get_hash_algorithm(String algorithm_name){
```

```
            if(algorithm_name.equalsIgnoreCase("md4")){ return new
MD4();}
            if(algorithm_name.equalsIgnoreCase("md5")){ return new
MD5();}
            if(algorithm_name.equalsIgnoreCase("sha1")){ return new
Sha160();}
            if(algorithm_name.equalsIgnoreCase("ripemd160")){ return new
RipeMD160();}
            return null;
      }

      public static void main(String[] args) {
            long execution_time = 0;
            int offset = 0;
            int numRead = 0;
            Digest hash_algorithm = get_hash_algorithm(args[0]);
            File file = new File(args[1]);

            try {
                FileInputStream fileInputStream = new
FileInputStream(file);
                int length_to_read = hash_algorithm.blockSize();
                if (file.length() - offset < hash_algorithm.blockSize()){
                  length_to_read = (int)(file.length() - offset);
                }
                byte[] input_block = new byte[length_to_read];
                while (offset < file.length()
                        && (numRead=fileInputStream.read(input_block, 0,
length_to_read)) >= 0) {

                  //long begining_time = System.nanoTime();
                  long begining_time = System.currentTimeMillis();
                        hash_algorithm.update(input_block);
                        //execution_time += System.nanoTime() -
begining_time;
                        execution_time += (System.currentTimeMillis() -
begining_time);

                  offset += numRead;
                      if (file.length() - offset <
hash_algorithm.blockSize()){
                        length_to_read = (int)(file.length() - offset);
                      }
                      input_block = new byte[length_to_read];
                }
                //long begining_time = System.nanoTime();
                long begining_time = System.currentTimeMillis();
                  byte[]message_digest = hash_algorithm.digest();
                  //execution_time += System.nanoTime() - begining_time;
                  execution_time += System.currentTimeMillis() -
begining_time;
                  System.out.println("\nRezultat: " +
Main.toString(message_digest));
                  System.out.println("Vreme: " + execution_time + " ms");
            }
            catch (FileNotFoundException e) {
                System.out.println("File Not Found.");
```

```
                e.printStackTrace();
            }
            catch (IOException e1){
                System.out.println("Error Reading The File.");
                e1.printStackTrace();
            }
    }
}
```

Main.java

```
public class Main {

    private static final char[] HEX_DIGITS =
"0123456789ABCDEF".toCharArray();

    public static String toString(int n) {
        char[] buf = new char[8];
        for (int i = 7; i >= 0; i--) {
           buf[i] = HEX_DIGITS[n & 0x0F];
           n >>>= 4;
        }
        return new String(buf);
     }

      public static String toString(byte[] ba) {
          return toString(ba, 0, ba.length);
    }

      public static final String toString(byte[] ba, int offset, int
length) {
          char[] buf = new char[length * 2];
          for (int i = 0, j = 0, k; i < length; ) {
             k = ba[offset + i++];
             buf[j++] = HEX_DIGITS[(k >>> 4) & 0x0F];
             buf[j++] = HEX_DIGITS[ k        & 0x0F];
          }
          return new String(buf);
      }

      public static void main(String[] args) {

        new GUI();
    }

}
```

MD4.java

```
public class MD4 extends Digest {


   //Duzina irezultata koji se dobija na izlazu ovog algoritma
   private static final int DIGEST_LENGTH = 16;

   //Velicina bloka podataka sa kojim radi ovaj algoritam
   private static final int BLOCK_LENGTH = 64;

   //Kljucevi za MD4
   private static final int A = 0x67452301;
   private static final int B = 0xefcdab89;
   private static final int C = 0x98badcfe;
   private static final int D = 0x10325476;

   private int a, b, c, d;

   public MD4() {
      super("MD4", DIGEST_LENGTH, BLOCK_LENGTH);
   }

   private MD4(MD4 that) {
      this();

      this.a = that.a;
      this.b = that.b;
      this.c = that.c;
      this.d = that.d;
      this.count = that.count;
      this.buffer = (byte[]) that.buffer.clone();
   }



   public Object clone() {
      return new MD4(this);
   }

   //Implementacija metoda iz nadklase

   protected byte[] getResult() {
      byte[] digest = {
         (byte) a, (byte) (a >>> 8), (byte) (a >>> 16), (byte) (a >>>
24),
         (byte) b, (byte) (b >>> 8), (byte) (b >>> 16), (byte) (b >>>
24),
         (byte) c, (byte) (c >>> 8), (byte) (c >>> 16), (byte) (c >>>
24),
         (byte) d, (byte) (d >>> 8), (byte) (d >>> 16), (byte) (d >>> 24)
      };
      return digest;
   }

