Provjerljivi izračuni opće namjene temeljeni na tehnologijama nultog znanja

Bibić, Luka

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:497072

Rights / Prava: Attribution 4.0 International/Imenovanje 4.0 međunarodna

Download date / Datum preuzimanja: **2024-12-19**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Engineering

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Provjerljivi izračuni opće namjene temeljeni na tehnologijama nultog znanja

Rijeka, studeni 2024.

Luka Bibić
0069082424

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Provjerljivi izračuni opće namjene temeljeni na tehnologijama nultog znanja

Mentor: prof.dr.sc. Kristijan Lenac

Rijeka, studeni 2024.

Luka Bibić
0069082424

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, studeni 2024.

_____

Luka Bibić

Zahvala

Ovim putem želim izraziti zahvalnost svima koji su doprinijeli izradi ovog diplomskog rada.

Prije svega, zahvaljujem svom mentoru na stručnom vodstvu, savjetima i podršci tijekom rada na zadanoj temi. Vaše smjernice i konstruktivni komentari bili su od velike pomoći u svim fazama istraživanja.

Zahvaljujem i RISC Zero razvojnom timu, koji su mi pružili podršku odgovarajući na moja pitanja te omogućili pristup testiranju unutar platforme Bonsai. Vaša stručnost i suradnja značajno su doprinijeli uspjehu ovog istraživanja.

Posebnu zahvalnost upućujem svojoj obitelji, osobito majci, koja mi je svojom podrškom omogućila da se u potpunosti posvetim studiranju. Vaša pomoć bila je neprocjenjiva.

Hvala svima na podršci i povjerenju.

Sadržaj

Popis slika

Popis tablica

Popis programskih kodova

Poglavlje 1

Uvod

1.1 Ciljevi i svrha istraživanja

Cilj ovog rada je upoznati se s tehnologijama nultog znanja (eng. Zero knowledge (ZK)), njihovim prednostima i manama, primjenama u praksi i na konkretnim primjerima zabilježiti uočene rezultate u kontekstu sigurnosti i performansi. Tehnologije nultog znanja omogućuju dokazivanje točnosti neke informacije, izračuna, izjave ili slično bez da se otkrije sama informacija ili način na koji je ostvarena.

Ovaj rad će se fokusirati na istraživanje postojećih rješenja za provjerljive izračune temeljene na tehnologijama nultog znanja. Bit će ispitani različiti provjerljivi izračuni s posebnim naglaskom na vremensku složenost samih izračuna. Također će biti analizirana i praktičnost implementacija ovih tehnologija u kontekstu delegiranja izračuna trećim stranama. Delegiranje izračuna je sve učestalija praksa zbog potrebe za optimizacijom resursa, smanjenja troškova, povećanja učinkovitosti sustava, te unapređenja skalabilnosti i fleksibilnosti poslovnih modela, ali i zbog sve veće važnosti sigurnosti podataka i zaštite privatnosti. Omogućuje organizacijama da koriste napredne tehnologije i specijalizirane resurse bez ugrožavanja povjerljivosti podataka, što je ključno u kontekstu sigurnosnih zahtjeva modernih računalnih sustava, osobito onih temeljenih na oblaku.

1.2 Struktura rada

Struktura ovog rada organizirana je kako bi čitatelju omogućila jasan i logičan slijed informacija, počevši od teorijskih osnova do praktičnih primjena i analize specifičnih slučajeva. Rad je podijeljen u sljedeća poglavlja:

- Teorijska osnova: Obuhvaća osnovne pojmove i tehnologije nultog znanja, uključujući ključne algoritme i protokole koji omogućuju njihovu primjenu.
- Pregled postojećih tehnologija: Analiza trenutno dostupnih rješenja i tehnologija za provjerljive izračune, uspoređujući njihove metode, performanse i praktične primjene.
- RISC Zero: Detaljna analiza tehnologije RISC Zero zkVM, uključujući njezinu arhitekturu, način rada, prednosti, nedostatke i praktične primjene.
- Rad u zkVM-u: Objašnjenje potpunog proces razvijanja aplikacija koje stvaraju dokaze nultog znanja, uključujući analizu često korištenih algoritama u kriptografiji te korištenje alata za tu analizu.
- Razni programi unutar RISC Zero zkVM: Uvid u različite programe pisane u zkVM-u, njihov pristup rješavanja i analiza koristeći načine prethodno opisane u radu.
- Zaključak: Završno poglavlje koje sažima ključne nalaze istraživanja, pruža preporuke za buduća istraživanja i zaključuje rad.

Svako poglavlje osmišljeno je tako da čitatelju pruži dubinsko razumijevanje pojedinih aspekata tehnologija nultog znanja i njihovih primjena u provjerljivim izračunima, s posebnim naglaskom na sigurnost, privatnost i učinkovitost.

Poglavlje 2

Teorijska osnova

2.1 Definicija i povijest

U modernom digitalnom dobu, sve veća količina podataka i složenost izračuna stvara potrebu za prijenosom računskih zadataka na moćnije računarske resurse. Međutim, prijenos tih zadataka nameće izazove vezane uz povjerenje i sigurnost. Kako bi se osiguralo da rezultat prenesenih zadataka bude ispravan bez potpunog povjerenja u vanjske resurse, razvijeni su sustavi za provjerljive izračune (eng. Verifiable Computation (VC)).

Ideja provjerljivog računarstva temelji se na tome da klijent može delegirati složene računske zadatke radniku (serveru), uz mogućnost učinkovite provjere ispravnosti dobivenih rezultata. Proces uključuje korištenje kriptografskih ključeva kako bi se osigurala sigurnost i integritet komunikacije između klijenta i radnika.

U tipičnom VC sustavu, klijent najprije generira ključ ili par ključeva koji se koriste u protokolu. U početku VC sheme nisu bile javno provjerljive. To znači da su zahtijevale točno određenog provjeritelja koji zna tajni VK_F ključ. U takvim sustavima, tajni ključ se koristio za pripremu ulaznih podataka i generiranje dodatnih informacija potrebnih za izračun. Radnik zatim izvršava zadani izračun nad dobivenim podacima i vraća rezultat zajedno s kriptografskim dokazom ispravnosti. Provjeritelj, koji posjeduje tajni ključ, koristi ga za provjeru dokaza i potvrđivanje ispravnosti rezultata.

Međutim, moderni VC sustavi evoluirali su kako bi omogućili javno provjeravanje. Klijent može generirati javni verifikacijski ključ koji omogućuje bilo kome da provjeri ispravnost rezultata bez potrebe za poznavanjem tajnog ključa. Tajni ključ i dalje se koristi za pripremu ulaznih podataka ili generiranje dodatnih informacija koje olakšavaju izračun, ali provjera se može obaviti pomoću javnog ključa. Navedeni način rada poboljšava fleksibilnost sustava i omogućuje širu primjenu u različitim scenarijima gdje je potrebno povjerenje među više sudionika.

Potreba za VC nastala je zbog sve većeg zahtjeva za sigurno delegiranje računalnih zadataka (eng. outsourcing). Klijent s ograničenim računalnim resursima može prenijeti zadatak evaluacije funkcije na moćniji resurs (radnik), ali je klijent i dalje u mogućnosti provjeriti ispravnost rezultata s manjom količinom rada nego što bi bilo potrebno za samostalnu evaluaciju funkcije.

Sustavi za VC moraju zadovoljavati ispravnost, sigurnost i efikasnost (eng. correctness, security and efficiency) sustava koristeći sljedeća pravila [1] (notacija i pojmovi su objašnjeni u tablici 2.1):

- Ispravnost: Za bilo koju funkciju F, i bilo koji ulaz u za F, ako se pokrene $(EK_F, VK_F) \leftarrow \text{KeyGen}(F, 1^\lambda)$ i $(y, \pi_y) \leftarrow \text{Compute}(EK_F, u)$, uvijek će se dobiti $1 = \text{Verify}(VK_F, u, y, \pi_y)$.
  - Ako su algoritmi ispravno pokrenuti, verifikacija uvijek mora potvrditi točnost rezultata koji je izračunat od strane radnika.
- Sigurnost: Za bilo koju funkciju F i bilo kojeg protivnika probabilističkog polinomskog vremena $\mathcal{A}$,

  $$\Pr[(\hat{u}, \hat{y}, \hat{\pi}_y) \leftarrow \mathcal{A}(EK_F, VK_F) : F(\hat{u}) \neq \hat{y} \text{ and } 1 = \text{Verify}(VK_F, \hat{u}, \hat{y}, \hat{\pi}_y)] \leq \text{negl}(\lambda).$$

- Efikasnost: KeyGen se smatra jednokratnom operacijom čiji se trošak amortizira kroz mnoge izračune, ali zahtjeva se da je Verify manje računalno zahtjevniji od evaluacije funkcije F.

Javni verificirani komputacijski sustav sastoji se od skupa od 3 algoritma polinomskog vremena [1]:

- $(EK_F, VK_F) \leftarrow \text{KeyGen}(F, 1^\lambda)$: Nasumični algoritam za generiranje ključeva uzima funkciju F koju treba odraditi i sigurnosti parametar λ. Vraća javni evaluacijski ključ EK_F i javni verifikacijski ključ VK_F.
- $(y, \pi_y) \leftarrow \text{Compute}(EK_F, u)$: Deterministički algoritam radnika koristi javni ključ za evaluaciju EK_F i ulaz u. Vraća $y = F(u)$ i dokaz π_y o ispravnosti y.
- $\{0, 1\} \leftarrow \text{Verify}(VK_F, u, y, \pi_y)$: S obzirom na verifikacijski ključ VK_F, deterministički verifikacijski algoritam vraća 1 ako je $F(u) = y$, u suprotnom vraća 0.

Tablica 2.1 Opis notacije, simbola i funkcija

Simbol	**Značenje**
F	Funkcija koja se evaluira; funkcija koju treba obraditi delegacijom.
u	Ulaz funkcije F.
y	Izlaz funkcije F za dani ulaz u, tj. $y = F(u)$.
λ	Sigurnosni parametar koji određuje razinu sigurnosti sustava.
(EK_F, VK_F)	Par ključeva generiranih pomoću $\text{KeyGen}(F, 1^\lambda)$; EK_F je javni evaluacijski ključ, a VK_F je javni verifikacijski ključ.
$\text{KeyGen}(F, 1^\lambda)$	Nasumični algoritam generacije ključeva za funkciju F uz sigurnosni parametar λ; izlaz su EK_F i VK_F.
(y, π_y)	Par izlaznog rezultata y i dokaza π_y generiranog pomoću $\text{Compute}(EK_F, u)$.
$\text{Compute}(EK_F, u)$	Deterministički algoritam koji koristi javni evaluacijski ključ EK_F za računanje izlaza y i dokaza π_y za ulaz u.
π_y	Dokaz koji potvrđuje ispravnost izračunatog izlaza y.
$\text{Verify}(VK_F, u, y, \pi_y)$	Deterministički algoritam koji koristi verifikacijski ključ VK_F za provjeru ispravnosti dokaza π_y za ulaz u i izlaz y; vraća 1 ako je dokaz ispravan, u suprotnom 0.
$\mathcal{A}$	Probabilistički protivnik polinomskog vremena koji pokušava prevariti sustav.
$\hat{u}, \hat{y}, \hat{\pi}_y$	Ulaz $\hat{u}$, izlaz $\hat{y}$ i dokaz $\hat{\pi}_y$ koje generira protivnik $\mathcal{A}$ u pokušaju da prevari verifikacijski algoritam.
$\leftarrow$	Strelica ulijevo označava dodjelu ili izlaz algoritma, npr. $(EK_F, VK_F) \leftarrow \text{KeyGen}(F, 1^\lambda)$ označava da su (EK_F, VK_F) izlaz algoritma KeyGen.
$\text{negl}(\lambda)$	Zanemariva vjerojatnost, funkcija koja brzo opada sa sigurnosnim parametrom λ.
$\Pr[\ldots]$	Vjerojatnost određenog događaja, npr. da protivnik uspješno prevari sustav.
$F(\hat{u}) \neq \hat{y}$	Označava da izračun funkcije F za ulaz $\hat{u}$ ne daje rezultat $\hat{y}$, što znači da protivnik nije točno izračunao $F(\hat{u})$.
$1 = \text{Verify}(VK_F, \hat{u}, \hat{y}, \hat{\pi}_y)$	Označava da verifikacijski algoritam prihvaća dokaz $\hat{\pi}_y$ iako je rezultat netočan, što znači da je protivnik uspio prevariti sustav.

Dokazi nultog znanja (eng. Zero knowledge Proofs (ZKP)) su kriptografske metode koje omogućuju jednoj strani, koju se često naziva dokazivačem (eng. prover) da dokaže drugoj strani, zvanom provjeritelj (eng. verifier) da je određena izjava točna, a da pri tome ne otkrije ikakvu informaciju o tome što se dokazuje. Takav pristup predstavlja ključnu značajku dokaza nultog znanja jer uvelike povećuje privatnost i sigurnost informacija u mnogim okolnostima, poput transakcija i pametnih ugovora (eng. smart contracts) [2]

Na visokoj razini (eng. high level), ZKP funkcioniraju na način da provjeritelj zatražuje od dokazivača da odradi niz ispita ili akcija, koje je moguće odraditi samo ako dokazivač stvarno zna informaciju koju želi dokazati bez da ju otkrije.

Glavne 3 karakteristike koje ZKP moraju zadovoljiti su:

- Potpunost (eng. Completeness): Ako je tvrdnja točna, tada pošteni provjeritelj može biti uvjeren od strane poštenog dokazivača da posjeduje znanje o ispravnom unosu.
- Ispravnost (eng. Soundness): Ako je tvrdnja netočna, tada nijedan nepošteni dokazivač ne može uvjeriti poštenog provjeritelja da posjeduje znanje o točnom unosu.
- Nultog znanja (eng. Zero-knowledge): Ako je tvrdnja točna, provjeritelj ne saznaje ništa više od toga da je tvrdnja točna.

2.2 Osnovni koncepti i algoritmi

Jednostavan primjer koncepta nultog znanja može biti sljedeći: dokazivač želi uvjeriti provjeritelja koji ima povez na očima i drži po jednu kuglu u svakoj ruci da su kugle različite boje, iako ih sam provjeritelj ne može vidjeti. Dokazivač nudi provjeritelju opciju da zamijeni kugle između ruku ili da ih zadrži na istom mjestu. Nakon njegove odluke, dokazivač će reći provjeritelju je li zamijenio redoslijed kugli ili ne, iako nije mogao vidjeti provjeriteljevu odluku dok ju je provjeritelj donosio.

Ako dokazivač ne govori istinu o izjavi da su "kugle različite boje", dokazivač neće znati razliku između kugli te će tijekom prvog pogađanja imati 50% šansu da slučajno

pogodi izbor provjeritelja. Naravno, jedan pogodak neće biti dovoljan da provjeritelj bude uvjeren u točnost izjave, te će ponoviti eksperiment ponovno. Jednostavno je izračunati da je vjerojatnost da će dokazivač svaki sljedeći put pogoditi odabir provjeritelja upola manja, odnosno vjerojatnost da će u n provjera dokazivač pogoditi svih n izbora je sljedeća:

$$P(\text{točni pogodci}) = p^n = \left(\frac{1}{2}\right)^n$$

Recimo da je provjeritelj ovaj eksperiment odradio 100 puta, vjerojatnost da je dokazivač pogodio svih 100 puta iznosi 7.8886091e-31 (približno 0%), i provjeritelj će s vjerojatnošću od 1 - 7.8886091e-31 (približno 100%) biti uvjeren da je izjava dokazivača točna.

Opisanom igrom dokazivač će uspješno uvjeriti provjeritelja o njegovoj izjavi da su kugle različite, ako je svaki put točno odredio izbor redoslijeda kugli provjeritelja, iako mu nije otkrio ikakvu informaciju o boji samih kugli.

Iako su ZKP svoju popularnost počeli stvarat tek u zadnjih nekoliko godina, naglim interesom u kripto tehnologije, koncept ZKP nije nov. Povijest ovog koncepta može se pratiti do kasnih 1980.-ih kada su Shafi Goldwasser, Silvio Micali i Charles Rackoff predstavili koncept u radu pod nazivom “The Knowledge Complexity of Interactive Proof-Systems”. Autori izvorni koncept opisuju kao "interaktivni protokol" u kojem dokazivač i provjeritelj međusobno komuniciraju dok dokazivač ne uspije uvjeriti provjeritelja u istinitost neke izjave [3].

2.3 Fiat-Shamir transformacija

Interaktivni protokoli, iako su bili značajan napredak, pokazali su se vremenski i resursno zahtjevnima, osobito kada se radi o velikim količinama podataka. Kako bi se dokazi nultog znanja mogli skalirati, neophodno je bilo da postanu neinteraktivni.

1986. godine Amos Fiat i Adi Shamir su izumili Fiat-Shamir transformaciju, tehniku koja pretvara interaktivne dokaze u digitalne potpise bazirane na njima. Ovaj izum uspješno pretvara interaktivne dokaze u neinteraktivne dokaze nultog znanja.

U praksi, većina dokaza nultog znanja obično slijedi proces u tri koraka:

- Dokazivač generira neku slučajnu vrijednost, zvanu obveza (eng. commitment), i šalje je provjeritelju.
- Provjeritelj zatim odgovara izazovnom vrijednošću (eng. challenge value) generiranom jednoliko nasumično.
- Na kraju, dokazivač izračunava završni dokaz koristeći obje vrijednosti - obvezu i izazov.

Kao što se može vidjeti, ova struktura je interaktivna, jer provjeritelj zahtijeva odgovor od dokazivača.

Ideja iza Fiat-Shamir transformacije je da umjesto da provjeritelj šalje slučajnu izazovnu vrijednost dokazivaču, dokazivač tu vrijednost može sam izračunati koristeći pseudo-nasumičnu funkciju, poput hash funkcije [4].

Idući kod prikazuje vrlo jednostavan primjer implementacije Fiat-Shamir transformacije za dokazivanje poznavanja lozinke.

```
import hashlib
import random
import libnum
from sympy import randprime

KORISTI_KRIVU_LOZINKU = False

# U stvarnim slučajevima koristiti veći broj, npr. 1024
duljina_u_bitovima = 16
donja_granica = 2**duljina_u_bitovima
gornja_granica = 2 ** (duljina_u_bitovima + 1)

# FAZA REGISTRACIJE
# Zajednički parametri
text = "Točna lozinka"

```

```
# Dokazivač i provjeritelj se dogovaraju o ovoj vrijednosti
# (npr. putem interneta), neki veliki prosti broj
n = randprime(
    donja_granica, gornja_granica
)
print("n = ", n)
print("broj bitova za n = ", n.bit_length())
g = 16  # Generator grupe
print("g = ", g)

# Tajni i javni ključ dokazivača
x = int(hashlib.md5(text.encode()).hexdigest()[:8], 16) % n
y = pow(g, x, n)
print("Javni ključ y = ", y)  # y odgovara h u opisu

# FAZA PRIJAVE (LOGIN)
# Dokazivačev commitment (obvezivanje)
if KORISTI_KRIVU_LOZINKU:
    text = "Kriva lozinka"
    x = int(hashlib.md5(text.encode()).hexdigest()[:8], 16) % n

# Dokazivač odabire nasumičnu vrijednost v
v = random.randint(1, 999999)
t = pow(g, v, n)
print("Obveza (Commitment) t = ", t)

# Dokazivač izračunava izazov (challenge) interno
c_ulaz = (str(g) + str(n) + str(y) + str(t)).encode("utf-8")
c = int(hashlib.sha256(c_ulaz).hexdigest(), 16) % n
print("Izračunati izazov c = ", c)

# Dokazivač izračunava odgovor (response)
```

```
49 r = (v - c * x) % (n - 1)
50 print("Odgovor r = ", r)
51
52 # Dokazivač šalje (t, r) provjeritelju
53
54 # FAZA VERIFIKACIJE
55 # Koraci provjeritelja
56 # Provjeritelj izračunava izazov c koristeći istu hash funkciju
57 c_provjeritelj_ulaz = (str(g) + str(n)
58                        + str(y) + str(t)).encode("utf-8")
59 c_provjeritelj = int(hashlib.sha256(c_provjeritelj_ulaz)
60                      .hexdigest(), 16) % n
61 print("provjeritelj izračunao izazov c = ", c_provjeritelj)
62
63 # provjeritelj provjerava verifikacijsku jednadžbu
64 lijeva_strana = t % n
65 desna_strana = (pow(g, r, n) * pow(y, c_provjeritelj, n)) % n
66 print("Provjeritelj izračunava lijevu stranu = ", lijeva_strana)
67 print("Provjeritelj izračunava desnu stranu = ", desna_strana)
68
69 if lijeva_strana == desna_strana:
70     print("Dokazivač je autentificiran.")
71 else:
72     print("Autentifikacija nije uspjela.")
```

Programski kod 2.1 Primjenjivanje Fiat-Shamir heuristike za dokazivanje znanja lozinke

- Dokazivač koristi lozinku spremljenu u varijablu `text` i primjenjuje hash funkciju kako bi generirao vrijednost `x`, što predstavlja tajni ključ (linija 28). Vrijednost `x` je integer koji se računa kao MD5 hash lozinke, te se uzima samo prvi dio hash vrijednosti (8 znakova). Vrijednost `x` se zatim koristi u modulo operaciji s `n`, koji predstavlja veliki prosti broj (linija 28). Vrijednost `y` izra-

čunava se kao `y = pow(g, x, n)` i predstavlja javni ključ koji dokazivač šalje provjeritelju (linija 29) dok je `g` takozvani "generator grupe" koji služi kao baza za potenciranje u pow metodi. Vrijednost `y` je javna i provjeritelj je koristi za daljnju verifikaciju, dok je `x` privatna vrijednost koju samo dokazivač poznaje i ona je računalno jako složena za kalkulirati ako nije poznata lozinka.

- Obveza: Dokazivač generira nasumičnu vrijednost `v` (linija 39) pomoću funkcije `random.randint(1, 999999)` kako bi stvorio tzv. commitment (obvezu). Obveza `t` se izračunava kao `t = pow(g, v, n)` (linija 40) i ta vrijednost se šalje provjeritelju. Vrijednost `v` je nasumična i koristi se samo za ovu sesiju prijave, čime se osigurava da svaka prijava bude jedinstvena.
- Izazovna vrijednost: Izazovna vrijednost `c` (linija 45) generira se pomoću hash funkcije SHA-256 unutar same prijave (bez interakcije s provjeriteljem), što je omogućeno korištenjem Fiat-Shamir transformacije. Izazov `c` izračunava se kao hash kombinacije vrijednosti `g`, `n`, `y` i `t`, čime se osigurava da izazovna vrijednost bude povezana s trenutnom sesijom.
- Završni dokaz: Dokazivač izračunava završni dokaz `r` kao `r = (v - c * x) % (n - 1)` (linija 49). Vrijednost `r` koristi se u kombinaciji s obvezom `t` i izazovom `c` kako bi se dokazalo da dokazivač poznaje lozinku bez njenog izravnog otkrivanja. Nakon što je dokazivač poslao vrijednosti `t` i `r` provjeritelju, provjeritelj može potvrditi valjanost dokaza.
- Verifikacija: Provjeritelj ponovo generira izazov `c_provjeritelj` koristeći istu hash funkciju (linija 59). Verifikacija se provodi usporedbom dvije strane verifikacijske jednadžbe: lijeve strane `t % n` (linija 64) i desne strane `(pow(g, r, n) * pow(y, c_provjeritelj, n)) % n` (linija 65). Ako su ove dvije vrijednosti jednake, provjeritelj može biti siguran da dokazivač zna tajnu vrijednost `x`, koja je povezana s ispravnom lozinkom.

Kod pruža mogućnost podešavanja varijable `KORISTI_KRIVU_LOZINKU` kako bi se simulirao pokušaj prijave od treće strane koja ne zna stvarnu lozinku, tako da stvori vrijednost `x` koristeći krivu lozinku. U tom slučaju program pokazuje da dokazivač nije autentificiran.

```
n =  122273
broj bitova za n =  17
g =  16
Javni ključ y =  102652
Obveza (Commitment) t =  117000
Izračunati izazov c =  16237
Odgovor r =  76875
provjeritelj izračunao izazov c =  16237
Provjeritelj izračunava lijevu stranu =  117000
Provjeritelj izračunava desnu stranu =  117000
Dokazivač je autentificiran.
```

(a) Točna lozinka

```
n =  121343
broj bitova za n =  17
g =  16
Javni ključ y =  1515
Obveza (Commitment) t =  71188
Izračunati izazov c =  88650
Odgovor r =  53247
provjeritelj izračunao izazov c =  88650
Provjeritelj izračunava lijevu stranu =  71188
Provjeritelj izračunava desnu stranu =  63595
Autentifikacija nije uspjela.
```

(b) Kriva lozinka

Slika 2.1 Dokazivanje lozinke koristeći Fiat-Shamir transformaciju

Prikazani koraci u kodu zajedno čine osnovu Fiat-Shamir transformacije. Dokazivač generira obvezu `t`, koristi izazov `c` za izračunavanje završnog dokaza `r`, a provjeritelj potvrđuje autentičnost bez izravnog otkrivanja lozinke. Kod omogućuje jednostavnu i sigurnu autentifikaciju bez potrebe za interakcijom između dokazivača i provjeritelja, čime se eliminira potreba za izravnim izazovom od strane provjeritelja.

Fiat-Shamir transformacija, kao ključna inovacija, ne samo da je smanjila potrebu za interakcijom između dokazivača i provjeritelja, već je otvorila mogućnosti za primjenu ZKP-a u decentraliziranim financijama i blockchain tehnologijama, čime se unapređuje sigurnost i efikasnost sustava.

