VILNIAUS UNIVERSITETO
KAUNO FAKULTETAS

SOCIALINIŲ MOKSLŲ IR TAIKOMOSIOS INFORMATIKOS INSTITUTAS

Finansų technologijų studijų programa
Kodas 6211BX022

IGNAS BUDREIKA

MAGISTRO BAIGIAMASIS DARBAS

EFEKTYVUMO DIDINIMAS IR VYKDYMO MOKESČIŲ MAŽINIMAS „ETHEREUM“ IŠMANIOSIOSE SUTARTYSE TAIKANT NAŠIUOSIUS KOMPIUTERIJOS METODUS

Kaunas 2024

VILNIAUS UNIVERSITETO
KAUNO FAKULTETAS

SOCIALINIŲ MOKSLŲ IR TAIKOMOSIOS INFORMATIKOS INSTITUTAS

IGNAS BUDREIKA

MAGISTRO BAIGIAMASIS DARBAS

EFEKTYVUMO DIDINIMAS IR VYKDYMO MOKESČIŲ MAŽINIMAS „ETHEREUM“ IŠMANIOSIOSE SUTARTYSE TAIKANT NAŠIUOSIUS KOMPIUTERIJOS METODUS

Leidžiama ginti

Magistrantas _____
(parašas)

Darbo vadovas _____
(parašas)

_____
(darbo vadovo mokslo laipsnis, mokslo pedagoginis vardas, vardas ir pavardė)

Darbo įteikimo data

Registracijos Nr.

Kaunas 2024

TURINYS

SANTRUMPŲ SĄRAŠAS

EVM – „Ethereum“ virtuali mašina

OPCODE – EVM baitkodo interpretavimui naudojamos instrukcijos

BSC – „Binance“ sukurta, su EVM suderinama blokų grandinė (angl. „Binance Smart Chain“)

DAO – decentralizuota autonominė organizacija

CCIP – tarpusavio sąveikos tarp grandinių protokolas (angl. cross-chain interoperability protocol)

On-chain – numatytos logikos vykdymo atlikimas blokų grandinėje

Cross-chain – numatytos logikos vykdymo atlikimas naudojant daugiau nei vieną blokų grandinę

Off-chain – numatytos logikos vykdymo atlikimas už grandinės ribų

PAVEIKSLŲ SĄRAŠAS

LENTELIŲ SĄRAŠAS

Budreika Ignas (2024). *Increasing Efficiency and Reducing Execution Fees in Ethereum Smart Contracts Using High-Performance Computing Techniques*. MBA Graduation Paper. Kaunas: Vilnius University, Kaunas Faculty, Institute of Social Sciences and Applied Informatics. 76 p.

SUMMARY

Decentralised smart contracts based on blockchain technology and secured by cryptographic means have been developed to address the problems of traditional - centralised - systems. As the Ethereum platform is the first and main platform for smart contracts, an analysis of smart contracts on this platform has been carried out. The analysis, including technical specifications, helped to clarify execution fees problems arising from unoptimized smart contracts code. A few high-performance computing techniques, whose practical application in the context of smart contracts has not yet been extensively explored and which could help to address the optimization deficiencies in complex smart contracts, were analysed.

Introduced high-performance computing techniques were discussed in the context of a sample optimisation problem - the Knapsack problem. Different Knapsack problem algorithms and their time and space complexities were compared, with further experiments being carried out with the most optimal one. Alternative solutions of cross-chain implementation using “Chainlink CCIP“ and off-chain implementation using “Chainlink Functions“ were proposed.

A simulated smart contract, complying with a discussed optimisation problem, was introduced for the experiment. Proposed technical solutions were implemented and deployed on the Sepolia network. Different datasets were used to compare execution fees in relation to the algorithm time complexity and the volume of data between the solutions. The results showed, that both cross-chain and off-chain solutions reduce execution fees for example smart contract. Regarding other covered aspects and limitations, integration with “Chainlink Functions“ to implement off-chain solutions was selected as a better alternative to implementing complex smart contracts solely on-chain. Using the proposed off-chain solution may not only reduce the execution fees on the Ethereum platform but may also increase its efficiency by allowing more complex smart contracts to be implemented.

This paper consists of 64 pages, 18 tables and 22 figures.

ĮVADAS

Tradicinių centralizuotų sistemų problemoms spręsti sukurta įvairiais kriptografiniai metodais paremta decentralizuota blokų grandinės technologija ir išmaniosios sutartys, veikiančios šios technologijos pagrindu.

Nagrinėjant teorinius šaltinius buvo atlikta „Ethereum“ išmaniųjų sutarčių analizė, apimanti technines specifikacijas. Išsiaiškinus įprastas išmaniųjų sutarčių programinio kodo problemas, aptarti rinkoje jau pasiūlyti jų sprendimo būdai. Taip pat išanalizuoti tradicinio programavimo bei blokų grandinių našieji kompiuterijos metodai, kurių taikymas išmaniųjų sutarčių kontekste nėra plačiai ištirtas. Šie metodai galėtų padėti spręsti išmaniųjų sutarčių efektyvumo bei vykdymo mokesčių trūkumus.

Darbą sudaro 76 puslapiai, 18 lentelių ir 22 paveikslai.

Temos aktualumas

Besivystant blokų grandinės technologijai, Buterin (2014) pristatytas išmaniųjų sutarčių sprendimas, smarkiai praplėtęs modernių blokų grandinių pritaikymo galimybes. Išmaniosios sutartys vykdomos blokų grandinėje, todėl leidžia naudotojams decentralizuotu būdu atlikti įvairaus pobūdžio – tiek finansines, tiek ne finansines – operacijas tinkle.

Augant išmaniosios sutarties kompleksiškumui, didėja ir atliekamų operacijų skaičius, kuris tiesiogiai susijęs su išmaniosios sutarties naudojimo kaštais (1 lentelė). Neatsižvelgiant į įvairius išmaniųjų sutarčių techninius niuansus ar naudojant neefektyvius algoritmus, tai gali lemti didesnius vykdymo kaštus bei kitas susijusias problemas, pavyzdžiui saugumo (OWASP, 2023a, 2023b). Dėl šių priežasčių jų optimizavimas – vykdymo mokesčių ir efektyvumo prasme – yra itin opi tema. Sunaudojančios daugiau nei reikalinga mokesčių (Chen, J. et al., 2020) išmaniosios sutartys tiesiogiai finansine prasme atsiliepia tiek individualiems jų naudotojams, tiek verslams, teikiantiems ar naudojantiems įvairias decentralizuotas paslaugas, todėl tokios išmaniosios sutartys gali būti laikomos neoptimaliomis. Išmaniųjų sutarčių ekosistemai augant, svarbu atrasti tinkamus būdus kaip sutarčių vykdymą optimizuoti, išsprendžiant plečiamumo ir optimalumo santykio problemas egzistuojančiose išmaniųjų sutarčių platformose.

Kadangi „Ethereum“ platforma yra pirmoji ir lyderiaujanti išmaniųjų sutarčių platforma, jos svarbą išmaniųjų sutarčių kontekste galima laikyti stipriausia. „Ethereum“ blokų grandinės atveju, išmaniųjų sutarčių mokesčiai tiesiogiai priklauso nuo atliekamų operacijų skaičiaus, o mokslininkų Chen, J. et al. (2020) tyrimas parodė, kad nemaža dalis „Ethereum“ platformos išmaniųjų sutarčių turi neoptimalaus kodo fragmentų. Našieji kompiuterijos metodai galėtų padėti optimizuoti sprendžiamų problemų algoritmus išmaniųjų sutarčių kontekste, sumažinant jų atliekamų operacijų

skaičių blokų grandinėje, kuris anot Sherry, ir Thompson (2021) yra tiesiogiai susijęs su algoritmo sudėtingumu.

Problemos ištyrimo lygis

Išmaniųjų sutarčių optimizavimo problemos yra nagrinėtos taikant įvairius būdus ir sprendimo pasiūlymus.

Kadangi „Ethereum“ platformos atveju išmaniosios sutarties diegimo metu tai pat mokamas mokestis už sutarties baitkodą, vienas iš metodų kaip galima optimizuoti išmaniąsias sutartis yra jos programinio kodo apimties sumažinimas. Vieną iš būdų aptikti neefektyviai aprašytus programinio kodo ciklus bei šios problemos sprendimą aptarė Kaur, Tomar, ir Tripathi (2022). Jų pasiūlytas metodas orientuotas į išmaniųjų sutarčių transakcijų mokesčių mažinimą, sprendžiant programinio kodo ciklų semantines problemas. Deja, numatyti visus išmaniųjų sutarčių trūkumus iš anksto nėra įmanoma, o teisingai realizuota programa nebūtinai atliks užduoties sprendimą naudodama mažiausią operacijų skaičių.

Be kodo analizės, programinis išmaniųjų sutarčių kodas taip pat gali būti optimizuojamas naudojant įterptinį asemblerio – žemo lygio – programinį kodą. Chaliasos, Gervais, ir Livshits (2022) aptarė šio principo taikymo pranašumus vykdymo mokesčių prasme bei trūkumus kalbant apie saugumo aspektą. Tai vis daugiau dėmesio susilaukiantis, tačiau turintis savų niuansų saugumo aspektu sprendimas išmaniųjų sutarčių mokesčių optimizavimo problemoms spręsti.

Dar vienas metodas išmaniųjų sutarčių patobulinimui, reikalaujantis egzistuojančių išmaniųjų sutarčių platformų sprendimų pakeitimų – lygiagretaus išmaniųjų sutarčių apdorojimo galimybė. Tokį pasiūlymą pristatė mokslininkai Chen, Gao, Lu, Xu, ir Shi (2017), o šis sprendimas turėtų padidinti pačios išmaniųjų sutarčių platformos efektyvumą. Nepaisant to, jis nėra orientuotas į atskirų išmaniųjų sutarčių optimizacijos svarbą, todėl nėra aktualus vykdymo mokesčių problemoms spręsti.

Nors yra įvairių pasiūlytų išmaniųjų sutarčių vykdymo mokesčių mažinimo metodų, našiųjų kompiuterijos metodų taikymas šiai problemai spręsti moksliniuose straipsniuose nėra plačiai aptartas.

Darbo problema

Kaip padidinti išmaniųjų sutarčių efektyvumą ir sumažinti vykdymo mokesčius „Ethereum“ platformoje pritaikant našiuosius kompiuterijos metodus?

Darbo objektas

Išmaniųjų sutarčių kūrimas „Ethereum“ blokų grandinėje.

Darbo tikslas

Optimizuoti „Ethereum“ išmaniųjų sutarčių vykdymo mokesčius taikant našiuosius kompiuterijos metodus.