   protected void resetContext() {
```

```
      a = A; b = B;
      c = C; d = D;
   }


   protected byte[] padBuffer() {
      int n = (int) (count % BLOCK_LENGTH);
      int padding = (n < 56) ? (56 - n) : (120 - n);
      byte[] pad = new byte[padding + 8];

      pad[0] = (byte) 0x80;
      long bits = count << 3;
      pad[padding++] = (byte)  bits;
      pad[padding++] = (byte) (bits >>>  8);
      pad[padding++] = (byte) (bits >>> 16);
      pad[padding++] = (byte) (bits >>> 24);
      pad[padding++] = (byte) (bits >>> 32);
      pad[padding++] = (byte) (bits >>> 40);
      pad[padding++] = (byte) (bits >>> 48);
      pad[padding  ] = (byte) (bits >>> 56);

      return pad;
   }

   protected void transform(byte[] in, int i) {
      int X0 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X1 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X2 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X3 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X4 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X5 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X6 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X7 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X8 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X9 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X10 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X11 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X12 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X13 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X14 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X15 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i  ] << 24;
```

```
int aa, bb, cc, dd;

aa = a;  bb = b;  cc = c;  dd = d;

aa += ((bb & cc) | ((~bb) & dd)) + X0;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & bb) | ((~aa) & cc)) + X1;
dd = dd <<  7 | dd >>> -7;
cc += ((dd & aa) | ((~dd) & bb)) + X2;
cc = cc << 11 | cc >>> -11;
bb += ((cc & dd) | ((~cc) & aa)) + X3;
bb = bb << 19 | bb >>> -19;
aa += ((bb & cc) | ((~bb) & dd)) + X4;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & bb) | ((~aa) & cc)) + X5;
dd = dd <<  7 | dd >>> -7;
cc += ((dd & aa) | ((~dd) & bb)) + X6;
cc = cc << 11 | cc >>> -11;
bb += ((cc & dd) | ((~cc) & aa)) + X7;
bb = bb << 19 | bb >>> -19;
aa += ((bb & cc) | ((~bb) & dd)) + X8;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & bb) | ((~aa) & cc)) + X9;
dd = dd <<  7 | dd >>> -7;
cc += ((dd & aa) | ((~dd) & bb)) + X10;
cc = cc << 11 | cc >>> -11;
bb += ((cc & dd) | ((~cc) & aa)) + X11;
bb = bb << 19 | bb >>> -19;
aa += ((bb & cc) | ((~bb) & dd)) + X12;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & bb) | ((~aa) & cc)) + X13;
dd = dd <<  7 | dd >>> -7;
cc += ((dd & aa) | ((~dd) & bb)) + X14;
cc = cc << 11 | cc >>> -11;
bb += ((cc & dd) | ((~cc) & aa)) + X15;
bb = bb << 19 | bb >>> -19;

aa += ((bb & (cc | dd)) | (cc & dd)) + X0 + 0x5a827999;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & (bb | cc)) | (bb & cc)) + X4 + 0x5a827999;
dd = dd <<  5 | dd >>> -5;
cc += ((dd & (aa | bb)) | (aa & bb)) + X8 + 0x5a827999;
cc = cc <<  9 | cc >>> -9;
bb += ((cc & (dd | aa)) | (dd & aa)) + X12 + 0x5a827999;
bb = bb << 13 | bb >>> -13;
aa += ((bb & (cc | dd)) | (cc & dd)) + X1 + 0x5a827999;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & (bb | cc)) | (bb & cc)) + X5 + 0x5a827999;
dd = dd <<  5 | dd >>> -5;
cc += ((dd & (aa | bb)) | (aa & bb)) + X9 + 0x5a827999;
cc = cc <<  9 | cc >>> -9;
bb += ((cc & (dd | aa)) | (dd & aa)) + X13 + 0x5a827999;
bb = bb << 13 | bb >>> -13;
aa += ((bb & (cc | dd)) | (cc & dd)) + X2 + 0x5a827999;
aa = aa <<  3 | aa >>> -3;
dd += ((aa & (bb | cc)) | (bb & cc)) + X6 + 0x5a827999;
```