2.3.1 Prevencija napada ponovnog slanja

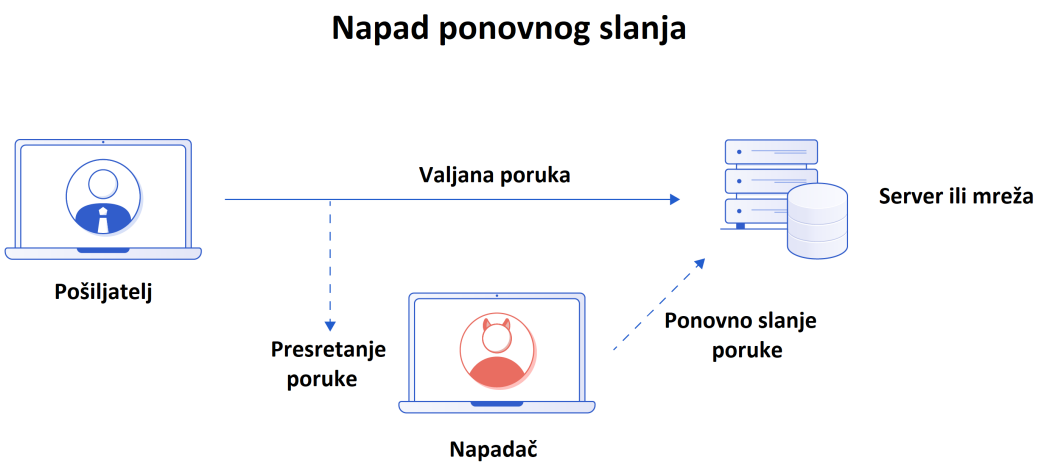
Baš kao i digitalni potpisi, dokazi nultog znanja mogu biti podložni napadima ponovnog slanja (eng. replay attacks). Ozbiljnost takvih napada ovisi o specifičnoj primjeni i kontekstu.

Napad ponovnog slanja (također poznat kao napad ponavljanja ili napad reprodukcije) predstavlja vrstu mrežnog napada u kojem se valjani prijenos podataka zlonamjerno ili obmanom ponavlja ili odgađa. Napadač presreće komunikaciju između dokazivača i provjeritelja, te ponovno šalje presretnute poruke s ciljem neovlaštenog pristupa ili lažnog predstavljanja.

U kontekstu dokaza nultog znanja, napadač može pokušati iskoristiti presretnute dokaze koje je dokazivač poslao provjeritelju. Na primjer, ako dokazivač pruža dokaz svog identiteta ili autorizacije za pristup određenom resursu, napadač može presresti taj dokaz i kasnije ga ponovno poslati provjeritelju. Cilj napadača je uvjeriti provjeritelja da je on zapravo legitimni dokazivač, čime bi stekao neovlašteni pristup resursima ili informacijama.

Jednostavniji način za opisivanje takvog napada je: „napad na sigurnosni protokol korištenjem ponovnog slanja poruka iz drugog konteksta u namjeravani (ili originalni i očekivani) kontekst, čime se varaju pošteni sudionici da misle kako su uspješno završili protokol."

Fiat-Shamir transformacija može se koristiti za prevenciju sličnih napada. U slučajevima kada primjena uključuje identitet vezan uz svaku stranu (npr. jedinstveni ID korisnika), preporučuje se uključiti ID dokazivača i provjeritelja unutar izračuna Fiat-Shamir hash funkcije. Provjeritelj bi prilikom provjere dokaza također trebao provjeriti odgovaraju li ID-ovi korišteni u hash funkciji njegovom vlastitom ID-u i ID-u dokazivača. Ova praksa sprječava zlonamjerne strane da ponovno iskoriste bilo koji dokaz koji nisu sami proizveli.



Slika 2.2 Prikaz replay napada, preuzeto iz [5]

2.4 Primjene tehnologija nultog znanja

2.4.1 Decentralizirani identitet

(ZKP) omogućuju provjeru identiteta osobe bez otkrivanja bilo kakvih osjetljivih privatnih informacija. Umjesto da korisnik dijeli detalje svog identiteta, on dokazuje da posjeduje određene vjerodajnice ili da zadovoljava određene uvjete. Primjer takvog pristupa je dokazivanje državljanstva neke države bez otkrivanja imena, prezimena ili osobnog identifikacijskog broja.

U DeFi (decentraliziranim financijama) aplikacijama, koje su financijske usluge izgrađene na blockchain tehnologiji bez potrebe za centraliziranim posrednicima poput banaka, često je nužno osigurati usklađenost s regulatornim zahtjevima i spriječiti prijevare. Kroz ZKP, korisnici mogu dokazati da dolaze iz dozvoljenih država ili da ispunjavaju druge potrebne kriterije, a da pritom ne dijele svoje osobne podatke.

Uloga trećih strana u ovom procesu je ključna za inicijalnu provjeru i izdavanje vjerodajnica. Treće strane, poput ovlaštenih institucija ili pružatelja identitetskih usluga, provjeravaju identitet korisnika na tradicionalan način. Nakon uspješne pro-

vjere, izdaju korisniku kriptografski potpisane vjerodajnice koje može koristiti u zero-knowledge dokazima. Kada korisnik želi pristupiti DeFi aplikaciji, prezentira ZKP koji dokazuje valjanost njegovih vjerodajnica bez otkrivanja ikakvih privatnih informacija.

Opisani pristup omogućava korisnicima da zaštite svoju privatnost od krađe i prevare, te smanjuje ovisnost o lozinkama koje su sklone napadima poput „phishinga“, pokušaja krađe podataka gdje napadači varaju korisnike kako bi otkrili osjetljive informacije poput lozinki ili financijskih podataka.

2.4.2 Transakcije koje štite privatnost

Dokazi nultog znanja omogućuju validaciju transakcija na blockchainu bez pristupa podacima o transakciji. Omogućavaju korisnicima da sakriju svoje transakcije na mreži koja štiti privatnost. ZKP-ovi mogu prikriti adrese pošiljatelja i primatelja, iznose transakcija i kodove pametnih ugovora, štiteći tako od neovlaštenog pristupa i javnog nadzora. ZKP-ovi su posebno korisni u represivnim režimima gdje se financijske transakcije strogo nadziru, a također pomaže u prevenciji krađe identiteta.

2.4.3 Sigurni i skalabilni rollup na sloju 2

Blockchain tehnologije često se sastoje od različitih slojeva koji služe specifičnim funkcijama. Sloj 1 (eng. layer 1) odnosi se na osnovnu mrežu blockchaina, poput Ethereum-a ili Bitcoin-a, koja obavlja ključne funkcije poput validacije transakcija i održavanja sigurnosti mreže. Često postaje zagušen zbog velikog broja transakcija, što može dovesti do visokih naknada i sporijeg procesiranja.

Sloj 2 (eng. layer 2) predstavlja dodatni sloj izgrađen na vrhu glavnog blockchaina (sloj 1) s ciljem poboljšanja skalabilnosti i brzine transakcija. Preuzima dio tereta iz glavne mreže procesiranjem transakcija izvan lanca te ih naknadno vraćaju na glavni blockchain. Na taj način, rješenja na sloju 2 rasterećuju osnovnu mrežu i omogućuju brže i jeftinije transakcije.

Rollup je tehnika koja se koristi na sloju 2 za poboljšanje skalabilnosti blockchain mreža. Rollup-ovi grupiraju veliki broj transakcija izvan glavne mreže i generiraju

jedinstveni dokaz o njihovoj ispravnosti, koji se potom objavljuje na glavnom blockchainu (sloj 1).

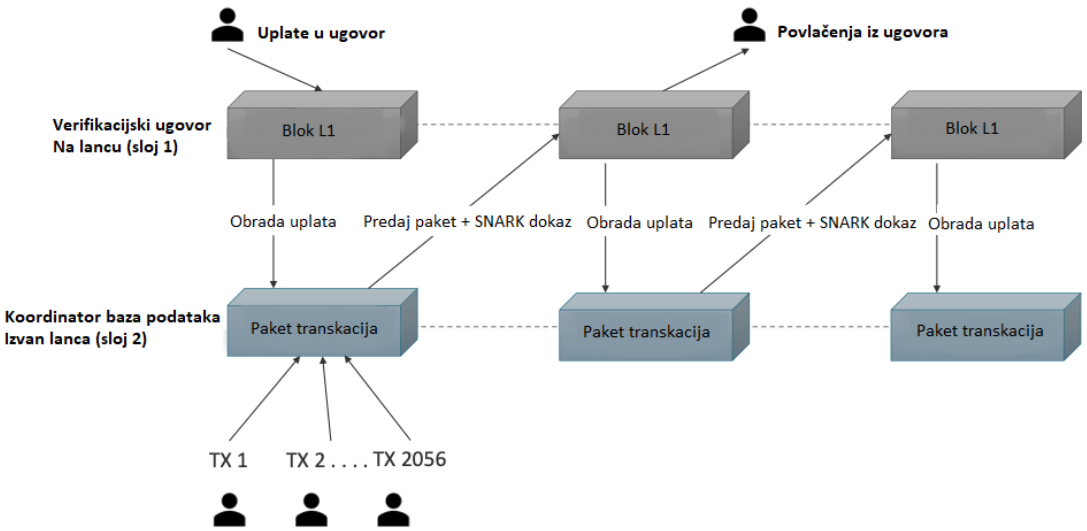
Postoje dvije glavne vrste rollup-ova: optimistični i rollup-ovi temeljeni na nultom znanju (zk-rollup). Optimistični rollup-ovi procesiraju transakcije izvan glavnog lanca i povremeno šalju sažetke tih transakcija natrag na glavnu mrežu. Naziv "optimistični" dolazi od pretpostavke da su sve transakcije ispravne, bez potrebe za trenutnim dokazom valjanosti. Ako se transakcija ospori, moguće je provjeriti njenu ispravnost kroz dokaze prijevare (eng. fraud proofs). Pristup koristeći dokaze prijevare omogućuje veće brzine transakcija i niže troškove, ali uključuje rizik od kašnjenja finalnosti transakcija zbog mogućnosti sporova.

Za razliku od optimističnih rollup-ova, zk-rollup-ovi pružaju trenutnu finalnost transakcija čim se objavi dokaz valjanosti temeljen na nultom znanju. ZK-rollup-ovi grupiraju tisuće transakcija i generiraju dokaz koji je mnogo manji od stvarnih podataka, smanjujući zagušenje mreže. ZK-rollup-ovi periodično pružaju valjane skupove transakcija obrađenih izvan lanca, koje se objavljuju kao sažetak promjena stanja. Glavna mreža potvrđuje promjene stanja putem matematički sigurnih dokaza, što korisnicima omogućuje brz pristup sredstvima, poboljšavajući efikasnost kapitala.

ZK-rollup-ovi se oslanjaju na glavnu mrežu za dostupnost podataka, namirenje i otpornost na cenzuru. Stanje rollupa održava pametni ugovor na mreži sloja 1. Kada korisnik pošalje transakciju, ona ide operatoru na sloju 2, koji može biti centralizirani „sequencer“ zadužen za izvršavanje, grupiranje i slanje transakcija na sloj 1. Alternativno, zk-rollup može koristiti proof-of-stake sustav gdje se izvršavanje i grupiranje transakcija rotira među validatorima koji su položili sredstva u staking ugovor.

Zbog navedenog, ova tehnologija omogućuje razvoj skalabilnih aplikacija koje podržavaju milijune korisnika uz niske troškove, dok održava visoku razinu sigurnosti karakterističnu za Web3. Razvojni programeri mogu graditi aplikacije koje su brze, sigurne i prilagođene širokoj upotrebi, bez žrtvovanja privatnosti i sigurnosti korisnika.

- Prednosti ZK-rollup:
  - Veća skalabilnost – Povećava ukupni kapacitet transakcija premještanjem



Slika 2.3 Vizualizacija ZK-rollup, preuzeto iz [6]

izvršavanja na sloj s većim kapacitetom.

- Sigurnost – Valjanost transakcija provjerava se na osnovnom sloju, zadržavajući sigurnost blockchaina.
- Smanjeni troškovi – Troškovi provjere dokaza su fiksni, smanjujući troškove po korisniku kako raste broj korisnika.
- Povlačenje sredstava – Korisnici mogu povući sredstva čak i ako rollup mreža ne radi ispravno.
- "1-of-n" model povjerenja – Dovoljno je da postoji samo jedan pošteni čvor za osiguranje valjanosti izvanlančanih izračuna.

- Mane ZK-rollup
  - Ograničenja osnovnog sloja – ZK-rollup-ovi ovise o osnovnom sloju za sigurnost i verifikaciju, što može povećati troškove i ograničiti performanse. Ako osnovni sloj, poput Ethereuma, ima visoke naknade ili ograničenu propusnost, to može negativno utjecati na učinkovitost zk-rollupa.
  - Fragmentacija likvidnosti – Kada postoji mnogo različitih rollupa ili L2

rješenja, sredstva mogu biti raspodijeljena među njima, što otežava trgovanje i prijenos sredstava. Nedostatak interoperabilnosti može zahtijevati složene procese za premještanje sredstava između različitih mreža, otežavajući korištenje aplikacija na različitim rollupima.

– Novi izazovi i rizici vezani uz nadogradnje – ZK-rollup-ovi su relativno nova tehnologija, pa nadogradnje mogu nositi rizike, poput sigurnosnih propusta ili nekompatibilnosti. Svaka nadogradnja može zahtijevati prilagodbu ili migraciju sredstava i aplikacija, što može biti rizično za korisnike ako nije pažljivo izvedeno.

2.4.4 Sustavi za glasanje

ZKP omogućuju stvaranje visoko sigurnih i provjerljivih glasačkih mehanizama. U ovim sustavima pojedinci mogu glasati bez otkrivanja svog identiteta ili za koga su glasali.

U tradicionalnim glasačkim sustavima, glasači moraju otkriti svoj identitet kako bi glasali, a administratori moraju čuvati te informacije povjerljivima. ZKP-bazirani glasački protokoli omogućuju pojedincima da dokažu valjanost svog dokumenta (npr. putovnice) bez otkrivanja identiteta.

ZKP također omogućuje glasačima da neovisno provjere je li njihov glas pravilno zabilježen u izbornom rezultatu bez otkrivanja svojih preferencija te neovisnim revizorima da provjere pravednost i točnost izbornog procesa.

Ovaj pristup značajno povećava sigurnost i povjerenje u glasačke sustave, omogućujući transparentne i privatne izbore. Navedeni su samo neki od primjera, ali postoje mnoge druge mogućnosti za primjenu ZKP-a. ZKP ima potencijal da značajno unaprijedi sigurnost i privatnost u različitim područjima.

Poglavlje 3

Pregled postojećih tehnologija

U ovom poglavlju bit će prikazan pregled postojećih tehnologija nultog znanja s posebnim naglaskom na sustave temeljene na tehnologijama Succinct Non-Interactive Arguments of Knowledge (SNARK) i Scalable Transparent Arguments of Knowledge (STARK). SNARK i STARK predstavljaju napredne oblike kriptografskih protokola koji omogućuju učinkovito i sigurno dokazivanje ispravnosti izračuna bez otkrivanja samih podataka. Rješenja temeljena na SNARK i STARK sve se češće primjenjuju u različitim područjima, od blockchain sustava do složenih izračuna u oblaku, te nude jedinstvene prednosti u smislu sigurnosti, skalabilnosti i transparentnosti. U nastavku se detaljno opisuju ključne karakteristike, prednosti i ograničenja ovih tehnologija, kao i njihova praktična primjena u stvarnim sustavima.

3.1 ZK-SNARK

Veliki napredak dogodio se 30.11.2011. kada su Bitansky, Canetti, Chiesa i Tromer objavili rad pod nazivom "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again" na Međunarodnom simpoziju o teoriji kriptografije [7].

Rad je dokazao mogućnost stvaranja sažetih neinteraktivnih argumenata znanja (eng. SNARK). Drugim riječima, stvoreni su ZKP koji su zbog male veličine skalabilni i brzo provjerljivi.

Sažeti neinteraktivni argument (eng. SNARG) je trostruki algoritam (P, GV, V). Za sigurnosni parametar k, provjeritelj pokreće $GV(1^k)$ kako bi generirao (vgrs, priv), gdje je vgrs (javni) niz referenci koje je generirao provjeritelj, a priv su odgovarajući privatni verifikacijski novčići; pošteni dokazivač $P(y, w, vgrs)$ proizvodi dokaz Π za tvrdnju $y = (M, x, t)$ s obzirom na svjedoka w; zatim $V(priv, y, \Pi)$ provjerava valjanost Π. SNARG je adaptivan ako dokazivač može odabrati tvrdnju nakon što vidi vgrs, inače je neadaptivan [7]. Notacija i pojmovi navedeni u idućim formulama su opisani tablici 3.1

Definicija: Trostruki algoritam (P, GV, V) je SNARG za relaciju $R \subseteq R_U$ ako su zadovoljeni sljedeći uvjeti [7]:

1. `Potpunost`: Za bilo koji par (y, w) iz relacije R,

$$\Pr\left[V(priv, y, \Pi) = 1 \mid (vgrs, priv) \leftarrow GV(1^k),\ \Pi \leftarrow P(y, w, vgrs)\right] = 1.$$

Uz to, $P(y, w, vgrs)$ radi u vremenu polinomske složenosti ovisno o k, |y|, i t.

2. `Sažetost`: Duljina dokaza Π koji $P(y, w, vgrs)$ proizvodi, kao i vrijeme izvršavanja $V(priv, y, \Pi)$, ograničeni su polinomom $p(k + |y|)$, gdje je p univerzalni polinom koji ne ovisi o R. Nadalje, $GV(1^k)$ radi u polinomskom vremenu ovisno o k; posebice, (vgrs, priv) su duljine polinomskog vremena ovisno o k.

3. `Ispravnost(Ovisno o pojmu adaptivnosti)`:
- *Neadaptivna ispravnost*: Za sve dokazivače polinomskog vremena P^*, dovoljno velik $k \in \mathbb{N}$, i $y \notin L_R$,

$$\Pr\left[V(priv, y, \Pi) = 1 \mid (vgrs, priv) \leftarrow GV(1^k),\ \Pi \leftarrow P^*(y, vgrs)\right] \leq \text{negl}(k).$$

- *Adaptivna ispravnost*: Za sve dokazivače polinomskog vremena P^* i dovoljno velik $k \in \mathbb{N}$,

$$\Pr\left[V(priv, y, \Pi) = 1 \mid (vgrs, priv) \leftarrow GV(1^k),\ (y, \Pi) \leftarrow P^*(vgrs),\ y \notin L_R\right] \leq \text{negl}(k).$$

SNARG znanja, ili skraćeno SNARK, je SNARG gdje je ispravnost pojačana na sljedeći način:

Definicija: Trostruki algoritam (P, GV, V) je SNARK ako je SNARG gdje je adaptivna ispravnost zamijenjena sa sljedećim jačim zahtjevom:

$$\Pr\left[\begin{array}{c}(vgrs, priv) \leftarrow GV(1^k),\ (y, \Pi) \leftarrow P^*(z, vgrs), \\ V(priv, y, \Pi) = 1,\ \text{a}\ (y, w) \leftarrow E_{P^*}(z, vgrs),\ w \notin R(y)\end{array}\right] \leq \text{negl}(k).$$

Tablica 3.1 Opis notacije, simbola i funkcija za SNARG i SNARK

Simbol	**Značenje**
P	Pošteni dokazivač koji generira dokaz Π za tvrdnju $y = (M, x, t)$ na temelju svjedoka w.
GV	Algoritam za generiranje javnog referentnog niza i privatnih verifikacijskih novčića.
V	Verifikacijski algoritam koji provjerava valjanost dokaza Π koristeći privatne novčiće i tvrdnju y.
k	Sigurnosni parametar koji određuje razinu sigurnosti sustava.
$(vgrs, priv)$	Par koji se sastoji od javnog referentnog niza $(vgrs)$ i privatnih verifikacijskih novčića $(priv)$ generiranih pomoću $GV(1^k)$.
$y = (M, x, t)$	Tvrdnja koju dokazivač pokušava dokazati; M označava stroj, x ulaz, a t vrijeme izvršavanja.
w	Svjedok za tvrdnju y, koji služi kao dokaz dokazivaču P.
Π	Dokaz generiran od strane dokazivača P za tvrdnju y uz svjedoka w.
$V(priv, y, \Pi)$	Verifikacijski algoritam koji provjerava valjanost dokaza Π za tvrdnju y koristeći privatne novčiće $priv$.
$R \subseteq RU$	Relacija koja sadrži parove (tvrdnja, svjedok) koji zadovoljavaju uvjete relacije.
LR	Jezik povezan s relacijom R; skup tvrdnji koje su prihvatljive prema relaciji.
$\text{negl}(k)$	Zanemariva vjerojatnost koja opada brže od bilo kojeg recipročnog polinoma u k.
$\leftarrow$	Strelica ulijevo označava dodjelu ili izlaz algoritma, npr. $(vgrs, priv) \leftarrow GV(1^k)$ označava da su $(vgrs, priv)$ izlaz algoritma GV.
$\Pr[\ldots]$	Vjerojatnost određenog događaja, npr. da verifikator prihvati dokaz Π za tvrdnju y.
P^*	Dokazivač koji može pokušati prevariti verifikatora, ne mora slijediti pravila poštenog dokazivača P.
$P^*(vgrs)$	Dokazivač koji može odabrati tvrdnju i generirati dokaz nakon što vidi javni referentni niz $vgrs$.

3.1.1 ZK-SNARK rješenja

Godine 2013. Pinocchio je bio jedan od prvih praktičnih dokaza koncepta SNARK-a. Pinocchio je inovativni sustav koji omogućuje učinkovitu verifikaciju općih izračuna koristeći napredne kriptografske metode. Klijent generira javni ključ za evaluaciju koji definira računanje, dok radnik koristi taj ključ za generiranje dokaza ispravnosti veličine samo 288 bajtova. Testiranje na više aplikacija pokazalo je da je Pinocchio izuzetno brz, s prosječnim vremenom verifikacije oko 10 milisekundi, što je znatno brže od prethodnih rješenja. Sustav također smanjuje napor radnika za 19-60 puta i podržava dokaze nultog znanja uz minimalne dodatne troškove. Pinocchio nudi alatni lanac za prevođenje podskupa C jezika u provjerljive programe.[1]

Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (ZK-SNARK) dokazi ovise o početnoj fazi povjerljivog postavljanja (eng. trusted setup) između dokazivača i provjeritelja, što znači da je potreban skup javnih parametara za izradu dokaza nultog znanja i privatnih transakcija. Ti parametri su poput pravila igre, kodirani su u protokolu i jedan su od nužnih faktora za dokazivanje valjanosti transakcije.

Generiranje tih parametara zahtijeva tajne informacije. Obično mala grupa ljudi generira te tajne (čime se stvara i problem centralizacije) i zatim ih koristi za stvaranje parametara. Nakon što su parametri stvoreni, tajne se uništavaju. Međutim, budući da tajne moraju generirati ljudi, moramo im "vjerovati". U blockchain okruženju nastoji se minimizirati povjerenje, zbog čega se povjerljiva postavljanja općenito izbjegavaju.

Godine 2016. predstavljen je Groth16, jedan od prvih protokola koji je učinio ZK-SNARK-ove učinkovitima i praktičnima za korištenje. Utjecaj Groth16 protokola na industriju bio je značajan te su ga mnogi odmah prihvatili. Veliki broj današnjih protokola temelje se na Groth16 zbog njegove jednostavnosti i performansi, a jedan od najpoznatijih primjera je ZCash koji ga je implementirao za svoje transakcijske protokole. Nedugo nakon toga, 2017. godine izašli su Bulletproofs, kratki neinteraktivni ZKP koji ne zahtijevaju povjerljivo postavljanje. Bulletproofs omogućuju dokazivanje da je enkriptirani tekst dobro strukturiran, npr. da se enkriptirani broj nalazi u određenom rasponu, bez otkrivanja bilo čega drugog o samom broju. Za raz-

liku od SNARK-ova, Bulletproofs ne zahtijevaju povjerljivo postavljanje, no njihova verifikacija traje duže nego kod SNARK-ova.

Bulletproofs su osmišljeni za učinkovite povjerljive transakcije u Bitcoin-u i drugim kriptovalutama, skrivajući preneseni iznos. Svaka povjerljiva transakcija sadrži kriptografski dokaz da je transakcija valjana. Smanjuju veličinu kriptografskog dokaza s preko 10 kB na manje od 1 kB, čime se značajno poboljšava učinkovitost. Osim toga, Bulletproofs podržavaju agregaciju dokaza, pri čemu se dokazivanje više transakcijskih vrijednosti dodaje samo O(log(m)) dodatnih elemenata na veličinu jednog dokaza, gdje `m` predstavlja broj transakcija za koje se stvara dokaz. Nažalost, iako se veličina kriptografskog dokaza Bulletproofsa povećava samo logaritamski, vrijeme verifikacije tih dokaza raste linearno, što je glavna mana Bulletproofs-a. Zbog toga su idealni za jednostavnije relacije. Ako bi sve Bitcoin transakcije bile povjerljive i koristile Bulletproofs, ukupna veličina skupa neiskorištenih izlaznih transakcija (UTXO) bila bi samo 17 GB, u usporedbi sa 160 GB koliko je trenutno s korištenim dokazima. [8]

Godina 2019. je bila osobito značajna za ZK-SNARK, u kojoj su se istaknula tri projekta: SONIC, MARLIN i PLONK. Sonic je sustav koji zahtijeva povjerljivo postavljanje, ali za razliku od konvencionalnih SNARK-ova, koristi strukturalni referentni niz (SRS) koji je ažurabilan i podržava sve aritmetičke krugove (eng. circuit) do određene veličine.

SRS se koristi u sustavima za dokazivanje tehnologijama nultog znanja za generiranje i verifikaciju kriptografskih dokaza, omogućujući sigurno i efikasno provođenje složenih matematičkih operacija. Osigurava fleksibilnost sustava, podržava različite aritmetičke krugove i održava sigurnost tijekom dokaznog procesa.

Aritmetički krugovi su matematičke strukture koje predstavljaju niz aritmetičkih operacija, poput zbrajanja, množenja i drugih logičkih operacija, te modeliraju izračune potrebne za verifikaciju specifičnih tvrdnji ili programa. Koriste se za definiranje složenih izračuna u kriptografiji i blockchain tehnologijama, gdje svaka operacija unutar aritmetičkog kruga predstavlja jedan korak u izračunu koji treba dokazati ili verificirati. Ovaj pristup rješava mnoge praktične izazove i rizike povezane s takvim postavkama. Referentni niz u Sonic-u raste linearno s veličinom podržanih aritmetičkih krugova, za razliku od kvadratnog rasta u nekim drugim sustavima, npr.

