|  |  |
| --- | --- |
| **Logo  Description automatically generated** | **Univerzitet u Beogradu**  **Elektrotehnički fakultet** |

**SISTEM ZA VIZUELNU REPREZENTACIJU BLOKČEJN TEHNOLOGIJE**

**DIPLOMSKI RAD**

Kandidat: Profesor:

Dimitrije Knežević 244/2017 prof. dr Žarko Stanisavljević

Beograd

februar 2023.

**SADRŽAJ**

[1. UVOD 4](#_Toc125192336)

[2. TEORIJSKA OSNOVA BLOCKCHAIN-A 5](#_Toc125192337)

[2.1. KAO STRUKTURA PODATAKA 6](#_Toc125192338)

[2.1.1. KRIPTOGRAFSKE HEŠ FUNKCIJE 6](#_Toc125192339)

[2.1.2. MAGIČNI BROJ 8](#_Toc125192340)

[2.1.3. BROJ VERZIJE 8](#_Toc125192341)

[2.1.4. VREMENSKI ŽIG 9](#_Toc125192342)

[2.1.5. SADRŽAJ BLOKA 9](#_Toc125192343)

[2.1.6. DIGITALNI POTPIS 9](#_Toc125192344)

[2.1.6.1. ECDSA 12](#_Toc125192345)

[2.1.7. NONCE I TEŽINA 13](#_Toc125192346)

[2.1.8. COINBASE 14](#_Toc125192347)

[2.2. KAO MREŽNI PROTOKOL 14](#_Toc125192348)

[2.2.1. TIPOVI BLOKCHAIN-A 15](#_Toc125192349)

[2.2.1.1. JAVNI 15](#_Toc125192350)

[2.2.1.2. PRIVATNI 15](#_Toc125192351)

[2.2.1.3. KONZORCIJUM 15](#_Toc125192352)

[2.2.2. KONSENZUS ALGORITMI 16](#_Toc125192353)

[2.2.2.1. PROOF OF WORK 16](#_Toc125192354)

[2.2.2.2. PROOF OF STAKE 17](#_Toc125192355)

[2.2.2.3. DELEGATED PROOF OF STAKE 18](#_Toc125192356)

[2.2.2.4. PROOF OF BURN 18](#_Toc125192357)

[2.2.2.5. PROOF OF CAPACITY 19](#_Toc125192358)

[2.2.2.6. PROOF OF ELAPSED TIME 19](#_Toc125192359)

[2.2.3. BLOCKCHAIN forking 19](#_Toc125192360)

[2.3. ISTORIJA 20](#_Toc125192361)

[2.4. Motivacija i ciljevi rada 25](#_Toc125192362)

[3. Projektovanje procesora i simulatora 27](#_Toc125192363)

[3.1. blok–šema procesora 27](#_Toc125192364)

[3.2. stepen if pipeline procesora 28](#_Toc125192365)

[3.2.1. Jedinica NewPC 28](#_Toc125192366)

[3.2.2. Jedinica PCache 30](#_Toc125192367)

[3.2.3. Jedinica Instruction Memory 31](#_Toc125192368)

[3.2.4. Pipeline registar RID 31](#_Toc125192369)

[3.3. stepen id pipeline procesora 32](#_Toc125192370)

[3.3.1. Jedinica DecodID 32](#_Toc125192371)

[4. Zaključak 33](#_Toc125192372)

[5. LITERATURA 35](#_Toc125192373)

# UVOD

Blokčejn (eng. *blockchain)* je trenutno jedan od najaktuelnijih izraza (eng. *buzzword*) u IT svetu (Slika 1). Iako sama tehnologija postoji već 30-ak godina, u poslednje vreme je došla u pažnju šire javnosti najviše zahvaljujući kriptovalutama (eng. *cryptocurrency*). Kriptovalute su trenutno najšira primena *blockchain*-a, ali ne i jedina. Kako se svet sve više digitalizuje tako će i moguće primene *blockchain*-a rasti, i već postoje pokušaji primene ove tehnologije u različim sferama – od poljoprivrede do politike.

Slika . Termin *blockchain* u naglom porastu u prethodnih nekoliko godina



Međutim, zahvaljujući prevelikoj pažnji koju su i klasični mediji i društvene mreže poklonili *blockchain*-u (Slika 2), a pritom bez razumevanja načina na koji on funkcioniše i pozadinskih mehanizama, entuzijasti pomenute tehnologije precenjuju trenutni domen njene korisnosti. Stoga, cilj ovog rada je da se demistifikuje princip rada *blockchain*-a i složenih mehanizama na kojima se on zasniva, objasni koji problemi se time rešavaju, a koji se stvaraju i tako doprinese boljem razumevanju mogućih primena ove tehnologije.

Postoji puno različitih implementacija *blockchain*-a, tako da je autor odlučio da vizuelno prikaže glavne karakteristike i mehanizame koje su zajedničke većinu verzija *blockchain*-a, i na nekim mestima pomene druge moguće verzije implementacije i pokaže njihove prednosti i mane. Ovo sve je urađeno u vidu web aplikacije na primeru imaginarne kriptovalute i njenih transakcija, ali je naravno prenosivo i na sve druge oblasti u kojima bi *blockchain* mogao da se koristi, jer se radi sa alfanumeričkim podacima (imajući u vidu da se i multimedijalni sadržaji takođe mogu pretvoriti u alfanumeričke podatke).

U drugom poglavlju data je teorijska osnova na kojoj se *blockchain* zasniva. Posebno će se razmatrati *blockchain* kao struktura podataka, a posebno mrežni deo, tj. *blockchain* kao mrežni protokol. U ovom poglavlju je takođe data kratka istorija *blockchain*-a. Detaljniji vizuelni prikaz i objašnjenje svih mehanizama koji se koriste u implementaciji *blockchain*-a će biti provučeni kroz ceo rad.

Slika . Reklama za aplikaciju Crypto koju je videlo više od milijardu ljudi tokom svetskog prvenstva u fudbalu 2022. god.



Treće poglavlje opisuje jezike i okvire (eng. *frameworks*) koji su korišćeni u realizaciji sistema za vizuelnu reprezentaciju *blockchain*-a.

Četvrto poglavlje služi za prolazak kroz ceo sistem web aplikacije, gde će se posebno razmatrati svaka komponenta *blockchain*-a kao strukture podataka i njena implementacija. Struktura će biti takva da se kreće od osnovnih pojmova i onda se daljim rešavanjem mogućih problema postepeno uvode ostali mehanizmi.

U petom poglavlju se daje zaključak u vidu rezimea celog sistema i daljih koraka za njegovo unapređenje, kao i kratak osvrt na *blockchain* kao samu tehnologiju.

# TEORIJSKA OSNOVA BLOCKCHAIN-A

*Blockchain* je struktura podataka (i protokol) koji predstavlja digitalnu implementaciju distribuirane glavne knjige (eng. *ledger*). *Ledger* je trajna kolekcija svih novčanih transakcija (mogu se pratiti i druge stvari) u hronološkom redosledu i koristi se već hiljadama godina, još od kada su ljudi prvobitno počeli da trguju. Ove transakcije su se kroz istoriju beležile na svemu od glinenih tablica do papira, a jedini značajniji pomak u ovoj oblasti pre *blockchain*-a je bila kompjuterizacija koja je prvobitno samo značila prenos sa papira u digitalnu formu.

*Blockchain* tehnologija je između ostaloga omogućila kreiranje elektronskog sistema za transfer novca bez potrebe za trećim licem koje bi proveravalo i odobravalo transakcije (npr. banke). Ovaj rad se bazira na jednom takvom imaginarnom sistemu. Krajnji korisnici mogu da prebacuju novac iz svojih digitalnih novčanika (eng. *wallet*), a da istovremeno ostanu anonimni i imaju pristup celom *ledger*-u gde su zabeležene sve njihove transakcije i transakcije svih drugih korisnika koji koriste istu kriptovalutu.

U ovom odeljku ćemo proći kroz teoriju mehanizama koje blockchain koristi, a detaljnije ćemo videti kako oni funkcionišu i kako se kombinuju u poglavlju 4.

KAO STRUKTURA PODATAKA

Svaki korisnik *blockchain* sistema ima ličnu kopiju čitavog lanca na svom uređaju. U nastavku ćemo proći kroz bitne elemente i tehnike koje *blockchain* lokalna struktura podataka mora ili može da koristi unutar svakog bloka.

KRIPTOGRAFSKE HEŠ FUNKCIJE

Neophodan element svake verzije implementacije *blockchain*-a je kriptografska heš funckija (eng. *cryptographic hash function – CHF*). *CHF* je jednosmerna funkcija (ne može se izračunati ulaz funkcije na osnovu izlaza) koja od ulaza proizvoljne dužine daje izlaz fiksne dužine. Ulaz može biti bilo šta, pa čak i multimedijalni sadržaj (iako neefikasan), jer se ulaz prvo pretvara u binarni oblik dok je izlaz uvek binarni. Vrednost koju dobijamo na izlazu heš funkcije se zove heš vrednost (ili heš kod).

Jednosmernost je najbitnija odlika *CHF*. Ove funkcije imaju veoma široku primenu u kriptografiji, internet bezbednosti, kompresiji i optimizaciji, i telekomunikacijama. Jednostavan primer je čuvanje lozinki na nekom web sajtu: Lozinke ne bi trebalo da se čuvaju u originalnom obliku (eng. *plain text*) u bazi podataka već da se čuvaju samo njihove heš vrednosti. Na taj način, čak i ako neautorizovano lice uspe da dođe do baze podataka, sa njom nikad neće moći da pristupi nalozima korisnika, jer ne postoji način da od heširanih vrednosti dođe do originalnih lozinki koje su potrebne da se unesu na ulazu sistema. Sa druge strane, autorizovan korisnik uvek može da pristupi svom nalogu sa validnom ulaznom lozinkom, jer će heširana vrednost lozinke uvek biti ista za isti ulaz.

Ovo nije savršen bezbednosni mehanizam, zbog toga što je izlaz uvek identične veličine, a ulaz proizvoljne, uvek će se više ulaza preslikavati u jedan isti izlaz. Na primer, to znači da bi korisnici tehnički mogli da pristupe svom nalogu i sa pogrešnom lozinkom koja daje istu heš vrednost kao i originalna lozinka, međutim šanse za ovo su matematički zanemarljive. Napomena: u većini sistema se dodaju i drugi bezbednosni mehanizmi kako bi smanjili šansu za gorepomenuti problem (npr. dodavanje *salt* vrednosti).

Upotreba *CHF* kod blockchain-a je sledeća: 1) Podaci iz trenutnog bloka se heširaju i ta heš vrednost se čuva u posebnom polju. 2) Heš vrednost iz prethodnog bloka je jedan od ulaznih podataka sledećeg bloka i na taj način se zapravo blokovi ulančavaju, stoga i naziv *blockchain* (lanac blokova). Pošto svaki blok ukazuje na svog prethodnika baš pomoću ove heš vrednosti, ukoliko bi se neki blok unutar lanca izmenio, ili ukoliko bi se dodao potpuno novi blok između već dva postojeća, morale bi ponovo da se računaju sve heš vrednosti blokova od tog mesta pa do najnovijeg bloka u lancu.

Slika . Jedan od mogućih načina korišćenja *CHF*-a za povezivanje blokova unutar *blockchain*-a



U stvarnim implementacijama dosta često postoje dva polja u koja se zasebno smeštaju ostali heširani podaci. Na primer može se iskoristiti Merkleovo stablo heševa (Slika 4) za transakcije u bloku, a da se iz prethodnog bloka koristi samo finalna heš vrednost Merkleovog stabla umesto svih transakcija radi dodatne sigurnosti i provere (Slika 3) [2]. U autorovoj implementaciji iskorišćen je idejno sličan, ali jednostavniji mehanizam heširanja svih podataka prethodnog bloka u samo jedno polje.

Slika . Način funkcionisanja Merkleovog stabla



MAGIČNI BROJ

Magični broj (eng. *magic number*) predstavlja jedinstven identifikator *blockchain*-a. Svaki blok unutar jednog lanca mora da ima isti magični broj, nezavisno od toga na kojoj mašini se nalazi. Na primer *Bitcoin*-ov magični broj je 0xD9B4BEF9 (tj. 3652501241 u decimalnom obliku), dok je magični broj autorovog izmišljenog *blockchain*-a X.

BROJ VERZIJE

Ukoliko je planirano da se način implementacije *blockchain*-a menja tokom vremena, potrebno je ubaciti i broj verzije *blockchain*-a (eng. *version number*) na kojoj je trenutni blok nastao. Buduće verzije implementacije *blockchain*-a moraju da omoguće kompatibilnost unazad (eng. *backwards compatibility*) sa prethodnim verzijama unutar istog lanca. U autorovom rešenju ne postoji ovo polje, jer nije od važnosti za razumevanje suštine funckionisanja *blockchain*-a.

VREMENSKI ŽIG

Vremenski žig (eng. *timestamp*) je još jedan veoma bitan deo *blockchain*-a koji služi za vremensko datiranje stvaranja bloka i potvrdu da sadržaj bloka (transakcije, potpisi dokumenata, itd.) nije nastao nakon naznačene vremenske oznake unutar bloka. Pruža i dodatnu sigurnost u vidu potvrđivanja redosleda blokova (ukoliko se krećemo od najnovijeg ka najstarijem bloku unutar lanca, sve vremenske oznake moraju biti u opadajućem redosledu). U autorovoj implementaciji radi jednostavnosti je korišćen autoinkrementirajući identifikacioni broj bloka koji ima sličnu svrhu.

SADRŽAJ BLOKA

Svaki blok (osim u nekim slučajevima prvog – *genesis* bloka) pored svih ostalih pomoćnih podataka mora sadržati i neku informaciju u sebi. U autorovom primeru sadržaj bloka su transakcije između korisnika koji koriste izmišljenu kriptovalutu.

U stvarnim implementacijama najčešći sadržaj i jesu transakcije kriptovaluta, ali postoje i druge vrste sadržaja: ostale ekonomske radnje (npr. novčane transakcije, kupovina i prodaja deonica, osiguranje, ...), identifikovanje i autentifikacija objekata ili korisnika, praćenje trenutnog vlasništva objekata, decentralizovani registri podataka, brojevi glasova, itd.. Međutim, bitno je napomenuti da je veliki broj *blockchain* aplikacija koje implementiraju prethodne vrste sadržaja još uvek u eksperimentalnoj fazi ili fazi razvoja.

U opštem slučaju, sadržaj bloka može biti bilo kakva informacija ili skup informacija, ali za većinu sistema nema potrebe čuvati podatke u *blockchain* strukturi, jer je neefikasna za pretraživanje i nije potreban (ili dozvoljen) javni decentralizovani uvid u podatke.

DIGITALNI POTPIS

Digitalni potpisi (eng. *digital signatures*) predstavljaju način za dokazivanje autentičnosti poruke (informacije) i njenog autora. Mogu biti sa i bez trećeg lica (arbitra) koji verifikuje potpis i poruku i mogu se koristiti za kreiranje elektronskih potpisa. Digitalni potpisi se implementiraju korišćenjem kriptografski asimetričnih algoritama. Osim što je nemoguće lažno reprodukovati digitalni potpis, on takođe može da potvrdi da je autor zaista potpisao dokument čak i kada autor negira da je to uradio. Neke od popularnih primena gde se koriste ovi potpisi su: *online* transakcije, *email*, *FTP* servisi, *VPN*.

U Srbiji zakonski kvalifikovani elektronski potpisi se formiraju korišćenjem nekog od sledeća tri algoritma: *RSA* (*Rivest Shamir Adleman*), *DSA* (*Digital Signature Algorithm*), *ECDSA* (*Elliptic Curve Digital Signature Algorithm*); sa propisanim neophodnim minimalnim parametrima.

Pošiljalac i primalac poseduju svoje privatne i javne ključeve (eng. *private and public keys*). Javni ključ se izvodi matematički na osnovu privatnog, ali je nemoguće dobiti privatni poznavajući javni (ideja slična heširanju, mada je matematički pristup drugačiji – koristi se teorija brojeva i prosti brojevi).

Ukoliko primalac poseduje javni ključ pošiljaoca on može garantovano tvrditi da je poruka potpisana privatnim ključem pošiljaoca. Pošiljalac može dodatno enkriptovati poruku javnim ključem primaoca i time zagarantovati da samo autentični primalac može da pročita poruku dekriptovajući je svojim privatnim ključem. Ovime je omogućeno obostrano poverenje čak i kada se poruka šalje kroz nepoznatu i nebezbednu sredinu (internet).

Autor u svojoj implementaciji koristi *ECDSA* algoritam za digitalno potpisivanje transakcija.

Slika 5. Primer šeme *ECDSA* autentifikacije poruke



#### ECDSA

*ECDSA* spada u grupu algoritama koji se često sreću u kriptografiji, a zasnovani su na jednačinama eliptičke krive (oblika *y*2 = *x*3 + a*x* + b). *ECDSA* je često korišćen kao algoritam za enkripciju *online* komunikacije između servera i web browsera zbog svoje brzine u poređenju sa drugim sličnim algoritmima. Koristi se i kod kriptovaluta gde su korisničke adrese u stvari javni ključevi (npr. *Bitcoin* i autorova implementacija).