Darbo uždaviniai

Siekiant įgyvendinti išsikeltą tikslą yra atliekami tokie uždaviniai:

1. Išanalizuoti „Ethereum“ išmaniųjų sutarčių veikimo principą ir techninius aspektus.
2. Aptarti išmaniųjų sutarčių optimizacijos problemas, egzistuojančius optimizavimo būdus bei našiuosius kompiuterijos metodus.
3. Pateikti siūlomą „Ethereum“ išmaniųjų sutarčių vykdymo mokesčių sumažinimo sprendimą naudojant našiuosius kompiuterijos metodus.
4. Atlikti išmaniųjų sutarčių eksperimentinį optimizavimą, pritaikant pasiūlytą sprendimą.
5. Įvertinti pasiūlyto sprendimo būdu optimizuotų išmaniųjų sutarčių vykdymo mokesčių bei efektyvumo aspektus.

Darbo struktūra

Pirmoje – „Ethereum“ išmaniųjų sutarčių vykdymo mokesčių optimizavimo metodų analizė – dalyje analizuojama techninė „Ethereum“ blokų grandinės išmaniųjų sutarčių dalis, nagrinėjamos jų vykdymo mokesčių ir efektyvumo problemos. Aptariami egzistuojantys šių problemų sprendimo būdai ir našieji kompiuterijos metodai bei jų taikymas išmaniųjų sutarčių kontekste.

Antroje dalyje – „Ethereum“ išmaniųjų sutarčių vykdymo mokesčių optimizavimo techninis pasiūlymas – nagrinėjama pavyzdinė optimizacijos problema – „0/1“ kuprinės problema. Įvertinamas šios problemos įgyvendinamas „Ethereum“ išmaniosiose sutartyse bei pasiūlomi alternatyvūs sprendimai vykdymo mokesčių optimizavimui ir efektyvumui padidinti.

Trečioje – eksperimentinis „Ethereum“ išmaniųjų sutarčių vykdymo mokesčių optimizavimas – dalyje realizuojami aptarti išmaniųjų sutarčių sprendimai. Atliekamas eksperimentinis jų testavimas ir aptariami bei palyginami jų rezultatai vykdymo mokesčių ir efektyvumo prasme.

Darbo pabaigoje pateikiamos atliktos analizės, techninio siūlymo ir įgyvendintų sprendimų išvados. Pateikiamas siūlymas „Ethereum“ išmaniųjų sutarčių vykdymo mokesčiams sumažinti bei efektyvumui padidinti.

Literatūros šaltiniai

Teorinėje dalyje remiamasi užsienio autorių moksline literatūra, susijusia su blokų grandinės ir išmaniųjų sutarčių techniniais aspektais, našiaisiais kompiuterijos metodais. Praktinei daliai naudotos pritaikytų technologijų dokumentacijos.

Darbo ir tyrimo metodai

Atliekant teorinę „Ethereum“ išmaniųjų sutarčių analizę naudojama mokslinės literatūros analizė ir sintezė, indukcija bei klasifikacija. Techniniam pasiūlymui skirti naudojama asimptotinė bei lyginamoji algoritmų analizė, technologijų funkcionalumo analizė. Eksperimentiniam tyrimui atlikti taikoma lyginamoji kaštų ir rezultatų analizė.

Darbo teorinė reikšmė

- Atlikta „Ethereum“ išmaniųjų sutarčių techninė analizė leido suprasti egzistuojančias jų vykdymo mokesčių ir efektyvumo problemas.
- Išnagrinėti našieji kompiuterijos metodai padėjo įžvelgti šių problemų sprendimo galimybes taikant algoritmų analizę bei orakulus.

Darbo praktinė reikšmė

- Išanalizavus „Ethereum“ išmaniųjų sutarčių efektyvumo ir vykdymo mokesčių trūkumus, pasiūlyti cross-chain bei off-chain sprendimai kaip skaičiavimų pačioje grandinėje alternatyva.
- Pritaikytos „Chainlink CCIP“ ir „Chainlink Functions“ technologijos pavyzdinei optimizacijos problemai spręsti „Ethereum“ blokų grandinėje.
- Remiantis sprendimų testavimo rezultatais įvertinti vykdymo mokesčių ir efektyvumo aspektai pasiūlytiems sprendimams. Nuspręstą, kad off-chain sprendimas pritaikant „Chainlink Functions“ integraciją gali padėti spręsti „Ethereum“ blokų grandinės efektyvumo ir vykdymo mokesčių problemas.
- Sukurtas „Google Chrome“ naršyklės įskiepis numatomų on-chain bei off-chain sprendimų vykdymo mokesčių įvertinimui realiu metu.

Darbo apribojimai ir sunkumai

- Praktinis skaičiavimo orakulų pritaikymas išmaniosiose sutartyse nėra plačiai aptartas nagrinėtuose literatūros šaltiniuose.
- „Chainlink CCIP“ ir „Chainlink Functions“ naujumas lėmė mokslinių straipsnių trūkumą, todėl remtasi šių technologijų techninėmis dokumentacijomis.

Darbo struktūra ir apimtis

Rašto darbą sudaro įvadas, 3 skyriai, išvados bei siūlymai. Pagrindinę darbo dalį apima 64 puslapiai, 22 paveikslai ir 18 lentelių. Taip pat pateikiami 7 priedai. Literatūros sąrašą sudaro 37 šaltiniai.

1. „ETHEREUM“ IŠMANIŲJŲ SUTARČIŲ VYKDYMO MOKESČIŲ OPTIMIZAVIMO METODŲ ANALIZĖ

Šioje dalyje nagrinėjamas išmaniųjų sutarčių veikimas ir techninės jų savybės, bei aptariamos jų efektyvumo problemos, vykdymo mokesčių mažinimo metodai.

1.1. Išmaniųjų sutarčių apžvalga

Viena svarbiausių technologinių blokų grandinės galimybių – išmaniosios sutartys. Jos leido praplėsti modernių blokų grandinių panaudojimo atvejų sritis, skaitmenizuojant sutarčių pasirašymą tokiu būdu, kuris nereikalautų trečiosios šalies įsiterpimo sutarties sąlygoms ir vykdymui užtikrinti.

1.1.1. Išmaniųjų sutarčių platformos

Nors pirminė išmaniųjų sutarčių idėja, kurią pristatė Szabo (1994) ir turi daug panašumų su dabartine jų realizacija, kai kurie šiuo metu taikomi principai šioms sutartims vykdyti skiriasi. Išmaniąsias sutartis, kurios šiuo metu plačiai naudojamos blokų grandinėse, pristatė Buterin (2014), išleisdamas decentralizuotą aplikacijų vykdymo platformą – „Ethereum“. „Ethereum“ rėmėsi Nakamoto (2008) pristatytos „Bitcoin“ kriptovaliutos pavyzdžiu, o didesnį dėmėsį skyrė pačiai blokų grandinės technologijai bei decentralizuotam konsensuso algoritmui. Remiantis šiomis technologijomis, „Ethereum“ pristatė blokų grandinę kartu su Tiuringo testus įveikiančia programavimo kalba, skirta kontraktams – kuriais laikomos išmaniosios sutartys – kurti. Anot Buterin (2014), išmaniosios sutartys gali būti pritaikytos tiek finansiniams sprendimams, tiek pusiau finansiniams sprendimams ar net visiškai ne finansinėms aplikacijoms realizuoti.

„Ethereum“ platforma revoliucionizavo „Bitcoin“ bei iš esmės visos blokų grandinės technologijos ekosistemą, nes pristatyta galimybė kurti išmaniąsias sutartis, pasinaudojant EVM teikiamomis galimybėmis. Finansinės ir ne tik aplikacijos pagaliau galėjo tapti decentralizuotos pritaikant blokų grandinės ir išmaniųjų sutarčių technologijas.

Mokslininkai Bartoletti et al. (2024) atliko palyginamąją išmaniųjų sutarčių programinio kodo kalbų bei platformų analizę. Pagrindinės aptartos išmaniųjų sutarčių kalbos bei pagrindinės jų blokų grandinės ir platformos – „Solidity“ kalba „Ethereum“ blokų grandinėje, „Rust“ kalba „Solana“ blokų grandinėje, „Aiken“ kalba „Cardano“ blokų grandinėje, „(Py)TEAL“ kalba „Algorand“ blokų grandinėje, „Move“ kalba „Aptos“ blokų grandinėje bei „SmartPy“ kalba „Tezos“ blokų grandinėje. Jos pasirinktos dėl išaugusio ir dažno jų pritaikymo decentralizuotoms aplikacijoms kurti. Priklausomai nuo platformos gali skirtis vykdymo mokesčių sistema, kuri „Ethereum“, „Cardano“, „Aptor“ bei „Tezos“ platformų atveju priklauso nuo operacijų skaičiaus. Anot mokslininkų Doss, Pan, ir Praitheeshan (2020) – iš visų išmaniųjų sutarčių kūrimo platformų, „Ethereum“ platforma šiuo metu yra lyderiaujanti.

Išmaniosioms sutartims kurti šiuo metu yra nemažas skirtingų programavimo kalbų bei platformų pasirinkimas. Kiekviena iš jų turi savus privalumus, todėl prieš kuriant išmaniąsias sutartis svarbu į juos atsižvelgti, siekiant išsirinkti labiausiai tinkančią platformą konkrečiam sutarties pritaikymo atvejui. Kadangi „Ethereum“ platforma yra pirmoji ir lyderiaujanti, jos svarbą išmaniųjų sutarčių kontekste galima laikyti stipriausia. O taip pat dėl taikomo mokesčių modelio – tiesiogiai susijusio su atliekamų operacijų skaičiaus – ši platforma pasirinkta nagrinėjamai problemai spręsti.

Tolimesniame skyrelyje apžvelgiamos būtent „Ethereum“ platformai būdingos išmaniųjų sutarčių techninės specifikacijos.

1.1.2. „Ethereum“ platformos išmaniųjų sutarčių techninės specifikacijos

Siekiant suprasti išmaniųjų sutarčių kontekste kylančias efektyvumo problemas svarbu žinoti techninę išmaniųjų sutarčių dalį, kuri aptariama šiame skyrelyje.

Detaliau išmaniųjų sutarčių technines savybes nagrinėjo mokslininkai Chen, W. et al. (2019). Anot jų, išmaniosios sutarties gyvavimo ciklas gali būti išskiriamas į 4 pagrindines dalis – sukūrimas, diegimas blokų grandinėje, vykdymas bei vykdymo užbaigimas. Išmaniųjų sutarčių diegimo procesas atliekamas transakcijos blokų grandinėje būdu. Įdiegus išmaniąją sutartį į blokų grandinę gali būti kviečiama sutarties logika. Tam inicijuoti reikalinga nauja transakcija. Išmanioji sutartis laikoma įvykusia, kai ši transakcija yra patvirtinama blokų grandinės validatorių.