Groth16.

Verifikacija dokaza u Sonic-u uključuje konstantan broj uparivanja (eng. pairing), a svi elementi dokaza su u istoj izvornoj grupi, što omogućava efikasniju verifikaciju više dokaza istovremeno. Dodatno, Sonic može postići bolju efikasnost uz pomoć neslužbene „pomoćne“ strane koja agregira skup dokaza, ubrzavajući verifikaciju. U blockchain aplikacijama, ovaj pomoćnik može biti klijent poput rudara koji već obrađuje transakcije. Sonic pruža konkurentna verifikacijska vremena u usporedbi s najnovijim ZK-SNARK-ovima, s veličinama dokaza od 256 bajtova i verifikacijom od približno 0,7 ms za male instance. [9]

Marlin je protokol nultog znanja i nadogradnja na Sonic. Marlin koristi univerzalni i ažurabilni strukturalni referentni niz (SRS), smanjujući veličinu dokaza i kompleksnost vremena potrebnog za verifikaciju. Također, Marlin znatno smanjuje troškove generiranja dokaza, čineći proces stvaranja dokaza deset puta efikasnijim, a proces provjere dokaza 4 puta efikasnijim u usporedbi sa Sonicom. [10]

Ključne prednosti Marlin-a:

- Manja veličina dokaza: Marlin smanjuje veličinu dokaza smanjujući broj elemenata u grupi i polju.
- Brža verifikacija: Verifikacija dokaza u Marlin-u je tri puta brža nego u Sonic-u.
- Efikasnije generiranje dokaza: Generiranje dokaza u Marlin-u je više od deset puta brže, postižući vremenske efikasnosti koje su blizu najboljim ZK-SNARK-ovima specifičnim za određene aritmetičke krugove, poput onih koje je razvio Groth16.
- Univerzalnost i ažurabilnost: SRS u Marlin-u je univerzalan i može se ažurirati, što pojednostavljuje primjenu i olakšava prilagodbu različitim primjenama.

Marlin koristi metodologiju za pretprocesiranje aritmetičkih krugova u "offline" fazi, koristeći niskostepene polinome za enkodiranje kruga i dokaza, rezultirajući visokom efikasnošću i omogućavajući Marlin-u da se koristi za širok raspon primjena uz zadržavanje sigurnosti i efikasnosti. Implementiran je u Rust biblioteci, a evaluacija pokazuje konkurentne performanse u usporedbi s najnovijim ZK-SNARK-ovima, što čini Marlin praktičnim za stvarnu upotrebu.

PlonK značajno poboljšava performanse dokazivanja i provjere dokaza u odnosu na Sonic. Dokazivač (eng. prover) u PlonK-u mora izračunati pet polinomskih obveza i dvije otvarajuće provjere, što smanjuje broj potrebnih eksponentacija. Ovisno o strukturi aritmetičkog kruga, PlonK može biti do 30 puta učinkovitiji u generiranju dokaza u usporedbi sa Sonicom. Dodatno, PlonK koristi univerzalni SRS koji je manji i efikasniji za pohranu i prijenos, što omogućava bržu verifikaciju dokaza.

Jedna od glavnih inovacija PlonK-a je omogućavanje korištenja prilagođenih vrata (specijalizirane aritmetičke ili logičke operacije, npr. hash funkcija) osim uobičajenih operacija zbrajanja i množenja. Radi toga je postalo moguće graditi zero-knowledge dokaze za puno složenije programe.

Također, uvođenjem PlonK-a, kripto zajednica je shvatila da mogu graditi Zero Knowledge Ethereum Virtual Machine (zkEVM), što omogućuje pretvaranje bilo kojeg koda pametnog ugovora na Ethereum-u u dokaze nultog znanja.

3.1.2 Prednosti zk-SNARK-ova

- Visoka učinkovitost: ZK-SNARK-ovi omogućuju vrlo brzu verifikaciju izračuna, često u milisekundama, što je značajno brže od tradicionalnih metoda, čime se smanjuje vrijeme obrade.
- Mala veličina dokaza: Dokazi generirani ZK-SNARK-ovima izuzetno su kompaktni, obično samo nekoliko stotina bajtova, bez obzira na složenost izračuna.
- Neinteraktivnost: ZK-SNARK-ovi ne zahtijevaju višestruku razmjenu poruka između dokazivača i provjeritelja, što pojednostavljuje proces, smanjuje komunikacijske zahtjeve i ubrzava provjeru.
- Privatnost: Omogućuju dokazivanje ispravnosti izračuna bez otkrivanja podataka ili metoda izračuna, čime se čuva privatnost.
- Skalabilnost: Dobro funkcioniraju čak i za složene i velike izračune, omogućujući primjenu u raznim industrijama.

3.1.3 Mane zk-SNARK-ova

- Složenost postavljanja: Generiranje ključnih parova za ZK-SNARK-ove može biti složeno i računalno zahtjevno, zahtijevajući specijalizirane algoritme i puno vremena.
- Snažna kriptografska pretpostavka: Sigurnost ZK-SNARK-ova ovisi o snažnim kriptografskim pretpostavkama, koje su složene za razumijevanje i pravilnu primjenu.
- Potencijalna centralizacija: Proces povjerljivog postavljanja može zahtijevati povjerenje u centralizirane entitete, što može narušiti načela decentraliziranih sustava za koje se ZK-SNARK-ovi često koriste.
- Visoki početni troškovi: Implementacija i postavljanje ZK-SNARK-ova može biti skupo i zahtijeva visoko specijalizirano znanje, što može ograničiti njihovu dostupnost.
- Ograničena postkvantna sigurnost: ZK-SNARK-ovi nisu sigurni u postkvantnom okruženju jer se oslanjaju na kriptografiju s javnim ključem. Kriptografija s javnim ključem temelji se na težini rješavanja diskretnog logaritamskog problema u određenim skupinama brojeva. Međutim, s razvojem kvantnih računala, kriptografija s javnim ključem je ugrožena jer bi kvantna računala mogla lako faktorizirati velike brojeve na proste faktore, što bi značilo da rješavanje diskretnog logaritamskog problema više ne bi predstavljalo izazov.

3.2 ZK-STARK

Zero-Knowledge Scalable Transparent Argument of Knowledge (ZK-STARK)-ovi predstavljaju sljedeću revoluciju u razvoju dokaza nultog znanja. Predstavljeni su 2018. u radovima autora Eli Ben-Sasson, Iddo Bentov, Yinon Horesh i Michael Riabzev. Njihova misija bila je riješiti problem povjerljivih postavljanja na način da javne verifikacije mogu skalirati.

Za korištenje ZK sustava za velike podatke (eng. big data), ključno je da javni proces verifikacije raste sublinearnom brzinom u odnosu na veličinu podataka. ZK-

STARK je razvijen kao odgovor na ove probleme. Za razliku od ZK-SNARK-a, ZK-STARK-ovi ne zahtijevaju povjerljivo postavljanje, čime se eliminira potreba za povjerenjem u treće strane. Također, ZK-STARK-ovi omogućuju eksponencijalno bržu provjeru dokaza u odnosu na veličinu podataka, što ih čini posebno korisnima za velike baze podataka.

Primjer upotrebe ZK-STARK-a je dokazivanje da DNA profil predsjedničkog kandidata ne postoji u forenzičkoj DNA bazi podataka, a da se pritom ne otkriva nikakva dodatna informacija o bazi podataka ili kandidatu. Dokaz je manji od baze podataka i može se provjeriti brže nego ručno pregledavanje baze. [11]

ZK-STARK nudi transparentnost i brzinu bez potrebe za povjerenjem u početne postavke, čime poboljšava sigurnost i efikasnost ZK sustava u usporedbi s ZK-SNARK-om. Iako su ZK-STARK-ovi izašli relativno kasno, uspjeli su pridobiti veliku podršku i primjenu. Ethereum Foundation, neprofitna organizacija posvećena podršci Ethereum mreže i sličnim tehnologijama, dodijelila je nagradu od 12 milijuna dolara STARKware-u, tvrtki koja se fokusira na skaliranje rješenja koristeći ZK-STARK.

3.2.1 Prednosti zk-STARK-ova

- Sigurnost u postkvantnom okruženju: Za razliku od ZK-SNARK-ova, ZK-STARK-ovi su otporni na napade kvantnih računala.
- Bez potrebe za povjerljivim postavljanjem: Izostanak potrebe za povjerljivim postavljanjem znači da su ZK-STARK-ovi manje podložni određenim vrstama napada i zloupotreba.
- Transparentnost: Kod ZK-STARK-ova ne postoje tajni kriptografski parametri koji bi mogli biti iskorišteni.

3.2.2 Mane zk-STARK-ova

- Velike veličine dokaza: Dokazi generirani ZK-STARK-ovima značajno su veći od onih iz ZK-SNARK-ova, što može biti problematično za aplikacije s ograni-

čenim prostorom za pohranu ili propusnošću.

- Manje razvijena zajednica i dokumentacija: Budući da su ZK-STARK-ovi noviji, imaju manje resursa za razvoj, biblioteka i podrške zajednice u usporedbi sa ZK-SNARK-ovima.

- Veći računalni troškovi: Veće veličine dokaza znače da je potrebno više računanja za provjeru svakog dokaza, što može povećati troškove transakcija u određenim sustavima.

Poglavlje 4

RISC Zero

4.1 Ideja i ciljevi

U prethodnim poglavljima opisano je kako se dokazi nultog znanja mogu implementirati izravno putem zk-SNARK i zk-STARK protokola. Iako su ove metode učinkovite za specifične izračune, zahtijevaju prilagodbu aritmetičkih krugova za svaki program, što može biti složeno i dugotrajno.

Virtualne mašine nultog znanja (eng. Zero Knowledge Virtual Machine (zkVM)) pojavile su se kao rješenje ovih izazova. Djelujući kao virtualni CPU-ovi, zkVM-ovi omogućuju izvršavanje različitih programa unutar sigurnog okruženja nultog znanja. Umjesto da se svaki izračun ručno prevodi u aritmetičke krugove, zkVM automatski pretvara trag izvršavanja programa u format kompatibilan sa ZK-SNARK ili ZK-STARK protokolima. Time se pojednostavljuje integracija dokaza nultog znanja u različite aplikacije i smanjuje potreban napor programera.

Ključna prednost zkVM-ova je ponovna upotrebljivost i prenosivost dokaza. Omogućuju korištenje istog okruženja za dokazivanje različitih izračuna, bez potrebe za redizajniranjem sustava za svaki program. Posebno su korisni u aplikacijama koje trebaju provjeravati raznolike izračune. Nasuprot tome, izravna primjena ZK-SNARK i ZK-STARK protokola veže dokaze za specifične krugove, što zahtijeva novi dizajn za svaki novi izračun.

Iako su zkVM-ovi fleksibilni jer podržavaju različite programe s minimalnim pos-

tavkama, ta fleksibilnost dolazi uz kompromis u učinkovitosti. Primorani su prevesti svaki trag izvršavanja programa u ograničenja ili polinome, što uvodi dodatno opterećenje. Nasuprot tome, izravna primjena ZK-SNARK i ZK-STARK protokola omogućuje optimizaciju za specifične izračune, ali zahtijeva više vremena i napora za razvoj.

RISC Zero je kompanija započeta 2021. godine s ciljem rješavanja jednog od najvećih izazova u industriji i društvu – rastuće centralizacije aplikacija i infrastrukture. Njihova tehnologija za zkVM kombinira dokaze nultog znanja s RISC-V arhitekturom kako bi pružila sigurnost, privatnost i skalabilnost u različitim industrijama.

4.2 Povijest RISC-V arhitekture

Vlasnički skup instrukcija arhitekture (eng. Instruction set architecture (ISA)) strogo su kontrolirali određeni proizvođači, ograničavajući pristup njihovoj arhitekturi i namećući licencne naknade. Nedostatak otvorenosti kočio je inovacije, obeshrabrivao konkurenciju i otežavao razvoj prilagođenih procesora manjim tvrtkama i akademskim institucijama. To je dovelo do pojave RISC-V-a.

Prije pojave RISC-V-a, tržištem su dominirali vlasnički RISC (Reduced Instruction Set Computer) procesori poput MIPS-a, SPARC-a i PowerPC-a, koji su bili poznati po svojoj učinkovitosti, ali su imali ograničen pristup zbog visokih troškova licenci i zatvorenog dizajna.

RISC-V je razvijen na Sveučilištu Kalifornija, Berkeley, kao odgovor na te izazove. U 2010. godini, istraživački tim vođen Krsteom Asanovićem, Yunsupom Leejem i Andrewom Watermanom započeo je rad na otvorenoj arhitekturi skupa instrukcija koja bi bila dostupna svima. Njihov cilj bio je razviti fleksibilan i proširiv ISA koji bi podržavao raznovrsne primjene, od osnovnih mikrokontrolera do složenih procesora za podatkovne centre. Prva verzija RISC-V ISA, "RV32I", objavljena je 2011. godine i bila je usmjerena na jednostavnost i učinkovitost, postavljajući temelje za buduće iteracije i proširenja.

Godine 2015. osnovana je "RISC-V Foundation" s ciljem promicanja usvajanja i standardizacije RISC-V arhitekture. Fondacija je okupila vodeće industrijske i aka-

demske institucije te individualne doprinositelje kako bi zajednički radili na razvoju i širenju RISC-V tehnologije. Od tada, fondacija je narasla na preko 200 članova, a RISC-V je postao temelj za razne aplikacije, uključujući mikrokontrolere, ugrađene sustave i visokoučinkovite procesore.

Razvoj RISC-V-a potaknut je potrebom za većom prilagodljivošću u dizajnu procesora, smanjenjem ovisnosti o vlasničkim ISA-ima i rastućom potražnjom za energetski učinkovitim rješenjima. Otvorena i modularna priroda RISC-V omogućila je inovacije u dizajnu procesora i potencijalno preoblikovala industriju poluvodiča.

4.3 RISC-V arhitektura

Svaki procesor temelji se ISA, koja definira kako softver kontrolira hardver procesora. Dok specijalizirani čipovi za umjetnu inteligenciju koriste vlastite ISA-e, za opće računalstvo dizajnerski timovi obično licenciraju postojeće ISA-e od tvrtki poput Intela ili Arma. Međutim, sve više tvrtki istražuje treću opciju: RISC-V.

Dizajnerski timovi mogu pristupiti i implementirati RISC-V besplatno, izbjegavajući skupe licencne naknade koje bi inače plaćali Intel-u ili Arm-u. Tvrtke širom svijeta, uključujući američke i kineske tvrtke, koriste RISC-V za dizajn čipova različite složenosti, od jednostavnih mikrokontrolera do složenih sustava na čipu.

4.4 Usporedba RISC i CISC

Reduced Instruction Set Computer (RISC) i Complex Instruction Set Computer (CISC) predstavljaju dva različita pristupa dizajnu procesora. Razlike između RISC i CISC arhitektura proizlaze iz njihovih temeljnih ciljeva i načina postizanja učinkovitosti.

RISC procesori koriste manji skup jednostavnih i brzih instrukcija, omogućavaju jednostavniji dizajn hardvera i brže izvršavanje instrukcija. Ključne karakteristike RISC arhitekture uključuju:

- Jednostavne instrukcije: Svaka instrukcija obavlja jednu jednostavnu operaciju.

- Fiksna dužina instrukcija: Instrukcije su obično jednake dužine, što pojednostavljuje dekodiranje.
- Više registara: RISC procesori često imaju veći broj registara za smanjenje broja pristupa memoriji.
- Load/Store arhitektura: Pristup memoriji je ograničen na posebne instrukcije za učitavanje i spremanje podataka.

CISC procesori koriste veći skup složenih instrukcija koje mogu izvršiti višestruke operacije unutar jedne instrukcije. Radi toga, smanjuju broj instrukcija potrebnih za obavljanje zadatka, ali povećavaju složenost procesora. Ključne karakteristike CISC arhitekture uključuju:

- Složene instrukcije: Instrukcije mogu obaviti složene operacije poput više koraka u jednoj instrukciji.
- Varijabilna dužina instrukcija: Instrukcije mogu biti različite dužine, što može otežati dekodiranje.
- Manje registara: CISC procesori često imaju manji broj registara, oslanjajući se više na pristup memoriji.
- Instrukcije s izravnim pristupom memoriji: Instrukcije mogu izravno raditi s memorijom bez potrebe za posebnim instrukcijama za učitavanje i spremanje.

Sljedeća jednadžba se obično koristi za izražavanje performansi računala:

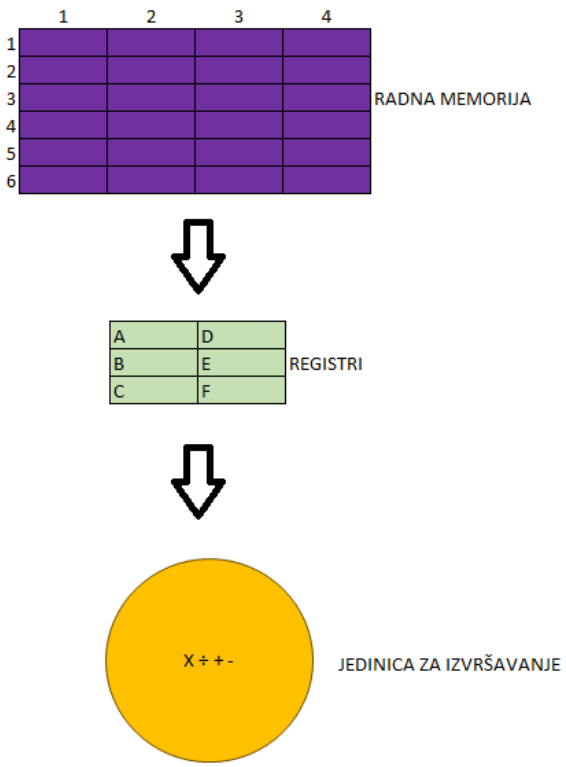
$$\frac{\text{vrijeme}}{\text{program}} = \frac{\text{vrijeme}}{\text{ciklus}} \times \frac{\text{ciklusi}}{\text{instrukcija}} \times \frac{\text{instrukcije}}{\text{program}} \tag{4.1}$$

CISC pristup nastoji minimizirati broj instrukcija po programu, žrtvujući broj ciklusa po instrukciji. RISC pristup radi suprotno, smanjujući broj ciklusa po instrukciji na račun broja instrukcija po programu.

Prednosti RISC-V arhitekture

Jednostavan način za analizirati prednost i mane RISC arhitekture je koristeći operacije u odnosu na njegovog prethodnika CISC.

Na slici je prikazana shema za pohranu podataka na generično računalo. Glavna memorija je podijeljena na lokacije označene brojevima. Prvi broj označuje redak u kojem se podatak nalazi, a drugi stupac. U ovom slučaju želimo pronaći umnožak broja na lokaciji 2:3 i broja na lokaciji 5:2.



Slika 4.1 Shema za pohranu podataka

RISC procesori koriste jednostavne instrukcije koje se izvode unutar jednog ciklusa takta. Za primjer množenja dva broja, RISC arhitektura bi koristila više jednostavnih instrukcija kako bi obavila zadatak. Proces bi izgledao ovako:

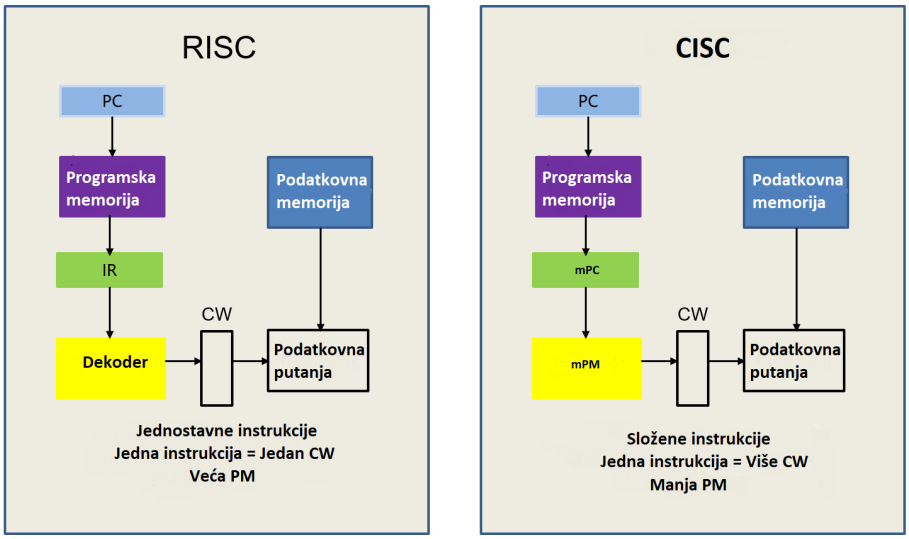
- `LOAD A, 2:3` - Učitaj vrijednost iz memorijske lokacije 2:3 u registar A.
- `LOAD B, 5:2` - Učitaj vrijednost iz memorijske lokacije 5:2 u registar B.

- `PROD A, B` - Pomnoži vrijednosti u registrima A i B.
- `STORE 2:3, A` - Pohrani rezultat iz registra A natrag u memorijsku lokaciju 2:3.

CISC procesori koriste složenije instrukcije koje mogu izvršiti višestruke operacije unutar jedne instrukcije. Za isti zadatak množenja dva broja, CISC procesor bi koristio jednu kompleksnu instrukciju:

- `MULT 2:3, 5:2` - Ova instrukcija učitava vrijednosti iz memorijskih lokacija 2:3 i 5:2, množi ih i pohranjuje rezultat natrag u memorijsku lokaciju 2:3.

Kao što se može vidjeti, CISC arhitektura omogućava da se zadatak obavi s manje linija koda, što može smanjiti potrebu za memorijom i olakšati rad prevoditelja (kompajlera). Međutim, složene instrukcije mogu zahtijevati više ciklusa takta za izvršavanje, što može smanjiti ukupnu brzinu procesora [12]. Na sljedećoj slici su prikazane RISC i CISC arhitekture.



(a) RISC arhitektura (a) (b) CISC arhitektura (b)

Slika 4.2 Usporedba RISC i CISC arhitekture

RISC arhitektura se sastoji od programskog brojača (PC), programske memorije, dekodera, registra instrukcija (IR) i podatkovne putanje, te koristi jednostavne ins-

trukcije koje dekoder obrađuje i šalje u podatkovnu putanju, dok CISC arhitektura uključuje mikroprogramski brojač (mPC), mikroprogramsku memoriju (mPM), podatkovnu memoriju i podatkovnu putanju, omogućujući složenije instrukcije kroz više upravljačkih riječi (eng. control words (CW)) uz pomoć mikroprogramske memorije.

4.5 Dizajn i implementacija RISC Zero zkVM-a

RISC Zero nudi open source virtualni stroj u kombinaciji sa sustavom dokaza bez otkrivanja znanja (zkVM). Kada se RISC-V binarna datoteka izvršava unutar zkVM-a, izlaz je uparen s računalnim potvrdom, što služi kao argument integriteta računalnog izračuna bez otkrivanja znanja. Sustav dokaza RISC Zero implementira ZK-STARK, baziran na HMAC-SHA-256, koji koristi FRI i DEEP-ALI protokole za funkcionalnost svojeg IOP protokola, objašnjene u poglavlju 4.7 [13].

4.6 Koncept i princip rada

Sustav argumenata koji stoji iza RISC Zero zkVM potvrda temelji se na izvršnoj traci (eng. execution trace) i nizu ograničenja koja osiguravaju provjere računalne ispravnosti. Ograničenja su pravila koja se primjenjuju na izvršnu traku kako bi se osigurala računalna ispravnost. Jedno od ograničenja može biti provjera jesu li su svi koraci obrade ispravno izvedeni i da nema grešaka u memorijskim operacijama.

Kada se komad koda izvršava na stroju, izvršna traka predstavlja zapis kompletnog stanja stroja u svakom taktu obrade. Drugim riječima, izvršna traka pruža detaljan prikaz kako se stanje stroja mijenja s vremenom tijekom izvršavanja koda. Obično se prikazuje kao pravokutna matrica. Svaki redak u ovoj matrici prikazuje kompletno stanje stroja u određenom trenutku, dok svaki stupac predstavlja vremenski zapis određenog aspekta obrade, poput vrijednosti pohranjene u određenom registru RISC-V procesora u svakom taktu. Korištenjem izvršne trake moguće je dobiti detaljan uvid u ponašanje programa na razini hardvera, što je korisno za dijagnostiku i optimizaciju performansi. Na primjer, ako se želi pratiti kako se vrijednost u određenom registru mijenja tijekom vremena, moguće je pogledati odgovarajući

stupac u izvršnoj traci. Izvršna traka je ključna za razumijevanje internih operacija procesora, omogućavajući inženjerima da analiziraju svaki korak izvršenja programa i identificiraju potencijalna uska grla ili greške.

U nastavku su objašnjene kategorije stupaca u RISC Zero izvršnoj traci:

- Kontrolni stupci - javni: Sadrže podatke koji opisuju RISC-V arhitekturu i razne kontrolne signale koji definiraju fazu izvršenja te tako određuju koja se ograničenja primjenjuju.
- Podatkovni stupci - privatni: Prikazuju stanje procesora i memorije tijekom izvođenja koda. Kako bi se učinkovito provjerila ispravnost memorijskih operacija u RISC-V arhitekturi, svaki registar povezan s memorijskim operacijama ima dva pridružena stupca: jedan u izvornom redoslijedu izvršenja i drugi sortiran prvo prema memorijskoj lokaciji, a zatim prema ciklusu takta.
- Akumulacijski stupci - privatni: Akumulacijski stupci koriste se za implementaciju "grand product" ograničenja za PLONK-provjeru permutacija i PLOOKUP-provjeru raspona. Grand product ograničenje je matematički izraz koji se koristi za provjeru permutacija elemenata unutar skupova podataka tijekom izračuna unutar dokaza nultog znjanja. Akumulacijski stupci sadrže podatke vezane uz akumulatore, koji akumuliraju vrijednosti potrebne za provjeru. Unosi u akumulacijskim stupcima i pridružena ograničenja stvaraju se tijekom faze nasumične predobrade.