```
        dd = dd <<  5 | dd >>> -5;
        cc += ((dd & (aa | bb)) | (aa & bb)) + X10 + 0x5a827999;
        cc = cc <<  9 | cc >>> -9;
        bb += ((cc & (dd | aa)) | (dd & aa)) + X14 + 0x5a827999;
        bb = bb << 13 | bb >>> -13;
        aa += ((bb & (cc | dd)) | (cc & dd)) + X3 + 0x5a827999;
        aa = aa <<  3 | aa >>> -3;
        dd += ((aa & (bb | cc)) | (bb & cc)) + X7 + 0x5a827999;
        dd = dd <<  5 | dd >>> -5;
        cc += ((dd & (aa | bb)) | (aa & bb)) + X11 + 0x5a827999;
        cc = cc <<  9 | cc >>> -9;
        bb += ((cc & (dd | aa)) | (dd & aa)) + X15 + 0x5a827999;
        bb = bb << 13 | bb >>> -13;

        aa += (bb ^ cc ^ dd) + X0 + 0x6ed9eba1;
        aa = aa <<  3 | aa >>> -3;
        dd += (aa ^ bb ^ cc) + X8 + 0x6ed9eba1;
        dd = dd <<  9 | dd >>> -9;
        cc += (dd ^ aa ^ bb) + X4 + 0x6ed9eba1;
        cc = cc << 11 | cc >>> -11;
        bb += (cc ^ dd ^ aa) + X12 + 0x6ed9eba1;
        bb = bb << 15 | bb >>> -15;
        aa += (bb ^ cc ^ dd) + X2 + 0x6ed9eba1;
        aa = aa <<  3 | aa >>> -3;
        dd += (aa ^ bb ^ cc) + X10 + 0x6ed9eba1;
        dd = dd <<  9 | dd >>> -9;
        cc += (dd ^ aa ^ bb) + X6 + 0x6ed9eba1;
        cc = cc << 11 | cc >>> -11;
        bb += (cc ^ dd ^ aa) + X14 + 0x6ed9eba1;
        bb = bb << 15 | bb >>> -15;
        aa += (bb ^ cc ^ dd) + X1 + 0x6ed9eba1;
        aa = aa <<  3 | aa >>> -3;
        dd += (aa ^ bb ^ cc) + X9 + 0x6ed9eba1;
        dd = dd <<  9 | dd >>> -9;
        cc += (dd ^ aa ^ bb) + X5 + 0x6ed9eba1;
        cc = cc << 11 | cc >>> -11;
        bb += (cc ^ dd ^ aa) + X13 + 0x6ed9eba1;
        bb = bb << 15 | bb >>> -15;
        aa += (bb ^ cc ^ dd) + X3 + 0x6ed9eba1;
        aa = aa <<  3 | aa >>> -3;
        dd += (aa ^ bb ^ cc) + X11 + 0x6ed9eba1;
        dd = dd <<  9 | dd >>> -9;
        cc += (dd ^ aa ^ bb) + X7 + 0x6ed9eba1;
        cc = cc << 11 | cc >>> -11;
        bb += (cc ^ dd ^ aa) + X15 + 0x6ed9eba1;
        bb = bb << 15 | bb >>> -15;

        a += aa; b += bb; c += cc; d += dd;
    }
}
```

MD5.java

```
public class MD5 extends Digest {


   private static final int BLOCK_SIZE = 64; // Velicina blokova na koje
se deli ulazni tekst

   private int h0, h1, h2, h3;


   public MD5() {
      super("MD5", 16, BLOCK_SIZE);
   }

   private MD5(MD5 md) {
      this();

      this.h0 = md.h0;
      this.h1 = md.h1;
      this.h2 = md.h2;
      this.h3 = md.h3;
      this.count = md.count;
      this.buffer = (byte[]) md.buffer.clone();
   }


   public Object clone() {
      return new MD5(this);
   }

   // Implementacija metoda nasledjenih iz nadklase

   protected synchronized void transform(byte[] in, int i) {
      int X0 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X1 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X2 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X3 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X4 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X5 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X6 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X7 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X8 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X9 =  (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X10 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
```

```
      int X11 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X12 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X13 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X14 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i++] << 24;
      int X15 = (in[i++] & 0xFF) | (in[i++] & 0xFF) << 8 | (in[i++] &
0xFF) << 16 | in[i  ] << 24;

      int A = h0;
      int B = h1;
      int C = h2;
      int D = h3;

      // hex constants are from md5.c in FSF Gnu Privacy Guard 0.9.2
      // round 1
      A += ((B & C) | (~B & D)) + X0 + 0xD76AA478;  A = B + (A << 7 | A
>>> -7);
      D += ((A & B) | (~A & C)) + X1 + 0xE8C7B756;  D = A + (D << 12 | D
>>> -12);
      C += ((D & A) | (~D & B)) + X2 + 0x242070DB;  C = D + (C << 17 | C
>>> -17);
      B += ((C & D) | (~C & A)) + X3 + 0xC1BDCEEE;  B = C + (B << 22 | B
>>> -22);