Taigi, išmaniųjų sutarčių veikimo principas paveldi įprastus programinės įrangos kūrimo principus, tačiau pats sutarties programinio kodo vykdymas užtikrinamas decentralizuotu būdu. Išmaniosios sutarties gyvavimo ciklas susideda iš kūrimo, diegimo bei vykdymo, o pastariesiems žingsniams inicijuoti reikalingos blokų grandinės transakcijos, kurių metu už atliekamas operacijas „Ethereum“ platformoje mokami transakcijos mokesčiai.

Technologinį „Ethereum“ išmaniųjų sutarčių aspektą nagrinėjo mokslininkai Oliva, Hassan, ir Jiang (2020). Visų pirma, išmaniosioms sutartims sukurti rašomas programinis kodas, kuriam anot jų dažniausiai naudojama „Solidity“ programavimo kalba. „Solidity“ kompiliatorius šį programinį kodą paverčia EVM baitkodu, kurį interpretuoja ir vykdo EVM. Šis baitkodas yra sudarytas iš OPCODE operacijų sekų, o už kiekvieną atliekamą operaciją transakcijos metu yra mokamas tam tikras dujų (toliau angl. gas) mokestis (1 lentelė).

EVM vykdo sukompiliuotą „Solidity“ ar kitos išmaniųjų sutarčių programavimo kalbos EVM baitkodą, kuris interpretuojamas OPCODE sekų pavidalu. „Solidity“ yra populiariausia „Ethereum“ blokų grandinės išmaniosioms sutartims naudojama programavimo kalba.

1 lentelė

EVM OPCODE operacijų gas kainos

Operacijos	**Gas**	**Paaiškinimas**
RETURN, REVERT	0	Vykdymo sustabdymo operacija
TIMESTAMP, NUMBER, GASLIMIT	2	Dabartinio bloko informacijos operacija
ADD, SUB	3	Aritmetinė operacija
MUL, DIV	5	Aritmetinė operacija
AND, OR, XOR	3	Bitų logikos operacija
LT, GT, SLT, SGT, EQ	3	Palyginimo operacija
POP	2	Dėklo operacija
PUSH, DUP, SWAP	3	Dėklo operacija
MLOAD, MSTORE	3	Atminties operacija
JUMP	8	Programos skaitiklio pakeitimo operacija
JUMPI	10	Programos skaitiklio pakeitimo operacija
SLOAD	100/2100	Saugyklos operacija
SSTORE	2900/20,000	Saugyklos operacija

Šaltinis: sudaryta autoriaus remiantis Wood, *G.* (2024) ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER, p. 28

Remiantis Wood (2024) pristatyta „Ethereum“ technine dokumentacija, sutarties sukūrimo transakcijos metu mokami operacijų vykdymo mokesčiai ir kodo depozito mokestis, kuris yra proporcingas sutarties programinio kodo apimčiai. Naudotojai norėdami naudotis išmaniosiomis sutartimis, patalpintomis „Ethereum“ blokų grandinėje, kviečia sutartyje aprašytas funkcijas transakcijų būdu. Remiantis „Ethereum“ technine specifikacija, transakcijos vykdymo metu kiekviena išmaniosios sutarties atliekama operacija kainuoja, o įvykdžius transakciją apskaičiuojamas galutinis jos mokestis, negalintis viršyti nustatyto bloko mokesčių limito – kuris „Ethereum“ blokų grandinėje šiuo metu siekia 30,000,000.

„Ethereum“ blokų grandinėje išmaniosios sutartys įdiegiamos ir yra pasiekiamos transakcijų būdu. Tiek diegimo, tiek vykdymo transakcijos turi vykdymo mokesčius, proporcingus atliktų operacijų skaičiui, o diegimo transakcija turi ir papildomą mokestį, proporcingą sutarties programinio kodo apimčiai.

Išmaniosios sutartys gali būti kuriamos pasitelkiant įvairias programavimo kalbas ir jas palaikančias platformas. Kiekviena iš jų turi savų privalumų ir niuansų, pagal kuriuos galima pasirinkti tinkamiausią sprendžiamai problemai. Pagrindinė išmaniųjų sutarčių platforma „Ethereum“ vykdo išmaniąsias sutartis EVM baitkodo pavidalu, kuris atitinka EVM palaikomų OPCODE operacijų seką ir yra gaunamas sukompiliavus „Solidity“ ar kitos aukštesnio lygio programavimo kalbos kodą. O pačios „Ethereum“ išmaniosios sutartys diegiamos ir vykdomos blokų grandinėje transakcijų metodu, už vykdomas operacijas mokant vykdymo mokesčius.

Tolimesnėje dalyje aptariamos išmaniųjų sutarčių vykdymo optimizacijos problemos.

1.2. Išmaniųjų sutarčių vykdymo optimizacijos problemos

Netinkamai aprašytos išmaniosios sutartys gali turėti pažeidžiamumų, kurių pagrindinė galima žala – finansiniai nuostoliai. Prieš kuriant išmaniąsias sutartis svarbu susipažinti su įmanomomis jų optimizacijos problemomis.

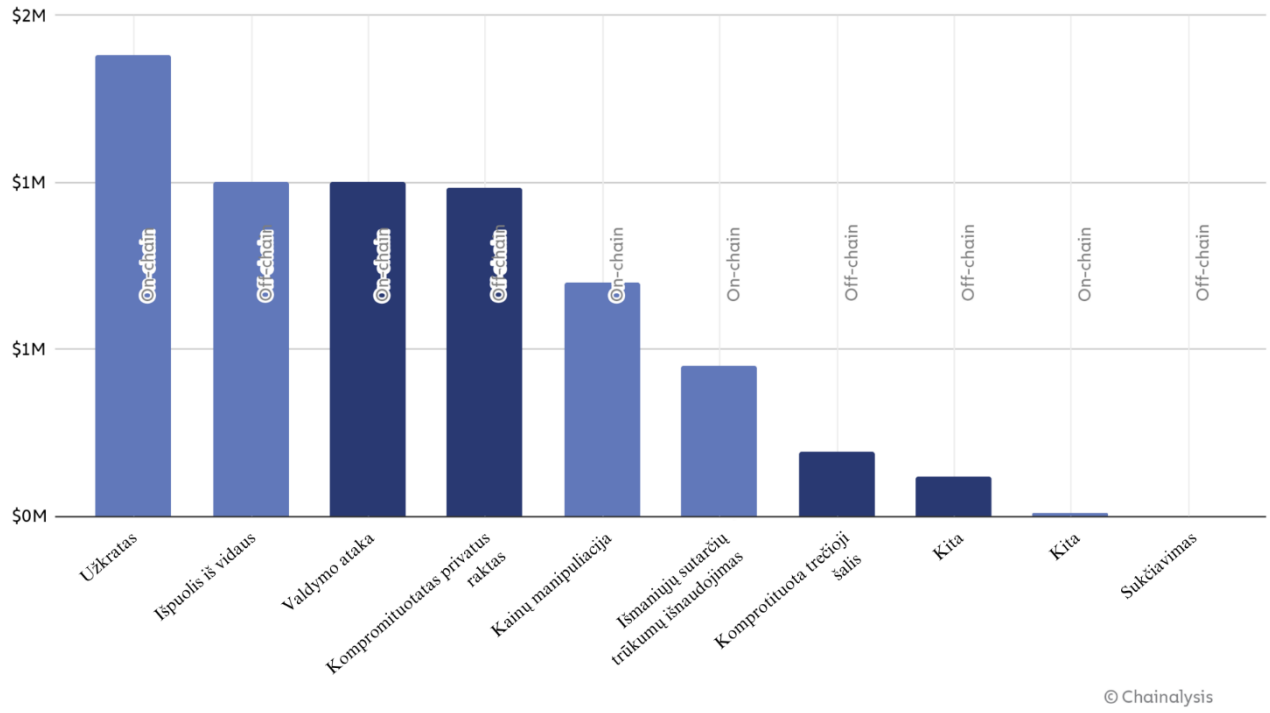
Viena iš esminių išmaniųjų sutarčių optimizacijos problemų, tiesiogiai susijusi su šių sutarčių vykdymo išlaidomis – neefektyvus programinis išmaniųjų sutarčių kodas. Šią problemą nagrinėjo mokslininkai Chen, T. et al. (2017). Kadangi „Ethereum“ platformos atveju transakcijos vykdymo mokesčiai tiesiogiai priklauso nuo sutarties atliekamų operacijų (1 lentelė), neefektyviai aprašytas išmaniųjų sutarčių programinis kodas lemia galimą permokėjimą už transakciją. Kai kurios operacijos, tokios kaip atminties (angl. memory) ar aritmetinės operacijos, yra sąlyginai pigios mokesčių atžvilgiu. Tačiau tokios operacijos kaip saugyklos (angl. storage) naudojimas (*SLOAD*, *SSTORE*) yra itin brangios, todėl turi būti naudojamos apgalvotai. Rezultate, visus sutarties programinio kodo atliekamų operacijų mokesčius sumoka piniginė, iš kurios inicijuota išmaniosios sutarties vykdymo transakcija.

Mokslininkai Chen, T. et al. (2017) nagrinėjo 7 neefektyvius išmaniųjų sutarčių programinio kodo modelius, kurie yra susiję su nenaudojamu kodu arba kode naudojamais ciklais. Nors nenaudojamas programinis kodas nepadidina vykdymo mokesčių, nes jo operacijos nėra pasiekiamos sutarties transakcijos metu, tačiau padidėjęs išmaniosios sutarties EVM baitkodas lemia didesnius sutarties diegimo mokesčius. Su programinio kodo ciklais susijusios problemos apėmė pasikartojančius skaičiavimus, perteklinius ciklus bei neefektyviai naudojamus palyginimus. Šių problemų aspektu neefektyvus programinis kodas nebuvo automatiškai optimizuotas naudoto „Solidity“ kompiliatoriaus.

Jei išmaniosios sutarties programinis kodas neefektyviai atlieka įvairias aritmetines ar kitas operacijas, naudotojai už tai permoka transakcijų vykdymo mokesčių pavidalu. Svarbu aptikti įprastas neefektyvaus programinio kodo optimizacijos problemas rašant kodą, nes „Solidity“ kompiliatorius ne visada geba aptikti ir optimizuoti šias problemas pats.