4.6.1 Provedba računalne ispravnosti kroz ograničenja

RISC Zero svoju izjavu da je zadovoljena računalna ispravnost ostvaruje kroz izjavu da izvršna traka zadovoljava određen set ograničenja. Ova ograničenja osiguravaju da je izvršavanje zkVM-a u skladu s RISC-V instrukcijskim skupom (ISA). Svako ograničenje predstavlja polinom niskog stupnja koji djeluje na ograničene vrijednosti. Izvršna traka je valjana samo ako svako ograničenje daje rezultat 0. U nastavku su navedeni primjeri ograničenja:

- Primjer 1: Ograničenje $(k)(k-1) = 0$ osigurava da je k ili 0 ili 1.

- Primjer 2: Ograničenje $j - 2k = 0$ osigurava da je $j = 2k$.

Implementacija RISC-V instrukcijskog skupa unutar zkVM-a zahtijeva tisuće takvih ograničenja, pri čemu se svako ograničenje izražava kao polinom s više varijabli, čime se osigurava sveobuhvatna provjera računalne ispravnosti u svakoj od idućih faza rada zkVM-a: [13]:

- Init: Inicijalizacija svih registara na 0, osiguravajući da je sustav u poznatom stanju prije početka.
- Setup: Priprema za učitavanje ELF datoteke, koja uključuje postavljanje početnih uvjeta za učitavanje.
- Load: Učitavanje RISC-V binarnog koda u memoriju, osiguravajući da su svi potrebni podaci pravilno pohranjeni.
- Reset: Završava fazu učitavanja i priprema sustav za izvršenje, osiguravajući da su svi resursi spremni.
- Body: Glavna faza izvršenja gdje se izvršava korisnički definirani kod, provjeravajući svaki korak za ispravnost.
- RamFini: Generira vrijednosti akumulacije grand produkta temeljene na memoriji za permutaciju memorije, provjeravajući točnost memorijskih operacija.
- BytesFini: Generira vrijednosti akumulacije grand produkta temeljene na bajtovima za PLOOKUP, osiguravajući da su svi bajtovi ispravno obrađeni. PLOOKUP je posebno koristan u okruženju nultog znanja za grupne dokaze, kao što su rasponi provjera, gdje je potrebno provjeriti sve vrijednosti polinoma unutar određenog raspona. Ovaj protokol koristi tri pomoćna polinoma za provjeru, što smanjuje složenost i poboljšava učinkovitost provjere raspona [14].

4.7 RISC Zero IOP protokol

Pečat na RISC Zero zkVM potvrdi koristi STARK te omogućava dokazivaču (eng. prover) da opravda tvrdnju o posjedovanju specifičnog znanja, poznatog kao svjedok `w`, koji zadovoljava određene uvjete `x`. U kontekstu računalne ispravnosti, svjedok

predstavlja izvršnu traku, dok uvjeti definiraju pravila koja osiguravaju ispravnost obrade.

Protokol je zasnovan na Interactive Oracle Proof (IOP)-u, koji uključuje niz interakcija između dokazivača i provjeritelja. Dokazivač koristi slučajnosti koje dobiva od provjeritelja tijekom protokola. IOP je teorijski model, dok u praksi RISC Zero koristi neinteraktivnu verziju gdje je sudjelovanje provjeritelja zamijenjeno HMAC-SHA-256 algoritmom putem Fiat-Shamir transformacije. HMAC-SHA-256 je kriptografski algoritam koji kombinira hash funkciju SHA-256 i Hash-based Message Authentication Code (HMAC). Koristi se za osiguranje integriteta i autentičnosti poruka tijekom prijenosa i da su poslani od autentičnog izvora. Koristi tajni ključ zajedno s hash funkcijom. SHA-256 je dio SHA-2 (Secure Hash Algorithm 2) obitelji i generira 256-bitni hash, koji je jedinstven za bilo koji ulazni podatak.

U kontekstu IOP protokola, pečat služi kao dokaz znanja. U implementaciji, ovaj dokaz postaje argument, preciznije STARK. Fiat-Shamir Heuristika omogućava izvedbu sigurnosnih svojstava za STARK na temelju analize ispravnosti IOP protokola. Koristeći Fiat-Shamir, neinteraktivno je osigurano da je svaki korak obrade validiran bez potrebe za povjerenjem trećim stranama, omogućujući visoku razinu sigurnosti i integriteta u RISC Zero zkVM-u.

4.7.1 Faze protokola

U nastavku su objašnjene faze IOP protokola.

Transparentna faza postavljanja (Set-up phase)

Transparentna faza postavljanja se izvršava samo jednom za svaki program, a javni parametri koji su potrebni za generiranje dokaza izvršenja mogu se koristiti više puta. Sadržaj faze uključuje:

- Specifikaciju kruga i ograničenja za zkVM
- Kriptografski identifikator programa koji se izvršava
- Različite parametre koji određuju način rukovanja raspršivanjem (eng. ha-

shing), DEEP-ALI i FRI

Navedeni parametri mogu se distribuirati dokazivačima i provjeriteljima putem pouzdanog kanala ili izračunati neovisno na temelju definicije programa (npr. izvorni kod). Mogućnost da se parametri izračunaju neovisno, koristeći samo javno dostupne informacije, upravo definira svojstvo "transparentnosti" same faze.

Glavna faza (Main phase)

Glavna faza IOP protokola sastoji se od 3 komponente: nasumične predobrade, DEEP-ALI protokola i FRI protokola.

1. Nasumična predobrada (Randomized preprocessing):

   Nakon faze postavljanja, dokazivač pokreće program koji generira kontrolne stupce i podatkovne stupce, a zatim započinje nasumičnu predobradu. Konstrukcija ove faze uključuje:

   - Generiranje akumulatora za memoriju i bajtove
   - Slanje obveza za kontrolne i podatkovne stupce
   - Konstrukciju akumulatora pomoću slučajnih vrijednosti koje generira provjerivač
   - Generiranje polinoma valjanosti (fvalidity): Dokazivač koristi ograničenja i trag izvršenja za generiranje polinoma valjanosti. Polinom sažima sve potrebne provjere koje osiguravaju ispravnost izvršenja programa unutar protokola.
   - Slanje obveze za f_{validity}: Dokazivač konstruira svjedoka valjanosti (w_{validity}), koji predstavlja dokaz o ispravnosti polinoma, i šalje obvezu (commitment) za ovaj polinom provjerivaču. Obveza služi kao "zapečaćeni" dokaz da su sve potrebne provjere zadovoljene.
   - Provjera valjanosti pomoću slučajne vrijednosti z: Provjerivač odabire slučajnu vrijednost z iz polja $\mathbb{F}_q$ za evaluaciju polinoma valjanosti. Polje $\mathbb{F}_q$ predstavlja konačno polje koje sadrži q elemenata, gdje je q obično

prost broj ili potencija prostog broja. U kontekstu protokola, polje $\mathbb{F}_q$ omogućuje sigurno i efikasno računanje s elementima koji zadovoljavaju specifične matematičke uvjete. Procjenom polinoma na slučajnoj točki z, provjerivač može utvrditi da li polinom zadovoljava sva postavljena ograničenja bez potrebe za potpunom evaluacijom svih mogućih točaka.

- Korištenje DEEP kvocijentne tehnike: Nakon evaluacije na točki z, dokazivač koristi DEEP kvocijentnu tehniku za izradu DEEPAnswerSequence, koja omogućuje potvrdu da je provjera polinoma ispravna i da su sve interakcije u skladu s protokolom.

2. DEEP-ALI protokol: DEEP-ALI (Doubly Efficient Evaluation Proofs with Algebraic Lookup Indexing) protokol koristi se za kombiniranje ograničenja u složeniji polinom, čime se omogućuje učinkovitija provjera ispravnosti unutar dokaza nultog znanja. Omogućuje dodatne provjere pomoću algebraičkih odnosa i pretraživanja, što olakšava potvrdu valjanosti složenih polinoma.

   - Generiranje slučajnih brojeva: Provjerivač generira slučajne brojeve koji se koriste za kombiniranje pojedinačnih ograničenja u jedan "kombinirani polinom" i pomažu u provjeri različitih uvjeta unutar polinoma na siguran način.
   - Konstrukcija polinoma valjanosti: Dokazivač koristi dobivene slučajne brojeve i ograničenja za konstruiranje polinoma valjanosti (f_{validity}). Polinom valjanosti objedinjuje sva potrebna ograničenja i sažima ih u jedan matematički izraz koji dokazuje ispravnost svih potrebnih uvjeta.
   - Slanje obveze za polinom valjanosti: Nakon konstrukcije, dokazivač šalje obvezu za polinom valjanosti provjerivaču kako bi osigurao da su sve informacije o polinomu valjanosti sigurno zabilježene, ali i da se stvarni podaci ne otkrivaju provjerivaču.
   - Generiranje DEEPAnswerSequence Koristeći DEEP kvocijentnu tehniku, dokazivač generira niz odgovora (DEEPAnswerSequence) koji omogućuju provjeritelju da efikasno provjeri ispravnost kombiniranog polinoma na odabranim slučajnim točkama, smanjujući potrebu za potpunom evalu-

acijom i omogućuje učinkovitu i brzu provjeru valjanosti.

3. Batched FRI protokol: Nakon kombiniranja ograničenja, oba sudionika primjenjuju FRI kako bi provjerili blizinu koda RS[$\mathbb{K}$,D,ρ] (Reed-Solomon kod).

   - K: Predstavlja broj informativnih simboličkih vrijednosti unutar polinoma koje sadrže stvarne podatke.
   - D: Označava domenu evaluacije, odnosno skup točaka na kojima se polinom evaluira tijekom kodiranja ili provjere.
   - ρ: Označava stopu koda, što je omjer broja informativnih vrijednosti u odnosu na ukupan broj kodiranih vrijednosti, određujući učinkovitost kodiranja i otpornost na pogreške.

   Reed-Solomon kod je vrsta linearno-korektivnog koda koji se koristi za detekciju i ispravljanje pogrešaka u digitalnim komunikacijama i skladištenju podataka. Kodiranje se postiže evaluacijom polinoma na različitim točkama unutar definirane domene (D). Reed-Solomon kod može popraviti do $\frac{n-k}{2}$ pogrešaka, gdje je n ukupan broj kodiranih simboličkih vrijednosti, a k broj informativnih vrijednosti. Koristi se u sustavima poput CD-ova, DVD-ova, digitalna televizija i komunikacijskih protokola zbog svoje visoke učinkovitosti u ispravljanju pogrešaka.

   Ovi parametri su ključni za definiranje strukture polinoma i provjera unutar Reed-Solomon koda, čime se osigurava da polinomi koje generiraju sudionici protokola zadovoljavaju algebraička ograničenja i pravila za provjeru valjanosti.

4.7.2 Provjera (eng. verification)

Provjera valjanosti potvrde (detaljnije objašnjena u nastavku rada) odvija se kroz niz logičkih provjera koje provjerivač mora izvršiti kako bi potvrdio valjanost primljenih podataka. Ako bilo koja od ovih provjera zakaže, provjerivač odbacuje primljenu potvrdu. Glavne provjere koje se provode tijekom verifikacije su:

- Provjera polinoma valjanosti: Provjerivač koristi zadane točke (eng. "taps"), ograničenja c_{set}, i generirane slučajne vrijednosti λ kako bi izračunao vrijednost

polinoma valjanosti $f_{\text{validity}}(z)$ na odabranoj slučajnoj točki z. Provjerivač zatim uspoređuje dobivenu vrijednost s navedenom vrijednošću na pečatu (eng. seal). Bilo kakva razlika između izračunatih i navedenih vrijednosti ukazuje na neispravnost, zbog čega bi se potvrda odbacila.

- Provjera DEEPAnswerSequence za konzistentnost: Provjerivač pregledava DEEPAnswerSequence kako bi osigurao unutarnju konzistentnost podataka, proces uključuje interpolaciju stupaca za ponovno izračunavanje navedenih vrijednosti. Provjera DEEPAnswerSequence je ključna za potvrđivanje točnosti prijavljenog skupa vrijednosti (eng. "tapset"), jer osigurava da interpolacije odgovaraju očekivanim rezultatima.
- Provjera FRI upita: Za svaki FRI upit j, provjerivač koristi koeficijente u i α_{FRI} kako bi izračunao vrijednost $f_{\text{FRI}}(j)$. Rezultat se zatim uspoređuje s prijavljenom vrijednošću na potvrdi. Svaka razlika ukazuje na pogrešku ili potencijalno lažiranje podataka.
- Dodatna provjera unutar FRI upita: Provjerivač detaljno analizira unutarnje elemente svakog FRI upita kako bi osigurao ispravnost svih izvedenih koraka. Postupak uključuje provjeru matematičkih izraza, struktura podataka i svih povezanih vrijednosti unutar upita, osiguravajući dodatnu razinu sigurnosti jer provjerivač potvrđuje da su svi proračuni i upiti ispravno provedeni, čime se jamči cjelokupna valjanost.

4.8 Virtualna mašina nultog znanja (zkVM)

Prvi koraci u razvoju zkVM-a bili su fokusirani na stvaranje alata koji bi omogućio programerima da koriste postojeće jezike poput Rust-a i C++, čime bi se eliminirala potreba za specijaliziranim kriptografskim jezicima i omogućilo programerima da brzo razvijaju i implementiraju aplikacije koje mogu dokazati ispravnost izvršenja koda bez otkrivanja samog koda ili osjetljivih podataka. U nastavku su objašnjene komponente zkVM-a.

1. Gost program (Guest Program): Gost program je ključna komponenta zkVM aplikacije koja se dokazuje. Program se kompajlira u ELF binarni format i

izvršava unutar zkVM-a.

2. Izvršitelj (Executor): Izvršitelj pokreće ELF binarni kod i bilježi sesiju izvršenja, stvarajući izvršnu traku koja bilježi stanje stroja u svakom ciklusu takta.
3. Dokazivač (Prover): Dokazivač provjerava i dokazuje ispravnost sesije izvršenja, rezultirajući izdavanjem potvrde (eng. receipt) koja se može koristiti za verifikaciju.

Tehnološke prednosti RISC Zero zkVM-a su iduće:

- Jednostavnost i pristupačnost: Korištenjem jezika poput Rust-a, zkVM omogućava programerima da koriste zrele ekosustave i postojeće alate, smanjujući vrijeme i resurse potrebne za razvoj aplikacija.
- Učinkovitost i performanse: zkVM koristi GPU akceleraciju za CUDA i Metal, omogućavajući paralelnu obradu velikih programa i povećavajući brzinu generiranja dokaza.
- Široka primjena: zkVM omogućava razvoj raznovrsnih aplikacija, od provjere Ethereum blokova do dokazivanja prisutnosti objekata u slikama, sve dok održava privatnost podataka.

4.8.1 Potvrde

Potvrda (eng. Receipt) u kontekstu RISC Zero's zkVM-a je ključni element koji služi kao kriptografski dokaz da je određeni gost program izvršen na ispravan način. Potvrda sadrži rezultate izvršenog programa i dokaz njegove ispravnosti, pružajući kriptografsko jamstvo da su izračuni obavljeni pomoću očekivanog programa i očekivanog identifikatora slike (Image ID). Potvrde se sastoje od idućih komponenti:

- Dnevnik (eng. Journal): Ovaj dio sadrži javne izlaze programa, odnosno podatke koji su dostupni javnosti nakon izvršenja.
- Tvrdnja potvrde (eng. Receipt Claim): sadrži dnevnik zajedno s ostalim važnim detaljima i čini javni izlaz programa. Osim dnevnika, tvrdnja uključuje informacije o Image ID-u, izlaznom statusu (npr. je li program uspješno završio ili je naišao na grešku) te stanje memorije na kraju izvršenja. Tvrdnja potvrde

predstavlja ono što potvrda dokazuje. Na primjer, jednostavna tvrdnja može biti: "Izvršio sam kalkulator za Fibonaccijev niz s ulazom '21' i dobio izlaz '10946'." RISC Zero pruža formalan sustav za definiranje i dokazivanje takvih tvrdnji

- Pečat (eng. Seal): Ova kriptografska komponenta osigurava integritet i valjanost potvrde. Pečat je neprozirni blok podataka koji kriptografski potvrđuje valjanost tvrdnje unutar potvrde.

Kada korisnik pokrene zkVM aplikaciju, generira se potvrda koja se može poslati drugoj strani (npr. za potrebe verifikacije). Primatelj tada može verificirati potvrdu kako bi osigurao da su izračuni ispravno izvedeni koristeći očekivani program.

Prije izvršenja programa, korisnici se dogovaraju o gost programu i njegovom identifikatoru slike (Image ID) koji predstavlja jedinstveni identifikator binarnog programa koji će biti izvršen. Image ID osigurava da se točno određeni program izvršava, omogućujući provjeru da su rezultati došli od ispravnog programa.

U nastavku su opisani procesi korištenja potvrde:

1. Inspekcija potvrde: Potvrda sadrži dnevnik, koji prikazuje javne izlaze izračuna. Dnevnik se može izvući iz potvrde korištenjem metode receipt.journal pomoću koje je omogućeno korisnicima da pregledaju rezultate izračuna koji su javno dostupni.
2. Verifikacija potvrde: Verifikacija potvrde pruža kriptografsko jamstvo da je dnevnik stvoren pošteno koristeći gost program s očekivanim ID-om slike. Proces verifikacije osigurava da:
   - Izvršenje je bilo valjano.
   - Gost program koji je izvršen dosljedan je očekivanom ID-u slike.
   - Verifikacija potvrde se izvršava koristeći ugrađenu metodu objekta receipt, kojoj se daje `Image ID` kao parametar: receipt.verify(image_id).
3. Serijalizacija i deserijalizacija potvrda: Potvrde se mogu serijalizirati i deserijalizirati koristeći različite formate kodiranja. Serde je alat koji podržava mnoge formate kodiranja, omogućujući lako pohranjivanje i prijenos po-

tvrda. Primjer serijalizacije potvrde u bincode formatu: let bytes = bincode::serialize(&receipt);. Navedeni kod pohranjuje potvrdu u binarnom formatu, što je korisno za prijenos i arhiviranje.

4.8.2 Gost program

Gost kod u kontekstu RISC Zero zkVM-a odnosi se na aplikacijski kod koji se izvršava unutar virtualnog stroja. Može biti napisan u programskom jeziku Rust, koji je podržan i integriran unutar RISC Zero ekosustava. Nakon što je kod napisan, kompajlira se u ELF binarni format koji je kompatibilan s RISC-V arhitekturom. Moguće je i koristiti Rust knjižnice, poznate kao crates koje su napisane u C ili C++.

Osnovne funkcionalnosti za koje je odgovoran kod gost programa su iduće:

1. Čitanje ulaza
2. Pisanje izlaza "domaćinu" (eng. host), odnosno sustavu koji pokreće zkVM.
3. Predavanje javnih podataka u dnevnik

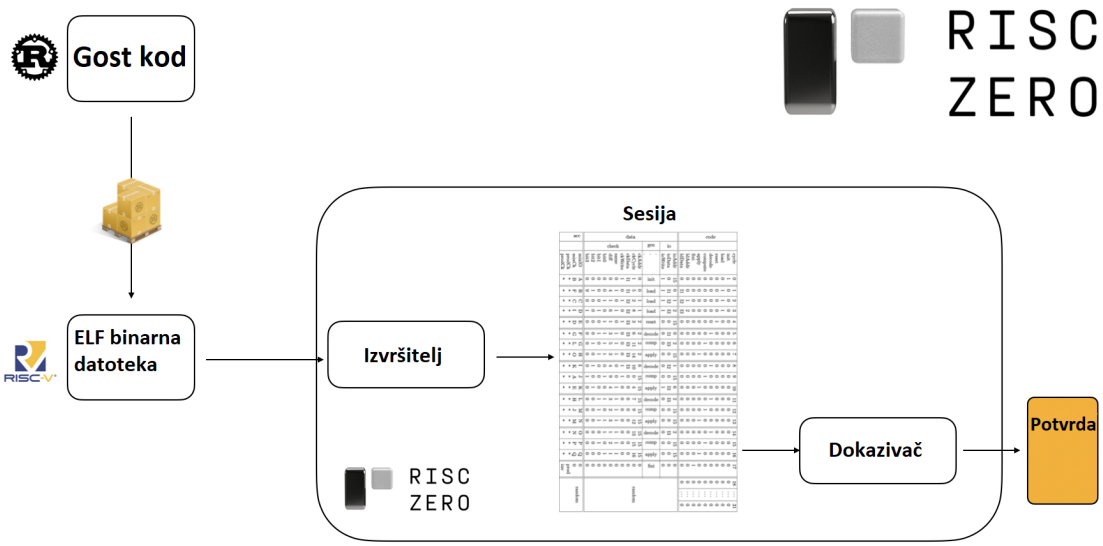
Za rad u gost programu, najbitniji je objekt `env`. Objekt `env` je ključan za rad gost programa u zkVM-u jer pruža sučelje za interakciju s okolinom gost stroja. U nastavku su navedene ključne uloge `env` objekta:

- Čitanje podataka: Funkcije poput `env::read` omogućuju gost programu da čita ulazne podatke koji su potrebni za izvršenje koji mogu biti u obliku jednostrukih vrijednosti ili nizova (eng. slices).
- Pisanje privatnih izlaza: Funkcije poput `env::write` omogućuju gost programu da piše privatne izlaze natrag domaćinu. Koriste se za rezultate koji se ne trebaju otkriti javno, već samo internim sustavima.
- Predavanje javnih izlaza: Funkcije poput `env::commit` omogućuju gost programu da predaje javne izlaze u dnevnik, koji su dostupni svim korisnicima i pružaju transparentnost i mogućnost verifikacije rezultata.

Program domaćin

Kod domaćina (eng. host code) u RISC Zero zkVM-u ima ključnu ulogu u orkestraciji izvršenja gost koda, upravljanju podacima i verifikaciji rezultata. Za razliku od gost koda, kod domaćina ne mora biti povjerljiv i može se izvršavati u neosiguranom okruženju.

- Učitavanje i izvršenje gost koda: Kod domaćina učitava gost kod u zkVM i pokreće njegovo izvršenje uključujući inicijalizaciju virtualnog stroja, učitavanje binarne datoteke gost koda i pokretanje izvršnog procesa. Kod domaćina može koristiti funkcije za pokretanje gost koda i upravljanje njegovim izvršenjem.
- Komunikacija s gost kodom: Kod domaćina pruža mehanizme za komunikaciju s gost kodom tijekom njegovog izvršenja poput slanja ulaznih podataka gost kodu i primanje rezultata iz izvršenja.
- Verifikacija rezultata: Nakon završetka izvršenja gost koda, kod domaćina može verificirati potvrde kako bi osigurao ispravnost i integritet rezultata. Primitci sadrže kriptografske dokaze koji potvrđuju da je gost kod izvršen ispravno i da su rezultati točni. Funkcije poput `Receipt::verify` omogućuju domaćinu da provjeri primitke i osigura povjerenje u rezultate.
- Upravljanje dnevnikom: Kod domaćina također upravlja dnevnikom koji sadrži javne izlaze izračuna gost koda te može koristiti funkcije za čitanje i pisanje u dnevnik kako bi osigurao pravilno bilježenje rezultata.



Slika 4.3 Vizualni prikaz "od Rust-a do dokaza", preuzeto iz [15]

Poglavlje 5

Rad u zkVM-u

5.1 Instalacija razvojnog okruženja za RISC Zero

Instalacija i konfiguracija razvojnog okruženja za RISC Zero zkVM zahtijeva nekoliko koraka, uključujući instalaciju Rust programskog jezika i potrebne podrške za RISC Zero zkVM. Slijede detaljni koraci za postavljanje ovog okruženja u UNIX sustavima (Linux, MacOS...).

5.1.1 Instalacija Rust programskog jezika

Rust, programski jezik razvijen od strane Mozilla Researcha, prošao je nevjerojatan put od svojeg početka kao sporednog projekta do statusa najbrže rastućeg jezika na svijetu. Razlog tome leži u njegovim jedinstvenim značajkama koje adresiraju sigurnost i performanse, što ga čini idealnim za razne vrste razvoja softvera.

Rust je stvoren kao odgovor na izazove s kojima su se suočavali programeri koristeći jezike poput C i C++. Glavni problemi uključuju sigurnost memorije i složenost višedretvenog programiranja. Rust koristi sustav vlasništva (eng. ownership system) koji pomaže u upravljanju memorijom bez potrebe za "sakupljanjem smeća", što smanjuje mogućnost grešaka i poboljšava performanse.

Popularnost Rust-a značajno je porasla zahvaljujući njegovoj sposobnosti da osigura sigurnost bez žrtvovanja brzine. Mnogi veliki tehnološki divovi, poput Micro-

softa i Amazona, počeli su koristiti Rust u svojim projektima, što je dodatno povećalo njegovu popularnost i prihvaćenost u industriji.

Rust ima vrlo aktivnu i podržavajuću zajednicu programera. Svake godine se provodi istraživanje među korisnicima Rust-a, koje pruža dragocjene uvide u potrebe i izazove s kojima se suočavaju programeri. Rezultati tih istraživanja koriste se za daljnje poboljšanje jezika i alata koji ga prate.

Instalacija Rust programskog jezika se izvršava idućom naredbom:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Nakon instalacije Rust programskog jezika, vrlo je bitno postaviti Cargo.

5.1.2 Cargo alat za upravljanje paketima u Rust-u

Cargo je upravitelj paketa i alat za izgradnju projekata u Rust programskom jeziku. Omogućava jednostavno upravljanje ovisnostima, kompilaciju koda, pokretanje testova i stvaranje distribucija softvera. Cargo je jedan od ključnih razloga zašto je Rust postao popularan među programerima, jer znatno pojednostavljuje proces razvoja softvera.