Slika . Dva primera izgleda eliptičke krive



Sam algoritam je matematički relativno komplikovan, i pošto nije tema ovog rada, nećemo previše duboko zalaziti u njega. Ispod je dat površan primer generisanja ključeva i provere validnosti poruke na osnovu njih pomoću ovog algoritma:

1. Recimo da Ana želi da pošalje potpisanu poruku Branku. Pre same komunikacije oni moraju da ustanove tri unapred poznata parametra: jednačinu krive i polje nad kojim je definisana ***C***, baznu tačku krive u okviru grupe reda nekog prostog broja ***G*** i ***n*** koje predstavlja multiplikativni red od ***G*** ikoje na osnovu Bezuove teoreme mora biti prost broj.
2. Ana kreira privatni ključ ***d­A*** nasumičnim izborom celog broja između 1 i ***n*** - 1 i javni ključ izborom tačke na krivoj ***QA*** = ***d­A*** × ***G***, gde × predstavlja skalarno množenje tačke na krivoj.
3. Ana potpisuje poruku ***m*** prateći sledeće korake:

3.1) Računa ***h*** = ***HASH***(***m***), gde je ***HASH*** unapred poznata heš funkcija i pretvara izlaz u decimalni ceo broj.

3.2) Uzima ***Ln*** najviših bitova od ***h*** i naziva ih ***l***, gde ***Ln*** predstavlja dužinu grupe reda ***n*** u bitima.

3.3) Generiše kriptografski sigurno i nasumično ***k*** koje predstavlja ceo broj između 1 i ***n*** - 1.

3.4) Računa tačku na krivi (***x1***, ***y1***) = ***k*** × ***G***.

3.5) Računa ***r*** = ***x1*** mod ***n***. Ukoliko dobije da je ***r*** = 0, vraća se na korak 3.3.

3.6) Računa ***s*** = ***k***-1 ∙ (***l*** + ***r*** ∙ ***d­A***) mod ***n***. Ukoliko dobije da je ***s*** = 0, vraća se na korak 3.3.

3.7) Potpis je uređeni par (***r***, ***s***). (Validan potpis je i uređen par (***r***,-***s*** mod ***n***) zbog geometrije krive).

Napomena: Veoma je bitno da se za svaki potpis generiše različito ***k*** u tački 3.3, jer u suprotnom je moguće rešiti jednačinu 3.6 po ***d­A***, ukoliko napadač ima pristup više od jednoj poruci. U decembru 2010. god. je zbog ovog propusta *Sony*-jev *PlayStation3* bio hakovan, omogućavajući napadačima kompletnu kontrolu nad operativnim sistemom, a avgusta 2013. nekoliko Android aplikacijaza *Bitcoin* transakcije zbog nepravilne implementacije unutar *Java* klase *SecureRandom* koja je ponekad generisala koliziju za ***k*** vrednosti, što je omogućilo napadačima da se lažno predstavljaju ukradenim privatnim ključevima.

1. Branko dobija poruku ***m***,Anin javni ključ ***QA*** i potpis (***r***, ***s***). Proverava validnost potpisa za tu poruku prateći sledeće korake:

4.1) Proverava da ***QA*** nije jednako jediničnom elementu ***O***, da ***QA*** zaista leži na krivoj i da ***n*** × ***QA*** = ***O***. Ukoliko nešto od prethodnog nije ispunjeno zna da Anin potpis nije validan i ne može da veruje da je poruku zaista dobio od nje.

4.2) Proverava da su ***r*** i ***s*** celi brojevi između 1 i ***n*** – 1. Ukoliko nisu potpis nije validan.

4.3) Računa ***h*** i ***l*** isto kao u 3.1 i 3.2.

4.4) Računa ***u1*** = ***l*** ∙ ***s***-1 mod ***n*** i ***u2*** = ***r*** ∙ ***s***-1 mod ***n***.

4.5) Računa tačku na krivoj (***x2***, ***y2***) = ***u1*** × ***G*** + ***u2*** × ***QA***. Ukoliko je (***x2***, ***y2***) = ***O*** potpis nije validan.

4.6) Potpis je validan ako i samo ako je ***r*** mod ***n*** = ***x2*** mod ***n***. U suprotnom nije validan i ne može verovati da je poruku zaista dobio od Ane.

NONCE I TEŽINA

*Nonce* (*number once*), u prevodu broj sa jednom svrhom, i težinu (eng. *difficulty*) imaju blokovi u *blockchain* sistemima koji kao metod validacije blokova koriste dokaz radom (eng. *proof of work).* Najpoznatiji takav sistem jeste *Bitcoin*, i autor je u svojoj implementaciji takođe koristio tu tehniku dokaza validnosti blokova.

Težina predstavlja broj na osnovu kog se računa neophodan broj prefiksnih nula heš vrednosti celog bloka. Autor je radi jednostavnosti i lakšeg razumevanja implementirao jedan na jedan preslikavanje između težine i broja prefiksnih nula, ali u opštem slučaju postoji funkcija zavisnosti i težina može biti predstavljena čak i decimalnim brojem. Menjanjem težine se u zavisnosti od raspložive procesorske snage svih rudara eksplicitno određuje očekivana vremenska razlika između dodavanja novih blokova.

*Nonce* je broj čija je jedina svrha da kad se hešira zajedno sa ostatkom bloka proizvede validnu heš vrednost, tj. heš vrednost koja započinje sa onoliko nula koliko je odredila težina bloka. Novi blok se dodaje tek kada se pronađe *nonce* koji ovo omogućava, a pronalaženje takvog *nonce*-a se naziva rudarenje (eng. *mining*) u *proof of work* sistemima.

COINBASE

Poslednji deo blockchain-a kao strukture podataka koji je bitan za kriptovalute jeste coinbase. *Coinbase* predstavlja transakciju koja nema pošiljaoca, već samo primaoca koji je u stvari čvor koji je validirao ovaj blok i dodao ga u lanac. To je zapravo nagrada koju validator dobija za uspešnu validaciju bloka i svaki validator prilikom validacije stavlja samoga sebe kao primaoca.

Novi novčići jedino ovako ulaze u sistem. Većina kriptovaluta ima limiran broj tokena koji može da bude u sistemu, i ta ograničenost im daje vrednost i sprečava inflaciju (za razliku od tradicionalnog novca koji nema gornju količinsku granicu). Ograničenost se postiže tako što je na svakin *N* blokova *coinbase* vrednost smanjena.

Primer *Bitcoin*-a: *Coinbase* nagrada se prepolovi na svakih 210,000 blokova. Nagrada na početku *Bitcoin*-a za validaciju bloka je bila čak 50 *BTC*. Sada možemo da izračunamo ukupnu količinu tokena koji može da uđe u sistem preko beskonačne sume:

Do januara 2023. godine je u cirkulaciju ušlo 19,300,000 BTC, a prognozira se da će se do maksimalnog kapaciteta doći oko 2140. godine, jer su procesori naravno limitirani u pogledu najmanjih vrednosti sa kojima mogu da rade, te suma naravno nije beskonačna. Nakon što *Bitcoin* dođe do gornje granice količine tokena, plan je da se izbaci *coinbase* nagrada, a da rudari budu nagrađivani transakcijskim troškovima koji se uzimaju od pošiljaoca/primaoca.

KAO MREŽNI PROTOKOL

*Ledger*-i su neophodna stavka za finansijsku reviziju u svakoj organizaciji. U tradicionalnom digitalnom formatu *ledger* se čuva na centralnom serveru dok su ostalim čvorovima (korisnicima na sopstvenim računarima) (eng. *nodes*) u mreži dodeljene privilegije za čitanje/pisanje (eng. *read/write privilege*). Ovo naravno podrazumeva da postoji čitav bezbednosni mehanizam unutar sistema koji bi korisnike autentifikovao i autorizovao i zatim obezbedio siguran pristup samom centralnom serveru koji bi zapisavao dobijene transakcije i čitao zatražene u/iz *ledger*-a.

Mane ovako opisanog centralizovanog sistema su: ne tako dobra skalabilnost (sa rastom broja čvorova u mreži opada svima brzina pristupa *ledger*-u), *single point of failure* (ako centralni server postane nedostupan iz bilo kog razloga onda niko neće imati pristup *ledger*-u), zavisnost od centralnog autoriteta (sadržaj *ledger*-a i pristup njemu zavise od „dobre volje“ onoga ko kontroliše centralni server).

Alternativan sistem se naziva siguran distribuirani *ledger* (eng. *secure distributed ledger -SDL*).

Siguran znači da se podaci ne smeštaju u jedinstvenu bazu podataka već u nadogradivu *blockchain* strukturu koja onemogućava kasnija menjanja već postojećih podataka za razliku od centralnog servera.

Distribuiran znači da ne postoji centralni autoritet koji nadgleda transakcije i određuje ko ima kakvu vrstu pristupa *ledger*-u već se transakcije i njihova validnost verifikuju postizanjem konsenzusa u mreži, tj. potrebno je da se većina čvorova složi da je transakcija validna pre nego što se ona izvrši. Kao što je već spomenuto svaki korisnik *SDL* sistema (u nastavku teksta se podrazumeva da se misli na *SDL* sistem kad se kaže *blockchain* protokol) ima zasebnu kopiju čitavog lanca na svom lokalnom uređaju (ne važi u svakom trenutku zbog slanja podataka kroz mrežu).

*Blockchain* tako rešava sve tri mane centralizovanog sistema, ali kreira i neke nove: neefikasno korišćenje ukupne računarske memorije i snage (otprilike broj čvorova puta više nego kod centralizovanog sistema), pretraga podataka je spora, jer mora da se ide unazad kroz lanac od bloka do bloka (morale bi da se koriste dodatne strukture podataka za poboljšanje pretrage, što bi dovelo do još veće neefikasnosti korišćenja ukupne memorije), novi korisnici moraju da skinu celu kopiju lanca prilikom prvog pristupa u *blockchain* mrežu (npr. trenutno 435GB kod *Bitcoin*-a i nastaviće da se povećava).

U nastavku ovog poglavlja ćemo videti kako je moguće da anonimna lica unutar decentralizovanog sistema dođu do konsenzusa i koji su to mogući načini validacije ispravnosti lanaca.

TIPOVI BLOKCHAIN-A

#### JAVNI

Javni *blockchain* je ubedljivo najčešća verzija *blockchain*-a. Njemu može da pristupi bilo ko preko interneta. Smatraju se da su zaista u potpunosti decentralizovani, jer ne postoji centralni autoritet koji dodeljuje dozvole čvorovima za pristupanje *ledger*-u ili za njegovo dopunjavanje novim transakcijama. Prednosti su transparentnost podataka i laka revizija, međutim u to ime se žrtvuje privatnost podataka i generalno imaju veću ukupnu potrošnju energije za održavanje ovakvog sistema. Bitcoin je najpoznatiji primer javnog *blockchain*-a. Autorova implementacija se takođe zasniva na javnoj verziji blockchain-a.

#### PRIVATNI

Privatni *blockchain* dozvoljava samo entitetima sa autoritetom ili vlasnicima da dodaju nove transakcije u okviru lanca (npr. jedna kompanija). Dozvole za čitanje podataka mogu biti javne (dostupno svima na internetu) ili privatne (vlasnik dodeljuje dozvole čvorovima kojima on želi). Privatni *blockchain* je druga najpopularnija verzija *blockchain*-a ali dosta zaostaje za javnim. Ne smatraju se decentralizovanim sistemima.

#### KONZORCIJUM

Kod konzorcijuma unapred izabrani skup čvorova odobrava transkacije. Na primer 9 različitih firmi učestvuju u istoj *blockchain* mreži od kojih transakciju mora da validira barem 6 da bi se ona dodala u lanac. Postoje različite varijacije gde čitanje i pisanje mogu biti javni ili privatni. Ne smatraju se poptuno decentralizovanim sistemima.

KONSENZUS ALGORITMI

Najbitnije pitanje kod distribuiranog decentralizovanog sistema jeste kako zagarantovati da svi korisnici mogu da veruju u tačnost informacija koje dobijaju bez centralnog autoriteta koji bi verifikovao transakcije.

Sa tačke gledišta jednog novog korisnika nekog *blockchain* sistema, on je ušao u mrežu gde ne zna sa koliko drugih čvorova komunicira, niti ko su, a zapravo mu je zagarantovano da može da veruje informacijama koje dobija od ostatka mreže. To se postiže uz pomoć konsenzus algoritama, tj. algoritama kojima se umreženi čvorovi usaglašavaju oko validnosti informacija unutar sistema.

Postoji dosta različitih konsenzus algoritama, a u ovoj glavi ćemo proći kroz nekoliko najpoznatijih.

#### PROOF OF WORK

Najpoznatiji, najkorišćeniji i najstariji konsenzus mehanizam (algoritam) jeste dokaz radom (eng. *Proof of Work – PoW*). Njega koristi *Bitcoin*, kao i autorova implementacija, a teorija iza njega postoji već tridesetak godina.

Kod ovog mehanizma posebni čvorovi unutar mreže koji validiraju blokove se zovu rudari (eng. *miners*). Njihov zadatak jeste da pronađu *nonce* vrednost koja daje odgovarajuću heš vrednost kad se hešira zajedno sa ostatkom bloka. Pronalaženje te vrednosti se naziva rudarenje (eng. *mining*).

Rudar koji prvi pronađe jedno moguće *nonce* rešenje kao nagradu dobija *coinbase* svotu kriptovalute. Rudari imaju inicijativu da ulože u što više hardverske opreme za rudarenje (npr. grafičke kartice – ovo je razlog zašto su dosta poskupele od kada je kripto rudarenje postalo popularno), jer time povećavaju šanse da baš oni budu ti koji su pronašli odgovarajuću *nonce* vrednost.

Zbog atributa koje poseduje *CHF*, *nonce* vrednost koju je rudar teško pronašao se lako validira od strane ostatka mreže.

Vremenom kako ukupna procesorska snaga mreže raste tako raste i težina matematičkog problema (*difficulty* raste), tj. smanjuje se šansa pronalaska validnog *nonce*-a i tako se srednje vreme dodavanja novog bloka održava konstantnim.

Rudari se mogu udružiti u veće grupe zvane bazeni (eng. *pools*) i da se ravnomerno svakom članu srazmerno njegovoj procesorskoj snazi dodeljuje *coinbase* ukoliko neki rudar iz tog *pool*-a reši blok.

Mane PoW-a:

* 51% napad: Ukoliko jedan entitet poseduje više od polovine procesorske snage u mreži, onda je moguće da zloupotrebi algoritam u svoju korist, tako što će matematički on biti taj koji najviše transakcija validira i može da u njih unese, npr. nepostojeće transakcije zarad lične koristi. Ovaj napad postoji i kod većine drugih konsenzus algoritama, samo sa drugačijim preduslovima.
* Vremenski i resursno zahtevno: Rudari moraju da potroše dosta vremena i resursa (električne energije, hardver) da bi validirali transakcije (tj. izrudarili blok). Ovo je glavni razlog zašto novije kriptovalute ne koriste ovaj algoritam, jer sistem postaje ekonomski neisplativ tokom vremena (težine blokova su sve veće i to zahteva sve više resursa, a nagrade za njihovo rešavanje su sve manje).
* Transakcije su spore: Zbog spore validacije transakcije nisu instantne. Proces potvrđivanja transakcije u glavnom traje između 10 i 60 minuta trenutno za *Bitcoin*.

#### PROOF OF STAKE

Drugi po redu najpoznatiji i najkorišćeniji konsenzus algoritam jeste dokaz ulogom (eng. *Proof of Stake – PoS*). Na primer kriptovaluta *Ethereum* je prešao sa korišćenja *PoW*-a na *PoS*, jer je smatrao da je ovo bolji način dokazivanja validnosti transakcija.

Čvorovi koji validiraju blokove se zovu validatori i oni stavljaju neku svotu svog novca kao ulog kladeći se na blokove za koje smatraju da su validni i da se mogu dodati u lanac. Nakon toga na osnovu blokova koji su zaista dodati, validatori dobijaju nagradu srazmernu njihovim ulozima.

Mreža onda dodeljuje nekom od validatora zadatak da dodaju novi blok u lanac nakon čega ukoliko se ostali validatori slože da je taj blok validan, svi koji su se kladili na taj blok dobijaju nagradu srazmernu njihovom ulogu.

Ukoliko se ostatak validatora ne slaže da je novododati blok zapravo validan (npr. može biti potrebno da se više od trećine validatora ne slaže), proces se ponavlja, a validator koji je bio prethodno izabran da doda taj blok gubi ceo ili deo svog uloga. Na ovaj način postoji finansijska inicijativa da se validatori slože i takođe postoje kazne za čvorove koji su maliciozni.

*PoS* je kao sistem ekonomski mnogo isplativiji od *PoW*-a, jer ne postoji takmičenje u uloženim i potrošenim resursima (hardverskim, mrežnim, električnim, itd.) među čvorovima, i potrebno je manje vremena da se transakcije potvrde. Međutim i dalje postoji 51% napad, samo što u ovom slučaju napadač mora da kontroliše više od 50% novčanog uloga u validaciji što je teže nego kod *PoW*-a da kontroliše više od 50% resursa, jer je broj validatora mnogo veći (lakše dostupno validiranje znači da veliki broj čvorova može da učestvuje u validaciji).