      A += ((B & C) | (~B & D)) + X4 + 0xF57C0FAF;  A = B + (A << 7 | A
>>> -7);
      D += ((A & B) | (~A & C)) + X5 + 0x4787C62A;  D = A + (D << 12 | D
>>> -12);
      C += ((D & A) | (~D & B)) + X6 + 0xA8304613;  C = D + (C << 17 | C
>>> -17);
      B += ((C & D) | (~C & A)) + X7 + 0xFD469501;  B = C + (B << 22 | B
>>> -22);

      A += ((B & C) | (~B & D)) + X8 + 0x698098D8;  A = B + (A << 7 | A
>>> -7);
      D += ((A & B) | (~A & C)) + X9 + 0x8B44F7AF;  D = A + (D << 12 | D
>>> -12);
      C += ((D & A) | (~D & B)) + X10 + 0xFFFF5BB1; C = D + (C << 17 | C
>>> -17);
      B += ((C & D) | (~C & A)) + X11 + 0x895CD7BE; B = C + (B << 22 | B
>>> -22);

      A += ((B & C) | (~B & D)) + X12 + 0x6B901122; A = B + (A << 7 | A
>>> -7);
      D += ((A & B) | (~A & C)) + X13 + 0xFD987193; D = A + (D << 12 | D
>>> -12);
      C += ((D & A) | (~D & B)) + X14 + 0xA679438E; C = D + (C << 17 | C
>>> -17);
      B += ((C & D) | (~C & A)) + X15 + 0x49B40821; B = C + (B << 22 | B
>>> -22);

      // round 2
      A += ((B & D) | (C & ~D)) + X1 + 0xF61E2562;  A = B + (A << 5 | A
>>> -5);
```

```
      D += ((A & C) | (B & ~C)) + X6 + 0xC040B340;  D = A + (D << 9 | D
>>> -9);
      C += ((D & B) | (A & ~B)) + X11 + 0x265E5A51; C = D + (C << 14 | C
>>> -14);
      B += ((C & A) | (D & ~A)) + X0 + 0xE9B6C7AA;  B = C + (B << 20 | B
>>> -20);

      A += ((B & D) | (C & ~D)) + X5 + 0xD62F105D;  A = B + (A << 5 | A
>>> -5);
      D += ((A & C) | (B & ~C)) + X10 + 0x02441453; D = A + (D << 9 | D
>>> -9);
      C += ((D & B) | (A & ~B)) + X15 + 0xD8A1E681; C = D + (C << 14 | C
>>> -14);
      B += ((C & A) | (D & ~A)) + X4 + 0xE7D3FBC8;  B = C + (B << 20 | B
>>> -20);

      A += ((B & D) | (C & ~D)) + X9 + 0x21E1CDE6;  A = B + (A << 5 | A
>>> -5);
      D += ((A & C) | (B & ~C)) + X14 + 0xC33707D6; D = A + (D << 9 | D
>>> -9);
      C += ((D & B) | (A & ~B)) + X3 + 0xF4D50D87;  C = D + (C << 14 | C
>>> -14);
      B += ((C & A) | (D & ~A)) + X8 + 0x455A14ED;  B = C + (B << 20 | B
>>> -20);

      A += ((B & D) | (C & ~D)) + X13 + 0xA9E3E905; A = B + (A << 5 | A
>>> -5);
      D += ((A & C) | (B & ~C)) + X2 + 0xFCEFA3F8;  D = A + (D << 9 | D
>>> -9);
      C += ((D & B) | (A & ~B)) + X7 + 0x676F02D9;  C = D + (C << 14 | C
>>> -14);
      B += ((C & A) | (D & ~A)) + X12 + 0x8D2A4C8A; B = C + (B << 20 | B
>>> -20);

      // round 3
      A += (B ^ C ^ D) + X5 + 0xFFFA3942;  A = B + (A << 4 | A >>> -4);
      D += (A ^ B ^ C) + X8 + 0x8771F681;  D = A + (D << 11 | D >>> -11);
      C += (D ^ A ^ B) + X11 + 0x6D9D6122; C = D + (C << 16 | C >>> -16);
      B += (C ^ D ^ A) + X14 + 0xFDE5380C; B = C + (B << 23 | B >>> -23);

      A += (B ^ C ^ D) + X1 + 0xA4BEEA44;  A = B + (A << 4 | A >>> -4);
      D += (A ^ B ^ C) + X4 + 0x4BDECFA9;  D = A + (D << 11 | D >>> -11);
      C += (D ^ A ^ B) + X7 + 0xF6BB4B60;  C = D + (C << 16 | C >>> -16);
      B += (C ^ D ^ A) + X10 + 0xBEBFBC70; B = C + (B << 23 | B >>> -23);