5.1.3 Značajke i koristi Cargo Alata

Cargo nudi brojne značajke koje olakšavaju razvoj u Rust-u:

- Upravljanje ovisnostima: Cargo omogućava jednostavno dodavanje, ažuriranje i uklanjanje ovisnosti unutar projekta. Ovisnosti su definirane u `Cargo.toml` datoteci, gdje se navode sve biblioteke koje projekt koristi.
- Kompilacija i izgradnja: Cargo automatski preuzima sve potrebne ovisnosti i kompilira projekt koristeći optimalne postavke. Podržava različite profile kompilacije, uključujući razvojne, testne i produkcijske profile.
- Pokretanje testova: Alat omogućava jednostavno pokretanje testova unutar projekta, čime se osigurava da je kod ispravan i funkcionalan. Testovi se definiraju unutar samih Rust modula, a Cargo ih automatski prepoznaje i izvršava.

- Stvaranje distribucija: Cargo olakšava stvaranje distributivnih paketa (binarnog ili izvornog koda) koji se mogu dijeliti s drugima ili distribuirati putem Rust-ovog paketa, `crates.io`.

5.1.4 Instalacija Cargo alata

Cargo se instalira zajedno s Rust programskim jezikom putem `rustup` alata. Bitno je dodati Cargo u PATH kako bi bio jednostavno pristupan u terminalu u bilo kojem trenutku.

1. Dodavanje Cargo-a u PATH: `source $HOME/.cargo/env`
2. Provjera instalacije: `cargo --version`

5.1.5 Osnovna upotreba Cargo alata

Nakon instalacije, Cargo se koristi za upravljanje Rust projektima. U nastavku su osnovne naredbe za rad s Cargo alatom:

- Kreiranje novog projekta:
  ```
  cargo new naziv_projekta
  cd naziv_projekta
  ```

- Izgradnja projekta:
  ```
  cargo build
  ```

- Pokretanje projekta:
  ```
  cargo run
  ```

- Pokretanje testova:
  ```
  cargo test
  ```

- Dodavanje ovisnosti:

  ```
  # Dodavanje novih ovisnosti u Cargo.toml datoteku:
  [dependencies]
  naziv_ovisnosti = "verzija"
  ```

  ili koristeći naredbu:
  `cargo add <naziv_ovisnosti>`

- Ažuriranje Ovisnosti:
  `cargo update`

Instalacija RISC Zero programskog okruženja

Postoje specifični paketi koje RISC Zero koristi za svoj zkVM i oni se instaliraju idućim naredbama koristeći cargo alat:

```
cargo install cargo-binstall
cargo binstall cargo-risczero
cargo risczero install
```

Bitno je napomenuti, ako se koristi x86/64 macOS, umjesto naredbe

```
cargo risczero install
```

potrebno je koristiti naredbu:

```
cargo risczero build-toolchain
```

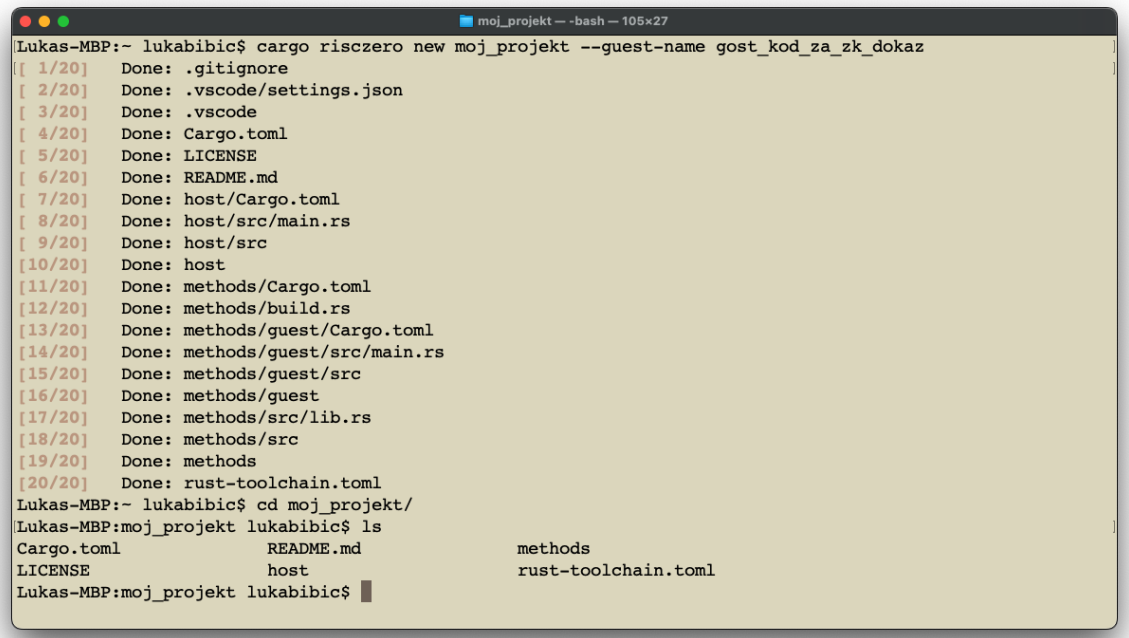
5.2 Izrada programa koristeći RISC Zero

Kako bi se pojednostavila izrada ZK projekata koristeći RISC Zero, koristit će se `cargo risczero` paket koji automatski stvara "šablonski" (eng. template) rust projekt kojeg je moguće dalje uređivati kako bi se manje vremena izgubilo na samo okruženje za zkVM kod.

Prateći prethodne upute za instalaciju, moguće je pokrenuti sljedeću naredbu koja stvara taj projekt:

```
cargo risczero new <ime_projekta> --guest-name <ime_gost_koda>
```

Naredba će korijenu direktorija cijelog projekta stvoriti direktorije `host` koja sadrži sve što je potrebno da se pokrene kod domaćina i `methods` koji sadrži sve što je potrebno da se izvrši kod unutar zkVM-a odnosno gost kod, te datoteke `Cargo.toml` koja sadrži potrebne pakete za domaćina, `rust-toolchain.toml` i `Readme.md` s uputama za korištenja i LICENSE datoteku.



Slika 5.1 Shema za pohranu podataka

Primjećuje se da postoje i `src` direktoriji unutar `host` i `methods` direktorija. U njima se nalaze unaprijed napravljene rust skripte za domaćina i gosta koje pružaju jasne primjere korištenja i mogu se prilagoditi za vlastitu upotrebu. U nastavku su automatski generirani kodovi domaćina i gosta.

```
use methods::{
    GOST_KOD_ZA_ZK_DOKAZ_ELF, GOST_KOD_ZA_ZK_DOKAZ_ID
};
use risc0_zkvm::{default_prover, ExecutorEnv};

fn main() {
    // Initialize tracing. In order to view logs,
    // run `RUST_LOG=info cargo run`
    tracing_subscriber::fmt()
        .with_env_filter
        (tracing_subscriber::filter::EnvFilter::from_default_env())
        .init();

    let input: u32 = 15 * u32::pow(2, 27) + 1;
    let env = ExecutorEnv::builder()
        .write(&input)
        .unwrap()
        .build()
        .unwrap();

    // Obtain the default prover.
    let prover = default_prover();

    let prove_info = prover
        .prove(env, GOST_KOD_ZA_ZK_DOKAZ_ELF)
        .unwrap();

    // extract the receipt.
    let receipt = prove_info.receipt;

    let _output: u32 = receipt.journal.decode().unwrap();

```

```
    // The receipt was verified at the end of proving,
    // but the below code is an
    // example of how someone else could verify this receipt.
    receipt
        .verify(GOST_KOD_ZA_ZK_DOKAZ_ID)
        .unwrap();
}
```

Programski kod 5.1 Kod domaćina

```
use risc0_zkvm::guest::env;

fn main() {
    // TODO: Implement your guest code here

    // read the input
    let input: u32 = env::read();

    // TODO: do something with the input

    // write public output to the journal
    env::commit(&input);
}
```

Programski kod 5.2 Kod gosta

U kodu domaćina može se vidjeti kako se koristi RISC Zero zkVM za stvaranje okruženja za izvršenje (ExecutorEnv), provođenje dokaza (eng. proving) i verifikaciju potvrda (eng. receipt verification).

Proces inicijalizacije sustava za praćenje (tracing) odvija se na sljedeći način: koristeći biblioteku `tracing_subscriber`, najprije se poziva funkcija `fmt()` koja postavlja format izlaza. Zatim se koristi metoda `with_env_filter()` kako bi se definirali filtri iz zadanih postavki okoliša (eng. environment), pri čemu se koriste funkcionalnosti `EnvFilter::from_default_env()`. Na kraju se poziva `init()` kako bi se inicijalizirao cijeli sustav praćenja.

Nakon inicijalizacije sustava za praćenje izvršenja programa, potrebno je koristiti naredbu `RUST_LOG=info cargo run` koja omogućuje prikaz informativnih poruka tijekom izvršenja programa.

`ExecutorEnv` je okruženje za izvršenje koje opisuje konfiguracije za zkVM, uključujući ulazne podatke programa. U ovom primjeru, prvo se definira ulazna vrijednost `input` pomoću izraza `15 * u32::pow(2, 27) + 1`, a zatim se kreira `ExecutorEnv` koristeći `ExecutorEnv::builder()`, gdje se ta vrijednost dodaje u okruženje putem metode `write()`. Nakon toga, eventualne greške pri dodavanju ulazne vrijednosti rješavaju se korištenjem metode `unwrap()`, dok se kompletno okruženje za izvršenje postavlja pozivom metode `build()`. Konačno, metoda `unwrap()` osigurava ispravno rukovanje eventualnim iznimkama, a postavljeno okruženje koristi se kao ulaz za gostujući program unutar zkVM-a.

Funkcija `default_prover()` vraća zadani objekt prover, koji se koristi za provođenje dokaza unutar zkVM sustava i pohranjuje se u varijablu `prover`, čime se omogućuje daljnje korištenje ovog objekta za generiranje i verifikaciju kriptografskih dokaza.

Metoda `prove` koristi se za provođenje dokaza temeljenog na specificiranoj ELF binarnoj datoteci, u kombinaciji s konfiguracijama iz objekta `env`. U ovom primjeru, metoda `prove` poziva se na objektu `prover`, s `env` i `GOST_KOD_ZA_ZK_DOKAZ_ELF` kao argumentima. Rezultat ovog procesa je objekt potvrde (`receipt`), koji sadrži dokaz da je izvršenje programa bilo ispravno.

Dnevnik (`journal`) unutar potvrde sadrži javne izlaze izvršenog izračuna, koje

je moguće dekodirati pomoću metode `decode()`. U ovom slučaju, dekodirani izlaz pohranjuje se u varijablu `_output`. Nakon toga, metoda `verify()` koristi identifikator slike `GOST_KOD_ZA_ZK_DOKAZ_ID` kako bi verificirala primitak i osigurala da je izvršenje bilo konzistentno s očekivanim rezultatom. Ukoliko je verifikacija uspješna, metoda `unwrap()` omogućuje rješavanje mogućih iznimki.

Gost program sastoji se od nekoliko ključnih koraka potrebnih za njegovo izvršenje unutar RISC Zero zkVM okruženja.

Prvi korak je inicijalizacija, koja isključuje korištenje standardne biblioteke i glavne funkcije programa, korištenjem direktiva `#![no_main]` i `#![no_std]`. Ovaj pristup je uobičajen za razvoj u okruženjima s ograničenim resursima, gdje se želi smanjiti upotreba memorije i drugih resursa.

Zatim se koristi `env` modul, koji pruža funkcije za interakciju s okolinom gostujućeg stroja. `Env` omogućava čitanje ulaznih podataka i zapisivanje izlaznih podataka, a u kodu se uvozi pomoću naredbe `use risc0_zkvm::guest::env;`.

Ulazna točka programa definira se korištenjem makro naredbe `risc0_zkvm::guest::entry!(main)`, koja specificira funkciju `main` kao početnu točku izvršenja gostujućeg programa.

Za čitanje ulaznih podataka koristi se funkcija `env::read()`, koja čita ulaz tipa `u32`. Nakon što se ulazni podatak obradi, rezultat se zapisuje kao javni izlaz u dnevnik pomoću funkcije `env::commit()`, koja osigurava da će izlazni podaci biti dostupni za verifikaciju u kasnijim fazama.

Opisani primjer prikazuje osnovne korake za izradu gostujućeg programa koji može čitati ulazne podatke, obrađivati ih i zapisivati rezultate u dnevnik za kasniju verifikaciju. Dodatno, korištenje `no_std` okruženja omogućava optimizaciju resursa, čime se poboljšava efikasnost i sigurnost aplikacije.

Pokretanje projekta se provodi kao i kod svakog drugog Rust projekta. Komanda `cargo build` (uz opcionalnu `--release` zastavicu za produkcijsko okruženje) koristi se za kompiliranje projekta, dok se komanda `cargo run` koristi za njegovo pokretanje.

5.3 Analiza različitih programa

U nastavku će se ispitati različiti programi, alati kojima se procjenjuju njihove performanse, te način na koji ti programi funkcioniraju i njihovi koncepti. Analiza će obuhvatiti detaljan pregled različitih programskih rješenja i metoda koje se koriste za optimizaciju performansi. Cilj je pružiti uvid u različite pristupe programiranju unutar RISC Zero zkVM okruženja.

5.3.1 ZKP SHA256 Hash-a

Ideja iza zadatka za SHA-256 ZKP je demonstrirati kako koristiti RISC Zero zkVM za stvaranje dokaza da je određeni SHA-256 hash izračun izveden ispravno, a da se pritom ne otkriva ulazni podatak. Primjer poput ovog služi za ilustraciju praktične primjene ZKP u provjeri integriteta izračuna na način koji čuva privatnost te se može iskoristiti u sustavima za prijavu u kojima bi korisnik mogao dokazati da zna lozinku bez da ju otkrije.

U nastavku je prilagođen gost kod za ovaj koncept.

```
#![no_main]

use risc0_zkvm::guest::env;
use risc0_zkvm::sha::{Impl, Sha256};
use sha_core::DateAndHash;

risc0_zkvm::guest::entry!(main);


fn main() {
    // čitanje ulaza
    let input: String = env::read();
    let date_now: u64 = env::read();
    let sha_hash = Impl::hash_bytes(&input.as_bytes());

```

```
16      // stvaranje strukture DateAndHash
17      let sha_receipt = DateAndHash {
18          date: date_now,
19          hash: *sha_hash,
20      };
21
22      // zapisivanje javnog izlaza u dnevnik
23      env::commit(&sha_receipt);
24  }
```

Programski kod 5.3 Gost kod za SHA hash

Razlike u odnosu na prvotni kod su sljedeće:

- **Podrška za std biblioteku**: Uklonjena je direktiva `no_std` kako bi se omogućila podrška za standardnu biblioteku, koja je eksperimentalna u ovom okruženju i potrebna za korištenje `String` strukture.
- **Uvođenje biblioteka**:

  ```
  use risc0_zkvm::guest::env;
  use risc0_zkvm::sha::{Impl, Sha256};
  use sha_core::DateAndHash;
  ```

  Dodane su dodatne biblioteke potrebne za SHA-256 hash i korištenje strukture `DateAndHash`.
- **Struktura DateAndHash**:

  ```
  struct DateAndHash {
      date: u64,
      hash: [u8; 32],
  }
  ```

  Struktura `DateAndHash` sprema datum i SHA-256 hash vrijednost. Struktura

je napravljena u zasebnom direktoriju nazvanom `core` te je uključena u gost kod modificiranjem Cargo.toml datoteke za gosta.

Modificirani gost kod omogućuje čitanje dodatnog ulaznog podatka (trenutnog datuma) i spremanje hash vrijednosti zajedno s datumom u strukturu `DateAndHash`, što je korisno za aplikacije koje zahtijevaju dokazivanje integriteta podataka u određenom vremenskom kontekstu.

Vremenska analiza

Moguće je pokrenuti program na način u kojem se dodatne informacije o izvođenju prikazuju u terminalu koristeći:

```
RUST_LOG=info cargo run
```

```
Lukas-MBP:moj_projekt lukabibic$ RUST_LOG=info cargo run
   Compiling methods v0.1.0 (/Users/lukabibic/moj_projekt/methods)
gost_kod_za_zk_dokaz: Starting build for riscv32im-risc0-zkvm-elf
gost_kod_za_zk_dokaz:     Finished release [optimized] target(s) in 0.04s
   Compiling host v0.1.0 (/Users/lukabibic/moj_projekt/host)
    Finished dev [optimized + debuginfo] target(s) in 1.21s
     Running `target/debug/host`
2024-08-05T09:07:50.411587Z  INFO executor: risc0_zkvm::host::server::exec::executor: execution time: 1.1ms
2024-08-05T09:07:50.411617Z  INFO executor: risc0_zkvm::host::server::session: number of segments: 1
2024-08-05T09:07:50.411620Z  INFO executor: risc0_zkvm::host::server::session: total cycles: 65536
2024-08-05T09:07:50.411621Z  INFO executor: risc0_zkvm::host::server::session: user cycles: 4597
2024-08-05T09:07:50.411623Z  INFO executor: risc0_zkvm::host::server::session: cycle efficiency: 7%
2024-08-05T09:07:50.411626Z  INFO risc0_zkvm::host::server::prove::prover_impl: prove_session: cpu, exit_code = Halted(0), journal = Some("669
6b06600000000d580754468330f66e5c155e16968a2be2a7828eca0885b42a0ce356594bfeab9")
Date: 1722848870
Hash: Digest(d580754468330f66e5c155e16968a2be2a7828eca0885b42a0ce356594bfeab9)
Execution time: 5.655054s
```

Slika 5.2 Izlazne informativne poruke za SHA program

`Execution time` pokazuje da je izvršavanje trajalo 1.5 milisekundi unutar gosta.
`Number of segments` pokazuje da je izvršavanje bilo podijeljeno u jedan segment izvšenja.
`Total cycles` pokazuje da je izvršenje trajalo 65536 CPU ciklusa.
`User cycles` pokazuje broj ciklusa koji se korišteni za izvršavanje korisničkog koda, odnosno specifičnih operacija koje je definirao programer unutar aplikacije.
`Cycle efficiency` pokazuje da je samo 7% ukupnih ciklusa bilo iskorišteno za korisničke zadatke.

Dodavanjem informativnih poruka koje zabilježavaju početno i krajnje vrijeme unutar host programa i njihovim oduzimanjem se dobilo da ukupno trajanje programa iznosi oko 5.5 sekundi na M3 Pro MacBook Pro. Navedeno vrijme izvršavanje je relativno sporo za zadatak koji je računalno jednostavan za odraditi. Iz ovog razloga, RISC Zero nudi mogućnost ubrzavanja programa koristeći GPU ubrzanje.

GPU akceleracija je ključna tehnologija u RISC Zero koja omogućava ubrzanje procesa dokazivanja koristeći moć paralelne obrade GPU-ova. U nastavku je detaljan pregled implementacije i prednosti korištenja GPU akceleracije u RISC Zero sustavu, s posebnim naglaskom na korištenje Metal API-a za ubrzanje zadataka na M3 Pro MacBook Pro koji koristi ARM arhitekturu.

5.3.2 Prednosti GPU akceleracije

- Ubrzano izvođenje dokaza:
  - GPU-ovi su dizajnirani za paralelnu obradu velikih količina podataka, što je idealno za kriptografske operacije koje zahtijevaju intenzivne računalne resurse.
  - Ubrzanje omogućuje brže generiranje dokaza, smanjujući ukupno vrijeme potrebno za verifikaciju.
- Povećana efikasnost:
  - Korištenje GPU-ova može smanjiti opterećenje na CPU, oslobađajući ga za druge zadatke.
  - Bolja raspodjela računalnih resursa rezultira efikasnijim radom cijelog sustava.
- Smanjenje troškova:
  - Ubrzavanjem procesa dokazivanja može se smanjiti potreba za skupim hardverskim resursima, što može dovesti do smanjenja troškova infrastrukture.

5.3.3 Tehnički detalji implementacije

- Ubrzanje prover-a:
  - U RISC Zero, proces dokazivanja može biti značajno ubrzan korištenjem specijaliziranih GPU algoritama koji su optimizirani za masivno paralelne operacije.
  - To uključuje korištenje CUDA (Compute Unified Device Architecture) platforme koja omogućuje programiranje i izvršavanje izračuna na NVIDIA GPU-ovima. Međutim, za M3 Pro MacBook Pro, koristi se Metal API koji je prilagođen za ARM arhitekturu.
- Paralelno procesiranje:
  - GPU-ovi mogu izvršavati tisuće niti paralelno, što omogućava brže izvođenje složenih kriptografskih operacija.
  - Paralelizacija se postiže razbijanjem velikih problema na manje zadatke za istovremeno izvršavanje.

5.3.4 Korištenje Metal API-a za ARM Arhitekturu

- Instalacija i konfiguracija:
  - Prvo je potrebno osigurati da je Metal API pravilno instaliran i konfiguriran na sustavu.
  - Prilagodba Cargo.toml datoteke u host direktoriju kako bi se omogućilo korištenje Metal API-a za GPU akceleraciju.

host > Cargo.toml

```
1   [package]
2   name = "host"
3   version = "0.1.0"
4   edition = "2021"
5
6   [dependencies]
7   methods = { path = "../methods" }
8   risc0-zkvm = { version = "0.21.0" }
9   tracing-subscriber = { version = "0.3", features = ["env-filter"] }
10  serde = "1.0"
11  sha-core = { path = "../core" }
12
13  [features]
14  metal = ["risc0-zkvm/metal"]
```

Slika 5.3 Dodavanje Metal API-ja u Cargo.toml datoteku u host-u direktoriju

Sljedeća slika pokazuje da se vrijeme izvršavanja smanjilo na samo 1.5 sekundi korištenjem Metal API-ja, što predstavlja poboljšanje od čak 72.7%.

```
Lukas-MBP:moj_projekt lukabibic$ RUST_LOG=info cargo run -F metal
   Compiling methods v0.1.0 (/Users/lukabibic/moj_projekt/methods)
gost_kod_za_zk_dokaz: Starting build for riscv32im-risc0-zkvm-elf
gost_kod_za_zk_dokaz:     Finished release [optimized] target(s) in 0.05s
   Compiling host v0.1.0 (/Users/lukabibic/moj_projekt/host)
    Finished dev [optimized + debuginfo] target(s) in 2.64s
     Running `target/debug/host`
2024-08-05T09:20:12.127702Z  INFO executor: risc0_zkvm::host::server::exec::executor: execution time: 1.4ms
2024-08-05T09:20:12.127734Z  INFO executor: risc0_zkvm::host::server::session: number of segments: 1
2024-08-05T09:20:12.127737Z  INFO executor: risc0_zkvm::host::server::session: total cycles: 65536
2024-08-05T09:20:12.127738Z  INFO executor: risc0_zkvm::host::server::session: user cycles: 4597
2024-08-05T09:20:12.127740Z  INFO executor: risc0_zkvm::host::server::session: cycle efficiency: 7%
2024-08-05T09:20:12.127755Z  INFO risc0_zkvm::host::server::prove::prover_impl: prove_session: metal, exit_code = Halted(0), journal = Some("4
c99b0660000000d580754468330f66e5c155e16968a2be2a7828eca0885b42a0ce356594bfeab9")
Date: 1722849612
Hash: Digest(d580754468330f66e5c155e16968a2be2a7828eca0885b42a0ce356594bfeab9)
Execution time: 1.512786s
```

Slika 5.4 Izlazne informativne poruke za SHA program

5.4 Profiliranje gost koda u RISC Zero zkVM

U ovom poglavlju analizirat će se različiti programi i alati koji se koriste za profiliranje performansi gost koda u RISC Zero zkVM. Profiliranje je ključno za identifikaciju i optimizaciju dijelova koda koji troše najviše resursa, odnosno omogućuje efikasniji rad aplikacija.

5.4.1 Instalacija alata

Za profiliranje koristit će se alat zvan **Pprof**. Razvio ga je Google te pomaže u analiziranju i vizualiziranju podataka o performansama aplikacija. Omogućava korisnicima da identificiraju uska grla u izvedbi koda i optimiziraju ih. Pprof podržava različite vrste profila, uključujući CPU, memoriju, blokiranje, i trajanje korutina (eng. coroutine). Alat generira vizualizacije poput grafova plamena (eng. Flame graph), koje pružaju jasnu sliku o tome koji dijelovi koda najviše troše resurse. Pprof je posebno koristan za razvojne inženjere koji žele poboljšati efikasnost svojih aplikacija.

Za instalaciju Pprof alata, potrebno je instalirati Go programski jezik te u njemu dolazi kao standardni alat. To se može napraviti idućom naredbama za Linux odnosno MacOS:

```
# Ubuntu instalacija
sudo apt install golang-go

# MacOs instalacija
brew install go
```

Pprof koristi uzorkovanje, što znači da povremeno bilježi pozivni stog tijekom izvršavanja programa. Uzorkovanje je selektivno i fokusira se na određene segmente koda, ignorirajući druge dijelove izvršenja. Selektivno uzorkovanje omogućuje Pprof-u alatu pružanje detaljnih informacija o performansama bez značajnog utjecaja na ukupno izvršenje programa.

Ukupan broj ciklusa prikazan od strane `risc0_zkvm::host::server::session` uključuje sve cikluse koje virtualni stroj izvršava, uključujući sve operacije, prebacivanja konteksta i režijske troškove. To znači da se bilježe svi aspekti izvršenja, bez obzira na njihov utjecaj na performanse.

Zbog različitih načina rada, broj ciklusa dobiven korištenjem Pprof-a i RISC Zero informativnih poruka je različit, ali pprof dobro ukazuje na dijelove koda koji troše najviše ciklusa.