Neefektyvaus išmaniųjų sutarčių programinio kodo lemiami didesni vykdymo mokesčiai – tik viena išmaniųjų sutarčių optimizacijos problemos dalis. Ne mažiau svarbus blokų grandinės kontekste yra ir saugumo aspektas. Atakos pasinaudojant išmaniųjų sutarčių pažeidžiamumais įprastai lemia didelę finansinę žalą (1 pav.), o tokios techninės išmaniųjų sutarčių savybės kaip programinio kodo nekeičiamumas bei skaidrumas palengvina atakų galimybes. OWASP išskirtuose pagrindiniuose 10 išmaniųjų sutarčių pažeidžiamumų, 2 iš jų yra tiesiogiai susiję su išmaniosios sutarties vykdymo mokesčiais.

Paskirstyta paslaugos trikdymo ataka (angl. DDoS) išmaniosiose sutartyse gali būti įvykdoma keliais būdais, o vienas iš jų yra mokesčių resursų išnaudojimas blokų grandinės bloke (OWASP, 2023b). „Ethereum“ blokų grandinės atveju kiekvienas blokas turi tilpti į nustatytus bloko mokesčių limitus, o juos viršijus blokas nebus patvirtintas validatorių. Ši problema tiesiogiai susijusi su išmaniųjų kontraktų programinio kodo operacijų mokesčiais bei gali lemti nepasiekiamas išmaniąsias sutartis ir trikdyti blokų grandinės veiklą (OWASP, 2023a). Todėl, neefektyvūs ar nekorektiški algoritmai ne tik reiškia didesnius transakcijos vykdymo mokesčius, bet ir gali trikdyti išmaniosios sutarties vykdymą, tam tikrais atvejais net įšaldant išmaniojoje sutartyje esančias lėšas.



Šaltinis: Chainalysis Team (2024), https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2024/

1 pav. Kriptovaliutų vagysčių pagal atakos tipus 2023 metais žalos medianos

Taigi, saugumo aspektas išmaniųjų sutarčių kontekste ne mažiau svarbus už vykdymo mokesčius. Išmaniųjų sutarčių pažeidžiamumai turi tiesioginę sąsają su finansiniais nuostoliais bei vis dar yra aktuali problema.

Apžvelgiant išmaniųjų sutarčių optimizavimo problemas galima teigti, kad išmaniųjų sutarčių programinio kodo efektyvumas daro įvairiapusę įtaką išmaniosios sutarties vykdymui. Neoptimalus operacijų naudojimas lemia didesnius transakcijų vykdymo mokesčius. Be to, neoptimizuoto ar nekorektiško išmaniųjų sutarčių programinio kodo sukelti mokesčių limito pažeidimai gali sutrikdyti išmaniųjų sutarčių veikimą, taip įšaldant naudotojų ar sutarties savininkų lėšas šiose sutartyse.

Kitoje dalyje aptariami egzistuojantys greitaveikos problemų sprendimo būdai, padedantys optimizuoti išmaniųjų sutarčių diegimą ir vykdymą.

1.3. Išmaniųjų sutarčių optimizavimo metodai

Išmaniųjų sutarčių programinis kodas gali būti optimizuotas įvairiais būdais. Šiame poskyryje aptariami moksliniuose straipsniuose nagrinėti išmaniųjų sutarčių optimizavimo metodai bei techniniai pasiūlymai.

Vienas iš principų išmaniosioms sutartims optimizuoti – programinio kodo analizė. Neoptimaliai mokesčių prasme parašytas programinis kodas gali būti pakeičiamas tą patį funkcionalumą garantuojančiu, bet efektyviau veikiančiu kodu, o tai atliekama pritaikant įvairius būdus bei įrankius.

Vieną iš tokių įrankių – „GASOL“ – pristatė mokslininkai Albert, Correas, Gordillo, Román-Díez, ir Rubio (2020). Įrankis skirtas „Eclipse“ integruotos kūrimo aplinkos „Solidity“ programavimo kalba parašytoms išmaniosioms sutartims „Ethereum“ blokų grandinei analizuoti bei optimizuoti. Anot šių mokslininkų, „Solidity“ kompiliatorius gali įvertinti tik konstantines vykdymo mokesčių vertes, o joms esant parametrinėms išraiškoms – ne. „GASOL“ įrankis leidžia įvertinti šias mokesčių išraiškas ir apskaičiuoti parametrines reikšmes. Tai leidžia palyginti skirtingus sprendimus vykdymo mokesčių prasme ir padeda išvengti permokos už mokesčius. Be to, ši programinio kodo analizė leidžia automatiškai optimizuoti išmaniąsias sutartis saugyklos operacijų aspektu. Įrankis gali aptikti ir optimizuoti išmaniųjų sutarčių, neefektyviai naudojančių saugyklos operacijas cikluose, programinį kodą.

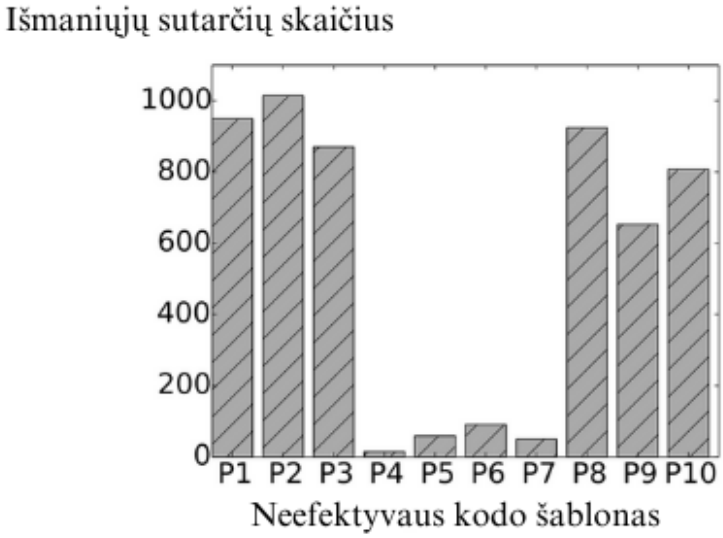
„GASOL“ įrankis orientuotas į „Solidity“ išmaniųjų sutarčių programinio kodo vykdymo mokesčių parametrinių išraiškų įvertinimą. Tai gali padėti įvertinti programinio kodo efektyvumą mokesčių prasme, numatyti bei išvengti išmaniųjų sutarčių pažeidžiamumų ir padidinti saugumą. Taip pat įrankis leidžia aptikti ir optimizuoti atliekamas saugyklos operacijas. Tokiu programinio kodo analizės principu būtų galima spręsti ir kitas išmaniųjų sutarčių programinio kodo problemas.

Panašaus tipo kodo analizę populiarių „Solidity“ išmaniųjų sutarčių bibliotekų programiniame kode atliko mokslininkai He, M. et al. (2024). Jie nagrinėjo bibliotekų programinio kodo pakeitimų istorijas, ieškodami su išmaniųjų sutarčių optimizavimu susijusių pakeitimų. Pagrindinės išskirtos problemų kategorijos – dėklo (angl. stack) problemos, atminties problemos,

saugyklos problemos, funkcijų argumentų duomenų (angl. calldata) problemos. Atrastų problemų sprendimai taip pat sukategorizuoti į duomenų saugojimo vietos pakeitimus, dėklo operacijų pakeitimus, skaičiavimų eliminavimą bei kitus. Pagal šias įžvalgas sukurtas „PeCatch“ statinės analizės įrankis „Solidity“ programavimo kalbos išmaniųjų sutarčių neefektyvaus programinio kodo aptikimui.

Abu įrankiai skirti „Solidity“ programavimo kalbos programinio kodo analizei atlikti. Atlikti tyrimai parodo, kad yra daug skirtingų programavimo praktikų, specifinių „Solidity“ programavimo kalbai, kurios gali padėti optimizuoti išmaniųjų sutarčių mokesčius.

Kiek kitokį principą pritaikė mokslininkai Chen, J. et al. (2020) ir išskyrė 10 neefektyvių išmaniųjų sutarčių programinio kodo metodų vykdymo mokesčių kontekste. Šiuos metodus sugrupavo į 4 pagrindines kategorijas – nenaudojamas programinis kodas, ciklai, nereikalingos saugyklos operacijos, neefektyvios operacijų sekos. Atlikti šių neefektyvių metodų paieškos rezultatai (2 pav.) parodė, kad nemaža dalis išmaniųjų sutarčių „Ethereum“ blokų grandinėje galėtų būti optimizuotos. Viso analizei naudoti 1500 išmaniųjų sutarčių EVM baitkodai. Šio įrankio skirtumas nuo anksčiau minėtųjų „GASOL“ bei „PeCatch“ yra, kad atliekama ne statinė „Solidity“ programinio kodo analizė, bet ieškomos neefektyvios jau sukompiliuoto EVM baitkodo sekos.



Šaltinis: Chen, J. et al. (2020) *GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts*, p. 10

2 pav. Neefektyvaus išmaniųjų sutarčių programinio kodo metodų analizės rezultatai

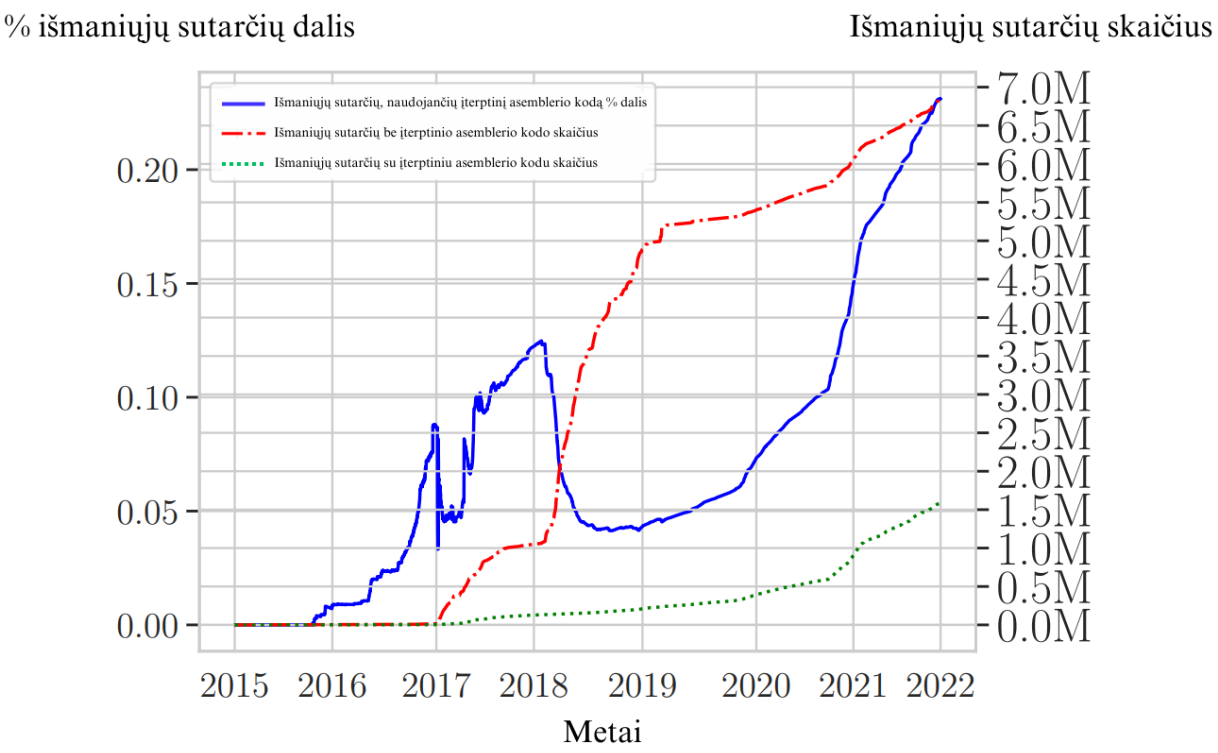
Išmaniųjų sutarčių programinio kodo optimizacijos problemos gali būti atliktos ne tik programinio kodo analizės, tačiau ir EVM baitkodo analizės būdu. Tyrimai parodė, kad nemaža dalis „Ethereum“ išmaniųjų sutarčių turi žinomų vykdymo mokesčių optimizacijos problemų, kas leidžia teigti, kad ši problema yra aktuali.