      A += (B ^ C ^ D) + X13 + 0x289B7EC6; A = B + (A << 4 | A >>> -4);
      D += (A ^ B ^ C) + X0 + 0xEAA127FA;  D = A + (D << 11 | D >>> -11);
      C += (D ^ A ^ B) + X3 + 0xD4EF3085;  C = D + (C << 16 | C >>> -16);
      B += (C ^ D ^ A) + X6 + 0x04881D05;  B = C + (B << 23 | B >>> -23);

      A += (B ^ C ^ D) + X9 + 0xD9D4D039;  A = B + (A << 4 | A >>> -4);
      D += (A ^ B ^ C) + X12 + 0xE6DB99E5; D = A + (D << 11 | D >>> -11);
      C += (D ^ A ^ B) + X15 + 0x1FA27CF8; C = D + (C << 16 | C >>> -16);
      B += (C ^ D ^ A) + X2 + 0xC4AC5665;  B = C + (B << 23 | B >>> -23);

      // round 4
```

```
      A += (C ^ (B | ~D)) + X0 + 0xF4292244;  A = B + (A << 6 | A >>> -
6);
      D += (B ^ (A | ~C)) + X7 + 0x432AFF97;  D = A + (D << 10 | D >>> -
10);
      C += (A ^ (D | ~B)) + X14 + 0xAB9423A7; C = D + (C << 15 | C >>> -
15);
      B += (D ^ (C | ~A)) + X5 + 0xFC93A039;  B = C + (B << 21 | B >>> -
21);

      A += (C ^ (B | ~D)) + X12 + 0x655B59C3; A = B + (A << 6 | A >>> -
6);
      D += (B ^ (A | ~C)) + X3 + 0x8F0CCC92;  D = A + (D << 10 | D >>> -
10);
      C += (A ^ (D | ~B)) + X10 + 0xFFEFF47D; C = D + (C << 15 | C >>> -
15);
      B += (D ^ (C | ~A)) + X1 + 0x85845dd1;  B = C + (B << 21 | B >>> -
21);

      A += (C ^ (B | ~D)) + X8 + 0x6FA87E4F;  A = B + (A << 6 | A >>> -
6);
      D += (B ^ (A | ~C)) + X15 + 0xFE2CE6E0; D = A + (D << 10 | D >>> -
10);
      C += (A ^ (D | ~B)) + X6 + 0xA3014314;  C = D + (C << 15 | C >>> -
15);
      B += (D ^ (C | ~A)) + X13 + 0x4E0811A1; B = C + (B << 21 | B >>> -
21);

      A += (C ^ (B | ~D)) + X4 + 0xF7537E82;  A = B + (A << 6 | A >>> -
6);
      D += (B ^ (A | ~C)) + X11 + 0xBD3AF235; D = A + (D << 10 | D >>> -
10);
      C += (A ^ (D | ~B)) + X2 + 0x2AD7D2BB;  C = D + (C << 15 | C >>> -
15);
      B += (D ^ (C | ~A)) + X9 + 0xEB86D391;  B = C + (B << 21 | B >>> -
21);

      h0 += A;
      h1 += B;
      h2 += C;
      h3 += D;
   }

   protected byte[] padBuffer() {
      int n = (int)(count % BLOCK_SIZE);
      int padding = (n < 56) ? (56 - n) : (120 - n);
      byte[] result = new byte[padding + 8];

      result[0] = (byte) 0x80;

      long bits = count << 3;
      result[padding++] = (byte) bits;
      result[padding++] = (byte)(bits >>>  8);
      result[padding++] = (byte)(bits >>> 16);
      result[padding++] = (byte)(bits >>> 24);
      result[padding++] = (byte)(bits >>> 32);
      result[padding++] = (byte)(bits >>> 40);
      result[padding++] = (byte)(bits >>> 48);
```

```
      result[padding  ] = (byte)(bits >>> 56);

      return result;
   }

   protected byte[] getResult() {
      byte[] result = new byte[] {
         (byte) h0, (byte)(h0 >>> 8), (byte)(h0 >>> 16), (byte)(h0 >>>
24),
         (byte) h1, (byte)(h1 >>> 8), (byte)(h1 >>> 16), (byte)(h1 >>>
24),
         (byte) h2, (byte)(h2 >>> 8), (byte)(h2 >>> 16), (byte)(h2 >>>
24),
         (byte) h3, (byte)(h3 >>> 8), (byte)(h3 >>> 16), (byte)(h3 >>>
24)
      };

      return result;
   }

   protected void resetContext() {

      h0 = 0x67452301;
      h1 = 0xEFCDAB89;
      h2 = 0x98BADCFE;
      h3 = 0x10325476;
   }