Za korištenje pprof alata potrebno je stvoriti datoteku u proto formatu koja

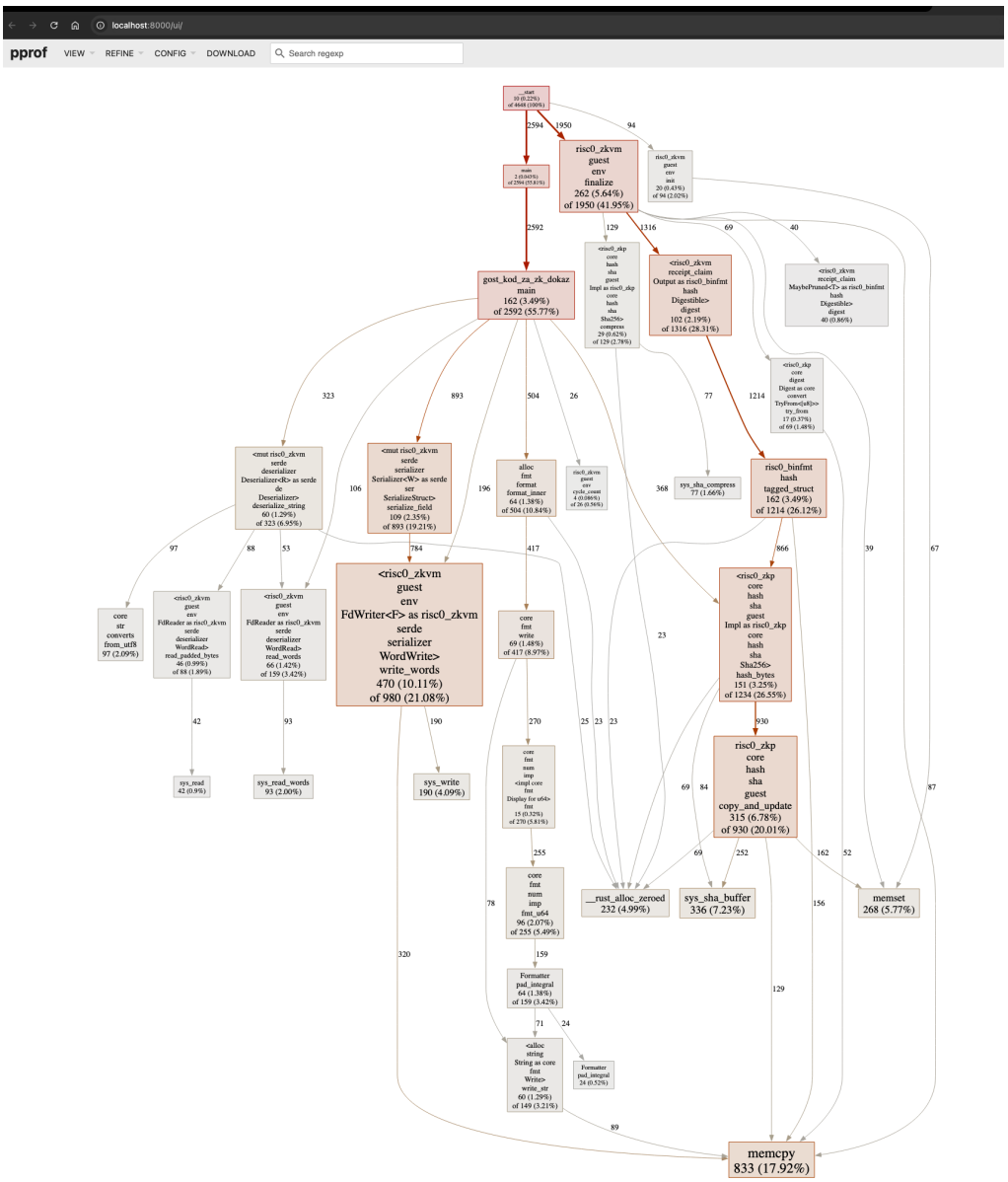
opisuje skup callstackova i informacije o simbolizaciji. Stvaranje datoteke se može ostvariti postavljanjem varijable okruženja tijekom pokretanja programa:

```
RISC0_PPROF_OUT=profile.db cargo run --release -F metal
```

Ova komanda će stvoriti profile.db datoteku koju je zatim moguće analizirati pprof alatom u lokalnom web pregledniku idućom naredbom:

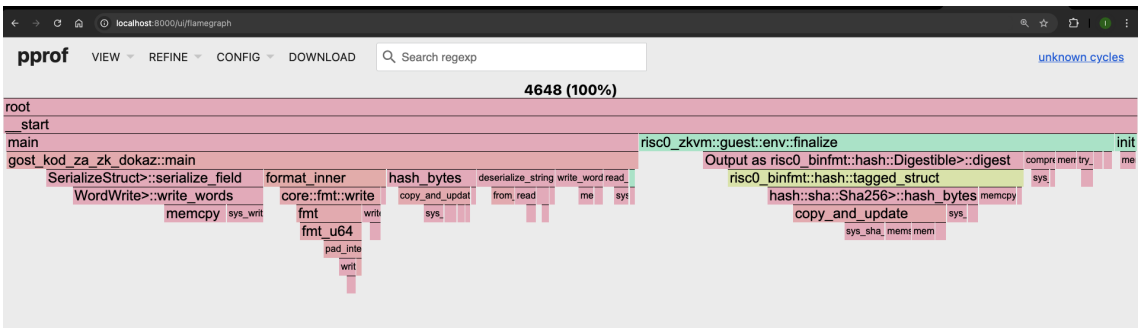
```
go tool pprof -http :8000 profile.db
```

Korištenjem prethodno generirane datoteke profile.db u proto formatu, alat pprof kreira web aplikaciju koja je dostupna korisniku putem preglednika na adresi http://localhost:8000/ui/. Na idućoj slici vidljiva je početna stranica te web aplikacije koja sadrži graf stabla koji opisuje dijelove koda i cikluse koji su utrošeni na njihovo izvršavanje.



Slika 5.5 Prikaz početna stranice pprof alata za SHA256 primjer

Takav graf stabla može biti teži za čitati, ali zato postoje i mnogi drugi grafovi s kojima se vizualizira trošenje CPU ciklusa. Pritiskom na gumb `View` moguće je odabrati druge grafove. Na idućoj slici je prikazan graf plamena s kojim je jednostavnije shvatiti koji dijelovi programa troše više resursa.



Slika 5.6 Graf plamena pprof alata za SHA256 primjer

Svi grafovi pprof alata su interaktivni te omogućuju klikanje na svaki od objekata kako bi se otvorilo više informacija i komponentama koje ju grade. Prelaskom miša preko jednog od pravokutnika, ispisuje se točan broj ciklusa koji gradi tu komponentu te koliki je njen postotak u odnosu na ukupan broj ciklusa programa. Analizom grafa koristeći ove funkcionalnosti, moguće je primjetiti iduće činjenice:

- Funkcija `main` troši 55.8%, a funkcije env::finalize i env::init zajedno iznose 44%. Od približno 2600 CPU ciklusa kojih zauzima main funkcija, `SerializeStruct` funkcija zauzima približno 900 ciklusa, odnosno 34.6%.
  - Analiza ukazuje na problematične dijelove koda, te omogućava programerima da ne troše vrijeme optimizirajući dijelove koda koji iznose malene postotke ukupnog broja ciklusa.
  - Optimizacija dijelova koda koji zahtjevaju veliki broj ciklusa je ključna za delegiranje zadataka trećim stranama, npr. **Bonsai** (opisan kasnije u radu) čije naplaćivanje ovisi o broju ciklusa.
  - Serializacija podataka u DateAndHash strukturu napravljenu od ulaznih podataka koje je proslijedio domaćin je skupa operacija te se na ovakve dijelove programa može dati posebna pažnja kako bi se optimizirali.

5.5 Delegiranje programa trećim stranama

Glavna ideja zkVM-a je osigurati povjerenje u rezultate, čak i kada su generirani na udaljenom serveru (od treće strane). U ovom radu za treću stranu će se koristiti Bonsai. Bonsai je RISC Zero platforma koja omogućuje delegiranje generiranja kriptografskih dokaza na udaljene servere, umjesto da se to radi lokalno na uređaju. Od velike je koristi za uređaje s ograničenim računalnim resursima jer omogućava skalabilnost bez kompromisa u sigurnosti. Bonsai smanjuje vrijeme generiranja dokaza koristeći paralelizaciju kroz stotine GPU-ova.

5.5.1 Determinističke kompilacije

Determinističke kompilacije odnose se na proces kompilacije koji uvijek proizvodi identičan binarni fajl, bit po bit. U kontekstu razvoja aplikacija za RISC Zero, takve kompilacije su ključne za osiguravanje jasne veze između izvornog koda i generiranog `Image ID-a`. Da bi se postigle determinističke kompilacije za zkVM aplikaciju, koristi se naredba `cargo risczero build`, koja pokreće rustc kompajler unutar Docker okruženja, čime se osigurava dosljednost i reproducibilnost rezultata.

Image ID se prosljeđuje Bonsai-ju te se provjera Bonsai vraćenog dokaza provodi koristeći prethodno naveden `receipt.verify()`.

Image ID se može programski dobiti i koristeći `compute_image_id` funkciju risc0_zkvm paketa

```
let image_id = hex::encode(
compute_image_id(GOST_KOD_ZA_ZK_DOKAZ_ELF)?);
```

Iako je ovo možda jednostavnije jer ne postoji ovisnost o Dockeru, prednost `cargo risczero build` jest u tome što se izvodi u kontejnerskom okruženju te se time osigurava apsolutno isti izračun Image ID-ja neovisno na kojem računalu ga netko pokreće u odnosu na programsko rješenje kroz Rust paket.

```
Lukas-MBP:moj_projekt lukabibic$ cargo risczero build --manifest-path methods/guest/Cargo.toml
cargo:rerun-if-env-changed=RISC0_SKIP_BUILD
Docker context: "/Users/lukabibic/moj_projekt"
Building ELF binaries in gost_kod_za_zk_dokaz for riscv32im-risc0-zkvm-elf target...
Docker version 26.1.0, build 9714adc6c7
[+] Building 94.9s (10/10) FINISHED                                                                    docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                0.0s
 => => transferring dockerfile: 758B                                                                                0.0s
 => [internal] load metadata for docker.io/risczero/risc0-guest-builder:v2024-01-31.1                               2.1s
 => [build 1/5] FROM docker.io/risczero/risc0-guest-builder:v2024-01-31.1@sha256:b30cf76019ff0d06e96d0d974ac5710  37.2s
 => => resolve docker.io/risczero/risc0-guest-builder:v2024-01-31.1@sha256:b30cf76019ff0d06e96d0d974ac5710074bab9  0.0s
 => => sha256:29b46c88354680433d1e95aca12bdad953038b061db22432925ce5f15d7273ad 3.51kB / 3.51kB                      0.0s
 => => sha256:6377b8b8fce7450cef3d5591a7ebdec7e0fdc1fc8c744a8e4f4c2815e2d92194 33.69MB / 33.69MB                    2.3s
 => => sha256:6620614d7af044f2209279d91dd6d54490a48e6ab7c9e3565381a97dadd70a81 126.21MB / 126.21MB                  3.7s
 => => sha256:b30cf76019ff0d06e96d0d974ac5710074bab9fd2e6550eea034f2548810d527 1.80kB / 1.80kB                      0.0s
 => => sha256:7007490126efaae58924972668256aaeb4858e6c4537eb4257e1978719b958c7 28.58MB / 28.58MB                    0.8s
 => => sha256:4f69ba4ede49a40e1b961f4dd6c57dea4254e44cf6120a406d323a8a0861358b 259.46MB / 259.46MB                  7.1s
 => => extracting sha256:7007490126efaae58924972668256aaeb4858e6c4537eb4257e1978719b958c7                           0.6s
 => => extracting sha256:6377b8b8fce7450cef3d5591a7ebdec7e0fdc1fc8c744a8e4f4c2815e2d92194                           0.3s
 => => sha256:649fb5f2ea6e78d08dd0cf4333770fb64d00f4842b5e32c500bb966a83b03c23 218.07MB / 218.07MB                  7.4s
 => => extracting sha256:6620614d7af044f2209279d91dd6d54490a48e6ab7c9e3565381a97dadd70a81                           2.5s
 => => sha256:21f0f0b272d724034b6fc235460caa781a32ee3cdcec666d1147f4a0526ea921 37.36MB / 37.36MB                    5.6s
 => => sha256:1eace69f893f60c1e99d47ed656116a27b1224ce9e4134e21187dcb351e352c4 508.28MB / 508.28MB                 14.1s
 => => extracting sha256:4f69ba4ede49a40e1b961f4dd6c57dea4254e44cf6120a406d323a8a0861358b                          12.0s
 => => extracting sha256:649fb5f2ea6e78d08dd0cf4333770fb64d00f4842b5e32c500bb966a83b03c23                          10.1s
 => => extracting sha256:21f0f0b272d724034b6fc235460caa781a32ee3cdcec666d1147f4a0526ea921                           0.5s
 => => extracting sha256:1eace69f893f60c1e99d47ed656116a27b1224ce9e4134e21187dcb351e352c4                           7.1s
 => [internal] load build context                                                                                   0.0s
 => => transferring context: 117.67kB                                                                               0.0s
 => [build 2/5] WORKDIR /src                                                                                        0.6s
 => [build 3/5] COPY . .                                                                                            0.1s
 => [build 4/5] RUN cargo +risc0 fetch --locked --target riscv32im-risc0-zkvm-elf --manifest-path methods/guest/C  9.2s
 => [build 5/5] RUN cargo +risc0 build --release --locked --target riscv32im-risc0-zkvm-elf --manifest-path meth  45.0s
 => [export 1/1] COPY --from=build /src/target/riscv32im-risc0-zkvm-elf/release /gost_kod_za_zk_dokaz               0.1s
 => exporting to client directory                                                                                   0.4s
 => => copying files 45.90MB                                                                                        0.4s
ELFs ready at:
ImageID: bd8abea046781c8cd8325650a7f3f558566e5864796fe7b804aec4a98e513e0e - "target/riscv-guest/riscv32im-risc0-zkvm-elf/docker/gost_kod_za_zk_dokaz/gost_kod_za_zk_dokaz"
Lukas-MBP:moj_projekt lukabibic$
```

Slika 5.7 Pokretanje naredbe cargo risczero build

Kao što se na gornjoj slici može vidjeti, naredba je generirala ELF datoteku i ispisala Image ID koji se dalje može koristit za Bonsai i verifikaciju.

5.5.2 Delegacija izvršavanja Bonsai-ju

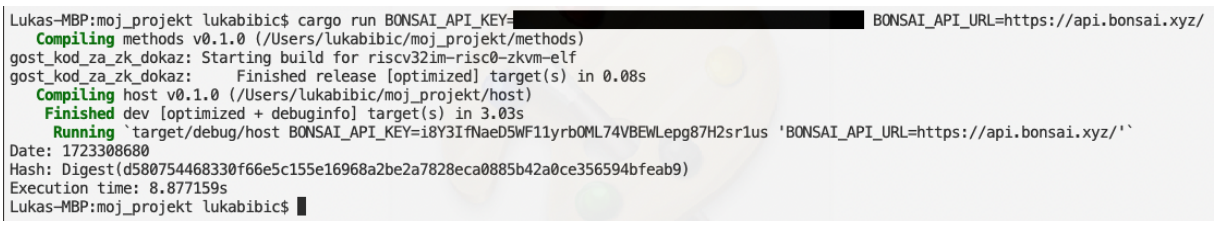
Delegiranje izvršavanja Bonsai-ju je moguće na više načina:

- Koristeći `cargo risczero` - ovo je rješenje tijekom samog razvijanja za eksperimentiranje ili za aplikacije koje nisu na blockchainu. Pokreće se postavljajući varijable okruženja tijekom korištenja cargo run naredbe.
- Koristeći Risc Zero Foundry Template - za integraciju s Ethereum-om.
- Bonsai SDK - programski paket za rad u Rust-u.
- Bonsai Rest API - najfleksibilniji, omogućava korištenje u mnogim sustavima, daje programu najveću slobodu u stvaranju svojih aplikacija koje delegiraju na Bonsai.

Bonsai je u trenutku pisanja ovog rada u razvojnom procesu i nije dostupan u produkciji javnosti. S toga je bilo potrebno kontaktirati razvojni tim kako bi se pridobio API ključ za testiranje.

Neke od ograničenja Bonsai API-ja su sljedeća:

- Broj CPU ciklusa - svaki korisnik ima ograničen broj CPU na raspolaganju, a troše se u istoj količini kao i u prethodno prikazanoj analizi programa.
- Dopušteni broj izvođenja - osim dodijeljenog broja ciklusa, korisnik ima pravo ograničen broj delegacija Bonsai-ju.
- Ograničenje ciklusa izvršitelja - najveći broj ciklusa koje izvršitelj može dokazati za pojedinačni dokaz



Slika 5.8 Delegiranje Bonsai-ju koristeći cargo run i varijable okruženja

Za jednostavan primjer, poput SHA-256, delegacija Bonsai-ju će rezultirati dužim izvršavanjem nego lokalno izvršavanje na M3 Pro Macbook Pro računalu s uključenim Metal API-jem.

```
    fn run_bonsai(input_data: Vec<u8>) -> Result<()> {
    let client = Client::from_env(risc0_zkvm::VERSION)?;

    let image_id = hex::encode(
        compute_image_id(GOST_KOD_ZA_ZK_DOKAZ_ELF)?);

    client.upload_img(&image_id,
        GOST_KOD_ZA_ZK_DOKAZ_ELF.to_vec())?;

    // Priprema ulaznih podataka i slanje ulaza Bonsai-ju
    let input_data = to_vec(&input_data).unwrap();
    let input_data = bytemuck::cast_slice(&input_data).to_vec();
    let input_id = client.upload_input(input_data)?;

    // List pretpostavki, ne koriste se u ovom primjeru,
    // samo potrebno za slanje
    let assumptions: Vec<String> = vec![];

    // Odabir načina pokretanja programa
    let execute_only = false;

    // Start a session running the prover
    let session = client.create_session(
        image_id, input_id, assumptions, execute_only)?;
    loop {
        let res = session.status(&client)?;
        if res.status == "RUNNING" {
            eprintln!(
                "Current status: {} - state: {}
                - continue polling...",
                res.status,
                res.state.unwrap_or_default()
```

```
            );
            std::thread::sleep(Duration::from_millis(100));
            continue;
        }
        if res.status == "SUCCEEDED" {
            // Preuzimanje potvrde koja sadrži izlaz
            let receipt_url = res
                .receipt_url
                .expect("API error,
                    missing receipt on completed session");

            let receipt_buf = client.download(&receipt_url)?;
            let receipt: Receipt = bincode::deserialize(
                &receipt_buf)?;
            receipt
                .verify(GOST_KOD_ZA_ZK_DOKAZ_ID)
                .expect("Receipt verification failed");
        } else {
            panic!(
                "Workflow exited: {} - | err: {}",
                res.status,
                res.error_msg.unwrap_or_default()
            );
        }

        break;
    }

    Ok(())
}
```

Programski kod 5.4 Delegiranje Bonsai-ju koristeći Bonsai SDK

Funkcija `run_bonsai` se poziva unutar main funkcije i zamjenjuje skoro cijeli

dio koda domaćina. Može se primijetiti da ulaz u funkciju zahtjeva `Vec<u8>` tip ulaza umjesto običnog String ili u64 tipa koji su se prethodno koristili. S toga se u main funkciji prethodno navedeni tipovi podataka kombiniraju u Vec<u8> kako je prikazano na idućem kodu.

```
    let string_to_hash: String = "Luka".to_string();

    let time_now_as_seconds = SystemTime::now()
        .duration_since(SystemTime::UNIX_EPOCH)
        .unwrap()
        .as_secs();

    let combined_data = format!("{}{}", string_to_hash,
        time_now_as_seconds);
    let input_data: Vec<u8> = combined_data.into_bytes();

    run_bonsai(input_data)?;
```

Programski kod 5.5 Prilagođavanje ulaznih varijabli na strani domaćina za Bonsai SDK

5.6 Testiranje vremenskih performansi često korištenih algoritama u kriptografiji

U ovom poglavlju će se analizirati performanse različitih često korištenih algoritama u kriptografiji kao što su BLAKE2b, BLAKE3, Keccak, SHA-2, ECDSA, ED25519 i te drugi nasumični zadaci poput Fibonnaci i Sudoku. Testiranje performansi ovih algoritama omogućava uvid u njihovu učinkovitost u kontekstu stvarnih primjena, kao i predviđanje kako će se ponašati u različitim scenarijima unutar kriptografskih sustava. Na primjer, rezultati pokazuju trajanje izvršenja, trajanje generiranja dokaza i ukupne cikluse, što može pomoći u optimizaciji izbora algoritma za specifične primjene u kriptografiji.

- SHA2: Široko korišten u mnogim sigurnosnim protokolima, SHA-2 se koristi za hashiranje i generiranje digitalnih potpisa.
- BLAKE2b: Alternativa SHA-2 s bržim hashiranjem i većom sigurnošću, koristi se za verifikaciju podataka i generiranje potpisa. Optimiziran za 64 bitne platforme.
- BLAKE3: Poboljšana verzija BLAKE2, optimizirana za brže hashiranje i široko korištena u modernim aplikacijama. Brži je od MD5, SHA-1, SHA-2 i SHA-3.
- Keccak: Hash funkcija koja je temelj SHA-3 standarda, koristi se za sigurnost podataka i kriptografske potpise. Ima istu duljinu hash-a kao SHA-2, ali različiti su.
- ECDSA: Algoritam za digitalne potpise, koji se koristi u mnogim sigurnosnim protokolima, uključujući Bitcoin.
- Ed25519: Algoritam za digitalne potpise s visokom sigurnošću i brzim performansama, koristi se u sigurnosnim protokolima poput SSH-a.
- fibonacci: Računa Fibonaccijev niz za zadani broj.
- membership: Izračunava dokaz o članstvu za zadani autentificirani put od lista do korijena Merkle-ovog stabla, koristeći SHA-256 kao hash funkciju.
- sudoku: Provjerava rješenje zadanog Sudoku zadatka.

Način na koji se generiraju ove statistike je da se kombinira Rust i bash kako bi se odredili poslovi te komande s kojima se oni pokreću. Iduće slike prikazuju tablicu performansi različitih kriptografskih algoritama koji su izvršeni na Apple M3 Pro računalu, te koje će se analizirati u odnosu na njihovo izvršavanje bez da se stvaraju dokazi nultog znanja.

Metal ubrzanje na Apple M3 Pro

Algoritam	Veličina	Brzina	Trajanje izvršavanja	Trajanje dokazivanja	Ukupno trajanje	Trajanje provjere	Ukupno ciklusa	Korisnički ciklusi	Izlazni bajtovi	Bajtovi dokaza
big_blake2b-1024	1024	203.07ns	8.42ms	5.03s	5.04s	11.38ms	262144	116716	32	217.45KB
big_blake2b-2048	2048	249.95ns	10.80ms	8.18s	8.19s	11.59ms	524288	230436	32	217.45KB
big_blake2b-4096	4096	287.27ns	18.81ms	14.24s	14.26s	11.36ms	1048576	457876	32	217.45KB
big_blake2b-8192	8192	586.68ns	36.09ms	13.93s	13.96s	11.43ms	1048576	912756	32	217.45KB
big_blake3-1024	1024	210.37ns	6.23ms	4.86s	4.87s	11.67ms	262144	78987	32	217.45KB
big_blake3-2048	2048	417.43ns	9.11ms	4.90s	4.91s	11.40ms	262144	157724	32	217.45KB
big_blake3-4096	4096	533.08ns	14.61ms	7.67s	7.68s	11.75ms	524288	315732	32	217.45KB
big_blake3-8192	8192	567.88ns	25.00ms	14.40s	14.43s	11.41ms	1048576	631748	32	217.45KB
big_keccak-1024	1024	134.08ns	8.59ms	7.63s	7.64s	11.58ms	524288	248899	32	217.45KB
big_keccak-2048	2048	143.59ns	14.30ms	14.25s	14.26s	11.42ms	1048576	495000	32	217.45KB
big_keccak-4096	4096	281.16ns	24.13ms	14.54s	14.57s	11.52ms	1048576	961489	32	217.45KB
big_keccak-8192	8192	260.98ns	45.17ms	31.34s	31.39s	11.61ms	2097152	1894454	32	217.45KB
big_sha2-1024	1024	289.39ns	5.10ms	3.53s	3.54s	11.49ms	131072	43649	32	217.45KB
big_sha2-2048	2048	420.00ns	7.19ms	4.87s	4.88s	11.41ms	262144	84673	32	217.45KB
big_sha2-4096	4096	834.15ns	11.83ms	4.90s	4.91s	11.55ms	262144	166721	32	217.45KB
big_sha2-8192	8192	1.07µs	21.10ms	7.66s	7.68s	11.41ms	524288	330817	32	217.45KB
ecdsa_verify	1	0.07ns	30.50ms	14.66s	14.69s	11.36ms	1048576	708000	408	217.45KB
ed25519_verify	1	0.07ns	24.12ms	14.30s	14.32s	11.50ms	1048576	534300	400	217.45KB
fibonacci-10	10	3.67ns	3.14ms	2.72s	2.73s	11.60ms	65536	2220	8	217.45KB
fibonacci-50	50	18.45ns	3.22ms	2.71s	2.71s	11.51ms	65536	2351	8	217.45KB
fibonacci-90	90	32.92ns	2.83ms	2.73s	2.73s	11.50ms	65536	2433	8	217.45KB
iter_blake2b-1	1	0.29ns	3.51ms	3.46s	3.46s	11.46ms	131072	12022	32	217.45KB
iter_blake2b-10	10	2.06ns	4.94ms	4.84s	4.85s	11.39ms	262144	99453	32	217.45KB
iter_blake2b-100	100	5.12ns	24.19ms	19.49s	19.51s	11.38ms	1114112	973263	32	217.45KB
iter_blake3-1	1	0.36ns	3.20ms	2.76s	2.76s	11.39ms	65536	5166	32	217.45KB
iter_blake3-10	10	2.93ns	3.29ms	3.41s	3.42s	11.61ms	131072	28541	32	217.45KB
iter_blake3-100	100	13.07ns	7.10ms	7.65s	7.65s	11.44ms	524288	262091	32	217.45KB
iter_keccak-1	1	0.29ns	3.91ms	3.43s	3.43s	11.53ms	131072	28338	32	217.45KB
iter_keccak-10	10	1.31ns	7.61ms	7.65s	7.66s	11.71ms	524288	263951	32	217.45KB
iter_keccak-100	100	2.14ns	46.41ms	46.79s	46.84s	11.45ms	3145728	2620061	32	217.45KB
iter_sha2-1	1	0.36ns	3.45ms	2.74s	2.74s	11.44ms	65536	2825	32	217.45KB
iter_sha2-10	10	3.67ns	3.15ms	2.72s	2.72s	11.35ms	65536	6057	32	217.45KB
iter_sha2-100	100	29.15ns	3.51ms	3.43s	3.43s	11.56ms	131072	38367	32	217.45KB
membership-10	10	3.64ns	3.62ms	2.74s	2.75s	11.41ms	65536	13449	64	217.45KB
membership-20	20	5.83ns	4.33ms	3.43s	3.43s	11.50ms	131072	22819	64	217.45KB
sudoku	1	0.37ns	3.11ms	2.73s	2.73s	11.89ms	65536	9640	32	217.45KB

Slika 5.9 Performanse različitih algoritama na M3 Pro koristeći Metal ubrzanje

Ako se ne koristi Metal ubrzavanje, dobivaju se znantno lošiji rezultati za sve algoritme i zadatke.