Dar vienas būdas „Ethereum“ išmaniųjų sutarčių, parašytų „Solidity“ programavimo kalba, optimizavimui yra įterptinio asemblerio kodo naudojimas. Remiantis „Solidity“ v.0.8.28

dokumentacija (The Solidity Authors, 2024), tam atlikti naudojama žemo lygio „Yul“ programavimo kalba. Pagrindiniai šios kalbos tikslai yra:

1. „Yul“ programinis kodas turėtų būti nesudėtingai skaitomas.
2. Valdymo srautas turi būti lengvai suprantamas.
3. „Yul“ programinio kodo transformacija į EVM baitkodą turėtų būti kuo tiesiogiškesnė.
4. „Yul“ kodas turėtų padėti optimizuoti „Solidity“ programas.

Anot mokslininkų Chaliasos, Gervais, ir Livshits (2022) atlikto mokslinio tyrimo, iš analizuotų 6.8 milijono „Ethereum“ blokų grandinės išmaniųjų sutarčių net 23% jų naudojo įterptinį asemblerio kodą. Pasak jų atliktos analizės, atliktos naudojantis „Etherscan“ išmaniųjų sutarčių naršykle, šis procentas yra augančios tendencijos (3 pav.). Taip pat anot šios analizės, įterptinis asemblerio kodas pagrinde naudojamas būtent programinio kodo optimizacijoms atlikti (2 lentelė). Taip pat pastebėta, kad įterptinio asemblerio kodo naudojimo ypatybės priklauso ir nuo programuotojų pasirinkimo, nėra konkrečios tendencijos kaip įterptinio asemblerio kodo fragmentų skaičius priklauso nuo sutarties sudėtingumo. Nors įterptinio asemblerio kodo naudojimas ir gali paspartinti kai kurį funkcionalumą, šis metodas taip pat turi trūkumų, nes naudojantis „Yul“ yra apeinami kai kurie saugumo patikrinimai. Taip pat, kadangi pats „Solidity“ kompiliatoriaus optimizavimo įrankis yra tobulinamas, ateityje įterptinio asemblerio kodui sugeneruotas EVM baitkodas gali tapti perteklinis. Be to, šio principo taikymas apsunkina išmaniųjų sutarčių saugumo auditus, nes įterptinis „Yul“ kodas yra sunkiau suprantamas ir kompleksiškesnis nei vien su „Solidity“ kalba parašytas išmaniųjų sutarčių programinis kodas.



Šaltinis: Chaliasos, S., Gervais, A., ir Livshits, B. (2022) A Study of Inline Assembly in Solidity Smart Contracts, p. 13

3 pav. „Ethereum“ blokų grandinės išmaniųjų sutarčių asemblerio kodo naudojimo statistika

2 lentelė

Įterptinio asemblerio kodo panaudojimo atvejų klasifikacija

Pobūdis	**Funkcionalumas nėra prieinamas „Solidity“**	**Optimizacija**
Instrukcija neprieinama „Solidity“	+	-
Deserializacija	+	+
String, bytes bei matematiniai įrankiai	+	+
Proxy įgyvendinimas	+	+
Klaidų valdymas	+	+
Kitų sutarčių kvietimas	-	+
Kriptografinių metodų optimizavimas	-	+
Duomenų tvarkymas	+	+
Tiksli atminties ir saugyklos kontrolė	+	+

2 lentelės tęsinys

Pobūdis	**Funkcionalumas nėra prieinamas „Solidity“**	**Optimizacija**
Rankinis asemblerio kodo eilutės įterpimas	-	+

Šaltinis: Chaliasos, S., Gervais, A. ir Livshits, B. (2022). A Study of Inline Assembly in Solidity Smart Contracts, p. 19

Taigi, įterptinio „Yul“ asemblerio kodo naudojimas gali padėti optimizuoti „Solidity“ išmaniąsias sutartis „Ethereum“ blokų grandinėje. Šis principas vis populiarėja, nepaisant jo keliamų saugumo ir kompleksiškumo iššūkių.

Be to, remiantis „Solidity“ v.0.8.28 dokumentacija (The Solidity Authors, 2024), „Ethereum“ blokų grandinės išmaniųjų sutarčių „Solidity“ programinis kodas kartu su įterptiniu „Yul“ asemblerio kodu yra automatiškai optimizuojamas „Solidity“ kompiliatoriaus. „Solidity“ kompiliatorius turi integruotą optimizavimo įrankį, kuris atlieka optimizacijas 3 lygmenimis:

1. Optimizacijos atliekant tiesioginę „Solidity“ kodo analizę.
2. „Yul“ tarpinės „Solidity“ programinio kodo reprezentacijos (angl. intermediate representation) kodo optimizavimas.
3. OPCODE sekų optimizacijos.

„Solidity“ programavimo kalba parašytos išmaniosios sutartys automatiškai optimizuojamos „Solidity“ kompiliatoriuje integruotu optimizavimo įrankiu, kuris veikia keliais skirtingais lygmenimis. Iš esmės „Solidity“ optimizavimo įrankis supaprastina kompleksiškas išmaniųjų sutarčių programinio kodo išraiškas, sumažina kodo apimtį, o kai kuriais atvejais ir vykdymo mokesčius.

Apibendrinant, „Ethereum“ išmaniosios sutartys turi nemažai skirtingų metodų, kuriais gali būti optimizuojamos. Šie metodai apima tiek „Solidity“ kalbos programinio kodo analizę, tiek sukompiliuoto EVM baitkodo analizę ir net įterptinį asemblerio „Yul“ kodą. Atlikti tyrimai parodė, kad nemaža dalis „Ethereum“ išmaniųjų sutarčių turi neefektyvaus programinio kodo fragmentų, kurie galėtų būti optimizuoti. Taip pat verta paminėti, kad pastebimos kylančios optimizacijos metodų, tokių kaip įterptinio asemblerio kodo naudojimo, tendencijos.

Toliau bus aptariami našieji kompiuterijos metodai, kurie galėtų būti pritaikyti išmaniųjų sutarčių vykdymo mokesčių optimizavimui.

1.4. Našiųjų kompiuterijos metodų analizė

Šio darbo kontekste našiaisiais kompiuterijos metodais laikomi sprendimai, kurie gali būti pritaikyti programinio kodo efektyvumui padidinti. Svarbu suprasti esminius tokių metodų principus, siekiant juos pritaikyti išmaniųjų sutarčių vykdymo mokesčių mažinimui.

1.4.1. Tradicinio programavimo metodai

Tradicinio programavimo metodai darbo kontekste atitinka skirtingus algoritmų tipus, kurie priklausomai nuo sprendžiamos problemos gali būti naudojami optimaliam sprendimui rasti.

Godieji algoritmai

Vienas iš bendrinių optimizacijos problemų sprendimo būdų yra godieji algoritmai. Tokių algoritmų principus nagrinėjo bei juos suklasifikavo Curtis (2003). Godžiųjų algoritmų principas gana tiesmukiškas – kiekvieno algoritmo žingsnio atveju rinktis tuo metu esantį tinkamiausią variantą kitam žingsniui. Priklausomai nuo sprendžiamos problemos, pritaikytas godusis algoritmas gali atrasti tikslų teisingą rezultatą, tačiau tam tikroms problemoms gali veikti tik euristiškai – tik artėjant prie optimalaus sprendimo. Dėl šios priežasties svarbu įvertinti sprendžiamą problemą, suprantant ar godusis algoritmas tinkamai ją sprendžia. Anot Curtis (2003), godieji algoritmai sudaryti iš 2 pagrindinių elementų – optimalaus lokalaus pasirinkimo kriterijaus ir dedamųjų žingsnių.

Taigi, godieji algoritmai yra paprastas optimizacijos problemų sprendimo būdas, kiekviename algoritmo žingsnyje pasirenkantis tuo momentu optimaliausią variantą. Priklausomai nuo sprendžiamos problemos šie algoritmai gali rasti optimalius sprendimus arba tik prie jų priartėti.