Mana ovog algoritma jeste da se favorizuju čvorovi sa velikim ulozima, jer je veća šansa da dobiju zadatak da validiraju blok i ukoliko to uspešno urade još više povećavaju svoj ulog u narednim rundama.

#### DELEGATED PROOF OF STAKE

Delegirani dokaz ulogom (eng. *Delegated Proof of Stake – DPoS*) je sličan običnom *PoS*-u osim što validatori glasaju za entitete za koje smatraju da treba da zapravo validiraju blokove (jačina glasa je srazmerna ulogu), umesto da ga algoritam sam izabere. U većini sistema broj validatora koji mogu da budu izabrani je između 21 i 101.

Izabrani validatori se zovu svedoci (eng. *witnesses*). Kada svedok uspešno validira blok, on i njegovi glasači dobijaju novčanu nagradu. Svedoci imaju ograničeno vreme da validiraju novi blok i ukoliko ne uspeju blok se smatra promašenim, transakcije ostaju i dalje nevalidirane, a nagrada se dodaje na sledeću nagradu koju dobija sledeći svedok koji uspešno validira taj blok.

Ukoliko svedok pokuša da uradi nešto maliciozno, on gubi svoj ulog isto kao kod *PoS*-a, tako da čvorovi nemaju nikakvu inicijativu da napadaju sistem. Određivanje da li je blok koji je svedok validirao mogu da rade svi čvorovi u mreži, mada za ovo ne postoje nagrade, već je inicijativa održavanje tačnosti i stabilnosti mreže.

U mreži postoje i posebni čvorovi zvani delegati koji nadgledaju celu mrežu i predlažu promene, npr. broj svedoka, veličine blokova, nagradne svote, itd.. Svi čvorovi mogu da učestvuju u biranju delegata i u glasanju za predložene promene.

Prednosti u odnosu na PoS algoritam jeste što je mreža više demokratizovana i finansijski inkluzivnija, jer je generalno potreban mali ulog. Manji ulog znači da više čvorova učestvuje u validaciji transakcija i napad na mrežu je teži.

#### PROOF OF BURN

Dokaz spaljivanjem (eng. Proof of Burn – PoB) je sličan PoS-u međutim umesto da validatori stavljaju novac kao ulog, koji kasnije može da se povuče ukoliko čvor ne želi više da bude validator, validatori spaljuju novac tako što ga šalju na nedostupnu adresu sa koje ne mogu nikako da ga vrate.

Mreža isto kao u PoS-u određuje koji validator će dobiti pravo da validira sledeći blok, a veću prednost imaju čvorovi koji su više novca spalili. Ukoliko validator uspešno validira blok dobija nagradu, a ukoliko ne uspe dobija zabranu od dalje validacije. Ovime se validatori dugoročno obavezuju da će ispravno validirati blokove, jer im je u suprotnom finansijski neisplativo.

Problem *PoB*-a je što se kao i kod *PoW*-a nepotrebno troše resursi (novac u ovom slučaju), i isto kao kod *PoS*-a čvorovi sa većim novčanim sredstvima su u prednosti da više zarađuju.

Primer kriptovalute koja koristi PoB je *BitShares*.

#### PROOF OF CAPACITY

Dokaz kapacitetom (eng. *Proof of Capacity – PoC*) je jedan od najnovijih algoritama koji je sličan *PoW*-u osim što rudari ne ulažu u procesorsku snagu već kapacitet hard diskova, što ga čini dosta ekonomski isplativijim i boljim ekološkim rešenjem, jer troši mnogo manje električne energije.

Rudari pre procesa samog rudarenja moraju da na svojim hard diskovima sačuvaju velike fajlove pre-izračunatih heš funkcija koje se inače veoma teško računaju. Čuvanje ovog fajla se naziva *plotting* i može da potraje i nedeljama. U ove heš funkcije ulaze i podaci o rudaru tako da su unikatne za svakog rudara. Svaki *nonce* se sastoji od nekog broja *scoop*-ova (npr. 4096 kod *Burstcoin*-a), gde *scoop* predstavlja dve uzastopne heš vrednosti.

Rudar zatim izrudari (na sličan način kao kod *PoW*-a, samo što je problem dosta lakši) broj *scoop*-a koji ubacuje u svaki *nonce* (npr. rudarenjem dobije broj 12, i onda redom posećuje svaki 12-ti scoop svakog *nonce*-a koji ima sačuvan na hard disku) ne bi li dobio što manju *deadline* vrednost. Očigledno što više nonce-ova ima sačuvanih, to je veća šansa da dobije manji *deadline*. *Deadline* vrednost predstavlja broj sekundi od generisanja prethodnog bloka koji ostali rudari imaju da naprave novi validan blok. Ukoliko ne uspeju, rudar koji je izračunao taj *deadline* dobija nagradu i on je izabran da validira sledeći blok.

Iako je brži i pristupačniji od *PoW*-a, mogu se javiti slični problemi gde rudari masovno kupuju velike količine hard diskova ne bi li imali bolje šanse da dobiju *coinbase* nagradu. Takođe *malware* se dosta teže detektuje nego kod *PoW*-a, a i sam algoritam nije puno rasprostranjen među *blockchain* sistemima.

#### PROOF OF ELAPSED TIME

Dokaz proteklim vremenom (Proof of Elapsed Time – PoET) je najviše fer algoritam od svih koje smo do sada pogledali, tj. svi čvorovi imaju podjednake šanse da budu izabrani kao validatori sledećeg bloka, međutim algoritam se u glavnom koristi na privatnim mrežama ili mrežama gde čvorovi nisu anonimni.

Algoritam je osmislio Intel 2016. godine. Funkcioniše tako što je svakom čvoru dodeljeno određeno nasumično vreme koje procesori moraju da budu u *sleep* režimu. Procesor koji se prvi probudi, tj. kome je bilo dodeljeno najmanje vreme dobija pravo kreiranja novog bloka i nagradu.

BLOCKCHAIN FORKING

Decentralizovanost sistema znači da korisnici moraju da dođu do konsenzusa oko trenutnog stanja lanca. Ukoliko se svi čvorovi slože onda postoji samo jedna verzija lanca, međutim to često nije slučaj.

Zbog toga što je potrebno vreme da informacije prođu kroz mrežu moguće je da se čvorovi ne slažu jednoglasno oko toga kako lanac trenutno izgleda. Tada dolazi do razdvajanja *blockchain*-a u više paralelnih lanaca istovremeno i ovo se naziva živo slučajno/trenutno razdvajanje (eng. *live accidental/temporary fork*). Ovo se veoma često dešava u *blockchain* sistemima, ali se lako razrešava tako što čvorovi preuzimaju tok lanca od sledećeg rudara.

Živo namerno razdvajanje (eng. *live intentional fork*). može da bude meko (eng. *soft*) i tvrdo (eng. *hard*). Meko označava da je nova verzija blockchain-a kompatibilna unazad (eng. *backwards compatible*), tj. da su uvedena nova strožija pravila u sistemu i da su novi blokovi smatrani validnim od strane starijih verzija softvera (npr. uvedene nove adrese). Tvrdo razdvajanje označava suprotnu stvar, tj. uvedena su nova pravila koja su u suprotnosti sa prethodnim pravilima, i čvorovi koji pređu na novu verziju sistema ne mogu da interaguju sa čvorovima koji su na staroj (npr. kada je Ethereum prešao sa *PoW*-a na *PoS*).

Postoji i *codebase fork* koji predstavlja mogućnost da dođe do razdvajanja lanca prilikom izlaska potpuno nove verzije softvera zasnovane na nekom već postojećem *blockchain* sistemu (npr. nova kriptovaluta zasnovana na Bitcoin-u, sa minimalnim izmenama sistema), ali to nećemo detaljno razmatrati.

ISTORIJA

Iako se *blockchain* kao termin koristi tek od skoro, njegova istorija počinje pre 40 godina. Trezorni sistem (eng. *vault system*) je tehnologija nalik *blockchain*-u koja se prvi put pominje još 1982. godine u doktorskoj disertaciji pod nazivom “Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups” od strane autora Dejvida Čauma (eng. *David Chaum*) [1]. Dejvid se smatra pionirom kriptografije i tehnologija koje čuvaju privatnost i izumiteljem digitalnog novca. U svojoj disertaciji dao je kod pomoću kog bi se implementirao protokol kao i skoro sve elemente *blockchain*-a koje danas koristi *Bitcoin*, čak 27 godina pre nego što je on uopšte i izumljen.

Sledeći veliki korak u razvoju *blockchain*-a daju 1991. god. kriptografi Stjuart Hejber (eng. *Stuart Haber*) i Skot Storneta (eng. *Scott Stornetta*). Prvobitna upotreba ove verzije *blockchain*-a nije imala veze sa novcem, već su njegovi tvorci zamislili da se on koristi za dokazivanje autentičnosti digitalnih dokumenata korišćenjem vremenskih oznaka (eng. *timestamp*) i kriptografije. Oni su prvi implementirali Merkleova stabla u ove svrhe, a *New York Times* još od 1995. god. koristi njihovu ideju za dokazivanje autentičnosti digitalnih dokumenata heširanjem (Slika 7).

Dalji doprinos razvoju *blockchain*-a dali su u svojim radovima iz polja kriptografije 1996. Ros Anderson (eng. *Ross Anderson*), a potom 1998. Brus Šnajer (eng. *Bruce Schneier*) i Džon Kelsi (eng. *John Kelsey*).

1998. god. Nik Šabo (eng. *Nick Szabo*) daje teorijsku osnovu za jednu od prvih kriptovaluta ikada zvanu *bit gold*. Iako nikad nije zapravo bila implementirana, u teoriji ona koristi dosta mehanizama koje i današnje stvarne kriptovalute koriste. Nik je takođe dao teorijsku osnovu za pametne ugovore (eng. *smart contracts*) koje danas koriste novije verzije kriptovaluta.

2000. god. Stefan Konst (eng. *Stefan Konst*) objavljuje u naučnom radu način rada kriptografski ulančanih lanaca podataka i daje svoje ideje za implementaciju istog.

Konačno 2008. god. objavljen je rad koji čini prekretnicu u ovom polju od strane anonimnog lica ili grupe ljudi pod pseudonimom Satoši Nakamoto (eng. *Satoshi Nakamoto*). Naziv rada je “Bitcoin: A peer-to-peer electronic cash system” i u njemu je opisan *blockchain* kao protokol i decentralizovana distribuirana baza podataka koja omogućava korišćenje danas najpoznatije kriptovalute *Bitcoin*.

Slika . New York Times koristi Merkleovo stablo heširanih digitalnih dokumenata kao dokaz njihove autentičnosti



2009. god. Nakamoto implementira prvi moderan *blockchain* koji je opisao u svom radu prethodne godine. Satoši je u svom radu zasebno koristio reči *block* i *chain*, a termin *blockchain* se pojavio tek oko 2014. godine.

Danas se Satošijeva verzija *blockchain*-a naziva *Blockchain 1.0*. On je unapredio prvobitnu Dejvid Čaumovu zamisao koristeći mehanizam koji se zove dokaz radom (eng. *Proof of Work*) i dodao parametar težine (eng. *difficulty*) kako bi ograničio količinu kriptovalute koja ulazi u cirkulaciju.

Slika . Sadržaj *genesis block*-a *Bitcoin*-a sa obeleženom skrivenom porukom



Na Slici 8 prikazan je izgled koda prvog bloka *Bitcoin*-ovog *blockchain*-a. Prvi blok u lancu se zove blok postanka (eng. *genesis block*) i on u većini *blockchain* implementacija ima unapred zadate vrednosti promenljivih u kodu (eng. *hardcoded*). Zanimljiv detalj je da je Nakamoto ubacio poruku u dve promenljive (coinbase i potpis koje ćemo objasniti u poglavlju *x*) koja glasi “The Times 03/Jan/2009 Chancellor on brink of second bailout for banks” referišući jedan od naslova na naslovnoj stranici čuvenih britanskih novina *The Times*. Postoje razne spekulacije oko značenja ove poruke, ali je sigurno da je ta poruka služila, između ostalog, kao dokaz da blok nije nastao pre 3. januara 2009. godine.

Nakon Satošijevog rada koji je dao teorijsku osnovu za kriptovalute, tržište je preplavljeno različitim implementacijama *blockchain*-a i kriptovalutama vezanim za iste. Sledeći veliki skok se desio 2014. god. nastankom nove verzije *blockchain*-a – *Blockchain 2.0*. Ono što je razlikovalo novu verziju od stare jeste upotreba pametnih ugovora, koji su sada mogli da se ubace u blokove u okviru *blockchain*-a. Prva kriptovaluta (i danas druga najpoznatija, odmah iza *Bitcoin*-a) koja je implementirala *Blockchain 2.0* je bila *Ethereum*. Pametni ugovori su omogućili programerima da implementiraju kompleksnu logiku i razvijaju čitave aplikacije u okviru postojećih *blockchain* platformi.

*Blockchain 2.0* je između ostalog omogućio i kreiranje nekoliko popularnih, mada ne i pravno jasno definisanih entiteta i procedura:

* Decentralizovana autonomna organizacija (eng. *decentralized autonomous organization - DAO*) – Organizacija bez centralnog nadležnog tela koje ne odgovara nikome i uglavnom se bavi posredovanjem kriptovaluta.
* Prva ponuda novčićima (eng. *initial coin offering - ICO*) – Metod prvobitnog izlaska na tržište nekog preduzeća, sličan prvoj javnoj ponudi, koji se u ovom slučaju sastoji od kupovine novčića (eng. *coin*) ili tokena popularnim kriptovalutama (npr. *Bitcoin*-om ili *Ethereum*-om) od strane vlasnika budućih deonica. Dodatni novčani prihodi se najčešće koriste da finansiraju tehnologije i platforme kojima bi vlasnici tokena prvi (ili jedini) imali pristup.
* Nezamenljivi tokeni (eng. *non-fungible tokens - NFTs*) – Jedinstveni tokeni (često u obliku neke slike ili audio zapisa) koje vlasnici skupljaju u kolekcije.

Trenutno poslednji korak u evoluciji blockchain-a predstavlja *Blockchain 3.0* koji se trenutno i dalje razvija. Ova verzija nema tako jasno određenu definiciju, međutim konsenzus jeste da je primenljivija u mnogo više oblasti nego prethodne verzije koje su bile ograničene na isključivo ekonomsku i finansijsku primenu. Mane trenutnih verzija blockchain-a su održivost, skalabilnost, isplativost, a postoji i dosta prostora za dalju decentralizaciju i povećanje sigurnosti. Želja *Blockchain*-a *3.0* jeste da poboljša sve prethodno pomenute nedostatke i da omogući neprimetno uključivanje korporativnih aplikacija i baza podataka u decentralizovane sisteme radi sigurnosti i transparentnosti.

# Implementacija

Autor je odlučio da implementira sistem za vizualizaciju *blockchain* tehnologije koristeći popularne web tehnologije, jezike i njihove okvire (eng. *frameworks*) – HTML, CSS i Javascript zbog toga što pružaju jednostavnost i automatizaciju kreiranja tekstualnih formi koje čine srž sistema.

Tekstualne forme su izabrane pošto je glavni cilj rada vizualna reprezentacija i autor je smatrao da je najbolje prikazati rad *blockchain*-a pomoću interaktivnog sadržaja kojim su predstavljeni podaci unutar izmišljenog *blockchain* sistema sa kojim korisnici web aplikacije mogu da interaguju i da vide u realnom vremenu kako izmene u jednom bloku utiču na ostale podatke unutar istog bloka kao i čitavog lanca.

Korišćene Tehnologije

HTML

*HyperText Markup Language – HTML* je jednostavan platformski nezavisan jezik za označavanje (eng. *markup language*) dokumenata koji su namenjeni za prikazivanje u web čitačima (eng. *browser*). Ti dokumenti se nazivaju web stranice.

Markup language qweqweqwe

*HTML* semantički opisuje strukturu web stranice pomoću *HTML* elemenata zvanih tagovi koji se navode između karaktera < i >. Web *browser*-i ne prikazuju tagove u obliku teksta već pomoću njih struktuiraju web stranice. *HTML* fajlovi se završavaju ekstenzijom .htm ili .html.

Pomoću *HTML* elemenata se umeću ostale tehnologije poput *CSS*-a i *Javascript*-a u web stranice.

#### Pug

*Pug* (prethodno poznat kao *Jade*) je *Node*-ov generator šablona (eng. *template engine*). Kompajlira se u *HTML* i ima jednostavniju sintaksu. *Pug* fajlovi se završavaju ekstenzijom .pug.

CSS

*Cascading Style Sheets – CSS* je jezik koji služi za opis prezentacije dokumenta koji je napisan u *markup* jeziku. Dizajniran je tako da može da omogući razdvajanje sadržaja stranice od izgleda stranice.

qweqweqweqwe

#### SASS

*Syntatically Awesome Stylesheet – SASS* je *CSS* preprocesor koji omogućava korišćenje odlika koje ne postoje u čistom *CSS*-u poput promenljivih, nasleđivanja, funkcija, ugnežđenih pravila, ...