}
```

RIPEMD160.java

```
public class RipeMD160 extends Digest {

   private static final int BLOCK_SIZE = 64;


   private static final int[] R  = {
      0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
      7,  4, 13,  1, 10,  6, 15,  3, 12,  0,  9,  5,  2, 14, 11,  8,
      3, 10, 14,  4,  9, 15,  8,  1,  2,  7,  0,  6, 13, 11,  5, 12,
      1,  9, 11, 10,  0,  8, 12,  4, 13,  3,  7, 15, 14,  5,  6,  2,
      4,  0,  5,  9,  7, 12,  2, 10, 14,  1,  3,  8, 11,  6, 15, 13
   };

   private static final int[] Rp = {
       5, 14,  7,  0,  9,  2, 11,  4, 13,  6, 15,  8,  1, 10,  3, 12,
       6, 11,  3,  7,  0, 13,  5, 10, 14, 15,  8, 12,  4,  9,  1,  2,
      15,  5,  1,  3,  7, 14,  6,  9, 11,  8, 12,  2, 10,  0,  4, 13,
       8,  6,  4,  1,  3, 11, 15,  0,  5, 12,  2, 13,  9,  7, 10, 14,
      12, 15, 10,  4,  1,  5,  8,  7,  6,  2, 13, 14,  0,  3,  9, 11
   };
```

```
private static final int[] S  = {
   11, 14, 15, 12,  5,  8,  7,  9, 11, 13, 14, 15,  6,  7,  9,  8,
    7,  6,  8, 13, 11,  9,  7, 15,  7, 12, 15,  9, 11,  7, 13, 12,
   11, 13,  6,  7, 14,  9, 13, 15, 14,  8, 13,  6,  5, 12,  7,  5,
   11, 12, 14, 15, 14, 15,  9,  8,  9, 14,  5,  6,  8,  6,  5, 12,
    9, 15,  5, 11,  6,  8, 13, 12,  5, 12, 13, 14, 11,  8,  5,  6
};

private static final int[] Sp = {
    8,  9,  9, 11, 13, 15, 15,  5,  7,  7,  8, 11, 14, 14, 12,  6,
    9, 13, 15,  7, 12,  8,  9, 11,  7,  7, 12,  7,  6, 15, 13, 11,
    9,  7, 15, 11,  8,  6,  6, 14, 12, 13,  5, 14, 13, 13,  7,  5,
   15,  5,  8, 11, 14, 14,  6, 14,  6,  9, 12,  9, 12,  5, 15,  8,
    8,  5, 12,  9, 12,  5, 14,  6,  8, 13,  6,  5, 15, 13, 11, 11
};


private int h0, h1, h2, h3, h4;

//16 x 32-bitnih reci
private int[] X = new int[16];



public RipeMD160() {
   super("RipeMD160", 20, BLOCK_SIZE);
}


private RipeMD160(RipeMD160 md) {
   this();

   this.h0 = md.h0;
   this.h1 = md.h1;
   this.h2 = md.h2;
   this.h3 = md.h3;
   this.h4 = md.h4;
   this.count = md.count;
   this.buffer = (byte[]) md.buffer.clone();
}







public Object clone() {
   return (new RipeMD160(this));
}


protected void transform (byte[] in, int offset) {
   int A, B, C, D, E, Ap, Bp, Cp, Dp, Ep, T, s, i;
```

```
      // encode 64 bytes from input block into an array of 16 unsigned
integers
      for (i = 0; i < 16; i++) {
         X[i] = (in[offset++] & 0xFF)       |
                (in[offset++] & 0xFF) <<  8 |
                (in[offset++] & 0xFF) << 16 |
                 in[offset++]         << 24;
      }

      A = Ap = h0;
      B = Bp = h1;
      C = Cp = h2;
      D = Dp = h3;
      E = Ep = h4;

      for (i = 0; i < 16; i++) { // runda 0...15
         s = S[i];
         T = A + (B ^ C ^ D) + X[i];
         A = E; E = D; D = C << 10 | C >>> 22; C = B;
         B = (T << s | T >>> (32 - s)) + A;

         s = Sp[i];
         T = Ap + (Bp ^ (Cp | ~Dp)) + X[Rp[i]] + 0x50A28BE6;
         Ap = Ep; Ep = Dp; Dp = Cp << 10 | Cp >>> 22; Cp = Bp;
         Bp = (T << s | T >>> (32 - s)) + Ap;
      }

      for ( ; i < 32; i++) { // runda 16...31
         s = S[i];
         T = A + ((B & C) | (~B & D)) + X[R[i]] + 0x5A827999;
         A = E; E = D; D = C << 10 | C >>> 22; C = B;
         B = (T << s | T >>> (32 - s)) + A;