Lygiagretus programavimas

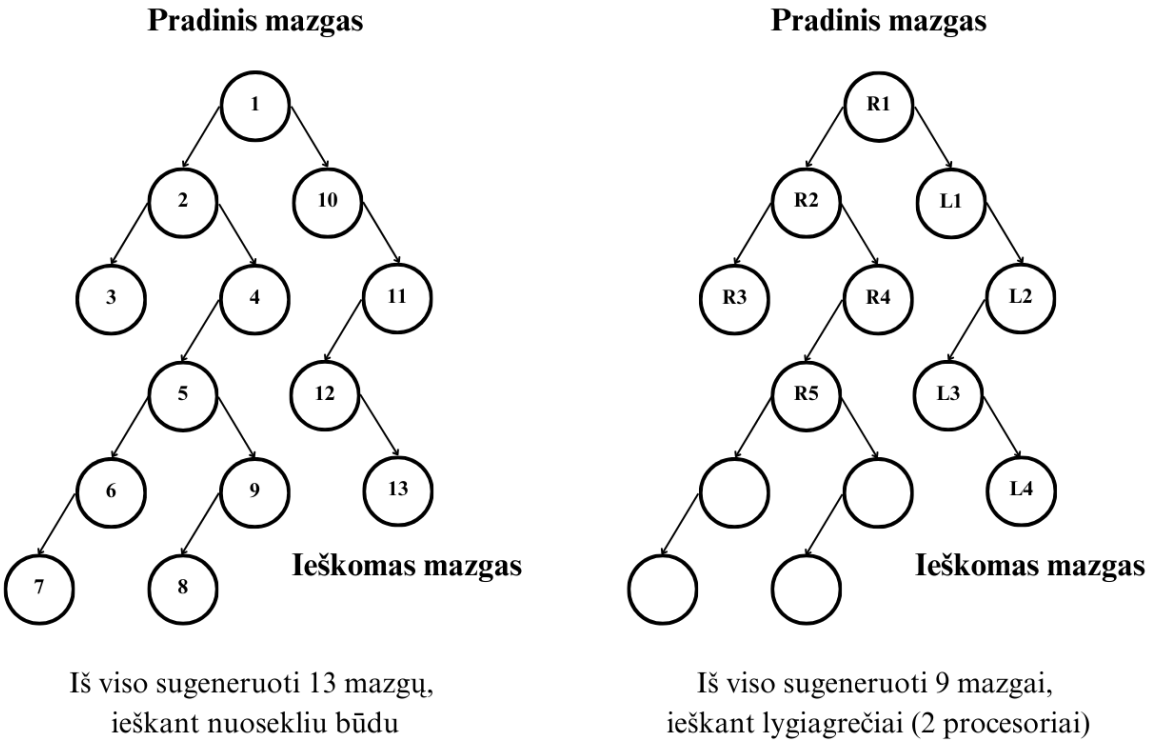
Mokslininkas Schnabel (1984) išnagrinėjo lygiagretaus programavimo architektūras ir pristatė šio programavimo principo galimybes algoritmų optimizavimo kontekste. Anot jo, lygiagretus programavimas atsirado dėl augančių programinės įrangos poreikių ir sudėtingėjančių jų sprendžiamų problemų. Taip pat šis principas padeda tokias problemas spręsti kuo pigesniu bei greitesniu būdu. Schnabel (1984) suprato, kad lygiagretus programavimas tuo metu buvo dar tik labai ankstyvoje stadijoje, tačiau gali būti pritaikytas visuotiniam optimizavimui atlikti.

Lygiagretaus programavimo ištakos atsirado dar praeito amžiaus 9 dešimtmetyje, jau tada buvo kalbama, kad lygiagretūs skaičiavimai gali padėti paspartinti neoptimalius nuoseklaus pobūdžio algoritmus.

Kiek vėliau, mokslininkai Kumar, Migdalas, ir Toraldo (2003) aptarė kitų mokslinių tyrimų metu nagrinėtus bei pasiūlytus lygiagrečių algoritmų, skirtų optimizacijai, sprendimus. Jie užsimena, kad žemo lygio lygiagretinimu laikomas toks sprendimas, kurio būdu identifikuojamos algoritmo operacijos galinčios vykti lygiagrečiai, nepakeičiant metodo sekos – algoritmas atliks tą patį sprendimą tik greičiau. Jie taip pat pristatė bei palygino kitus lygiagretinimo metodus ir pateikė išvadas, kad lygiagretus problemų sprendimas veda optimizacijos link.

Anot mokslininkų de Bruin, Kindervater, ir Trienekens (1995), nagrinėjusių lygiagrečių algoritmų anomalijas, kuriant lygiagretų algoritmą siekiama išvengti nuostolingos anomalijos, kai didinant lygiagrečiai vykdomų procesų skaičių įvyksta sprendimo sulėtėjimas. Vienas iš tokio pobūdžio pavyzdžių – paieškos gilyn algoritmo lygiagretinimas. Vykdant paieškos gilyn algoritmą

lygiagrečiai, priklausomai nuo ieškomo mazgo vietos, lygiagretus algoritmas gali veikti ir greičiau – reikalauja apeiti mažiau mazgų sprendimui rasti (4 pav.), ir lėčiau – kaip pateikta 5 pav.

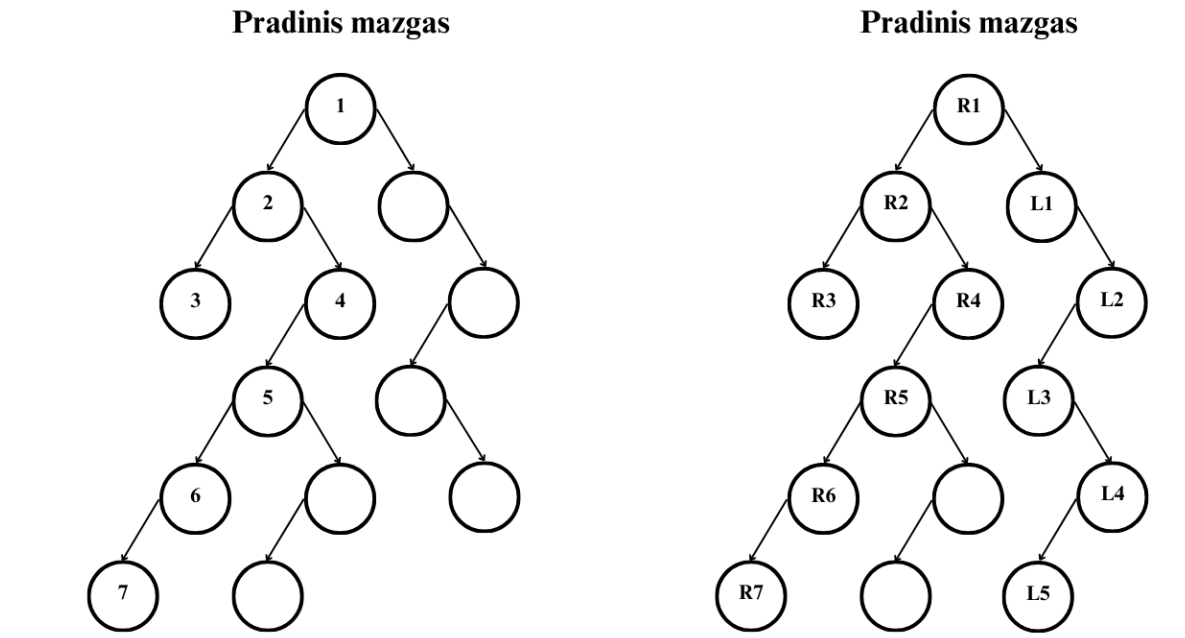


Iš viso sugeneruoti 13 mazgų, ieškant nuosekliu būdu

Iš viso sugeneruoti 9 mazgai, ieškant lygiagrečiai (2 procesoriai)

Šaltinis: http://users.atw.hu/parallelcomp/ch11lev1sec6.html

4 pav. Nuoseklaus ir lygiagretaus paieškos gilyn algoritmo palyginimas nr. 1



Šaltinis: http://users.atw.hu/parallelcomp/ch11lev1sec6.html

5 pav. Nuoseklaus ir lygiagretaus paieškos gilyn algoritmo palyginimas nr. 2

Apibendrinant, lygiagretūs sprendimai gali būti pritaikomi nuoseklių algoritmų optimizacijoms atlikti. Tačiau svarbu tinkamai juos realizuoti bei suprasti kaip išvengti nuostolingo lygiagretumo, siekiant pasiekti optimizuotą problemos sprendimą.

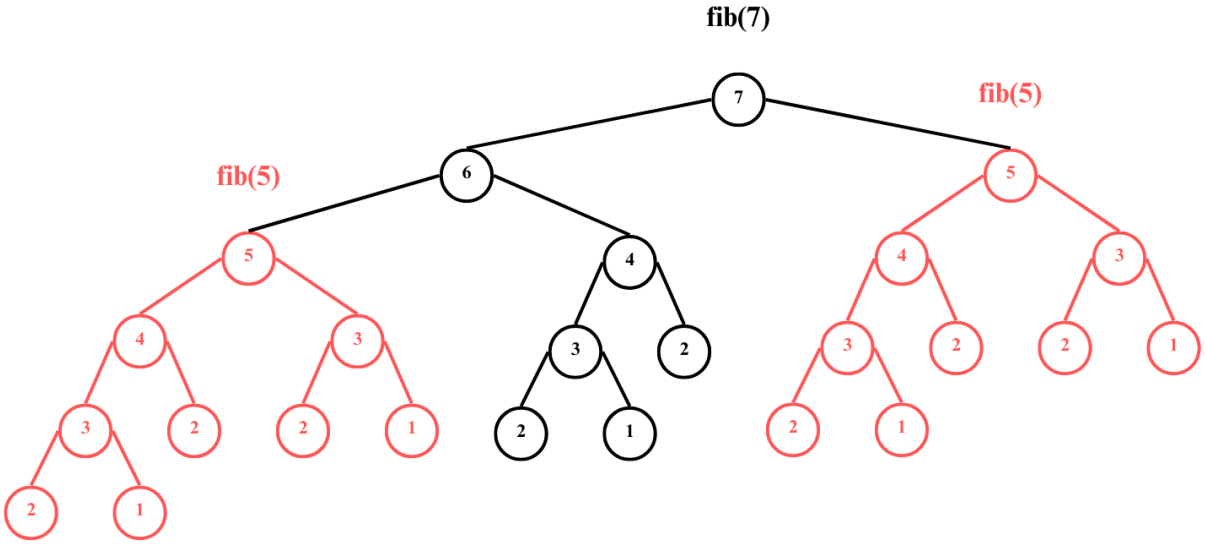
Dinaminis programavimas

Dinaminis programavimas yra dar vienas optimizavimo uždavinių sprendimo būdas. Šį principą pristatė Bellman (1957), kaip sprendimą problemoms, kurių pilno perrinkimo algoritmai turi daug persidengiančių užduočių aibių. Anot Rust (2006), dinaminio programavimo metodas, dar vadinamas atgaline indukcija, leido ekonomistams suformuluoti ir išspręsti didelę įvairovę problemų, susijusių su nuosekliu sprendimų priėmimu esant neužtikrintumams. Tai anot jo yra laikomas vienu svarbiausiu įrankiu ekonomikoje. Dinaminio programavimo algoritmai gali būti pritaikyti įvairiose srityse. Pavyzdžiui, sėkmingus tyrimo rezultatus pristatė mokslininkai He, S. et al. (2024), pritaikę dinaminius algoritmus energijos valdymo strategijų optimizavimui.

Dinaminio programavimo principas, pristatytas kaip nuoseklaus sprendimo priėmimo ir perrinkimo algoritmų keliamų problemų sprendimas, gali būti sėkmingai pritaikytas įvairiose srityse, kuriose reikalingi skaičiavimai su persidengiančiomis užduočių aibėmis.