*SASS* fajlovi se završavaju ekstenzijom .scss.

#### Bootstrap

*Bootstrap* je front-end *CSS framework* koji prema statistici koristi svaki peti web sajt. Napravljen je pomoću *SASS*-a, a inicijalno je bio razvijen u okviru kompanije *Twitter* radi lakšeg osiguravanja konzistencije stilova između različitih alata.

Javascript

Kao glavni programski jezik korišćen je *Javascript*. *Javascript*-ov *Node.js framework* je korišćen za uspostavljanje servera na kom se pokreće aplikacija, čist *Javascript* i *jQuery* okvir su korišćeni za front-end validaciju i raznolika računanja podataka. *Node* *packet manager* je korišćen za upravljanje bibliotekama.

#### Node.js

Qweqwewqe

#### jQuery

qweqweqweqwe

Organizacija fajlova

qweqweqwe

METADATA fajlovi

qweqweqwe

Index.js

qweqweqwe

Layout.pug

Qweqweqwe

Bs-config.js

Qweqweqwe

Globalne Funkcije

qweqweqwe

# Primer Rada sistema

U ovoj glavi ćemo proći kroz rad čitave aplikacije od početka do kraja i kroz interaktivne primere dodatno objasniti rad *blockchain*-a, ali sada sa praktičnog aspekta.

Stranice unutar aplikacije za koje će biti priložene slike će biti stavljene u zasebna poglavlja, a za neke kraće funkcije će dodatno biti priloženi i delovi koda pomoću kojih su prikazane funkcionalnosti implementirane.

Pokretanje aplikacije

Aplikacije se pokreće preko komandne linije iz *root* direktorijuma komandom npm run dev**,** gde dev u stvari predstavlja alijas za nodemon ./index.js, kako je definisano u fajlu package.json. Druga bitna komanda za pokretanje prilikom razvijanja aplikacije je npm run ui, gde je ui alijas za browser-sync start --config bs-config.js.

dev komanda pokreće lokalni server pokretanjem komandi definisanih u index.js fajlu, postavlja aplikaciju na njega i osluškiva port 8000, dok komanda ui konfiguriše aplikaciju tako da se prilikom njenog razvijanja (eng. *development*) automatski, bez potrebe za restartovanjem servera, najnovija verzija aplikacije nalazi na portu 8000 ili portu 3000 (koji je proxy za port 8000) prilikom svake promene *source* koda.

Aplikaciji se može pristupiti preko više *URL*-ova unutar web čitača:

* http://localhost:3000/
* http://localhost:8000/
* http://<*privatna IP adresa servera*>:3000

U poslednjem slučaju moguće je pristupiti aplikaciji i sa drugih uređaja unutar iste mreže kao što je prikazano na slici X ispod. Zahvaljujući *Bootstrap*-u, *CSS* stilovi su automatski rekonfigurisani za druge oblike ekrana.

Slika . Pristupanje aplikaciji sa mobilnog uređaja



Index.pug

Početna (Indeks) stranica sa kratkim upustvima za korišćenje sajta data je na slici X.

Slika . Početna (Index) stranica



Njoj se ponovo može pristupiti klikom na ikonicu lanca u navigacionom meniju. Navigacioni meni pored te ikonice sadrži za svaku stranicu po jedan hiperlink (Slika X).

Slika . Navigacioni meni



Aplikacija je zamišljena tako da se inkrementalno gradi čitav blokčejn sistem, dodavajući nove elemente na svakoj sledećoj stranici.

CHF.pug

*CHF* je skraćenica za *Cryptographic Hash Functions*, tj. kriptografske heš funkcije. Ova stranica se sastoji od jednog *TextArea* elementa promenljive veličine koji korisnik može da koristi za proizvoljan tekstualni unos i 4 *TextBox*-a koja predstavljaju izlaz različitih *CHF* u stvarnom vremenu u zavisnosti od trenutnog korisničkog unosa (Slika X).

Slika . Hash (CHF) stranica



Stranica demonstrira rad različitih kriptografskih heš funkcija, gde korisnik može da vidi da je izlaz svake od *CHF* uvek iste veličine za bilo koji unos, i njihove osobine poput: kompjuterski nemogućeg pronalaska dve iste poruke koje daju istu *hash* vrednost, lavinskog efekta – mala promena u početnoj poruci dovodi do velike promene u izlazu funkcije, nemogućnost pronalaska početne poruke znajući izlaznu vrednost.

Prelaskom miša preko *label*-a sa leve strane iskače tooltip u kom je dat kratak opis za svaku od *CHF* (Slika X). Ovo važi i za labele na drugim stranicama, međutim zbog dužine rada te slike neće biti priložene.

Slika . Primer *tooltip*-a na *label*-i SHA512



Primer rada *hash* funkcija će u nastavku biti priložen kroz vizuelne primere:

Slika . Primer izlaza hash funkcija za ulaz Hello world



Slika . Primer izlaza hash funkcija za ulaz *8576b8cd3ade0bed4379b6f22844eef4*



Slika . Primer izlaza hash funkcija za ulaz *Helo world*



Iako postoji jednoslovna razlika u ulazima funkcija na slikama X i Y, izlazi su veoma različiti, te je nemoguće uopšte i pretpostaviti da su razlika u ulazima toliko mala. Ukoliko izlaz *MD5* funkcije sa slike X dovedemo sada kao novi ulaz (Slika Y), takođe se dobija novi neprepoznatljiv niz heksadecimalnih znakova.

Treba imati u vidu da je *MD5* potpuno kriptografski razbijena, dok je *SHA1* podložna kolizionim napadima, tako da se ove dve funkcije ne smatraju bezbednim. U nastavku aplikacije i rada se koristi samo *SHA256*.

Za generisanje izlaza ovih kriptografskih funkcija je korišćena javno dostupna Javascript biblioteka *CryptoJS*. Poziv ovih funkcija je dosta jednostavan:

$('#sha256').val(sha256());

function sha256(chain, block) { return CryptoJS.SHA256(getText(chain, block)); }

function getText(chain, block){ return $('#data').val(); }

Blok koda . Primer poziva i korišćenja sha256 funkcije

U ovom slučaju nemamo parametre chain i block, tako da je ponašanje funkcije getText(chain, block) uvek isto – vraća vrednost korisničkog unosa iz elementa sa *ID*-jem data, a to je gorepomenuta *TextArea*. *Javascript* nam omogućava da pozovemo funkcije i bez parametara iz njihovih definicija, što nama u ovom slučaju odgovara, jer se oni svakako i ne koriste.

Svaka stranica *overload*-uje funkciju getText() novom definicijom, što je u suštini samo nadovezivanje (konkateniranje) svih mogućih vrednosti polja za korisnički unos iz jednog bloka u jedan string.

sha256(chain, block) vraća *hash* vrednost bloka prosleđenog kroz parametre pomoću poziva CryptoJS.SHA256 funkcija čija implementacija je data u nastavku:

(function (Math) {

        // Shortcuts

        var C = CryptoJS;

        var C\_lib = C.lib;

        var WordArray = C\_lib.WordArray;

        var Hasher = C\_lib.Hasher;

        var C\_algo = C.algo;

        // Initialization and round constants tables

        var H = [];

        var K = [];

        // Compute constants

        (function () {

            function isPrime(n) {

                var sqrtN = Math.sqrt(n);

                for (var factor = 2; factor <= sqrtN; factor++) {

                    if (!(n % factor)) { return false; }

                }

                return true;

            }

            function getFractionalBits(n) {

                return ((n - (n | 0)) \* 0x100000000) | 0;

            }

            var n = 2;

            var nPrime = 0;

            while (nPrime < 64) {

                if (isPrime(n)) {

                    if (nPrime < 8) {

                        H[nPrime] = getFractionalBits(Math.pow(n, 1 / 2));

                    }

                    K[nPrime] = getFractionalBits(Math.pow(n, 1 / 3));

                    nPrime++;

                }

                n++;

            }

        }());

        // Reusable object

        var W = [];

        /\*\*

         \* SHA-256 hash algorithm.

         \*/

        var SHA256 = C\_algo.SHA256 = Hasher.extend({

            \_doReset: function () {

                this.\_hash = new WordArray.init(H.slice(0));

            },

            \_doProcessBlock: function (M, offset) {

                // Shortcut

                var H = this.\_hash.words;

                // Working variables

                var a = H[0];

                var b = H[1];

                var c = H[2];

                var d = H[3];

                var e = H[4];

                var f = H[5];

                var g = H[6];

                var h = H[7];

                // Computation

                for (var i = 0; i < 64; i++) {

                    if (i < 16) {

                        W[i] = M[offset + i] | 0;

                    } else {

                        var gamma0x = W[i - 15];

                        var gamma0  = ((gamma0x << 25) | (gamma0x >>> 7))  ^

                                      ((gamma0x << 14) | (gamma0x >>> 18)) ^

                                       (gamma0x >>> 3);

                        var gamma1x = W[i - 2];

                        var gamma1  = ((gamma1x << 15) | (gamma1x >>> 17)) ^

                                      ((gamma1x << 13) | (gamma1x >>> 19)) ^

                                       (gamma1x >>> 10);

                        W[i] = gamma0 + W[i - 7] + gamma1 + W[i - 16];

                    }

                    var ch  = (e & f) ^ (~e & g);

                    var maj = (a & b) ^ (a & c) ^ (b & c);

                    var sigma0 = ((a << 30) | (a >>> 2)) ^ ((a << 19) | (a >>> 13)) ^ ((a << 10) | (a >>> 22));

                    var sigma1 = ((e << 26) | (e >>> 6)) ^ ((e << 21) | (e >>> 11)) ^ ((e << 7)  | (e >>> 25));

                    var t1 = h + sigma1 + ch + K[i] + W[i];

                    var t2 = sigma0 + maj;

                    h = g;

                    g = f;

                    f = e;

                    e = (d + t1) | 0;

                    d = c;

                    c = b;

                    b = a;

                    a = (t1 + t2) | 0;

                }

                // Intermediate hash value

                H[0] = (H[0] + a) | 0;

                H[1] = (H[1] + b) | 0;

                H[2] = (H[2] + c) | 0;

                H[3] = (H[3] + d) | 0;

                H[4] = (H[4] + e) | 0;

                H[5] = (H[5] + f) | 0;

                H[6] = (H[6] + g) | 0;

                H[7] = (H[7] + h) | 0;

            },

            \_doFinalize: function () {

                // Shortcuts

                var data = this.\_data;

                var dataWords = data.words;

                var nBitsTotal = this.\_nDataBytes \* 8;

                var nBitsLeft = data.sigBytes \* 8;

                // Add padding

                dataWords[nBitsLeft >>> 5] |= 0x80 << (24 - nBitsLeft % 32);

                dataWords[(((nBitsLeft + 64) >>> 9) << 4) + 14] = Math.floor(nBitsTotal / 0x100000000);

                dataWords[(((nBitsLeft + 64) >>> 9) << 4) + 15] = nBitsTotal;

                data.sigBytes = dataWords.length \* 4;

                // Hash final blocks

                this.\_process();

                // Return final computed hash

                return this.\_hash;

            },

            clone: function () {

                var clone = Hasher.clone.call(this);

                clone.\_hash = this.\_hash.clone();

                return clone;

            }

        });

        /\*\*

         \* Shortcut function to the hasher's object interface.

         \* @param {WordArray|string} message The message to hash.

         \* @return {WordArray} The hash.

         \* @static

         \* @example

         \*     var hash = CryptoJS.SHA256('message');

         \*     var hash = CryptoJS.SHA256(wordArray);

         \*/

        C.SHA256 = Hasher.\_createHelper(SHA256);

        /\*\*

         \* Shortcut function to the HMAC's object interface.

         \* @param {WordArray|string} message The message to hash.

         \* @param {WordArray|string} key The secret key.

         \* @return {WordArray} The HMAC.

         \* @static

         \* @example

         \*     var hmac = CryptoJS.HmacSHA256(message, key);

         \*/

        C.HmacSHA256 = Hasher.\_createHmacHelper(SHA256);

    }(Math));

Blok koda . Implementacija CryptoJS.SHA256 funkcije

Block.pug

Ova stranica prikazuje osnovna polja svakog bloka. *Data* polje sa prethodne stranice je sada razbijeno na tri polja i zadržana je samo *SHA256 hash* funkcija. Ta polja su predstavljena promenljivim *TextBox*-ovima za magični broj - *Magic* i *ID* (ili vremenski žig) – *Block (number)*, a *TextArea*-jom za same podatke unutar bloka, koji se takođe zovu *Data*. *Hash* vrednost stringa koji se dobije kada se sva prethodna polja konkatiniraju se nalazi u nepromenljivom *TextBox*-u (Slika X).

Slika . Izgled *Block* stranice



*Magic* i *Block* polja su podešena lambda funkcijama vezanim za događaje (eng. *events*) tako da mogu da primaju samo numeričke karaktere i da ne mogu da ostanu prazna, dok *Data* može da primi sve *ASCII* karaktere i može sadržati prazan unos. Kao i na prethodnoj stranici, *Hash* polje se računa u stvarnom vremenu pri svakoj promeni nekog od prethodnih polja bez poziva parametara.

          $('#magic').bind('input propertychange', function() {

                this.value = this.value.replace(/[^0-9]/g, '');

                $('#hash').val(sha256());

            });

            $('#magic').bind('focusout', function() {

                if(this.value==''){

                    this.value='0';

                    $('#hash').val(sha256());

                }

            });

$('#data').bind('input propertychange', function() {

                $('#hash').val(sha256());

            });

Blok koda . Primer korišćenja lambda funkcija vezanih za određene akcije

Blockchain.pug

Na blokčejn stranici uvodimo dosta novih stvari koje će se koristiti do kraja rada i aplikacije:

* Enumeraciju lanaca i blokova koji se koriste kao parametri funkcija
* Polje koje nam zapravo omogućava da ulančavamo blokove – *Prev*, skraćeno za *Previous Hash*
* *Pug* module koji u suštini predstavljaju stranice unutar stranica, time je realizovano ponavljanje blokova koji se referenciraju brojevima lanca (za sada imamo samo jedan) i bloka
* Provere validnosti blokova na osnovu različitih polja

Inicijalne vrednosti polja u blokovima su unapred izračunate i manuelno postavljene na određenu vrednost (eng. *hard coded*) prilikom inicijalizacije stranice. *Prev* vrednosti su podešene tako da su identične *Hash*-u prethodnog bloka, osim u prvom bloku (eng. *genesis block*) koji u suštini može da sadrži proizvoljnu *Prev* vrednost, jer nema prethodnika, tako da je radi jednostavnosti inicijializovana svim nulama. Pošto postoji veći broj blokova, stavljen je *scroll bar* kojim korisnik može da se kreće kroz lanac (Slika X).