         s = Sp[i];
         T = Ap + ((Bp & Dp) | (Cp & ~Dp)) + X[Rp[i]] + 0x5C4DD124;
         Ap = Ep; Ep = Dp; Dp = Cp << 10 | Cp >>> 22; Cp = Bp;
         Bp = (T << s | T >>> (32 - s)) + Ap;
      }

      for ( ; i < 48; i++) { // runda 32...47
         s = S[i];
         T = A + ((B | ~C) ^ D) + X[R[i]] + 0x6ED9EBA1;
         A = E; E = D; D = C << 10 | C >>> 22; C = B;
         B = (T << s | T >>> (32 - s)) + A;

         s = Sp[i];
         T = Ap + ((Bp | ~Cp) ^ Dp) + X[Rp[i]] + 0x6D703EF3;
         Ap = Ep; Ep = Dp; Dp = Cp << 10 | Cp >>> 22; Cp = Bp;
         Bp = (T << s | T >>> (32 - s)) + Ap;
      }

      for ( ; i < 64; i++) { // runda 48...63
         s = S[i];
         T = A + ((B & D) | (C & ~D)) + X[R[i]] + 0x8F1BBCDC;
         A = E; E = D; D = C << 10 | C >>> 22; C = B;
         B = (T << s | T >>> (32 - s)) + A;
```

```
            s = Sp[i];
            T = Ap + ((Bp & Cp) | (~Bp & Dp)) + X[Rp[i]] + 0x7A6D76E9;
            Ap = Ep; Ep = Dp; Dp = Cp << 10 | Cp >>> 22; Cp = Bp;
            Bp = (T << s | T >>> (32 - s)) + Ap;
        }

        for ( ; i < 80; i++) { // runda 64...79
            s = S[i];
            T = A + (B ^ (C | ~D)) + X[R[i]] + 0xA953FD4E;
            A = E; E = D; D = C << 10 | C >>> 22; C = B;
            B = (T << s | T >>> (32 - s)) + A;

            s = Sp[i];
            T = Ap + (Bp ^ Cp ^ Dp) + X[Rp[i]];
            Ap = Ep; Ep = Dp; Dp = Cp << 10 | Cp >>> 22; Cp = Bp;
            Bp = (T << s | T >>> (32 - s)) + Ap;
        }

        T =  h1 + C + Dp;
        h1 = h2 + D + Ep;
        h2 = h3 + E + Ap;
        h3 = h4 + A + Bp;
        h4 = h0 + B + Cp;
        h0 = T;
    }

    protected byte[] padBuffer() {
        int n = (int)(count % BLOCK_SIZE);
        int padding = (n < 56) ? (56 - n) : (120 - n);
        byte[] result = new byte[padding + 8];


        result[0] = (byte) 0x80;


        long bits = count << 3;
        result[padding++] = (byte) bits;
        result[padding++] = (byte)(bits >>>  8);
        result[padding++] = (byte)(bits >>> 16);
        result[padding++] = (byte)(bits >>> 24);
        result[padding++] = (byte)(bits >>> 32);
        result[padding++] = (byte)(bits >>> 40);
        result[padding++] = (byte)(bits >>> 48);
        result[padding  ] = (byte)(bits >>> 56);

        return result;
    }

    protected byte[] getResult() {
        byte[] result = new byte[] {
            (byte) h0, (byte)(h0 >>> 8), (byte)(h0 >>> 16), (byte)(h0 >>>
24),
            (byte) h1, (byte)(h1 >>> 8), (byte)(h1 >>> 16), (byte)(h1 >>>
24),
            (byte) h2, (byte)(h2 >>> 8), (byte)(h2 >>> 16), (byte)(h2 >>>
24),
```

```
            (byte) h3, (byte)(h3 >>> 8), (byte)(h3 >>> 16), (byte)(h3 >>>
24),
            (byte) h4, (byte)(h4 >>> 8), (byte)(h4 >>> 16), (byte)(h4 >>>
24)
         };

         return result;
      }

      protected void resetContext() {

         h0 = 0x67452301;
         h1 = 0xEFCDAB89;
         h2 = 0x98BADCFE;
         h3 = 0x10325476;
         h4 = 0xC3D2E1F0;
      }