Paprastas dinaminio programavimo algoritmų pritaikymo pavyzdys – Fibonačio skaičių sekos n-tojo elemento paieška. Kaip matoma 6 pav., norint gauti n-tojo Fibonačio sekos skaičiaus vertę, reikia susumuoti ankstesnius elementus. Ankstesnių elementų vertės apskaičiuojamos taip pat sumuojant prieš juos einančius sekos skaičius. Ši problema yra persidengianti, todėl jai spręsti gali būti pritaikytas dinaminis programavimas, kurio būdu nereikėtų pakartotinai skaičiuoti tų pačių sekos elementų kelis kartus.

Dinaminis programavimas gali būti pritaikytas problemoms, kurios turi persidengiančių užduočių aibes, pavyzdžiui Fibonačio skaičių sekos n-tojo elemento skaičiavimui. Tokiu būdu išvengiama nereikalingų pakartotinių skaičiavimų, taip paspartinant problemos sprendimo algoritmą.



Šaltinis: Ilyas, A. (2022), *Dynamic Programming: An Overview and Implementation*

6 pav. Fibonačio skaičių sekos n-tojo elemento paieškos išraiška

1.4.2. Išmaniųjų sutarčių metodai

„Ethereum“ išmaniųjų sutarčių kontekste tradicinio programavimo našieji kompiuterijos metodai gali neturėti tokio pačio našumo dėl EVM techninių specifikacijų, todėl svarbu suprasti kokių metodų taikymas gali būti aktualesnis išmaniosioms sutartims.

Anot mokslininko Beniiche (2020), blokų grandinės orakulai (angl. oracles) – tarpinė tarp išmaniųjų sutarčių blokų grandinėje bei realaus pasaulio duomenų. Iš esmės jie skirti išmaniosioms sutartims gauti duomenis, kurie nėra tiesiogiai pasiekiami blokų grandinėje, tačiau taip pat gali būti ir panaudoti sudėtingiems skaičiavimams, kurie nėra optimalūs transakcijos vykdymo mokesčių blokų grandinėje prasme, atlikti. Tai skaičiavimo orakulai, kurie gali būti naudojami intensyviems skaičiavimams atlikti už blokų grandinės ribų. Nors orakulų paslaugos yra teikiamos trečiųjų šalių, centralizuotumo išvengiama naudojant decentralizuotus orakulų tinklus, kurie veikia panašiais

principais kaip ir pati blokų grandinė. Tokiu būdu nėra pasitikima vienu informacijos šaltiniu, taip padidinant orakulų veikimo patitikimumą.

Pagrindinis našusis kompiuterijos metodas išmaniųjų sutarčių kontekste – orakulų naudojimas, leidžiantis gauti duomenis, kurie nėra laikomi blokų grandinėje, ar atlikti skaičiavimus, kurie būtų brangūs ar net neįmanomi dėl techninių blokų grandinės limitų. Skaičiavimų atlikimo orakuluose sprendimas literatūroje aptartas tik teoriniu pobūdžiu.

1.4.3. Apibendrinimas ir pritaikymo išmaniosioms sutartims galimybės

Apibendrinant, yra įvairių tradicinio programavimo našiųjų kompiuterijos metodų, galinčių padėti optimizuoti įvairių problemų sprendimus. Keli iš jų – godūs algoritmai, lygiagretus ir dinaminimis programavimas. Šie principai orientuoti į skirtingas problemas, todėl svarbu suprasti, ar sprendžiama problema atitinka jų keliamas sąlygas, kad galėtų būti jais optimizuota.

Lygiagretaus programavimo metodas jau yra nagrinėtas išmaniųjų sutarčių optimizavimo kontekste. Mokslininkai Dickerson et al. (2017), Ao et al. (2024) nagrinėjo bei pateikė pasiūlymus, kaip išmaniųjų sutarčių vykdymas lygiagrečiu būdu galėtų padidinti išmaniųjų sutarčių platformų transakcijų pralaidumą ir efektyvumą. Tiesa, verta paminėti, kad lygiagretumo principas šiuo atveju pritaikomas ne konkrečios išmaniosios sutarties algoritmo paralelizavimui, bet bloko transakcijų vykdymui ir validacijai lygiagrečiu būdu, kaip pavyzdžiui Yakovenko (2017) pristatytos „Solana“ blokų grandinės atveju. „Ethereum“ platformos atveju, lygiagretus išmaniosios sutarties programinio kodo vykdymas kol kas nėra techniškai įgyvendinimas, todėl šis metodas „Ethereum“ išmaniųjų sutarčių efektyvumo didinimui ar vykdymo mokesčių mažinimui nėra tinkamas.

Lygiagretus programavimas dėl EVM techninių limitų negali būti naudojamas išmaniųjų sutarčių programiniame kode. Godieji algoritmai ar dinaminis programavimas gali būti įvertinami sprendžiant išmaniųjų sutarčių vykdymo mokesčių problemas. Tuo tarpu orakulų panaudojimas galėtų būti naudojamas vykdymo mokesčių optimizacijoms atlikti, kartu pritaikant ir aptartus tradicinio programavimo našiuosius kompiuterijos metodus.

Kitame skyriuje remiantis atlikta „Ethereum“ išmaniųjų sutarčių teorine analize bus pateikiamas techninis jų vykdymo mokesčių mažinimo bei efektyvumo didinimo sprendimas.

2. „ETHEREUM“ IŠMANIŲJŲ SUTARČIŲ VYKDYMO MOKESČIŲ OPTIMIZAVIMO TECHNINIS PASIŪLYMAS

Šioje dalyje apžvelgiami optimizacijos problemų vykdymo išmaniųjų sutarčių kontekste niuansai bei pateikiamas siūlomas sprendimas konkrečios problemos sprendimo išmaniųjų sutarčių pagrindu vykdymo mokesčių sumažinimui. Nors aptariamas sprendimas tik vienai problemai, techninis pasiūlymas teoriškai gali būti pritaikytas ir kitoms problemoms išmaniųjų sutarčių kontekste spręsti.

2.1. Kuprinės (angl. knapsack) optimizacijos problema

Anot mokslininkų Assi, ir Haraty (2018), viena iš plačiausiai nagrinėjamų optimizavimo problemų – kuprinės problema. Darbo tyrimas orientuotas į šią problemą, nes ji plačiai išnagrinėta bei turi daug skirtingų sprendimo algoritmų. Šie algoritmai gali būti panaudoti palyginimui „Ethereum“ išmaniųjų sutarčių kontekste. Dėl techninių EVM savybių sprendimo algoritmų efektyvumas gali skirtis nuo tradiciniame programavime egzistuojančių rezultatų. Anot Assi, ir Haraty (2018), kuprinės problema yra kombinatorinio optimizavimo problema, kurios esmė iš elementų, kurių kiekvienas turi savo svorį bei vertę, sudaryti jų rinkinį, kuris neviršytų numatytos svorio ribos bei turėtų kuo didesnę bendrą vertę. O anot Cacchiani, Iori, Locatelli, ir Martello (2022), šios optimizacijos problemos pritaikymas svarbus įvairioms realaus pasaulio aplikacijoms – portfelių optimizavimui, logistikos problemoms, tiekimo grandinėms ir t.t., todėl problema taip pat aktuali ir išmaniųjų sutarčių kontekste.

Toliau aptariami skirtingi kuprinės optimizacijos problemos tipai, priklausantys nuo sprendžiamos užduoties aplinkybių bei sąlygų.

2.1.1. Kuprinės problemos tipai

Assi, ir Haraty (2018) išskyrė 4 pagrindinius kuprinės optimizacijos problemos pobūdžio tipus:

1. „0/1”.
2. Trupmeninė (angl. fractional).
3. Apribota (angl. bounded)
4. Neribota (angl. unbounded).

„0/1“ kuprinės problema

Pasak šių mokslininkų, „0/1“ kuprinės problemos variacija yra labiausiai paplitusi, reikalaujanti laikytis kelių esminių taisyklių:

1. Kiekvienas elementas turi svorį bei vertę.
2. Kuprinė turi svorio limitą.

3. Elementas arba gali būti pasirinktas visiškai, arba nepasirinktas iš viso.
4. Negalima rinktis to paties elemento daugiau nei vieną kartą.

Dėl šių sąlygų „0/1“ kuprinės problema laikoma diskrečiąja optimizavimo problema.

Trupmeninė kuprinės problema

Trupmeninė kuprinės problema iš principo panaši į „0/1“, o esminis jos skirtumas – elementas nebūtinai turi būti pasirinktas visiškai, arba nepasirinktas iš viso, o gali būti pasirinkta tik jo dalis. Dėl šio faktoriaus trupmeninė kuprinės problema laikoma nuolatine optimizacijos problema.

Apribota kuprinės problema

Apribota kuprinės problema yra „0/1“ problemos variacija, kurios išskirtinumas yra tai, kad kiekvienas elementas gali būti pasirinktas daugiau nei vieną, tačiau limituota kiekį kartų.

Neribota kuprinės problema

Neribota kuprinės problema neturi elemento pasirinkimo kiekio limito, todėl kiekvienas elementas gali būti pasirinktas tiek kartų, kiek tai yra reikalinga optimaliam sprendimui.

Visos problemos variacijos yra plačiai naudojamos ir turi skirtingus optimalius sprendimo algoritmus. Tyrimui atlikti pasirinktas „0/1” kuprinės optimizacijos problemos variantas. Tolesnėje dalyje aptariami egzistuojantys šios problemos sprendimo algoritmai.

2.1.2. „0/1“ kuprinės problemos algoritmai

Anot Hristakeva, ir Shrestha (2005), „0/1“ kuprinės problema turi įvairaus sudėtingumo sprendimo būdus – algoritmus. Tyrimui pasirinkta aptarti jų nagrinėtų algoritmų laiko bei atminties sudėtingumus ir panaudojimą „Ethereum“ išmaniųjų sutarčių vykdymo transakcijų mokesčių optimizavimo prasme. Pasirinkti nagrinėjimui algoritmai:

1. Rekursyvus algoritmas.
2. „Memoization“ algoritmas.
3. „Bottom-up“ algoritmas.
4. Optimizuotos vietos dinaminio programavimo algoritmas.

Dėl „0/1” kuprinės problemos pobūdžio, joks godusis algoritmas nėra visiškai tinkamas optimaliam sprendimui, nes anot Hristakeva, ir Shrestha (2005) šiuo algoritmu ne visada gaunamas optimalus – teisingas – rezultatas. Tai lemia euristinis algoritmo pobūdis. Dėl šios priežasties godusis algoritmas tyrime nebus nagrinėjamas. Kiti algoritmai, tokie kaip „branch and bound“, nėra tinkami, nes reikalauja sudėtingesnės duomenų struktūros – rikiuotų eilių – kurios „Solidity“ programavimo kalboje nėra prieinamos, o jos realizacija mokesčių prasme yra brangi.