Svaki blok je generisan na osnovu *Pug* fajla *includes/simpleblock.pug* čiji je sadržaj naveden ispod:

Slika . Izgled Blockchain stranice



div.card.bg-light(id='chain'+simpleblock.chain+'block'+simpleblock.block+'card')

    div.card-body

        form

            div.mb-3.row

                label.col-sm-2.control-label.pt-2(for='chain'+simpleblock.chain+'block'+simpleblock.block+'magic') Magic:

                div.col-sm-10

                    div.input-group

                        span.input-group-text #

                        input.form-control(id='chain'+simpleblock.chain+'block'+simpleblock.block+'magic', type='text', value=simpleblock.magic)

            div.mb-3.row

                label.col-sm-2.control-label.pt-2(for='chain'+simpleblock.chain+'block'+simpleblock.block+'blockid') Block:

                div.col-sm-10

                    div.input-group

                        span.input-group-text #

                        input.form-control(id='chain'+simpleblock.chain+'block'+simpleblock.block+'blockid', type='text', value=simpleblock.blockid)

            div.mb-3.row

                label.col-sm-2.col-form-label(for='chain'+simpleblock.chain+'block'+simpleblock.block+'data') Data:

                div.col-sm-10

                    textarea.form-control(id='chain'+simpleblock.chain+'block'+simpleblock.block+'data',rows='10')

            div.mb-3.row

                label.col-sm-2.control-label.pt-2(for='chain'+simpleblock.chain+'block'+simpleblock.block+'prev') Prev:

                div.col-sm-10

                    input.form-control(id='chain'+simpleblock.chain+'block'+simpleblock.block+'prev', type='text', value=simpleblock.prev)

            div.mb-3.row

                label.col-sm-2.control-label.pt-2(for='chain'+simpleblock.chain+'block'+simpleblock.block+'hash') Hash:

                div.col-sm-10

                    input.form-control(id='chain'+simpleblock.chain+'block'+simpleblock.block+'hash', type='text', disabled)

Blok koda . Sadržaj *simpleblock.pug* fajla

Kao što vidimo, svi *ID*-jevi elemenata su parametrizovani (npr. id='chain'+simpleblock

.chain+'block'+simpleblock.block+'hash') tako da jednostavno možemo da referenciramo svaki od njih u fajlu koji sadrži više ovakvih blokova:

div.row.row-horizon(style='flex-wrap: nowrap')

            - var blocks = []

            - blocks.push({chain: 1, block: 1, magic: 74567, blockid: 1, prev: '0000000000000000000000000000000000000000000000000000000000000000'})

            - blocks.push({chain: 1, block: 2, magic: 74567, blockid: 2, prev: '050a3c1fe50deb0b49aa719b33f55e99114e6ce000e10ef855cb6bb5ea2a0624'})

            - blocks.push({chain: 1, block: 3, magic: 74567, blockid: 3, prev: '91b8325cfe25967ce30e8d6ea5d124b2c026695aedfc5dc6cc7068fd5277119b'})

            - blocks.push({chain: 1, block: 4, magic: 74567, blockid: 4, prev: '3eae7011c3e4525f8d5f221f8c60e3facec881040caf74abd3b648ff835cf3a3'})

            - blocks.push({chain: 1, block: 5, magic: 74567, blockid: 5, prev: '4be027c3c4df33d226a352d337ba400dc7097c085632ade089bc56ff9c1e47eb'})

            - blocks.push({chain: 1, block: 6, magic: 74567, blockid: 6, prev: 'dfbfed7dd4c6e972ffad17325d87cc82f3a5663aea49344f8dd658b645f8251a'})

            each simpleblock in blocks

                div.col-lg-6

                    include includes/simpleblock

Blok koda . Korišćenje pug fajla unutar drugog pug fajla

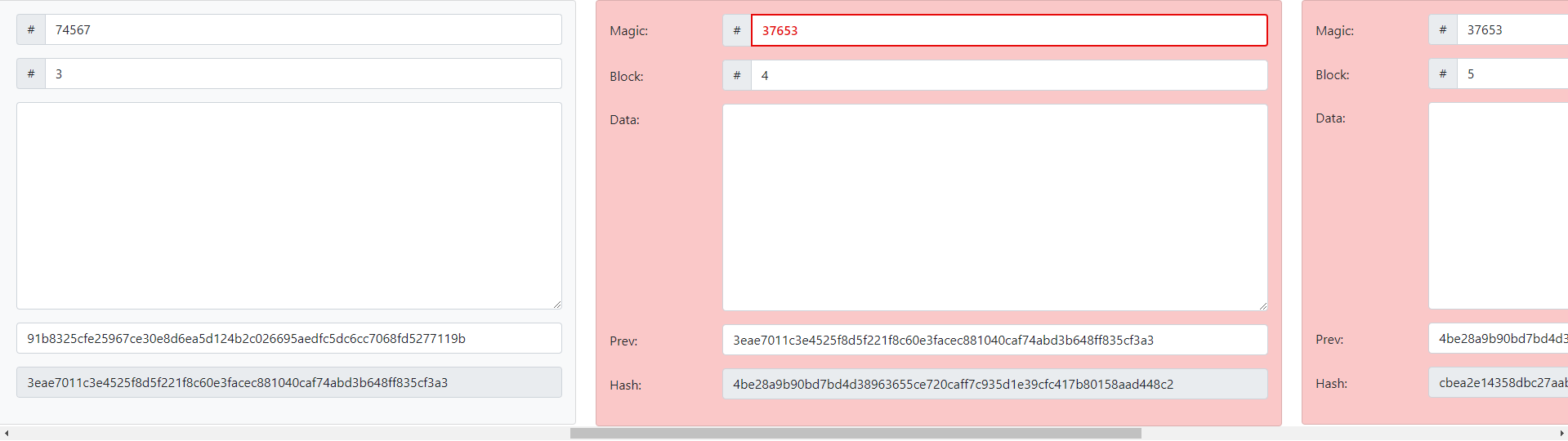
Kao što je već pomenuto, a i može se videti na gornjoj slici, *Prev* polja su izračunata unapred zbog bržeg učitavanja stranice i boljeg korisničkog iskustva. Sličan niz blokova će se koristiti na većini preostalih stranica.

Sada ćemo da prođemo kroz mehanizme provere validnosti blokova unutar ovog jednostavnog lanca:

1. Magični broj – Svi blokovi moraju da sadrže identičnu vrednost u *Magic* polju.

Vrednost magičnog broja se proverava sa vrednošću broja iza i tako sve do *genesis* bloka. Ovo znači da svi blokovi moraju da sadrže magični broj koji se nalazi u *genesis* bloku, inače se smatraju nevalidnim.

Slika . Primer nevalidnog magičnog broja



function validMagic(chain, block){

  if(block==1){ return true; }

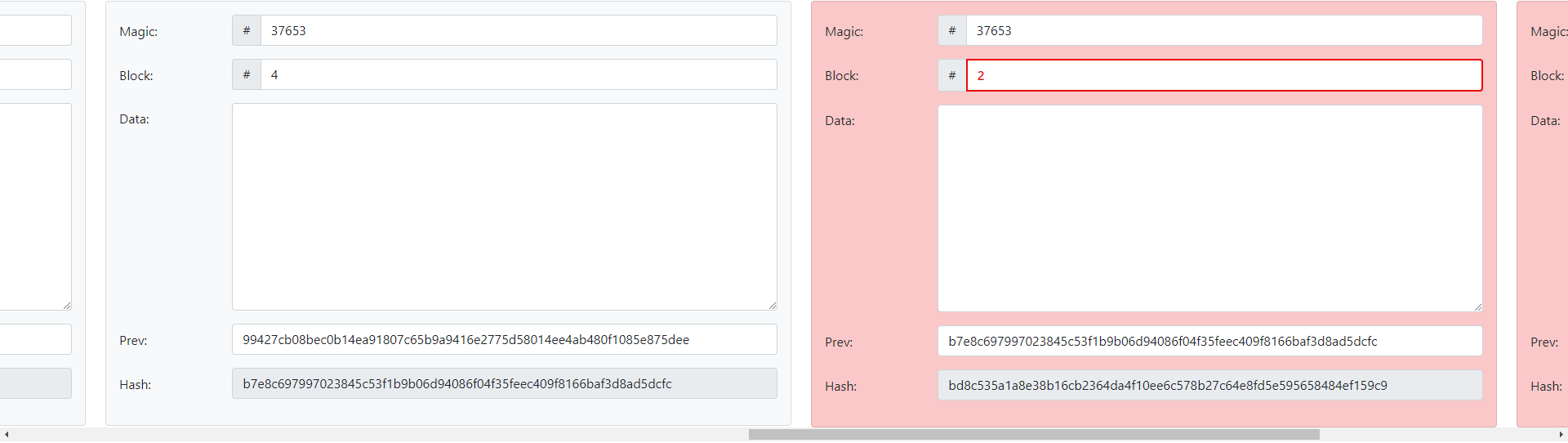
return $('#chain'+chain+'block'+block+'magic').val()==$('#chain'+chain+'block'+ (block-1).toString()+'magic').val();

}

Blok koda . Funkcija koja proverava validnost magičnog broja

2. Broj bloka – Svi blokovi osim prvog moraju da sadrže za jedan veću vrednost u *Block* polju od svog prethodnika. Ovo je pojednostavljena varijanta vremenskog žiga, gde blok koji je kasnije dodat ne može da se nađe pre ranije dodatog bloka u lancu.

Slika . Primer nevalidnog broja bloka



function validBlockId(chain, block){

if(block==1){ return true; }

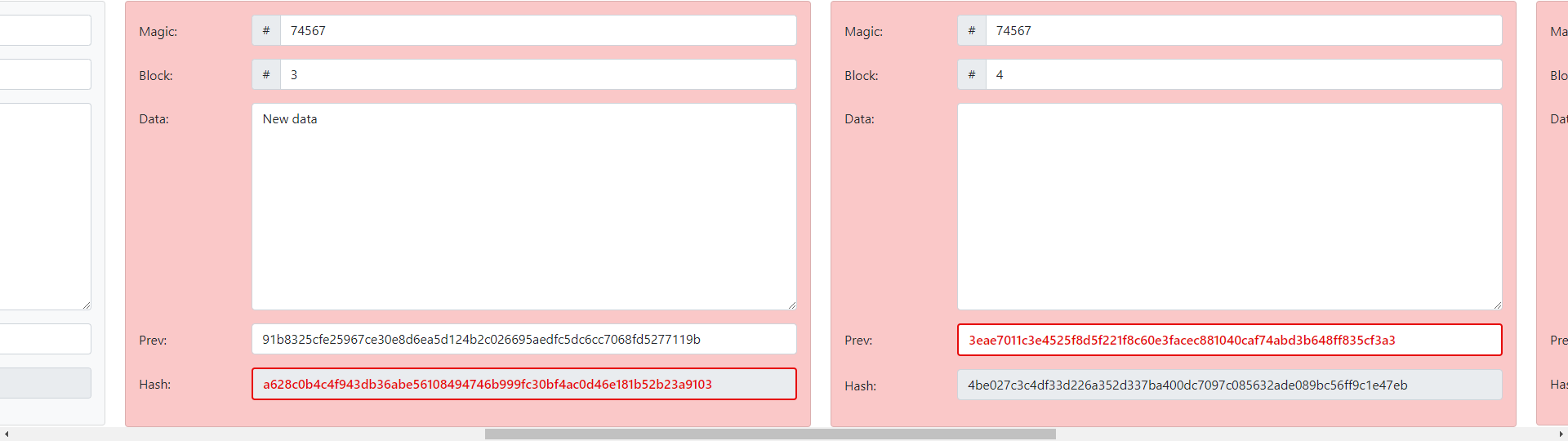
return parseInt($('#chain'+chain+'block'+block+'blockid').val())==parseInt ($('#chain'+chain+'block'+(block-1).toString()+'blockid').val())+1;

}

Blok koda . Funkcija koja proverava validnost broja bloka

3. Prethodni *Hash* – Svi blokovi osim prvog moraju da imaju vrednost *Prev* polja jednaku vrednosti *Hash* polja svog prethodnika. Ukoliko se te vrednosti ne poklapaju, a heš trenutnog bloka je validan onda je došlo do promene podataka u prethodnom bloku, tako da je on označen kao nevalidan.

Slika . Primer nevalidne prethodne heš vrednosti



function validPrev(chain, block){

  if(block==1){ return true; }

  return $('#chain'+chain+'block'+block+'prev').val()==$('#chain'+chain+'block'+ (block-1).toString()+'hash').val();

}

Blok koda . Funkcija koja proverava validnost heševa uzastopnih blokova

4. *Hash* – Svi blokovi moraju da imaju *hash* vrednost koja se dobija heširanjem svih drugih svojih konkateniranih polja (u ovom slučaju *Magic*+*Block*+*Data*+*Prev*) jednaku vrednosti u svom *Hash* polju. Radi jednostavnosti i brzine korišćenja aplikacije, autor je stavio da se pri promeni neke od ulaznih vrednosti unutar bloka, nova vrednost *Hash* polja za sada automatski izračunava, tako da je ovaj uslov validnosti trenutno uvek ispunjen.

Ove provere važe i za sve sledeće lance sa kojima ćemo se susresti, te tada nećemo prolaziti kroz njih, već samo kroz novouvedene validacije.

Na ovoj stranici je takođe uvedena vizuelna reprezentacija nevalidnih podataka i blokova unutar lanca (Slike X-X). Polje koje nije validno je označeno crvenom bojom, i svi blokovi do kraja lanca, uključujući i taj blok su takođe obojeni crvenom bojom. Ovo bi u stvarnom sistemu značilo da tačno znamo koji blok nije validan i zbog čega, tj. koji podaci su menjani unutar kog bloka.

Ova funkcionalnost je realizovana dodavanjem i oduzimanjem klasa određenim elementima na osnovu rezultata funkcija koje proveravaju validnosti polja. Slične funkcije su korišćene i na ostalim stranicama, te kroz njih nećemo prolaziti zbog dužine rada.

function updateBlockchain(chain, block, chainSize) {

  updateHash(chain, block);

  for(var i = 1; i<=chainSize; i++){

    if(validMagic(chain, i) && validBlockId(chain, i) && validPrev(chain, i)){  //no errors in the current block

      $('#chain'+chain+'block'+i+'magic').removeClass('input-error');

      $('#chain'+chain+'block'+i+'blockid').removeClass('input-error');

      $('#chain'+chain+'block'+i+'prev').removeClass('input-error');

      $('#chain'+chain+'block'+(i-1).toString()+'hash').removeClass('input-error');

      if($('#chain'+chain+'block'+(i-1).toString()+'card').hasClass('card-error')){ //previous card had an error

        $('#chain'+chain+'block'+i+'card').removeClass('bg-light').addClass('card-error');

      }

      else{ //previous card had NO errors

        $('#chain'+chain+'block'+i+'card').removeClass('card-error').addClass('bg-light');

      }

    }

    else{ //error(s) in the current block

      $('#chain'+chain+'block'+i+'card').removeClass('bg-light').addClass('card-error');

        if(!validPrev(chain, i)){

          $('#chain'+chain+'block'+(i-1).toString()+'card').removeClass('bg-light').addClass('card-error');

          $('#chain'+chain+'block'+(i-1).toString()+'hash').addClass('input-error');

          $('#chain'+chain+'block'+i+'prev').addClass('input-error');

        }

        else{ //previous and current hash match

          $('#chain'+chain+'block'+(i-1).toString()+'hash').removeClass('input-error');

          $('#chain'+chain+'block'+i+'prev').removeClass('input-error');

        }

        if(!validMagic(chain, i)){

          $('#chain'+chain+'block'+i+'magic').addClass('input-error');

        }

        else{ //magic number has NO error

          $('#chain'+chain+'block'+i+'magic').removeClass('input-error');

        }

        if(!validBlockId(chain, i)){

          $('#chain'+chain+'block'+i+'blockid').addClass('input-error');

        }

        else{ //block id has NO error

          $('#chain'+chain+'block'+i+'blockid').removeClass('input-error');

        }

    }

  }

}

Blok koda . Funkcija koja ažurira izgled i heš vrednosti blokova u lancu

.card-error{

  background-color: rgba(250, 200, 200)!important;

}

.input-error{

  color: rgb(230, 10, 10)!important;

  font-weight: 600!important;

  border: 2px solid rgb(230, 10, 10)!important;

}

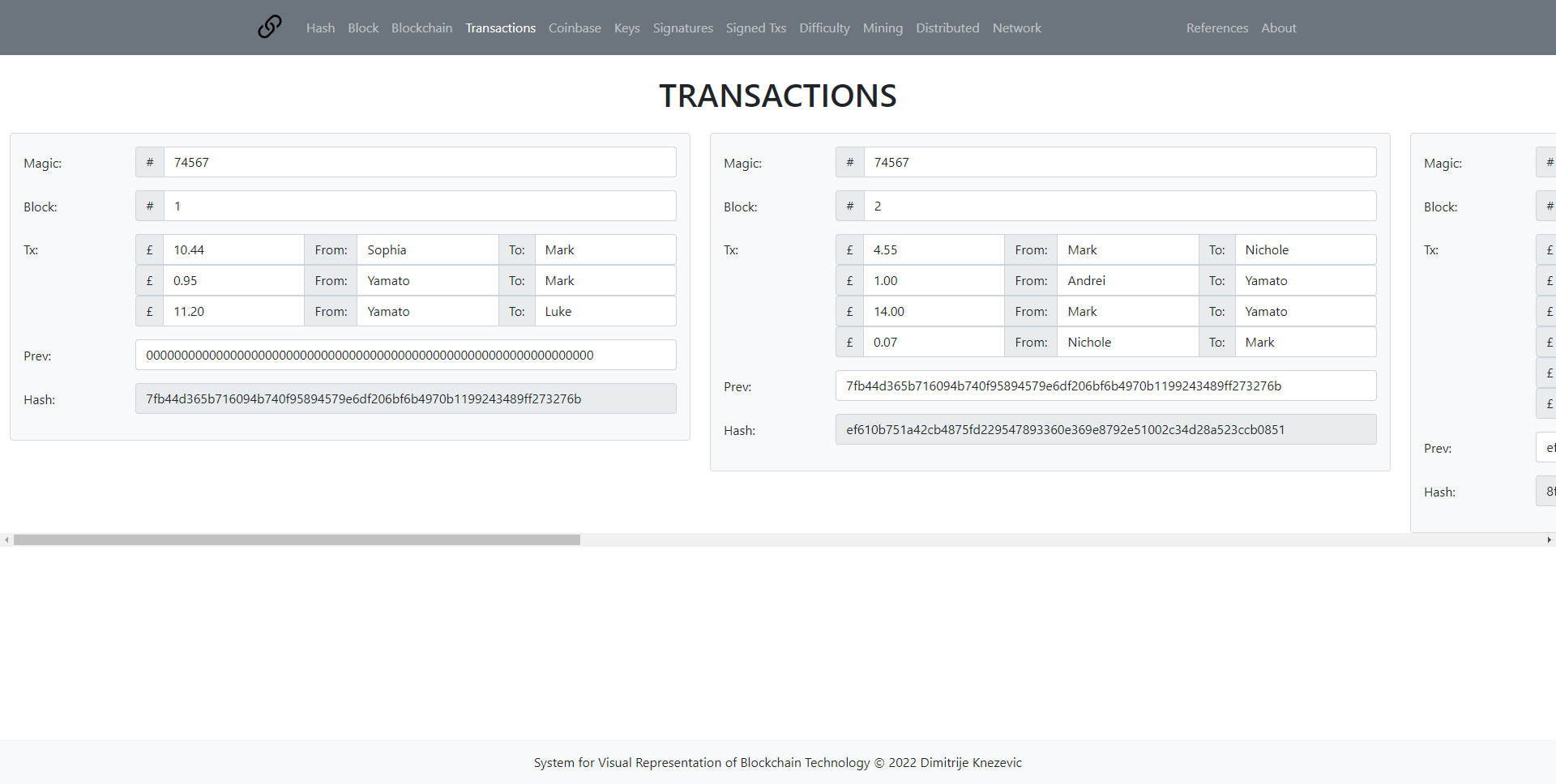
Blok koda . *CSS* stilovi koji se primenjuju na nevalidne elemente u lancu

Pravilo !important iz Bloka koda iznad se koristi da bi ovi stilovi nadjačali *Bootstrap* stilove koji se ne poklapaju sa njima. bg-light iz Bloka koda X je *Bootstrap* klasa.