Samo CPU na Apple M3 Pro

Algoritam	Veličina	Brzina	Trajanje izvršavanja	Trajanje dokazivanja	Ukupno trajanje	Trajanje provjere	Ukupno ciklusa	Korisnički ciklusi	Izlazni bajtovi	Bajtovi dokaza
big_blake2b-1024	1024	42.37ns	7.99ms	24.16s	24.17s	11.36ms	262144	116716	32	217.45KB
big_blake2b-2048	2048	52.80ns	11.32ms	38.78s	38.79s	11.35ms	524288	230436	32	217.45KB
big_blake2b-4096	4096	58.63ns	18.49ms	1.16min	1.16min	11.33ms	1048576	457876	32	217.45KB
big_blake2b-8192	8192	116.22ns	34.12ms	1.17min	1.17min	11.37ms	1048576	912756	32	217.45KB
big_blake3-1024	1024	42.69ns	5.38ms	23.98s	23.98s	11.36ms	262144	78987	32	217.45KB
big_blake3-2048	2048	85.14ns	8.55ms	24.05s	24.05s	11.34ms	262144	157724	32	217.45KB
big_blake3-4096	4096	104.66ns	13.51ms	39.12s	39.14s	11.37ms	524288	315732	32	217.45KB
big_blake3-8192	8192	118.58ns	23.70ms	1.15min	1.15min	11.33ms	1048576	631748	32	217.45KB
big_keccak-1024	1024	26.19ns	8.74ms	39.09s	39.10s	11.41ms	524288	248899	32	217.45KB
big_keccak-2048	2048	28.91ns	13.56ms	1.18min	1.18min	11.38ms	1048576	495000	32	217.45KB
big_keccak-4096	4096	58.23ns	24.18ms	1.17min	1.17min	11.33ms	1048576	961489	32	217.45KB
big_keccak-8192	8192	54.51ns	43.92ms	2.50min	2.50min	11.45ms	2097152	1894454	32	217.45KB
big_sha2-1024	1024	61.89ns	4.91ms	16.54s	16.55s	11.35ms	131072	43649	32	217.45KB
big_sha2-2048	2048	85.37ns	6.76ms	23.98s	23.99s	11.34ms	262144	84673	32	217.45KB
big_sha2-4096	4096	170.53ns	12.75ms	24.01s	24.02s	11.39ms	262144	166721	32	217.45KB
big_sha2-8192	8192	207.96ns	19.54ms	39.37s	39.39s	11.34ms	524288	330817	32	217.45KB
ecdsa_verify	1	0.01ns	29.43ms	1.18min	1.18min	11.33ms	1048576	708000	408	217.45KB
ed25519_verify	1	0.01ns	23.75ms	1.17min	1.17min	11.37ms	1048576	540854	400	217.45KB
fibonacci-10	10	0.78ns	2.91ms	12.85s	12.86s	11.58ms	65536	2220	8	217.45KB
fibonacci-50	50	3.89ns	2.85ms	12.86s	12.87s	11.33ms	65536	2351	8	217.45KB
fibonacci-90	90	6.97ns	3.09ms	12.91s	12.91s	11.37ms	65536	2433	8	217.45KB
iter_blake2b-1	1	0.06ns	3.22ms	16.55s	16.56s	11.37ms	131072	12022	32	217.45KB
iter_blake2b-10	10	0.42ns	5.27ms	23.98s	23.98s	11.33ms	262144	99453	32	217.45KB
iter_blake2b-100	100	1.07ns	24.24ms	1.55min	1.55min	11.39ms	1114112	973263	32	217.45KB
iter_blake3-1	1	0.08ns	2.83ms	12.89s	12.89s	11.60ms	65536	5166	32	217.45KB
iter_blake3-10	10	0.60ns	3.40ms	16.63s	16.63s	11.39ms	131072	28541	32	217.45KB
iter_blake3-100	100	2.51ns	7.01ms	39.76s	39.76s	11.33ms	524288	262091	32	217.45KB
iter_keccak-1	1	0.06ns	3.33ms	16.73s	16.74s	11.37ms	131072	28338	32	217.45KB
iter_keccak-10	10	0.26ns	7.78ms	38.60s	38.61s	11.34ms	524288	263951	32	217.45KB
iter_keccak-100	100	0.43ns	46.07ms	3.86min	3.86min	11.34ms	3145728	2620061	32	217.45KB
iter_sha2-1	1	0.07ns	2.76ms	13.61s	13.61s	11.36ms	65536	2825	32	217.45KB
iter_sha2-10	10	0.77ns	3.04ms	13.01s	13.01s	11.34ms	65536	6057	32	217.45KB
iter_sha2-100	100	5.92ns	3.58ms	16.89s	16.89s	11.42ms	131072	38367	32	217.45KB
membership-10	10	0.77ns	3.27ms	13.03s	13.04s	11.45ms	65536	13449	64	217.45KB
membership-20	20	1.20ns	3.01ms	16.62s	16.62s	11.40ms	131072	22819	64	217.45KB
sudoku	1	0.08ns	3.07ms	12.86s	12.86s	11.49ms	65536	9640	32	217.45KB

Slika 5.10 Performanse različitih algoritama na M3 Pro koristeći samo CPU

Faktori usporenja algoritama koristeći dokaze nultog znanja

Algoritam	Veličina	Ukupno trajanje bez ZKP (ms)	Ukupno trajanje s ZKP (s)	Faktor usporenja
big_blake2b-1024	1024	8.77	5.04	574.45
big_blake2b-2048	2048	10.43	8.19	785.33
big_blake2b-4096	4096	17.96	14.26	793.96
big_blake2b-8192	8192	33.19	13.96	420.70
big_blake3-1024	1024	5.39	4.87	903.52
big_blake3-2048	2048	8.15	4.91	602.45
big_blake3-4096	4096	14.59	7.68	526.46
big_blake3-8192	8192	26.04	14.43	554.18
big_keccak-1024	1024	7.91	7.64	965.85
big_keccak-2048	2048	14.48	14.26	985.50
big_keccak-4096	4096	24.49	14.57	595.07
big_keccak-8192	8192	43.52	31.39	721.29
big_sha2-1024	1024	4.49	3.54	788.42
big_sha2-2048	2048	6.60	4.88	739.39
big_sha2-4096	4096	11.49	4.91	427.27
big_sha2-8192	8192	19.14	7.68	401.26
ecdsa_verify	1	32.76	14.69	448.41
ed25519_verify	1	24.08	14.32	594.71
fibonacci-10	10	2.76	2.73	989.13
fibonacci-50	50	2.34	2.71	1158.12
fibonacci-90	90	2.46	2.73	1109.76
iter_blake2b-1	1	2.65	3.46	1305.66
iter_blake2b-10	10	4.46	4.85	1087.22
iter_blake2b-100	100	23.83	19.51	818.51
iter_blake3-1	1	2.59	2.76	1065.63
iter_blake3-10	10	3.04	3.42	1125.00
iter_blake3-100	100	6.67	7.65	1147.68
iter_keccak-1	1	2.71	3.43	1266.79
iter_keccak-10	10	6.81	7.66	1125.55
iter_keccak-100	100	46.44	46.84	1008.62
iter_sha2-1	1	2.97	2.74	922.90
iter_sha2-10	10	2.40	2.72	1133.33
iter_sha2-100	100	3.80	3.43	902.63
membership-10	10	9.01	2.75	305.22
membership-20	20	3.34	3.43	1026.95
sudoku	1	2.66	2.73	1026.32

Slika 5.11 Performanse različitih algoritama bez stvaranja dokaza nultog znanja

5.6.1 Analiza performansi

Za analizu će se gledati tablica koja koristi Metal GPU ubrzanje, jer daje puno bolje rezultate i otkriva potencijal Risc Zero zkVM-a.

Trajanje izvršavanja

Fokusirajući se na stupac **"Trajanje izvršavanja"** moguće je primijetiti iduće činjenice:

- Svi algoritmi se izvršavaju relativno brzo, u redu milisekunda
- Složeniji algoritmi poput `edcsa_verify` i `ed25519_verify` znatno dulja vremena izvršavanja nego jednostavniji algoritmi poput fibonacci-10.
- U konvencionalnom okruženju, izvan zkVM-a, isti kriptografski algoritmi obično imaju značajno kraća vremena izvršavanja. Razlog tome je direktni pristup CPU-u i izostanak dodatnih slojeva sigurnosti i verifikacije koji su karakteristični za zkVM. Na primjer, ecdsa_verify se može izvesti u nanosekundama na modernom procesoru, dok unutar zkVM-a traje milisekundama. Povećanje vremena je cijena koja se plaća za dodatne sigurnosne benefite i transparentnost koje pruža zkVM. [16]

Znatno veća vremena izvršavanja ne moraju nužno predstavljati problem u praksi jer čak i vremena unutar milisekunda mogu biti zadovoljavajuća ovisno o aplikaciji u kojoj će se koristiti.

Ipak, kao što se može vidjeti postoje i drugi stupci koje je potrebno analizirati, te stupac "Trajanje izvršavanja" nije dovoljan da se dovedu konkretni zaključci o zrelosti Risc Zero zkVM-a za opću primjenu nultog znanja.

Trajanje dokazivanja

Puno zanimljiviji stupac za analizu je stupac **"Trajanje dokazivanja"**. On ukazuje na vrijeme potrebno da se stvori dokaz za izvršen izračun. Bez dokaza, samo vrijeme izvršavanja (stupac "Trajanje izvršavanja") je nebitno, jer korisnik ne može imati povjerenje u rezultat.

Gledajući ovaj stupac, jasno se primjećuje glavna mana Risc Zero zkVM-a. Na primjer, za algoritme poput ecdsa_verify i ed25519_verify, ukupno trajanje generiranja dokaza je prilično dugo, što može ograničiti njihovu korisnost u aplikacijama koje zahtijevaju brze kriptografske operacije, kao što su digitalni potpisi ili brza

autentifikacija. Generiranje dokaza za ova dva algoritma iznosi oko 14.5 sekundi, što je čak 10^9 puta duže izvođenje nego vrijeme potrebno za sami algoritam ako se pokreću izvan zkVM-a. Navedeni algoritmi imaju duže vrijeme generiranja dokaza, što može predstavljati problem u sustavima koji zahtijevaju visoku propusnost i nisku latenciju.

S druge strane, algoritmi poput Fibonacci i sudoku imaju relativno kratko ukupno trajanje i vrijeme generiranja dokaza, što ih čini prikladnima za aplikacije gdje su brzina i efikasnost prioriteti, ali gdje također postoji potreba za kriptografskom sigurnošću, poput testiranja integriteta ili jednostavnih matematičkih verifikacija.

Analizirajući algoritme sha2 i keccak, vidimo da, iako su složeniji i zahtijevaju više ciklusa nego Fibbonaci i sudoku, njihovo ukupno trajanje nije toliko ekstremno kao kod nekih drugih algoritama, što ih čini pogodnim za korištenje u sigurnosnim aplikacijama koje zahtijevaju balans između brzine i sigurnosti.

Ostali stupci i zaključak

Od ostalih stupaca, korisno je i analizirati `"Korisnički ciklusi"` koji prikazuje točan broj ciklusa za izvršavanje samih algoritama (manji od stupca `"Ukupno ciklusa"` jer ne uračunava cikluse potrebne za postavljanje konteksta). Pregled broja ciklusa može pomoći u planiranju aplikacija prije početka razvijanja. Programeri koji imaju općeniti uvid u tehnologije koje će koristiti mogu s ovakvom tablicom unaprijed odlučiti je li Risc Zero zkVM dobro rješenje te mogu li koristit Bonsai za unaprjeđenje performansi svojeg programa pazeći na troškove koje bi takvo rješenje donijelo.

Također, moguće je vidjeti da je veličina dokaza i vrijeme za provjeru dokaza svih algoritama približno jednako i vrlo nisko. Vrijeme od 11.5ms za provjeru je vrlo brzo te ključno za skaliranje sustava koji koriste zkVM.

Gledajući tablicu faktora usporenja moguće je uočiti da kriptografske hash funkcije (`big_blake2b`, `big_blake3`, `big_keccak`, `big_sha2`) pokazuju prosječno usporenje od približno 650 puta kada se koriste dokazi s nultim znanjem (ZKP).

Provjera digitalnog potpisa (`ecdsa_verify`, `ed25519_verify`) ima prosječno usporenje od oko 522 puta u prisutnosti ZKP-a.

Iterativne hash funkcije (`iter_blake2b`, `iter_blake3`, `iter_keccak`, `iter_sha2`) doživljavaju prosječno usporenje od približno 1076 puta kada se implementiraju s ZKP-om.

Izračun Fibonaccijevih brojeva usporava se u prosjeku oko 1086 puta kada se koristi ZKP.

Dokazi članstva pokazuju prosječno usporenje od približno 666 puta uz korištenje ZKP-a.

Rješavanje sudoku-a usporeno je za oko 1026 puta kada se integrira s ZKP-om.

Na temelju performansi specifičnih algoritama poput provjera digitalnih potpisa (ecdsa_verify, ed25519_verify) i iterativnih hash funkcija (iter_blake2b, iter_blake3, iter_keccak, iter_sha2), može se zaključiti da Risc Zero zkVM nije pogodan za aplikacije koje zahtijevaju brzu obradu i nisku latenciju. Ovi algoritmi su ključni u aplikacijama kao što su online transakcije, autentifikacija korisnika u stvarnom vremenu, kriptovalute i sigurni komunikacijski protokoli. Zbog značajnog usporenja prilikom generiranja dokaza (usporenje od preko 500 do 1000 puta), korištenje Risc Zero zkVM-a u ovim kontekstima nije praktično.

S druge strane, za aplikacije gdje su sigurnost i povjerenje prioritet, a brzina nije kritičan faktor, RISC Zero zkVM može biti prikladan izbor. To uključuje sustave za dugoročnu pohranu podataka, blockchain glasovanje, kriptografske protokole koji zahtijevaju visok stupanj sigurnosti i aplikacije gdje je potrebno dokazati ispravnost operacija bez otkrivanja podataka. U takvim slučajevima, algoritmi poput kriptografskih hash funkcija (big_blake2b, big_blake3, big_keccak, big_sha2) i dokaza članstva pokazuju prihvatljiva usporenja, a dodatno vrijeme generiranja dokaza opravdano je zbog povećane sigurnosti i mogućnosti rada u nepovjerljivim okruženjima.

Poglavlje 6

Razni programi unutar Risc Zero zkVM

U ovom poglavlju istražit će se različiti programi unutar RISC Zero zkVM okruženja. Za svaki od ovih programa koristit će se alati za analizu performansi, prethodno opisani u radu. Na temelju rezultata analize, procijenit će se prikladnost RISC Zero zkVM-a za različite vrste aplikacija, uzimajući u obzir faktore poput vremena izvođenja i generiranja dokaza.

6.1 zk šah-mat

Ideja iza ovog programa je dokazati da u nekoj šahovskoj poziciji postoji šah-mat bez da se otkrije potez potreban za njega.

6.1.1 Pristup rješavanju

Program koristi shakmaty paket, s kojim je moguće prevesti šahovsku notaciju pozicije na ploči, provjeriti njenu valjanost i provjeriti svu potrebnu logiku iza odigranog poteza.

Domaćin prosljeđuje stanje šahovske ploče koristeći Forsyth-Edwards Notation (FEN) te odigrani potez koristeći algebarsku notaciju, npr. Qxf7 označava potez u

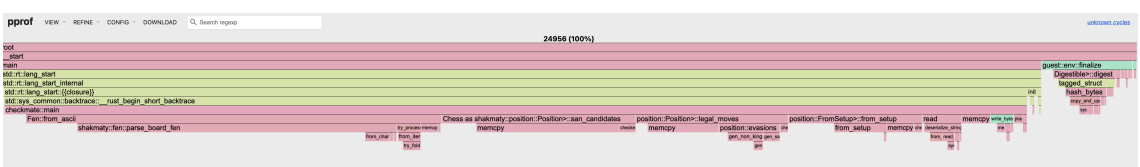
kojem kraljica (Q) uzima (x) figuru na polju f7. Qxf7 bi se čitalo kao "Kraljica uzima na f7". Gost kod na temelju ovih vrijednosti provjeri stanje šahovske partije na način da primjeni odigrani potez na zadanu poziciju te ustanovi je li odigran potez rezultirao šah-matom. Ako je, vratit će se Journal objekt s vizualnom reprezentacijom originalne ploče u kojoj su figure predstavljene slovima, a prazna polja točkama. Ako dobivena pozicija nije valjana ili ne rezultira matom, gost program se zaustavlja i ne vraća Journal objekt.

```
Lukas-MBP:chess lukabibic$ RUST_LOG=info cargo run -F metal -- --board "r1bqkb1r/pppp1ppp/2n2n2/4p2Q/2B1P3/8/PPPP1PPP/RNB1K1NR w KQkq - 4 4" --mv "Qxf7"
   Compiling chess-methods v0.1.0 (/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/chess/methods)
checkmate: Starting build for riscv32im-risc0-zkvm-elfs-methods(build)
checkmate:     Finished release [optimized] target(s) in 0.08s
   Compiling chess v0.1.0 (/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/chess)
    Finished dev [optimized + debuginfo] target(s) in 2.42s
     Running `/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/target/debug/chess --board 'r1bqkb1r/pppp1ppp/2n2n2/4p2Q/2B1P3/8/PPPP1PPP/RNB1K1NR w KQkq - 4 4' --mv Qxf7`
Checking for checkmate after move: Qxf7
There is a checkmate for White in this position:
r . b q k b . r
p p p p . p p p
. . n . . n . .
. . . . p . . Q
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R

Execution time: 3.239179s
```

Slika 6.1 Prikaz najpoznatije mat pozicije "Školski mat"

6.1.2 Profiliranje



Slika 6.2 Graf plamena za zadatak "ZK šah-mat"

Graf plamena jasno prikazuje broj ciklusa za izvršavanje ovog programa te iznosi 24956. Bitno je za primijetiti da za većinu tih ciklusa (90%) su odgovorne operacije vezane uz samu logiku oko stvaranja i provjere šahovske pozicije te logike oko odigranih poteza.

6.1.3 Zaključak o aplikaciji

S obzirom na to da su ulazi relativno maleni, zbog toga što se cijela šahovska pozicija može predstaviti FEN notacijom, nema velikih gubitaka za čitanje ulaza. Rezultat je ohrabravajuć u kontekstu praktičnosti rješavanja ovakvog problema putem zkVM-a. Mogući sustavi u kakvim bi se ovakve aplikacije mogle koristit su sustavi koji nagrađuju korisnike koji prvi stvore i objave dokaz, kojeg je moguće stvoriti samo ako korisnik stvarno zna potez za matirajuću poziciju (ili nekakav drugi problem). Bilo kakvo diranje izvornog koda bi rezultiralo neuspješnom verifikacijom dokaza zbog Image ID-ja koji se stvara deterministički. Ipak treba napomenuti da 3.3 sekunde za izvršavanje ovakvog programa znatno duže nego što bi to bilo bez da se generira ZKP, te ako ključno imati "real-time" performanse, ovakvo rješenje za to ne bi bilo prikladno. Uspoređujući istim programom koji ne generira ZKP, vrijeme izvršavanja zadatka iznosi 57 milisekundi.

6.2 Gdje je Waldo?

"Gdje je Waldo?" je Britanski serijal knjiga koji se sastoji od mnogih ilustracija. Serijal je osmišljen da bude izazovna igra za čitatelje u kojoj je cilj pronaći Walda, koji je poznat po svojoj crveno-bijeloj prugastoj majici, naočalama i šivanoj kapi.

Gdje je Waldo se često koristi kao analogija za zero-knowledge dokaze. Konkretno, postoji vizualna interpretacija gdje, ako se uzme slika Gdje je Waldo i pokrije je velikim komadom kartona s malim prorezom koji pokazuje samo Walda, može se dokazati da je poznato gdje se nalazi, a da se pritom ne otkrije točna lokacija.

6.2.1 Pristup rješavanju

Pristup rješavanju ovog zadatke je sličan analogiji. Cilj je uzeti cijelu sliku iz igre "Gdje je Waldo?" i izrezati samo Walda bez da se provjeritelju daju koordinate. Taj proces izrezivanja se događa unutar ZKVM gostujućeg programa.

Hashiranje cijele slike bi bilo skupo u ZKVM gostu zbog veličine slike. Zbog toga se koristi Merkelizacija, proces organiziranja podataka u Merkle stablo, koje je

strukturirano kao binarno stablo gdje su listovi hash vrijednosti pojedinačnih dijelova podataka (npr. dijelovi slike), a svaka razina iznad se stvara kombiniranjem i hashiranjem vrijednosti iz razina ispod. Rezultat je učinkovito i sigurno dokazivanje da je određeni dio podataka (npr. dio slike) dio originalnog skupa podataka bez potrebe za hashiranjem cijelog skupa. Koristi se u kriptografiji za osiguravanje integriteta podataka.

Tijekom izvršavanja gost i domaćin komuniciraju putem syscall-a za dinamičko dohvaćanje dijelova slike. Syscall naredbe omogućuju gostu da traži dijelove slike po potrebi, što se koristi za izrezivanje Walda. Navedeni pristup rezulitara fleksibilnijim i čitljivijim kodom.

Za manipulaciju slikom koristi se popularna image biblioteka. Implementacijom image::GenericImageView na ImageOracle, gost može koristiti mnoge operacije nad slikama unutar zkVM-a, kao što su izrezivanje (eng. crop) i maskiranje. Moguće su i druge operacije kao što su zamućenje (eng. blur) ili promjena razlučivosti slike unutar zkVM-a.

Za pokretanje programa, odnosno dokazivanje, potrebno je proslijediti sliku na kojoj se traži Waldo, te opcionalno masku kako bi se prekrili svi dijelovi rezultirajućeg izreska koji nisu sam lik Walda (zbog boljeg sakrivanja lokacija Walda provjeritelju). Pošto sam program ne pronalazi Walda, nego stvara isječak koji sadrži njegov lik, potrebno je proslijediti i koordinate gornjeg lijevog kuta pravokutnog isječka u kojem se nalazi te veličinu isječka. Primjer pokretanja programa za dokazivanje:

```
cargo run -F metal -release -bin prove - -i waldo.webp
-x 1150 -y 291 -width 58 -height 70 -m waldo_mask.png
```

Spremanjem Receipt objekta u binarnu datoteku omogućava njeno dijeljenje i provjeravanje dokaza ako je poznat Image ID, što bi trebalo bit poznato ukoliko su se pratile upute prethodno opisane u radu.

Pokretanje provjere dokaza se izvodi na sljedeći način i rezultira odrezanim isječkom Walda i crnom pozadinom ako je korištena maska:

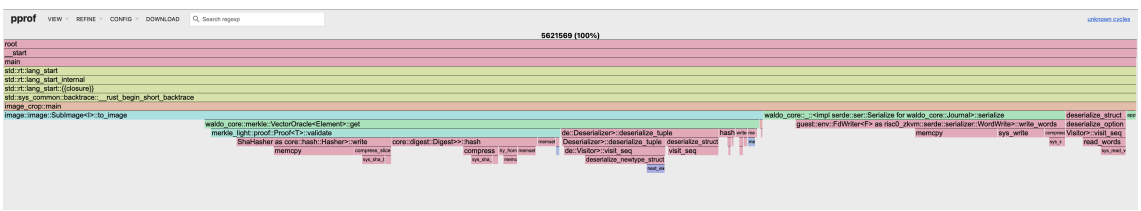
```
cargo run -release -bin verify - -i waldo.webp -r receipt.bin
```

```
Lukas-MBP:waldo lukabibic$ cargo run -r -F metal --bin verify -- -i waldo.webp -r receipt.bin
   Compiling waldo-methods v0.1.0 (/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/waldo/methods)
image_crop: Starting build for riscv32im-risc0-zkvm-elf-methods(build)
image_crop:     Finished release [optimized] target(s) in 0.08s
   Compiling waldo v0.1.0 (/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/waldo)
    Finished release [optimized + debuginfo] target(s) in 42.12s
     Running `/Users/lukabibic/Desktop/Risc_Zero/risc0/examples/target/release/verify -i waldo.webp -r receipt.bin`
Read image at waldo.webp with size: 2800 x 1760
Created Merkle tree from image with root Digest(631d3ed302c665a793fe4da7fedc0354016bd5946d2e1637fc813c409e39b6a8)
Verified receipt with 58x70 subimage
Saved Waldo cutout to ./waldo_cutout.png
```

```
Prover knows where this cutout is in the given image.
Do you recognize this Waldo?
Execution time: 403.348ms
```

Slika 6.3 Dokazano znanje lokacije Walda bez da je otkrivena

6.2.2 Profiliranje



Slika 6.4 Graf plamena za zadatak "Gdje je Waldo?"

Koristeći Pprof alat na način već opisan u radu, dobiva se gore prikazana slika. Jasno se može primijetiti da je riječ o puno složenijoj aplikaciji na temelju samog broja ciklusa (u prvom ZK šah-mat primjeru, broj ciklusa je iznosio oko 25 tisuća, a ovdje čak preko 5.6 milijuna ciklusa).