}
```

SHA160.java

```
public class Sha160 extends Digest {


   private static final int BLOCK_SIZE = 64; // Velicina blokova na koje
se deli ulazni tekst


   private static final int[] w = new int[80];

   private int h0, h1, h2, h3, h4;


   public Sha160() {
      super("SHA1", 20, BLOCK_SIZE);
   }

   private Sha160(Sha160 md) {
      this();

      this.h0 = md.h0;
      this.h1 = md.h1;
      this.h2 = md.h2;
      this.h3 = md.h3;
      this.h4 = md.h4;
      this.count = md.count;
      this.buffer = (byte[]) md.buffer.clone();
   }


   public static final int[]
   G(int hh0, int hh1, int hh2, int hh3, int hh4, byte[] in, int offset)
{
```

```
      return sha(hh0, hh1, hh2, hh3, hh4, in, offset);
   }


   public Object clone() {
      return new Sha160(this);
   }

   //Implementacija nasledjenih metoda

   protected void transform(byte[] in, int offset) {

      int[] result = sha(h0, h1, h2, h3, h4, in, offset);

      h0 = result[0];
      h1 = result[1];
      h2 = result[2];
      h3 = result[3];
      h4 = result[4];

   }

   protected byte[] padBuffer() {
      int n = (int)(count % BLOCK_SIZE);
      int padding = (n < 56) ? (56 - n) : (120 - n);
      byte[] result = new byte[padding + 8];


      result[0] = (byte) 0x80;


      long bits = count << 3;
      result[padding++] = (byte)(bits >>> 56);
      result[padding++] = (byte)(bits >>> 48);
      result[padding++] = (byte)(bits >>> 40);
      result[padding++] = (byte)(bits >>> 32);
      result[padding++] = (byte)(bits >>> 24);
      result[padding++] = (byte)(bits >>> 16);
      result[padding++] = (byte)(bits >>>  8);
      result[padding  ] = (byte) bits;

      return result;
   }

   protected byte[] getResult() {
      byte[] result = new byte[] {
         (byte)(h0 >>> 24), (byte)(h0 >>> 16), (byte)(h0 >>> 8), (byte)
h0,
         (byte)(h1 >>> 24), (byte)(h1 >>> 16), (byte)(h1 >>> 8), (byte)
h1,
         (byte)(h2 >>> 24), (byte)(h2 >>> 16), (byte)(h2 >>> 8), (byte)
h2,
         (byte)(h3 >>> 24), (byte)(h3 >>> 16), (byte)(h3 >>> 8), (byte)
h3,
         (byte)(h4 >>> 24), (byte)(h4 >>> 16), (byte)(h4 >>> 8), (byte)
h4
      };
```

```
      return result;
   }

   protected void resetContext() {

      h0 = 0x67452301;
      h1 = 0xEFCDAB89;
      h2 = 0x98BADCFE;
      h3 = 0x10325476;
      h4 = 0xC3D2E1F0;
   }


   //Metoda koja se poziva iz metode transform

   private static final synchronized int[] sha(int hh0, int hh1, int hh2,
int hh3, int hh4, byte[] in, int offset) {
      int A = hh0;
      int B = hh1;
      int C = hh2;
      int D = hh3;
      int E = hh4;
      int r, T;

      for (r = 0; r < 16; r++) {
         w[r] = in[offset++]         << 24 |
                (in[offset++] & 0xFF) << 16 |
                (in[offset++] & 0xFF) <<  8 |
                (in[offset++] & 0xFF);
      }
      for (r = 16; r < 80; r++) {
         T = w[r-3] ^ w[r-8] ^ w[r-14] ^ w[r-16];
         w[r] = T << 1 | T >>> 31;
      }

      // runda 0-19
      for (r = 0; r < 20; r++) {
         T = (A << 5 | A >>> 27) + ((B & C) | (~B & D)) + E + w[r] +
0x5A827999;
         E = D;
         D = C;
         C = B << 30 | B >>> 2;
         B = A;
         A = T;
      }

      // runda 20-39
      for (r = 20; r < 40; r++) {
         T = (A << 5 | A >>> 27) + (B ^ C ^ D) + E + w[r] + 0x6ED9EBA1;
         E = D;
         D = C;
         C = B << 30 | B >>> 2;
         B = A;
         A = T;
      }
```

```
      // runda 40-59
      for (r = 40; r < 60; r++) {
         T = (A << 5 | A >>> 27) + (B & C | B & D | C & D) + E + w[r] +
0x8F1BBCDC;
         E = D;
         D = C;
         C = B << 30 | B >>> 2;
         B = A;
         A = T;
      }

      // runda 60-79
      for (r = 60; r < 80; r++) {
         T = (A << 5 | A >>> 27) + (B ^ C ^ D) + E + w[r] + 0xCA62C1D6;
         E = D;
         D = C;
         C = B << 30 | B >>> 2;
         B = A;
         A = T;
      }

      return new int[] {hh0+A, hh1+B, hh2+C, hh3+D, hh4+E};
   }
}
```