Transactions.pug

Ova stranica uvodi koncept transakcija koje zamenjuju do sada uopšteno i apstrahovano polje *Data* u prethodnim primerima blokova. Transakcije su u autorovoj implementaciji sistema vezane za imaginarsku kriptovalutu i za sada će sadržati numeričku svotu kriptovalute, pošiljaoca i primaoca (u *plain text*-u).

Slika . Izgled Transactions stranice



Svaki blok je generisan na osnovu *Pug* fajla *includes/transactionblock.pug* koji umesto *Data* dela iz *includes/simpleblock.pug* sadrži sledeći kod:

div.mb-3.row

                label.col-sm-2.col-form-label(for='chain'+transactionblock.chain+ 'block'+transactionblock.block+'tx') Tx:

                div.col-sm-10

                    each tx, i in transactionblock.txs

                        div.input-group

                            span.input-group-text £

                            input.form-control(id='chain'+transactionblock.chain+ 'block'+transactionblock.block+'tx'+i+'value', type='text', value=tx.value)

                            span.input-group-text From:

                            input.form-control(id='chain'+transactionblock.chain+ 'block'+transactionblock.block+'tx'+i+'from', type='text', value=tx.from)

                            span.input-group-text To:

                            input.form-control(id='chain'+transactionblock.chain+ 'block'+transactionblock.block+'tx'+i+'to', type='text', value=tx.to)

Blok koda . Uvedena izmena u *transactionblock.pug* u odnosu na *simpleblock.pug*

Transakcije unutar bloka se prosleđuju iz roditeljske stranice detetu u vidu niza prilikom kreiranja bloka:

- blocks.push({

chain: 1,

block: 3,

magic: 74567,

blockid: 3,

txs: [

{value: '12.50', from: 'Andrei', to: 'Nichole'},

{value: '3.44', from: 'Yamato', to: 'George'},

{value: '9.03', from: 'Yamato', to: 'Mark'},

{value: '25.00', from: 'Luke', to: 'Andrei'},

{value: '25.00', from: 'Luke', to: 'Yamato'},

{value: '18.67', from: 'Andrei', to: 'George'}

],

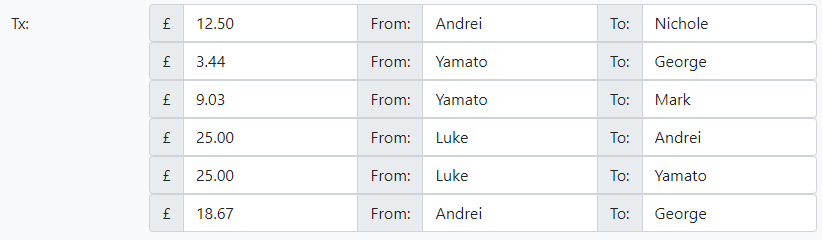
prev: 'ef610b751a42cb4875fd229547893360e369e8792e51002 c34d28a523ccb0851'

})

Blok koda . Primer prenosa transakcija između roditeljske stranice i stranice deteta

Uvećan primer izgleda dela bloka posvećenog transakcijama, gde se jasno može videti sadržaj polja generisanih u kodu iznad dat je na slici:

Slika . Primer transakcija u *plain text*-u



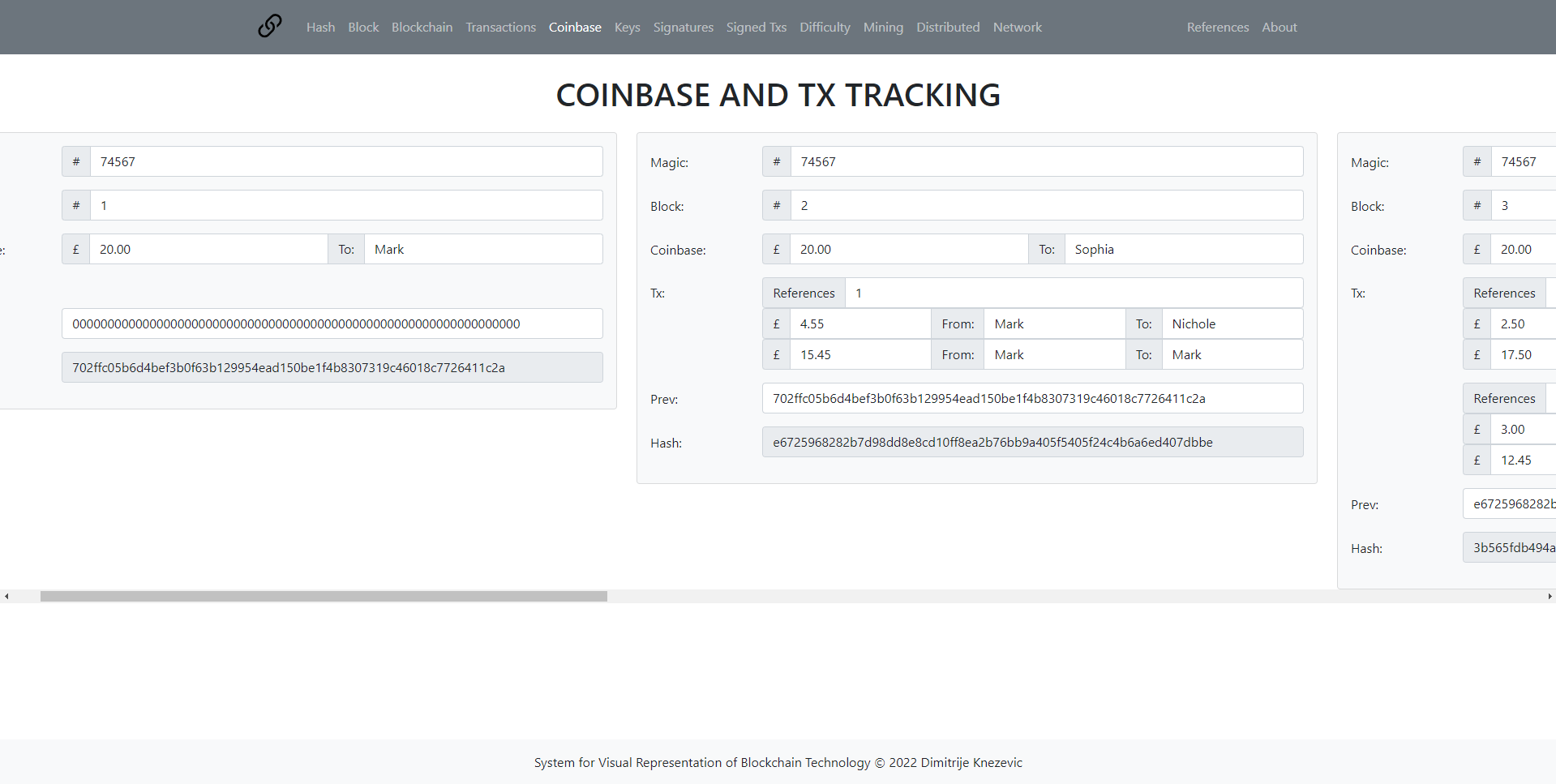
value polje koje predstavlja količinu kriptovalute je numeričnog tipa sa istim proveravama koje imaju i druga numerička polja (ne može biti prazno i ne može sadržati nenumeričke karaktere). from i to polja koja predstavljaju pošiljaoca i primaoca za sada nemaju dodatne provere.

Coinbase.pug

*Coinbase* stranica u sistem uvodi mehanizam za praćenje kreiranja i kretanja novca preko polja *Coinbase* i *References*. Sada je omogućeno da kriptovalutu šalju samo korisnici koji već iz prethodnih transakcija imaju dovoljnu količinu novca na njihovom računu. Radi efikasnosti algoritma provere, prilikom svake transakcije, pošiljalac mora sav preostali novac da pošalje sam sebi, time eliminišući potrebu da se svaki put ide nazad do početka lanca radi provere stanja na računu. Veoma sličan mehanizam koristi i *Bitcoin*.

*Coinbase* polje postoji u svakom bloku i sastoji se od količine kriptovalute koja se dodeljuje kao nagrada za kreiranje, tj. rudarenje bloka coinvalue (rudarenje će tek biti implementirano) i imena nagrađenog korisnika cointo. Posle svakog petog bloka, nagradna svota kriptovalute se prepolovi. Za ovo polje važe iste provere unosa kao i za druga slična polja koja smo već videli.

Slika . Izgled *Coinbase* stranice



function validCoinbase(chain, block){

  return parseFloat($('#chain'+chain+'block'+block+'coinvalue').val())== parseFloat(20.0/Math.pow(2,parseInt((block-1)/5)));

}

Blok koda . Funkcija koja proverava validnost *coinbase*-a

Slika . Primer nevalidnog *coinbase*-a



*References* polje je vezano za svaku transakciju. Transakcije više ne postoje u *genesis* bloku, jer u trenutku kreiranja bloka niko nije posedovao ni jedan novčić kriptovalute čiji blokčejn posmatramo. References sadrži niz celobrojnih vrednosti odvojenih jednim blanko znakom koje predstavljaju ID prethodnih blokova koji se koriste kao dokaz o postojanju određene količine kriptovalute na računu korisnika koji on pokušava da iskoristi u transakcijama ispod. Provere sada postaju malo složenije, a većina stvarnih sistema ima dodatnu strukturu podataka pored blokčejna koja sadrži ove informacije radi ubrzanja rada algoritma, međutim to u ovom kratkom primeru nije potrebno.

function validRef(chain, block, tx){

  if(block==1){

    return true;

  }

  var spentSum = parseFloat($('#chain'+chain+'block'+block+'tx'+tx+'value').val())+ parseFloat($('#chain'+chain+'block'+block+'tx'+tx+'returnedValue').val());

  var allowedSum = 0;

  const sender = $('#chain'+chain+'block'+block+'tx'+tx+'from').val();

  const refArray = $('#chain'+chain+'block'+block+'tx'+tx+'ref').val().split(' ');

  refArray.sort(function(a, b){return a - b});

  //array is empty || there are duplicates || referenced current block or greater

  if(refArray.length==0 || (new Set(refArray)).size!==refArray.length || refArray[refArray.length-1]>=block){

    return false;

  }

  var j = 1;

  for(let i = block-1; i>=refArray[0] || j<=refArray.length; i--){

    if(i==refArray[refArray.length-j]){ //if this block is referenced

      j++;

      let senderFound = false;

      if($('#chain'+chain+'block'+i+'cointo').val()==sender){ //add coinbase

        senderFound=true;

        allowedSum+=parseFloat($('#chain'+chain+'block'+i+'coinvalue').val());

      }

      for(let k = 0; $('#chain'+chain+'block'+i+'tx'+k+'value').length>0; k++){

//go through all tx in that block

        if($('#chain'+chain+'block'+i+'tx'+k+'to').val()==sender){

          allowedSum+=parseFloat($('#chain'+chain+'block'+i+'tx'+k+'value').val());

          senderFound = true;

        }

        if($('#chain'+chain+'block'+i+'tx'+k+'returnedTo').val()==sender){

          allowedSum+=parseFloat($('#chain'+chain+'block'+i+'tx'+k+'returnedValue').val());

          senderFound = true;

        }

      }

      if(!senderFound){

        return false;

      }

    }

    else{ //if this block isn't referenced

      for(let k = 0; $('#chain'+chain+'block'+i+'tx'+k+'value').length>0; k++){

//go through all tx in that block

        if($('#chain'+chain+'block'+i+'tx'+k+'from').val()==sender){

          let kRefArray = $('#chain'+chain+'block'+i+'tx'+k+'ref').val().split(' ');

          let tempArray = refArray;

          if(tempArray.filter(x=>kRefArray.includes(x)).length!=0){

//if arrays overlap

            return false;

          }

        }

      }

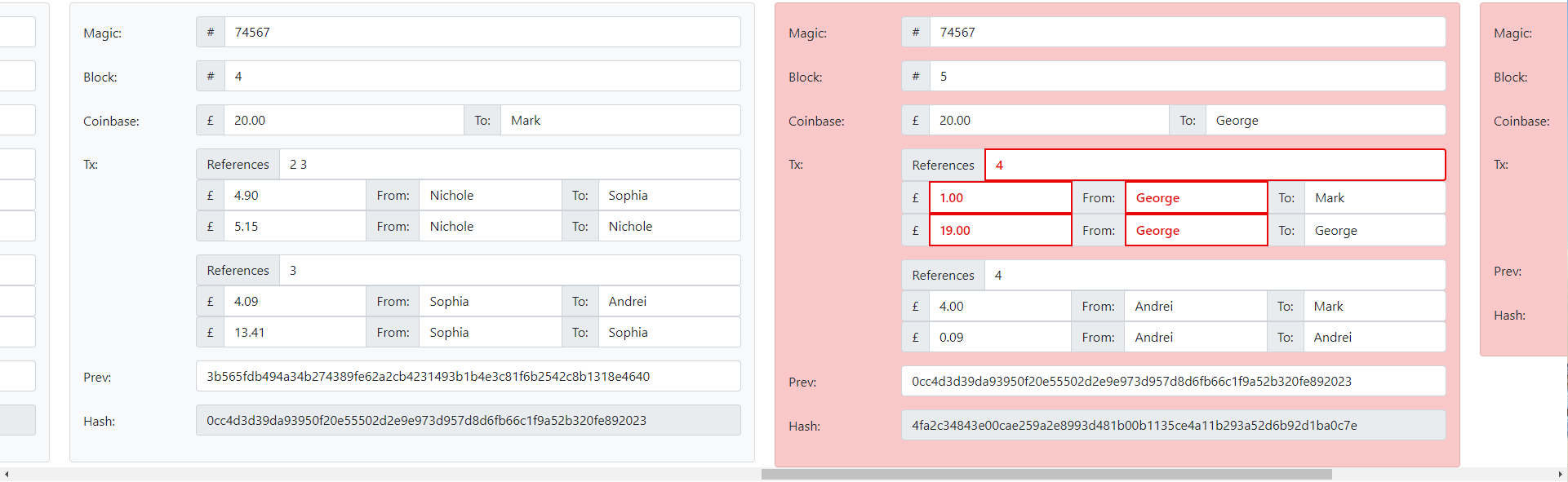
    }

  }

Blok koda . Funkcija koja proverava validnost *References* polja

Polje je validno ukoliko se zbir dve transakcije ispod (od kojih je jedna povraćaj novčića samome sebi) gde je osoba pošiljalac slaže sa zbirom transakcije iz blokova sa *ID*-jem iz *References* niza gde je ta ista osoba bila primalac. U svim drugim slučajevima polje je nevalidno, kao i u slučajevima gde je referenca već iskorišćena u nekom od prethodnih blokova. Takođe nije validno ni referencirati trenutni ili budući blok, referencirati neki prethodni blok više od jedanput, ili ne referencirati ni jedan blok.