Funkcija `SubImage<l>::to_image` je zaslužna za čak 67% ukupnog broja ciklusa. Funkcija je dio biblioteke `image` (ručno kreirana na slični način kao što je bila shacore biblioteka koja je sadržavala strukturu DateAndHash), koja pretvara izrezani dio slike (SubImage) natrag u cijelu sliku (Image).

Za funkciju je bilo potrebno stvoriti SubImage objekt koji je nastao idućim načinom:

- Kreira se ImageOracle, koji omogućuje gost kodu siguran pristup dijelovima slike koji su pohranjeni na sustavu domaćina. ImageOracle je baziran na Merkle stablu, što znači da svaki put kad gost kod zatraži dio slike, dobiva se odgovarajući kriptografski dokaz (Merkle putanja) koji potvrđuje autentičnost tog dijela.
- Koristi se funkcija crop_imm iz imageops modula kako bi izrezao pravokutni dio slike prema zadanim koordinatama
  `crop_location` i dimenzijama `crop_dimensions`. `crop_imm` funkcija koristi oracle za dohvaćanje potrebnih dijelova slike, pri čemu svaka operacija dohvaćanja provjerava autentičnost podataka koristeći Merkle dokaz.

Samo 1% ciklusa je odgovorno za maskiranje, vidljivo kao apply na krajnjoj desnoj strani grafa plamena. Operacija maskiranja slike uključuje primjenu ImageMask objekta na izrezani dio slike (subimage) kako bi se sakrili određeni dijelovi slike. Operacija maskiranja događa se unutar match izraza u gost kodu kada je maska prisutna.

Za ostalih 33%, približno 1.85 milijuna ciklusa gost koda, zaslužne su funkcije za serijalizaciju (stvaranje Journal objekta) i deserijalizaciju (čitanje ulaznih vrijednosti).

6.2.3 Zaključak o aplikaciji

Hashiranje i druge kriptografske operacije, poput onih povezanih s Merkle stablom, troše puno CPU ciklusa unutar zkVM-a. Zato se prvotno hashiranje slike provodi na strani domaćina te prosljeđuje gost kodu kao ulazni parametar. Rezanje slike i pretvaranje hash-a u sliku se isto pokazalo kao vrlo skupa operacija, te u ovom programu iznosi 67% potrošenih resursa. Serijalizacija i deserijalizacija podataka isto znatno pridonosi vremenu izvršavanja i broju cpu ciklusa. Povećanjem podataka (ulaznih ili izlaznih) bi ove procese učinili još računalno zahtjevnijima. Ukupno izvršavanje stvaranja dokaza za ovu aplikaciju traje oko 90 sekundi koristeći Metal GPU ubrzanje. Vrijeme potrebno za provjeru dokaza je puno prihvatljive te iznosi 400ms. Ukoliko se odluči ne stvarati ZKP, izvršavanje istog programa traje 498.5 milisekundi. Iz toga se može izračunati da se radi o povećanju izvršenja od 180 puta. Iako je vrijeme stvaranja dokaza značajno, prednosti koje pruža zkVM u smislu sigurnosti i privatnosti mogu opravdati ovo vrijeme, posebno u aplikacijama gdje je sigurnost od najveće važnosti. S toga, ulaganje u ovu tehnologiju može biti isplativo u kontekstu specifičnih sigurnosno osjetljivih aplikacija. Vrlo kratko vrijeme provjere dokaza opravdava ideju zkVM, a to je da provjera dokaza zahtjeva manje posla (u ovom kontekstu, računalnih resursa) nego samo dokazivanje.

6.3 Dokaz izvršavanja ML modela istreniranog u Pythonu

U ovom primjeru cilj je pokrenuti zaključivanje (eng. inference) modela treniranog za predviđanje `gas fee u gwei po transkaciji` za idući Ethereum blok, bez otkrivanja ulaznih podataka ili parametara modela.

6.3.1 Pristup rješavanju

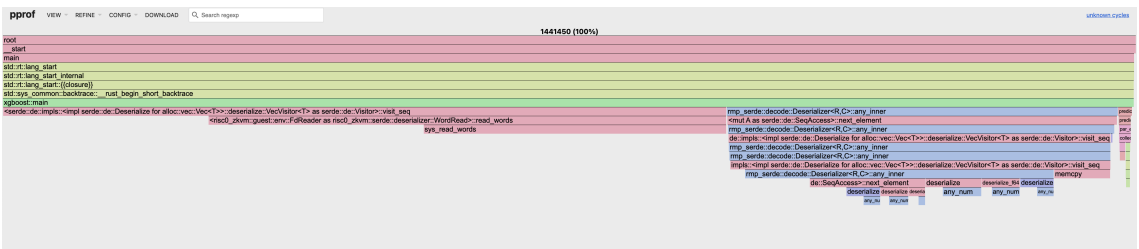
Prvo, model se trenira koristeći Python okruženje, posebno Jupyter Notebook. Za treniranje se koristi XGBoost algoritam implementiran u Forust paketu, koji je čista Rust implementacija s Python vezama. Omogućava programerima da iskoriste sve

dostupne alate u Python ekosustavu za prikupljanje podataka, postavljanje značajki i druge zadatke vezane za obradu podataka.

Nakon što se model istrenira, sprema se u izlaznu JSON datoteku. JSON format omogućava jednostavan prijenos modela u RISC Zero zkVM okruženje, gdje će se provoditi zaključivanje.

Kod domaćina je odgovoran za učitavanje modela iz JSON datoteke koristeći serde_json biblioteku te GradientBooster iz forust_ml biblioteke. Model se zatim serijalizira u polje byteova kako bi se prenio gostu. Domaćin je također odgovoran za prosljeđivanje broja bloka (broj koji identificira specifičan blok unutar Ethereum blockchain-a) te broja transakcija u tom bloku. Ove dvije vrijednosti se koriste kao ulazi za predviđanje gas fee-ja u tom bloku.

6.3.2 Profiliranje



Slika 6.5 Graf plamena za zadatak "Dokaz izvršavanja ML modela"

Gledajući gornji graf plamena za ovaj zadatak, vrlo je jasno da deserializacija predstavlja velike probleme unutar zkVM-a. Iako se ne radi cijelo treniranje modela nego samo učitanje na temelju JSON datoteke napravljene koristeći Python, broj ciklusa iznosi preko 1.4 milijuna. Deserijalizacija se provodi kod pretvaranja polja byteova u trenirani model. Što se tiče samog predviđanja (eng. inference), on se izvodi funkcijom `GradientBooster::predict` te iznosi samo 1.3% ukupnog broja ciklusa gost programa. S obzirom na njegov mali udio u broju ciklusa, teško ga je primjetiti na grafu.

6.3.3 Zaključak o aplikaciji

Kod se sastoji od nekoliko ključnih koraka: učitavanje podataka, deserijalizacija modela, formatiranje podataka u Matrix objekt, izvršavanje predikcije i zapisivanje rezultata. Međutim, analiza performansi, osobito uzimajući u obzir graf plamena, otkriva određene probleme s ovim pristupom.

Najproblematičniji dijelovi koda su povezani s deserijalizacijom podataka, osobito kod deserijalizacije bajtova u GradientBooster objekt.

Prema grafu plamena, deserijalizacija dominira u potrošnji resursa, dok stvarna predikcija modela (xgboost_model.predict) zauzima samo 1.3% ukupnog vremena izvršavanja. To znači da je stvarno korištenje modela izuzetno efikasno, ali taj se dobitak gubi zbog vremena potrebnog za pripremu modela i podataka.

Za ovakve aplikacije RISC Zero zkVM je dobro rješenje ako:

- je apsolutno kritično da podaci i modeli ostanu privatni te da se može kriptografski dokazati ispravnost izvršavanja modela bez otkrivanja ulaznih podataka ili modela, zkVM je odličan izbor.
- je potreban dokaz o ispravnosti rezultata koji može biti prihvaćen od strane treće strane bez izlaganja privatnih podataka

Loše je rješenje ako:

- Brzina izvođenja kritična i ako se često izvode predikcije, RISC Zero zkVM može postati neoptimalan zbog značajnog vremena koje se troši na deserijalizaciju podataka i modela. U ovakvim scenarijima, standardna okruženja za izvršavanje modela, koja ne zahtijevaju dodatne kriptografske dokaze, će biti puno brža i učinkovitija.
- se šalju velike količine podataka, proces deserijalizacije postaje značajno spor, što dodatno usporava cijeli proces i čini zkVM neprivlačnim izborom.

6.4 Dokaz provjere dokaza

Ovaj program koristi koncept zvan "Proof Composition", odnosno kompoziciju dokaza. Ideja je da se iskoristi gost program unutar zkVM-a koji će obaviti verifikaciju nekog drugog dokaza te vratiti "kompozitni" dokaz domaćinu. Koncept kompozicije dokaza omogućava razne mogućnosti u tome što se može dokazati. Na primjer, korisnik može kombinirati dokaze o svojim ETH i BTC sredstvima u jedan dokaz o ukupnim sredstvima. To otvara put prema povećanoj interoperabilnosti među lancima, omogućujući korisnicima da generiraju dokaze o aktivnosti na bilo kojem lancu, agregiraju više dokaza u jedan, i potvrde ih na bilo kojem lancu.

6.4.1 Pristup rješavanju

Iako se očito rješenje čini jednostavno pokrenuti dokazivača unutar gost koda, takvo rješenje ne bi rezultiralo sažimanjem dokaza. Risc Zero zkVM to rješava dodavanjem `pretpostavki (eng. assumption)` da je Receipt validan u "ReceiptClaim objekt".

Ključne metode za proof composition su: `add_assumption()` koja se dodaje na stranu domaćina te `env::verify()` koja se dodaje na strani gosta.

Kod domaćina poziva multiply metodu, koja izračunava produkt dva broja unutar zkVM-a (na isti način kao bilo koji drugi zkVM program, stvarajući ExecutorEnv i pružanjem ulaznih vrijednosti), te vraća Receipt i produkt tih brojeva Tuple objekt. Zatim se stvara novi ExecutorEnv kojemu se taj Receipt dodaje kao `pretpostavka` te ulazi za idući zkVM, od kojih je i produkt, spremljen u varijablu `n`.

```
let (multiply_receipt, n) = multiply(17, 23);
let env = ExecutorEnv::builder()
    .add_assumption(multiply_receipt)
    .write(&(n, 9u64, 100u64))
    .unwrap()
    .build()
    .unwrap();
```

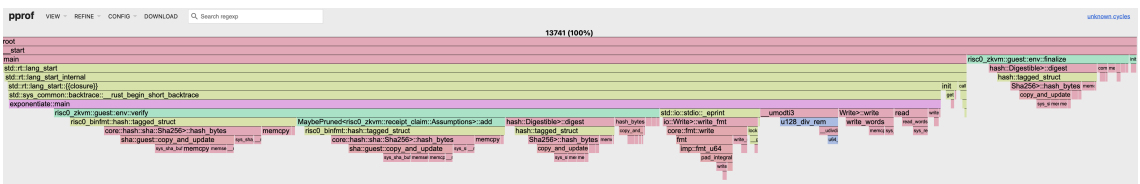
```
let receipt = default_prover()
    .prove(env, EXPONENTIATE_ELF)
    .unwrap()
    .receipt;
```

Za verifikaciju unutar dokaza je naravno potreban Image ID i izlazna vrijednost prošlog izvođenja (produkt poslan iz domaćina) te se provodi idućom linijom:

```
env::verify(MULTIPLY_ID, &serde::to_vec(&n).unwrap()).unwrap();
```

Ukoliko ijedna od te dvije vrijednosti nije točna, program će paničariti.

6.4.2 Profiliranje



Slika 6.6 Graf plamena zadatak "Proof Composition"

Kao što se može vidjeti, aplikacija nije zahtjevna. Verifikacija unutar zkVM-a troši 7334 CPU ciklusa. S obzirom na to da biblioteka `std::time::SystemTime` nije dostupna unutar zkVM-a, za računanje točnog vremena unutar zkVM-a, koristi se ukupno trajanje programa (mjerenog iz domaćina) pomnoženo sa postotkom CPU ciklusa za dio programa koji nas zanima. Pošto verifikacija zauzima ukupno 53.37% CPU ciklusa, a trajanje programa je 2.75 sekundi, vrijeme za verifikaciju dokaza unutar zkVM-a iznosi 1.47 sekundi.

6.4.3 Zaključak o aplikaciji

Kompozicija dokaza, kao koncept, je možda i najbitnija značajka RISC Zero zkVM-a. Kompozitni dokazi predstavljaju ključnu inovaciju u korištenju RISC Zero zkVM-

a, jer omogućavaju ne samo zaštitu privatnih ulaza, već i povećanje efikasnosti i sigurnosti u radu s velikim aplikacijama. Također uvodi značajnu fleksibilnost u dizajn sustava, omogućavajući podjelu odgovornosti među različitim komponentama bez kompromitiranja privatnosti. Kroz modularni pristup i mogućnost integracije različitih izvora podataka, "proof composition" potiče razvoj složenijih i robusnijih aplikacija u RISC Zero ekosustavu.

Konačno, kombiniranjem više dokaza u jedan, pojednostavljuje se proces verifikacije i smanjuje opterećenje na resurse, što je od presudnog značaja za skalabilnost i praktičnost u stvarnim primjenama. Ova značajka čini RISC Zero zkVM ne samo moćnim alatom za zaštitu privatnosti, već i temeljnom platformom za razvoj naprednih kriptografskih rješenja.

Poglavlje 7

Zaključak

U radu su analizirane različite tehnologije koje omogućuju provjerljive izračune opće namjene koristeći tehnologije nultog znanja (Zero Knowledge Proofs - ZKP). Razmotrena su rješenja poput ZK-SNARK-ova i ZK-STARK-ova, koja su se pokazala kao ključne tehnologije u pružanju sigurnih i skalabilnih provjerljivih izračuna.

ZK-SNARK-ovi (Succinct Non-Interactive Arguments of Knowledge) omogućuju stvaranje kompaktnih dokaza koji se mogu brzo provjeriti, ali zahtijevaju složeni proces "trusted setup"-a. Proces postavljanja javnih parametara može predstavljati rizik za sigurnost ako nije pravilno proveden, no unatoč tome, ZK-SNARK-ovi su široko prihvaćeni zbog svoje efikasnosti i kompaktne veličine dokaza.

S druge strane, ZK-STARK-ovi (Zero-Knowledge Scalable Transparent Arguments of Knowledge) nude slične sigurnosne garancije bez potrebe za "trusted setup"-om. Ova tehnologija koristi transparentne postavke, što uklanja potrebu za povjerenjem u treće strane. Iako ZK-STARK-ovi generiraju veće dokaze, njihova otpornost na kvantna računala i eliminacija potrebe za povjerenjem čine ih izuzetno atraktivnim za buduće primjene.

Analiza je također pokazala da su ZK-STARK-ovi skalabilniji za velike skupove podataka, jer njihov proces verifikacije raste sublinearnom brzinom u odnosu na veličinu podataka, što ih čini pogodnijima za aplikacije s velikim količinama podataka.

U radu su istražena i postojeća rješenja temeljena na tehnologijama nultog znanja, osobito RISC Zero. RISC Zero zkVM, kao primjer platforme koja koristi RISC-V

arhitekturu u kombinaciji sa ZKP tehnologijama, pokazuje kako se provjerljivi izračuni mogu učinkovito implementirati u stvarne sustave. Ova tehnologija omogućuje razvijateljima korištenje poznatih alata dok pruža sigurnost i privatnost, čime se olakšava integracija u postojeće ekosustave.

Programi unutar RISC Zero zkVM okruženja pokazali su se kao praktična rješenja za implementaciju sigurnih i provjerljivih izračuna. U radu su analizirani nekoliko konkretnih primjera, uključujući implementaciju ZKP SHA256 hash algoritma i GPU-akceleriranih izračuna unutar RISC Zero ekosustava. Ovi programi demonstriraju sposobnost RISC Zero ekosustava da efikasno obavlja složene kriptografske zadatke s visokim stupnjem sigurnosti, dok istovremeno zadržava jednostavnost i pristupačnost za korisnike.

U radu je provedena i analiza drugih često korištenih algoritama u kriptografiji, kako bi se zainteresiranim čitaocima na vrlo jednostavan i jasan način predočila efikasnost RISC Zero zkVM-a za njihove ideje za koje razmatraju ovaj sustav kao potencijalno rješenje za generiranje dokaza nultog znanja.

Jedan od ključnih aspekata implementacije ovih programa je jednostavnost korištenja alata poput Rust programskog jezika i Cargo alata za upravljanje paketima, koji omogućavaju programerima da brzo razvijaju i testiraju svoje aplikacije unutar RISC Zero ekosustava. Ova pristupačnost značajno smanjuje barijeru ulaska za nove korisnike i olakšava integraciju tehnologija nultog znanja u postojeće projekte.

U radu je isprobana i analizirana delegacija izračuna trećoj strani, koristeći Bonsai API. Delegacija izračuna Bonsai-ju nudi nekoliko prednosti, uključujući povećanje efikasnosti, zbog načina na koji sam paralelizira izvođenje zadataka, i mogućnost obavljanja složenih zadataka bez potrebe za lokalnim računalnim resursima. Međutim, ovaj pristup također donosi određene izazove, kao što su prilagodba ulaznih parametara u sustav te vrlo potrebnu optimizaciju resursa koji su ograničeni brojem izvođenja programa i CPU ciklusa dodijeljenih korisnikovom API ključu. Bonsai je u kratkim izračunima pokazao i sporija vremena izvršavanja, ali ovo mnogo ovisi o računalu korisnika. Također je bitno napomenuti da je Bonsai još uvijek u razvoju te zatvoren javnosti, u trenutku pisanja ovog rada.

Gost kod unutar RISC Zero zkVM-a suočava se s nekoliko ključnih ograničenja ve-

zanih uz IO operacije. Programi ne mogu izravno pristupati datotekama niti koristiti crate-ove poput `reqwest` za mrežnu komunikaciju ili internet pristup. Umjesto toga, svi ulazno-izlazni podaci moraju se prenositi kroz specijalizirane kanale, kroz kod domaćina, čime se osigurava stroga kontrola i sigurnost. Ova ograničenja sprječavaju neovlašten pristup ili manipulaciju podacima, ali zahtijevaju dodatnu optimizaciju i pažljivo planiranje resursa kako bi se efikasno iskoristile sve dostupne mogućnosti IO operacija unutar sustava. To može uključivati prilagodbu aplikacija za rad u strogo kontroliranom okruženju, što programerima dodaje dodatne izazove prilikom implementacije složenijih rješenja.

Još jedan od izazova jest izbor jezika u kojima je RISC Zero zkVM podržan: Rust (koji ima najveću programsku podršku za RISC Zero), C i C++. Iako ovo jesu relativno popularni jezici, statistike pokazuju da su znatno manje korišteni od jezika poput Pythona, TypeScripta i Jave, s toga razvijanje u ovim jezicima može predstavljat probleme zbog mnogo manje razvijenije, pogotovo porastom potražnje za strojnim učenjem. S toga smatram da bi programska podrška unutar tih jezika pomogla RISC Zero zkVM projektu da se jače probije na tržište i omogućila da se tehnologija nultog znanja primjeni u mnogo većem broju sustava.

Bibliografija

[1] B. Parno, J. Howell, Microsoft Research, C. Gentry, M. Raykova, and IBM Research, "Pinocchio: Nearly practical verifiable computation," 2013. , s Interneta, https://eprint.iacr.org/2013/279.pdf

[2] Transak, "What Are Zero-Knowledge Proofs?: A Detailed Explainer," 2023. , s Interneta, https://transak.com/blog/what-are-zero-knowledge-proofs-a-detailed-explainer

[3] E. Pepil, "History of the formation of zkp," 2024. , s Interneta, https://medium.com/@emilpepil/history-of-the-formation-of-zkp-151dd7001ffa

[4] Trail of Bits, "Fiat-shamir transformation." , s Interneta, https://www.zkdocs.com/docs/zkdocs/protocol-primitives/fiat-shamir/

[5] Chainlink. What is a replay attack? , s Interneta, https://cdn.prod.website-files.com/5f75fe1dce99248be5a892db/66d1a563e3c0a09d3f5531de_6695316cadf4120c1e1a4e32_Session-Replay-Attack_V2.png

[6] S. Brown. (2021) How zk-rollups work. , s Interneta, https://medium.com/fcats-blockchain-incubator/how-zk-rollups-work-8ac4d7155b0e

[7] N. Bitansky, R. Canneti, A. Chiesa, and E. Tromer, "From extractable collision resistanceto succinct non-interactive arguments of knowledge,and back again," 2011. , s Interneta, https://eprint.iacr.org/2011/443.pdf

[8] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs : Short proofs for confidential transactions and more," 2017. , s Interneta, https://eprint.iacr.org/2017/1066.pdf

[9] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal andupdatable structured reference strings," 2019. , s Interneta, https://eprint.iacr.org/2019/099.pdf

[10] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, "Marlin : Preprocessing zksnarkswith universal and updatable srs," 2020. , s Interneta, https://eprint.iacr.org/2019/1047.pdf

[11] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," 2018. , s Interneta, https://eprint.iacr.org/2018/046

[12] "Risc vs. cisc." , s Interneta, https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

[13] J. Bruestle, P. Gafni, and RISC Zero Team, "Risc zero zkvm: Scalable, transparent arguments of risc-v integrity," 2023. , s Interneta, https://dev.risczero.com/proof-system-in-detail.pdf

[14] A. Gabizon and Z. J. Williamson, "plookup: A simplified polynomial protocol for lookup tables," 2020. , s Interneta, https://eprint.iacr.org/2020/315.pdf

[15] RISC Zero. (2024) Receipts 101. RISC Zero. , s Interneta, https://dev.risczero.com/assets/images/from-rust-to-receipt-23117368c4f46d78c8cac3b753245a5a.png

[16] J. Jakobsson. , s Interneta, https://github.com/joelonsql/ecdsa_verify

Pojmovnik

CISC Complex Instruction Set Computer. 32, 33

HMAC Hash-based Message Authentication Code. 39

ISA Instruction set architecture. 31, 32

RISC Reduced Instruction Set Computer. 32, 33

SNARG Succinct Non-Interactive Argument. xii, 21, 22

SNARK Succinct Non-Interactive Arguments of Knowledge. xii, 20–24

STARK Scalable Transparent Arguments of Knowledge. 20, 38

VC Verifiable Computation. 3, 4

ZK Zero knowledge. 1, 17, 18, 27, 52, 86

ZK-SNARK Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge. 23–31

ZK-STARK Zero-Knowledge Scalable Transparent Argument of Knowledge. 27–31, 36

zkEVM Zero Knowledge Ethereum Virtual Machine. 26

ZKP Zero knowledge Proofs. 7, 8, 15, 16, 19, 20, 23, 58, 83, 87

zkVM Zero Knowledge Virtual Machine. 30, 36–38, 45, 49, 52, 53, 56–58, 63, 68, 77–79, 81, 83, 84, 87–92

Sažetak

U ovom diplomskom radu istražena su postojeća rješenja za provjerljive izračune opće namjene temeljene na tehnologijama nultog znanja. Tehnologije nultog znanja omogućuju dokazivanje točnosti određenih informacija ili izračuna bez otkrivanja samih informacija, što predstavlja ključnu komponentu za unapređenje sigurnosti i privatnosti u računalnim sustavima.

Rad se sastoji od nekoliko dijelova. U prvom dijelu predstavljena je teorijska osnova tehnologija nultog znanja, uključujući osnovne pojmove i algoritme koji se koriste za postizanje nultog znanja. U drugom dijelu analizirana su postojeća rješenja za provjerljive izračune, uključujući njihove arhitekture, metode implementacije i praktične primjene.

Posebna pažnja posvećena je različitim slučajevima provjerljivih izračuna s fokusom na vremensku složenost izračuna. U radu su detaljno ispitani primjeri provjerljivih izračuna kako bi se razumjele prednosti i nedostaci svake metode u kontekstu performansi i skalabilnosti.

U završnom dijelu rada, istražena je praktičnost implementacija ovih tehnologija u kontekstu delegiranja izračuna trećim stranama. Delegiranje izračuna predstavlja sve učestaliju praksu u modernim računalnim sustavima, posebno u oblaku, gdje korisnici žele osigurati da su izračuni koje obavljaju treće strane točni i pouzdani bez potrebe za potpunim povjerenjem u te strane.

Rezultati istraživanja ukazuju na značajne prednosti korištenja tehnologija nultog znanja za provjerljive izračune, ali također identificiraju izazove u pogledu složenosti implementacije i optimizacije performansi. Rad završava prijedlozima za buduća istraživanja i potencijalna poboljšanja koja bi mogla dodatno unaprijediti praktičnost i učinkovitost ovih sustava.

Ključne riječi — **provjerljivi izračuni, tehnologije nultog znanja, vremenska složenost, delegiranje izračuna, sigurnost, privatnost**

Abstract

This thesis investigates existing solutions for verifiable general-purpose computations based on zero-knowledge technologies. Zero-knowledge technologies enable the verification of the accuracy of certain information or computations without revealing the information itself, which is a crucial component for enhancing security and privacy in computing systems.

The thesis is divided into several parts. The first part presents the theoretical foundation of zero-knowledge technologies, including the basic concepts and algorithms used to achieve zero-knowledge proofs. The second part analyzes existing solutions for verifiable computations, including their architectures, implementation methods, and practical applications.

Special attention is given to various cases of verifiable computations with a focus on the time complexity of computations. Detailed examination of examples of verifiable computations is provided to understand the advantages and disadvantages of each method in terms of performance and scalability.

In the final part of the thesis, the practicality of implementing these technologies in the context of delegating computations to third parties is explored. Delegating computations is an increasingly common practice in modern computing systems, especially in the cloud, where users want to ensure that computations performed by third parties are accurate and reliable without the need for complete trust in those parties.

The research results indicate significant advantages of using zero-knowledge technologies for verifiable computations, but also identify challenges regarding the complexity of implementation and performance optimization. The thesis concludes with suggestions for future research and potential improvements that could further enhance the practicality and efficiency of these systems.

Keywords **— verifiable computations, zero-knowledge technologies, time complexity, delegation of computations, security, privacy**