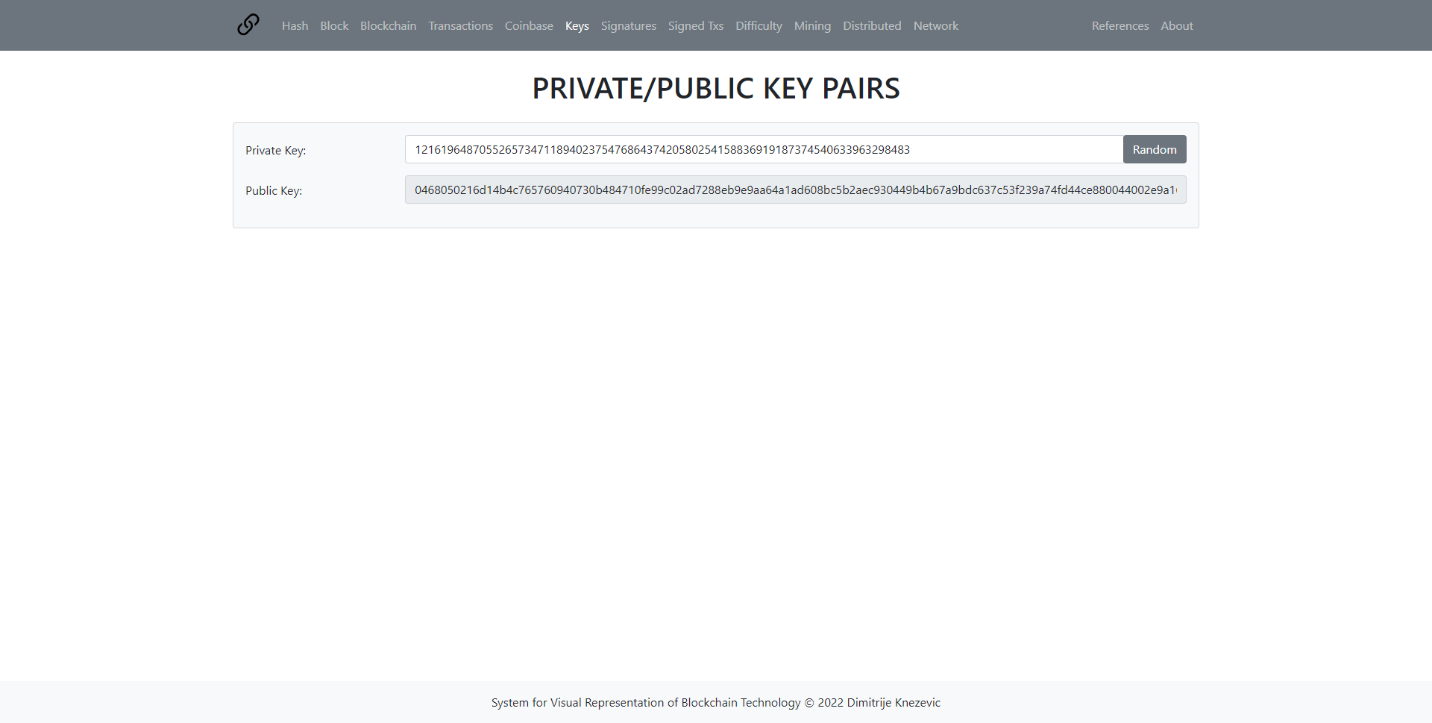
Slika 26. Primer nevalidne transakcione reference



Keys.pug

Stranica *Keys* uvodi veoma bitan pojam za kriptografiju, te stoga i blokčejn – privatne i javne ključeve.

Slika . Izgled stranice *Keys*



Interfejs stranice je dosta jednostavan, jedno polje predstavlja proizvoljan celobrojni unos privatnog ključa, a drugo polje predstavlja generisani heksadecimalni javni ključ na osnovu privatnog. Takođe postoji i dugme *Random* koje nam daje nasumičan celobrojni privatni ključ. Za implementaciju je korišćen javnodostupna *Javascript* biblioteka *elliptic.min.js* koja implementira datu funkcionalnost korišćenjem već pomenutog *ECDSA* algoritma.

Signatures.pug

Polje je validno

signed\_txs\_and\_ids.pug

Polje je validno

MiningDiff.pug

Polje je validno

Mining.pug

Polje je validno

Distributed.pug

Polje je validno

Network.pug

Polje je validno

References.pug

Polje je validno

About.pug

Polje je validno

Danas se biografija diplomiranog inženjera računarske tehnike i informatike, ne može zamisliti bez predmeta iz oblasti arhitekture i organizacije računara. Na Elektrotehničkom fakultetu u Beogradu postoji grupa predmeta, koja sistematski pokrivaju sve celine od značaja iz ove oblasti. U skladu sa savremenim načinima učenja, već nekoliko godina unazad, na Elektrotehničkom fakultetu u Beogradu, pored predavanja i vežbi na tabli, postoji i praktični deo predmeta, koji se obično izvodi ili kao samostalni projekat ili kao laboratorijske vežbe [1]. Kada govorimo o predmetima iz oblasti arhitekture i organizacije računara, to su obično laboratorijske vežbe na kojima studenti korišćenjem simulatora na praktičnim primerima mogu da provere koncepte, koje su učili na predavanjima. Iako je ovakvo izvođenje nastave u skladu sa IEEE preporukama [2], još jedan stav, koji se iznosi u ovim preporukama je da bi studentima trebalo omogućiti i da sami učestvuju u vežbama na kreativan način, a ne da samo pasivno prate tok vežbe.

Na Elektrotehničkom fakultetu u Beogradu razvijen je veliki broj simulatora sa kojima je u potpunosti pokriveno gradivo, koje se izlaže na predmetima iz oblasti arhitekture i organizacije računara. Sa druge strane, svi ovi simulatori su tzv. simulatori fiksnih računarskih sistema, koji studentima ne omogućavaju da upotrebe svoju kreativnost, već ih ostavlja da prate tok izvršavanja simulacije i da proveravaju naučene koncepte. Kako bi se ispoštovale najnovije preporuke IEEE, koje zahtevaju da student na aktivan način koristi simulatore i da mu se omogući da koristi kreativnost, autor je postavio cilj da razvije jedan simulator, koji bi omogućio pomenuto.

Da bi simulator bio u skladu sa pomenutim preporukama, potrebno je da se iskoristi neki od simulatora konfigurabilnih računarskih sistema, kao alat za realizaciju simulatora u kome bi studenti mogli da upotrebe svoju kreativnost. Zatim je potrebno izabrati pogodnu strukturu, dovoljno složenu, a istovremeno i sa pogodnim elementima na kojima bi studenti mogli da iskažu svoju kreativnost. Autor se odlučio za *pipeline* procesor. Kod *pipeline* procesora se istovremeno može izvršavati više različitih instrukcija, podelom instrukcija na faze u izvršavanju i preklapanjem različitih faza, tako da je sam osnovni model jednostavan. Iako je ovo jako korisno, ovakva organizacija procesora dovodi do situacija u kojima za neku instrukciju ne može da se izvrši faza predviđena stepenom *pipeline*-a i to se naziva hazard. Postoje strukturalni hazardi, hazardi podataka i upravljački hazardi. Za svaki od ovih problema postoje tehnike za rešavanje, koje omogućavaju različite realizacije rešenja i koje povećavaju složenost osnovnog modela, čineći pipeline procesor jednom od najsloženijih struktura. U savremenim procesorima se ovakva organizacija koristi za poboljšanje performansi računara.

Nakon izbora odgovarajuće strukture, autor je imao zadatak da izabere i odgovarajući alat u kome je najpogodnije realizovati simuator. Da bi došao do najpogodnijeg alata, autor je napravio pregled postojećih simulatora iz oblasti arhitekture i organizacije računara, sa posebnim osvrtom na postojeće simulatore *pipeline* procesora. Nakon toga izabran je alat, koji je u fazi razvoja na Elektrotehničkom fakultetu u Beogradu, pod nazivom *IgoVSoEDS* [3] [4] [5] [6]. Ovaj alat je već u upotrebi u nastavi na Elektrotehničkom fakultetu u Beogradu na nižim godinama, gde se koristi za pravljenje jednostavnijih struktura. Autor je dobio zadatak da nakon korišćenja ovog alata, da svoju procenu alata, da proveri koliko je pogodan za razvoj složenih struktura, kao i da iznese uočene prednosti i nedostatke.

U drugoj glavi daje se motivacija i predlog jednog rešenja. Najpre se daje motivacija za izradu rada, sa nacrtom planiranih ciljeva. Zatim se razmatra procesor sa *pipeline* organizacijom [7] i situacije koje čine *pipeline* procesor složenom strukturom, sa posebnim osvrtom na delove koji mogu biti lako izmenljivi, kao što je keš za predikciju skokova [8]. Daje se i pregled postojećih simulatora iz oblasti arhitekture i organizacije računara [9], kako bi se prikazale mogućnosti tih simulatora, sa posebnim osvrtom na postojeće simulatore *pipeline* procesora [10], kako bi se napravio pregled podržanih mogućnosti i kako bi se mogla dati procena realizovanog simulatora u skladu sa tim. Na kraju se daje predlog rešenja postavljenog problema.

U trećoj glavi je prikazana arhitektura i organizacija *pipeline* procesora, koji je realizovan. Najpre se razmatra arhitektura procesora, a zatim organizacija. Kompletna arhitektura i organizacija preuzeta je iz literature [11] uz saglasnost autora.

U četvrtoj glavi je prikazan alat koji je korišćen za izradu *pipeline* simulatora, kao i način korišćenja alata za projektovanje digitalnih struktura. Naziv alata iskorišćenog u realizaciji simulatora je *Interaktivni generator vizuelnih simulatora elektronskih digitalnih struktura* (*IGoVSoEDS*). Opis alata preuzet je iz literature [3] uz dozvolu autora.

U petoj glavi prikazan je način projektovanja procesora, uporedo sa prikazom projektovanja simulatora pomoću korišćenog alata. Ovakav način prikaza projektovanja procesora, daje mogućnost da se na konkretnom primeru projektovanja verifikuje korišćeni alat, kao i sam procesor. Šema je preuzeta iz literature [11] uz dozvolu autora i dopunjena u potrebnoj meri kako bi bilo moguće realizovati simulator u korišćenom alatu. Ovo podrazumeva da su sve komponente realizovane do nivoa prekidačkih mreža.

U šestoj glavi prikazuje se način korišćenja simulatora, odnosno funkcionalne mogućnosti simulatora. Najpre je prikazan način korišćenja alata za izvršavanje simulacija. Nakon toga opisan je rad simulatora kroz simulacije koje pokrivaju najveći deo specifičnosti *pipeline* procesora, tj. veliki broj različitih hazarda podataka i upravljačkih hazarda, kao i primene raznovrsnih tehnika za njihovo otklanjanje.

U sedmoj glavi daje se zaključak, kao kritički osvrt na ispunjenje ciljeva postavljenih na početku ovog rada, kao i rezime svega urađenog. Takođe se daje i procena korišćenog alata i iznose se uočene prednosti i nedostatci.

U ovoj glavi daje se motivacija i predlog jednog rešenja. Najpre se daje motivacija za izradu rada, sa nacrtom planiranih ciljeva. Zatim se razmatra procesor sa *pipeline* organizacijom i situacije koje čine *pipeline* procesor složenom strukturom, sa posebnim osvrtom na delove koji mogu biti lako izmenljivi, kao što je keš za predikciju skokova. Daje se i pregled postojećih simulatora iz oblasti arhitekture i organizacije računara, kako bi se prikazale mogućnosti tih simulatora, sa posebnim osvrtom na postojeće simulatore *pipeline* procesora, kako bi se napravio pregled podržanih mogućnosti i kako bi se mogla dati procena realizovanog simulatora u skladu sa tim. Na kraju se daje predlog rešenja postavljenog problema.

Motivacija i ciljevi rada

U nastavi iz oblasti arhitekture i organizacije računara na Elektrotehničkom fakultetu u Beogradu već dugi niz godina koriste se simulatori računarskih sistema, koji pomažu studentima da vide kako funkcionišu sistemi, koje uče na predavanjima. Razvijeni su simulatori, koji pokrivaju sve teme koje se obrađuju u nastavi i to su simulatori fiksnih računarskih sistema. Ovaj praktičan pristup izvođenju nastave je u skladu sa preporukama IEEE [2], u kojima osim pomenutog, stoji i preporuka da studenti ne bi trebalo da samo pasivno posmatraju šta se događa u simulatorima, već da bi im trebalo omogućiti da upotrebe svoju kreativnost. Kao odgovor na ovakav zahtev, mogu se upotrebiti simulatori konfigurabilnih računarskih sistema.

Da bi kreativnost studenata zaista mogla da dođe do izražaja, potrebno je napraviti neku od složenijih struktura iz oblasti arhitekture i organizacije računara, ali koja ima delova, koji se mogu realizovati na različite načine, a u samoj strukturi menjati jednostavno (*test bed*).

Jedna od veoma složenih struktura iz oblasti arhitekture i organizacije računara, koja odgovara prethodno pomenutom je procesor sa *pipeline* organizacijom. Ovakva organizacija procesora podrazumeva paralelno izvršavanje više instrukcija, pa samim tim i poboljšanje performansi računarskih sistema. U današnjim procesorima, ovakva organizacija se masovno koristi, kako bi se ubrzao rad procesora.

*Pipeline* organizacija podrazumeva da se izvršavanje operacije može razbiti na više logičkih celina, koje se nazivaju fazama (čitanje instrukcije, formiranje adresa operanada, čitanje operanada, izvršavanje operacije, itd…). Zatim za svaku od faza postoji posebna kombinaciona mreža i prihvatni registar, koji se koriste za izvršavanje te faze. Na taj način u zavisnosti od broja faza možemo imati veći broj instrukcija, koje se istovremeno izvršavaju u procesoru, tako što se nalaze u različitim fazama izvršavanja. Tipični *pipeline* ima pet faza izvršavanja: IF (*instruction fetch*), ID (*instruction decode and register fetch*), EX (*execute and effective address calculation*), MEM (*memory access and branch completion*) i WB (*write back*). Po redosledu izvršavanja faza instrukcije mogu se razlikovati statički i dinamički *pipeline*. Ako je redosled izvršavanja faza instrukcije uvek isti, takav *pipeline* naziva se statički (linearni). Ako se redosled stepeni kroz koje instrukcija prolazi može razlikovati od instrukcije do instrukcije, takav *pipeline* naziva se dinamički (nelinearni). U zavisnosti od toga kako se upravlja slanjem informacija iz stepena u stepen *pipeline*‑a, mogu se razlikovati asinhroni i sinhroni *pipeline*. U asinhronom *pipeline*‑u tok podataka između susednih stepeni kontroliše se *handshaking* protokolom. U sinhronom *pipeline*‑u upis u prihvatne registre svih stepeni realizuje se na isti signal takta. U nastavku rada biće razmatran sinhroni statički *pipeline* procesor. Ovakva organizacija procesora dovodi do situacija u kojima za neku instrukciju ne može da se izvrši faza predviđena redosledom izvršavanja, zbog međusavisnosti instrukcija, koje se javljaju uvođenjem ovakve organizacije, i to se naziva hazard. Postoje strukturalni hazardi, hazardi podataka i upravljački hazardi. Za svaki od ovih hazarda postoje tehnike za eliminisanje ili uklanjanje posledica, koje omogućavaju različite realizacije rešenja i koje u mnogome povećavaju složenost, čineći *pipeline* procesor jednom od najsloženijih struktura.

Nakon svega pomenutog, može se postaviti cilj ovog rada, a to je:

* napraviti pregled koncepata procesora *pipeline* organizacije, radi izbora odgovarajuće strukture za pravljenje simulatora,
* napraviti pregled postojećih simulatora računarskih sistema, sa posebnim osvrtom na simulatore *pipeline* procesora, radi izbora alata za rad i radi prikaza karakterističnih mogućnosti postojećih simulatora,
* realizovati simulator *pipeline* procesora u nekom od simulatora konfigurabilnih računarskih sistema, ali tako da se neki delovi mogu lako realizovati na različite načine (*test bed*) i
* dati analizu mogućnosti alata korišćenog za realizaciju.

# Projektovanje procesora i simulatora

U ovoj glavi prikazan je način projektovanja procesora, uporedo sa prikazom projektovanja simulatora pomoću korišćenog alata. Ovakav način prikaza projektovanja procesora, daje mogućnost da se na konkretnom primeru projektovanja verifikuje korišćeni alat, kao i sam procesor. Sve komponente su realizovane do nivoa prekidačkih mreža.

blok–šema procesora

U ovom odeljku data je blok‑šema procesora. Slika 27 prikazuje jedinice posmatranog procesora i najvažnije veze između njih. Šema se sastoji iz pet *pipeline* registara koji razdvajaju stepene procesora i u kojima se čuva sve što je potrebno za izvršavanje narednog stepena procesora i pet stepeni procesora, koji predstavljaju pet faza u izvršavanju instrukcije: IF, ID, EX, MEM, WB.

Diagram, schematic

Description automatically generated

Slika 27. Dopunjena blok‑šema procesora

stepen if pipeline procesora

Stepen IF (instruction fetch) *pipeline* procesora sastoji se od sledećih jedinica: New PC, PCache, Instruction Memory. *Pipeline* registar koji razdvaja stepene IF i ID označen je sa Rid. U sledećim sekcijama opisane su ove jedinice.

Jedinica NewPC

Programski brojač PC i jedinica NewPC, koja vrši izbor nove vrednosti programskog brojača, prikazani su na slici 28. 32‑bitni registar PC kao signal takta koristi signal CLK1 generisan u jedinici Stall. Jedinica NewPC sastoji se od multipleksera MPNEWPC i MPOPC, sabirača ADD, kodera prioriteta CD i pratećih logičkih kola. Multiplekser MPNEWPC je 32‑bitni multiplekser 8/1, pri čemu se koristi 5 od 8 ulaza. Izlaz ovog multipleksera se na sledeći signal takta upisuje u registar PC. Koder prioriteta CD generiše upravljačke signale za multiplekser MPNEWPC. To je koder 8/3, pri čemu se koristi 5 od 8 ulaza. Neiskorišćeni ulazi kodera CD fiksirani su na 0. Što je veća oznaka ulaza u koder to je viši prioritet tog ulaza. Multiplekser MPOPC je 32‑bitni multiplekser 2/1, koji propušta na ulaz sabirača ADD vrednost registra PC ili polja RMEM.PC, u zavisnosti od signala TNT. Sabirač ADD inkrementira izlaznu vrednost tog multipleksera za 1.

Ukoliko je signal PREKID aktivan a signal RMEM.TRAP neaktivan (pokreće se opsluživanje prekida koji nije izazvan instrukcijom **trap**), na izlaz multipleksera MPNEWPC propušta se adresa prekidne rutine i na sledeći signal takta upisuje u programski brojač. Ulaz 3 kodera CD biće aktivan ako se pokreće opsluživanje prekida izazvanog instrukcijom **trap** (aktivni su signali PREKID i RMEM.TRAP), ili ako se u stepenu MEM nalazi instrukcija bezuslovnog skoka (aktivno je polje RMEM.JUMP), ili instrukcija uslovnog skoka za koju je predviđanje bilo *not taken* a ishod je *taken* (NTT). (Prognoza ishoda skoka prikazana je vrednošću polja RMEM.CHIT, a stvarni ishod skoka vrednošću polja RMEM.COND.) Tada se na izlaz multipleksera MPNEWPC propušta vrednost RMEM.ALUOUT; to je odredišna adresa uslovnog ili bezuslovnog skoka ili instrukcije **trap**. Može se primetiti da mehanizam prekida zahteva vrlo malo dodatnog hardvera u ovoj jedinici, jer se pokretanje prekidne rutine sa stanovišta ove jedinice izvodi na isti način kao bezuslovni skokovi.

Ako je polje RMEM.CHIT aktivno a RMEM.COND neaktivno, to znači da je u stepenu MEM instrukcija uslovnog skoka za koju je predviđanje bilo *taken* a ishod je *not taken* (TNT). U registar PC treba upisati adresu instrukcije koja se nalazi u stepenu MEM (RMEM.PC) uvećanu za 1. Ako je aktivan signal CHIT, tj. ako u kešu za predikciju postoji ulaz za instrukciju određenu vrednošću registra PC, na izlaz multipleksera MPNEWPC propušta se vrednost pročitana iz keša za predikciju, PPC – prognozirana odredišna adresa skoka.

Za slučaj da nijedan od ulaza višeg prioriteta nije aktivan, što odgovara sekvencijalnom izvršavanju programa, ulaz 0 kodera CD fiksiran je na aktivnu vrednost. Tada je neaktivan signal TNT, multiplekser MPOPC propušta vrednost registra PC. Sabirač ADD inkrementira tu vrednost za 1, a multiplekser MPNEWPC propušta tu inkrementiranu vrednost.

Diagram, schematic

Description automatically generated

Slika 28. Registar *PC* i jedinica *NewPC*

Pravila na osnovu kojih je određen redosled ulaza u koder CD i multiplekser MPNEWPC su sledeća: 1) Koder CD mora davati prednost skokovima i prekidima u odnosu na sekvencijalno izvršavanje programa. U suprotnom bi, umesto da u njega bude upisana npr. odredišna adresa skoka, programski brojač samo bio inkrementiran. 2) Prognozirana odredišna adresa skoka (PPC) mora imati prednost u odnosu na sekvencijalno izvršavanje programa. 3) Instrukcija **trap** sa stanovišta jedinice NewPC ekvivalentna je instrukcijama bezuslovnog skoka. 4) Prekid, izvršena instrukcija skoka (bezuslovni ili uslovni sa pogrešnom predikcijom) i **trap** moraju imati viši prioritet od prognozirane odredišne adrese skoka (PPC) jer se u tim situacijama ispira *pipeline*. 5) Prekid koji nije izazvan instrukcijom **trap** mora imati viši prioritet od instrukcija skoka, jer se može javiti prilikom izvršavanja tih instrukcija. U suprotnom bi umesto adrese prekidne rutine u programski brojač bila upisana odredišna adresa skoka. 6) Međusobni prioritet instrukcija skoka i instrukcije **trap** nije bitan jer se ne mogu istovremeno naći dve takve instrukcije u stepenu MEM.

Jedinica PCache

Jedinica PCache (slika 29) je keš za predikciju odredišta i ishoda uslovnih skokova. Osim same keš memorije, ova jedinica sadrži 32‑bitni multiplekser 2/1 MPPCac čiji izlaz se vodi na adresni ulaz keša. Jedinica PCache koristi signale NTT i TNT generisane u jedinici NewPC.

Ako je predviđanje ishoda skoka bilo *taken* a ishod je *not taken*, potrebno je poništiti ulaz u keš memoriji. Zato se signal TNT koristi za invalidaciju ulaza na adresi RMEM.PC. Ako je predviđanje ishoda skoka bilo *not taken* a ishod je *taken*, potrebno je kreirati novi ulaz u keš memoriji. Signal NTT koristi se za upis u keš, a adresa ulaza u kešu takođe je RMEM.PC. U RAM deo keša upisuje se odredišna adresa skoka iz polja RMEM.ALUOUT. Ako nisu aktivni signali TNT ni NTT, proverava se da li u keš memoriji postoji ulaz koji odgovara vrednosti registra PC. Ako se ostvari pogodak u kešu, biće aktivan signal CHIT (*cache hit*), a prognozirano odredište skoka naći će se na izlazu RAM dela keša – PPC (*predicted PC*).

Diagram, schematic

Description automatically generated

Slika 29. Jedinica *PCache*

Na slici 30 prikazana je jedna od mogućih realizacija keš memorije. Prikazana realizacija predstavlja keš memoriju sa direktnim preslikavanjem, koja ima TAG, BADD i DATA deo. Svi delovi su realizovani uz pomoć modula MEM, koji je prikazan na slici 31.

Diagram, schematic

Description automatically generated

Slika 30. Jedinica *CACHE*

Diagram, schematic

Description automatically generated

Slika 31. Jedinica *MEM*

Jedinica Instruction Memory

Jedinica Instruction Memory (memorija za instrukcije) je analogna jedinici Data Memory. Kao adresa u memoriji za instrukcije se koristi vrednost registra PC. 32‑bitni izlazni podatak nosi oznaku IR; odgovarajući delovi tog podatka upisuju se u polja opcode, rd, rs1, rs2 i IMM *pipeline* registra RID. Signal za čitanje iz memorije za instrukcije je stalno aktivan. Upis u memoriju za instrukcije je izvan opsega ove knjige.

Pipeline registar RID

*Pipeline* registar RID sadrži 64 bita. Signal takta za taj registar je CLK1, a signal za brisanje , koje generiše jedinica Stall. Polja od kojih se sastoji *pipeline* registar RID opisana su u tabeli 4. IR je oznaka za izlazni podatak jedinice Instruction Memory, pročitan sa adrese određene vrednošću programskog brojača.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Polje | Broj bita | Biti | Izvor | Opis |
| VALID | 1 | 0 | Fiksno 1 | Sadržaj *pipeline* registra je validan |
| CHIT | 1 | 1 | PCache | Predikcija ishoda uslovnog skoka |
| PC | 32 | 2‑33 | Registar PC | Vrednost programskog brojača |
| rs1 | 5 | 34‑38 | IR17..13 | Adresa prvog izvorišnog registra |
| rs2 | 5 | 39‑43 | IR12..8 | Adresa drugog izvorišnog registra |
| IMM | 8 | 44‑51 | IR7..0 | Neposredni argument |
| opcode | 7 | 52‑58 | IR29..23 | Kod operacije |
| rd | 5 | 59‑63 | IR22..18 | Adresa odredišnog registra |

Tabela 4. Opis polja koja sačinjavaju *pipeline* registar *RID*

stepen id pipeline procesora

U procesorima klasične organizacije dekodovanje operacija najčešće obavlja jedna jedinica, koja na osnovu koda operacije generiše potrebne upravljačke signale. Međutim, kod *pipeline* procesora pri takvom načinu dekodovanja uz svaku instrukciju bi se u *pipeline* registrima morali voditi svi upravljački signali generisani pri dekodovanju operacije. Zato se u posmatranom procesoru dekodovanje vrši jednim delom u stepenu ID a drugim u stepenu EX.

Stepen ID (instruction decode and register fetch) *pipeline* procesora sastoji se od sledećih jedinica: DecodId, Registers, Extend. *Pipeline* registar koji razdvaja stepene ID i EX označen je sa Rex. U sledećim sekcijama opisane su ove jedinice.

Jedinica DecodID

Jedinica DecodID (slika 32) vrši dekodovanje operacija u stepenu ID. Pri tome generiše upravljačke signale potrebne u fazi ID i upravljačke signale potrebne na početku faze EX, takve da bi generisanje tih signala u fazi EX moglo zahtevati produženje periode signala takta. Pošto treba da generiše mali broj signala, jedinica DecodID realizovana je kao kombinaciona mreža koja prepoznaje odgovarajuće vrednosti kodova operacije.

Diagram, schematic

Description automatically generated

Slika 32. Jedinica *DecodID*

# Zaključak

U zaključku se daje kritički osvrt na ispunjenje ciljeva postavljenih u ovom radu. Analizira se šta je urađeno i daju se predlozi šta bi još moglo biti urađeno.

Napravljen je pregled koncepata procesora *pipeline* organizacije. Ova struktura je izabrana, kao često korišćena organizacija u savremenim procesorima, koja služi za poboljšanje njihovih performansi. Dat je kompletan opis svih koncepata *pipeline* organizacije, na koncizan i sažet način, pogodan za njihovo razumevanje. Objašnjeni su hazardi, koji ovu strukturu čine složenom, kao i tehnike za njihovo razrešavanja. Prikazane su i različite mogućnosti realizacije nekih elemenata, kao što je hardver za predikciju skoka.

Napravljen je pregled postojećih simulatora računarskih sistema, sa posebnim osvrtom na simulatore *pipeline* procesora. Pregled je kratak i bez mnnogo detalja, ali su pokriveni svi dominantni simulatori u upotrebi za svrhu nastave, a i za komercijalne upotrebe. Prikaz postojećih simulatora *pipeline* procesora, poslužio je kao odrednica za izbor odgovarajućeg alata za realizaciju simulatora.

Realizovan je simulator *pipeline* procesora u alatu *IGoVSoEDS*, ali tako da se neki delovi mogu lako realizovati na različite načine (*test bed*). Realizovani simulator *pipeline* procesora je potpuno verodostojan i ispravno funkcioniše. Takođe, ovaj simulator je realizovan najpre na blokovskom nivou, pa zatim sve do nivoa prekidačke mreže, s tim da alat *IGoVSoEDS* dozvoljava da se bilo koji od blokova realizuje na bilo koji drugi način, kao poseban modul i da se jednostavno zameni umesto postojećeg. Na taj način se postiže efekat *test bed*-a za dalja istraživanja studenata. Ono što je autor uočio kao manu, jeste da, iako složena struktura i iako modularno realizovan, *pipeline* procesor ima jedan blok (keš za predikciju skoka), koji ima smisla realizovati na druge načine. Za sve ostale blokove, nije veliki izazov isprobavati druge realizacije, jer su funkcionalnosti ovih blokova utvrđene postaljenom arhitekturom i organizacijom procesora. Naravno, kreativnim studentima bi ovakva situacija mogla biti podstrek, da umesto na nivou modula, izvrše promene i u organizaciji, isprobavajući tako različite varijante prikazanog modela *pipeline* procesora.

Na kraju, autor nakon upotrebe *IGoVSoEDS* alata, može dati njegovu procenu, kao i prednosti i nedostatke ovog alata. Opšti utisak autora je da je alat veoma dobar za realizaciju, kako jednostavnijih, tako i složenijih struktura. Takođe, alat je prilično jednostavan za upotrebu i ima dosta detaljno i dobro napisano uputstvo, koje pokriva sve mogućnosti ovog alata. Nedostaci alata, koje je uočio autor, mogu se podeliti u dve grupe: konceptualni i tehnički nedostatci. Pod konceptualnim nedostatcima podrazumevaju se funkcionalnosti, koje nedostaju, a koje bi bile korisne, dok se pod tehničkim nedostatcima podrazumevaju greške u izvršavanju programa i ograničenja, koja otežavaju korišćenje. Autor je uočio sledeće konceptualne nedostatke:

* ne postoji tabelarni pregled važnijih signala (trebalo bi obezbediti neki sličan vremenskom dijagramu signala), u koji bi se mogli dodati željeni signali i vrednosti registara, koje korisnik želi da prati,
* ne postoji podrška za dovođenje izlaza jednog kola na ulaz drugog, ukoliko je već izlaz drugog kola doveden na ulaz prvog (npr. kao kod flip-flopova) i nema rešenja za ovakvu situaciju,
* nema podrške za rotiranje elemenata na šemi, što je jako korisno kod ovakve vrste simulatora,
* nema podrške za pomeranje grupe elemenata.

Autor je uočio sledeće tehničke nedostatke:

* širina ulaza za podatke memorije je ograničena na maksimalno 16 bita, što može predstavljati otežavajuću okolnost u određenim situacijama,
* takođe, širina adresnog ulaza memorije je fiksirana na 16 bita i zbog ovakvih ograničenja, autor je morao da improvizuje memorije većeg kapaciteta, koristeći postojeće.
* na izlazu memorijskog modula se pojavljuje sadržaj sa adrese dovedene na adresni ulaz, na sledeći signal takta, što je dovelo do poteškoća u korektnom izvršavanju primera (morala se veštački dovoditi prava vrednost na izlaz),
* nema podrške za postavljanje Undefined signala na IN port modula, što je logično kod pravljenja složenijih struktura, ali isto tako i moguće rešiti zaobilaznim putem.

Pored ove dve grupe nedostataka, koje imaju veze sa samim simulatorom, postoji još jedan, koji je vezan za programsku realizaciju simulatora. Naime, kada se dogodi neka greška u projektovanju šeme ili izvršavanju simulacije, simulator ispiše poruku o grešci, koja je čisto tehničke prirode i nerazumljiva je običnom korisniku, ali još važnije, simulator se nakon toga samo ugasi, bez mogućnosti da korisnik sačuva svoj prethodni rad, što je jako loše, jer kod ovakvih vrsta programa, korisnik retko razmišlja o tome da često sačuva svoj rad, pa može doći do gubitka podataka i vraćanja korak u nazad u realizaciji nekog projekta.

Kada govorimo o dobrim stranama, to je već pomenuta jednostavnost u korišćenju, zatim dobar korisnički interfejs, sa dobrim rasporedom najčešće korišćenih komandi. Takođe, jako dobra strana je biblioteka sa osnovnim logičkim elementima, kao i mogućnost dodavanja novih modula u biblioteku. Zatim, u šemama je moguće prikazivati module sa ili bez portova, ali ipak povezivati "nevidljivim" vezama ovakve module, kao što se to obično radi u šemama logičkih struktura, kada je preveliki broj linija na šemi. Takođe, moguće je ograničiti pregled na nivou blokova, sa mogućnošću dozvole ili zabrane ulaženja u strukturu nekog bloka, pa se na taj način može napraviti simulator sa specifičnom namerom da demonstrira koncepte i/ili realizaciju koncepata. Prostorna uređenost je odlična, tako da jako složene šeme, mogu biti prikazane veoma jasno i pregledno. Simulacija se izvodi veoma jednostavno i po subjektivnom utisku autora precizno i korektno. Autor je uspeo da izvrši simulacije, koje su dale ispravne rezultate na procesoru, koji je realizovan u okviru ovog rada, kao što je i opisano ranije.

Krajnja ocena autora je da je korišćeni alat jako dobar i da uz ispravljanje nekih ili svih nedostataka može postati još bolji. Želja autora je da napravljeni simulator *pipeline* procesora, pomogne studentima da lakše shvate koncepte *pipeline* organizacije i da eksperimentišući sa ovim simulatorom dođu do novih ideja za poboljšanje performansi procesora.

# LITERATURA

|  |  |
| --- | --- |
| [1] | A. T. Sherman, F. Javani, H. Zhang and E. Golaszewski, "On the Origins and Variations of Blockchain Technologies," *IEEE Security & Privacy,* vol. 17, no. 1, pp. 72-77, 2019. |

1. D. Oberhaus, “The World’s Oldest Blockchain Has Been Hiding in the New York Times Since 1995”, Vice, Dostupno na: <https://www.vice.com/en/article/j5nzx4/what-was-the-first-blockchain> (Poslednji put pristupljeno decembra 2022.)

2. W. Stallings, “A Blockchain Tutorial”, The Internet Protocol Journal, Dostupno na: <http://ipj.dreamhosters.com/wp-content/uploads/2017/12/ipj20-3.pdf> (Poslednji put pristupljeno decembra 2022.)

3. Alan T. Sherman, Farid Javani, Haibin Zhang, Enis Golaszewski, “On the Origins and Variations of Blockchain Technologies”, Dostupno na: <https://ieeexplore.ieee.org/document/8674176> (Poslednji put pristupljeno decembra 2022.)

4. qwe

5. qwe

6. qwe