Rekursyvus algoritmas

7 pav. pavaizduotas rekursyvaus algoritmo „0/1” kuprinės problemai spręsti pseudokodas. Iš pseudokodo galima suprasti, kad pagrindinis algoritmo principas yra rekursiškai iteruoti per kuprinės elementus. Jei elemento svoris neviršija likusios kuprinės vietos, lyginami 2 variantai – kai n-tasis

elementas yra dedamas į kuprinę, bei kai n-tasis elementas praleidžiamas – ir pasirenkamas didesnę bendrą elementų vertę sugeneruojantis variantas.

```
// Programinio kodo šaltinis: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
FUNCTION knapsack(W, wt[], val[], n):
    // Pagrindinis atvejis: jei nėra elementų arba kuprinės talpa lygi 0
    IF n == 0 OR W == 0:
        RETURN 0

    // Jei n-tojo elemento svoris didesnis už likusią kuprinės talpą
    IF wt[n-1] > W:
        RETURN knapsack(W, wt, val, n-1)

    // Kitu atveju, pasirenkamas dvejų atvejų maksimumas:
    // 1. Neįskaitant n-tojo elemento
    // 2. Įskaitant n-tąjį elementą (pridedama jo vertė ir iš talpos atimamas svoris)
    ELSE:
        RETURN Max(
            knapsack(W, wt, val, n-1),                      // Neįskaitant n-tojo elemento
            val[n-1] + knapsack(W - wt[n-1], wt, val, n-1)  // Įskaitant n-tąjį elementą
        )

// Valdantysis kodas
FUNCTION main():
    // Elementų verčių masyvas
    val[] = [60, 100, 120]

    // Elementų svorių masyvas
    wt[] = [10, 20, 30]

    // Maksimali kuprinės talpa
    W = 50

    // Elementų skaičius
    n = Length(val)

    // Pateikiama maksimali vertė
    PRINT(knapsack(W, wt, val, n))
```

Šaltinis: sudarytas autoriaus naudojant *https://carbon.now.sh/* įrankį

7 pav. „0/1" kuprinės problemos rekursyvaus algoritmo pseudokodas

„Memoization" algoritmas

8 pav. pavaizduotas „memoization" algoritmo „0/1" kuprinės problemai spręsti pseudokodas. Pagal pseudokodą matoma, kad taip pat naudojamas rekursinis principas. Skirtumas nuo paprasto rekursyvaus algoritmo atsiranda dėl to, kad maksimalios tarpinės į kuprinę įterptų elementų vertės yra išsaugomos ir nereikalauja būti perskaičiuojamos iš naujo – taikomas dinaminio programavimo principas.

```
// Programinio kodo šaltinis: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
FUNCTION knapsackRec(W, wt[], val[], n, memo[][]):
    // Pagrindinis atvejis: jei nėra elementų
    IF n < 0:
        RETURN 0

    // Jei rezultatas jau apskaičiuotas - grąžinamas
    IF memo[n][W] != -1:
        RETURN memo[n][W]

    // Jei n-tojo elemento svoris didesnis už likusią kuprinės talpą
    IF wt[n] > W:
        memo[n][W] = knapsackRec(W, wt, val, n-1, memo) // Praleidžiamas n-tasis elementas
    ELSE:
        // Lyginami du atvejai:
        // 1. Įskaitant n-tąjį elementą
        // 2. Neįskaitant n-tojo elemento
        memo[n][W] = Max(
            val[n] + knapsackRec(W - wt[n], wt, val, n-1, memo), // Įskaitant
            knapsackRec(W, wt, val, n-1, memo)                   // Neįskaitant
        )

    // Grąžinamas dabartinis rezultatas
    RETURN memo[n][W]

FUNCTION knapsack(W, wt[], val[]):
    n = LENGTH(wt)                          // Elementų skaičius
    INITIALIZE memo[n][W+1] FILLED WITH -1 // Dvimatis [n][W+1] dydžio masyvas su -1 vertėmis

    // Kviečiama rekursyvi funkcija paskutiniam elementui
    Return knapsackRec(W, wt, val, n-1, memo)

// Valdantysis kodas
FUNCTION main():
    // Elementų verčių masyvas
    val[] = [60, 100, 120]

    // Elementų svorių masyvas
    wt[] = [10, 20, 30]

    // Maksimali kuprinės talpa
    W = 50

    // Pateikiama maksimali vertė
    PRINT(knapsack(W, wt, val))
```

Šaltinis: sudarytas autoriaus naudojant *https://carbon.now.sh/* įrankį

8 pav. „0/1" kuprinės problemos „memoization" algoritmo pseudokodas

„Bottom-up" algoritmas

9 pav. pavaizduotas „bottom-up" algoritmo „0/1" kuprinės problemai spręsti pseudokodas. Šio algoritmo atveju nebeatliekami rekursyvūs funkcijos kvietimai, o iteraciniu būdu apskaičiuojamos galimos kuprinės elementų sumos vertės iteruojant per elementus ir maksimalų svorį nuo apačios aukštyn. Šis algoritmas taip pat pritaiko dinaminio programavimo metodiką.

```
// Programinio kodo šaltinis: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
FUNCTION knapsack(W, wt[], val[]):
    n = LENGTH(wt)  // Elementų skaičius
    INITIALIZE dp[n+1][W+1] FILLED WITH 0 // Dvimatis [n+1][W+1] dydžio masyvas su 0 vertėmis

    // dp masyvas pildomas „bottom-up" principu
    FOR i = 0 TO n:      // Iteruojama per elementus
        FOR w = 0 TO W:  // Iteruojama per talpas
            IF i == 0 OR w == 0:
                dp[i][w] = 0  // Pagrindinis atvejis: jei nėra elementų arba talpa lygi 0
            ELSE IF wt[i-1] <= w:
                // Lyginami atvejai įskaitant ir neįskaitant i-tojo elemento
                dp[i][w] = Max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w])
            ELSE:
                dp[i][w] = dp[i-1][w]  // i-tojo elemento svoris viršija likusią kuprinės talpą

    // Grąžinama maksimali vertė
    RETURN dp[n][W]

// Valdantysis kodas
FUNCTION main():
    // Elementų verčių masyvas
    val[] = [60, 100, 120]

    // Elementų svorių masyvas
    wt[] = [10, 20, 30]

    // Maksimali kuprinės talpa
    W = 50

    // Pateikiama maksimali vertė
    PRINT(knapsack(W, wt, val))
```

Šaltinis: sudarytas autoriaus naudojant *https://carbon.now.sh/* įrankį

9 pav. „0/1" kuprinės problemos „bottom-up" algoritmo pseudokodas

Optimizuotos vietos dinaminio programavimo algoritmas

10 pav. pavaizduotas optimizuotos vietos dinaminio programavimo algoritmo „0/1" kuprinės problemai spręsti pseudokodas. Iš pseudokodo pastebima, kad algoritmas veikia iteraciniu principu kaip ir „bottom-up" algoritmas, tik iteruojama per elementus ir svorį nuo viršaus žemyn. Be to, tarpiniai rezultatai saugomi ne matricoje, o vienmačiame masyve. Būtent dėl pastarosios savybės algoritmas ir vadinamas optimizuotos vietos dinaminio programavimo algoritmu.

```
// Programinio kodo šaltinis: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
FUNCTION knapsack(W, wt[], val[], n):
    // Sukuriamas W+1 dydžio masyvas maksimalioms kiekvienos talpos vertėms su 0 vertėmis
    INITIALIZE dp[W+1] FILLED WITH 0

    // dp masyvas pildomas „bottom-up" principu
    FOR i = 1 TO n:                 // Iteruojama per elementus
        FOR w = W DOWN TO wt[i-1]:  // Iteruojama per talpas atvirkštine tvarka
            IF wt[i-1] <= w:
                // Lyginami atvejai įskaitant ir neįskaitant i-tojo elemento
                dp[w] = Max(dp[w], dp[w - wt[i-1]] + val[i-1])

    // Grąžinama maksimali vertė
    RETURN dp[W]

// Valdantysis kodas
FUNCTION main():
    // Elementų verčių masyvas
    val[] = [60, 100, 120]

    // Elementų svorių masyvas
    wt[] = [10, 20, 30]

    // Maksimali kuprinės talpa
    W = 50

    // Pateikiama maksimali vertė
    PRINT(knapsack(W, wt, val, n))
```

Šaltinis: sudarytas autoriaus naudojant *https://carbon.now.sh/* įrankį

10 pav. „0/1" kuprinės problemos dinaminio programavimo algoritmo pseudokodas

Taigi, tik rekursyvus bei „memoization" algoritmai naudoja rekursyvius funkcijų kvietimus. EVM toks algoritmo pobūdis yra brangus mokesčių atžvilgiu. Kiekvienas funkcijos iškvietimas yra kainuojanti operacija, o funkcijų argumentų perdavimas priklausomai nuo duomenų pobūdžio taip pat reikalauja papildomų gas mokesčių. Abu algoritmai reikalauja perduoti duomenų masyvus. Saugant ir perduodant saugyklos duomenis jų kopijuoti nereikia bei perduodama jų nuoroda, tačiau duomenų skaitymas bei rašymas į saugyklą yra itin brangi EVM operacija – 100/2100 gas vienetų už skaitymą, 2900/20000 – rašymą (1 lentelė). Kitas būdas duomenims perduoti yra saugant juos atmintyje, nes operacijos su tokiais duomenimis yra žymiai pigesnės. Tačiau perduodant atminties duomenų masyvus jie yra kopijuojami, todėl priklausomai nuo duomenų kiekio taip pat išauga vykdymo mokesčiai bei perduodama tik jų vertė, o ne nuoroda.

Kita problema su rekursyviais algoritmais yra tai, kad EVM veikia dėklo struktūros principu, o kiekvienas rekursyvus funkcijos kvietimas kartu su argumentais yra saugomas dėkle. Dėklo limitas yra 1024 elementai, todėl rekursyvūs algoritmai su daug skaičiavimų gana greit pasiekia šį limitą, dėl ko rekursyvūs sprendimai nėra itin tinkami išmaniųjų sutarčių vykdymo kontekste.

Tolimesniame poskyryje analizuojami tyrimui pasirinktų algoritmų laiko bei atminties sudėtingumai.

2.2. Algoritmų analizė

Algoritmų efektyvumui bei optimalumui įvertinti ganėtinai svarbūs laiko bei atminties sudėtingumai. Laiko sudėtingumas leidžia suprasti kiek skaičiavimų bus reikalinga algoritmui įvykti bei kaip smarkiai išaugs algoritmo vykdymo laikas didėjant įvesties duomenų apimčiai. Atminties sudėtingumas parodo, kiek atminties reikalauja algoritmas.

Tolimesniuose skyreliuose palyginami pasirinktų algoritmų teoriniai laiko bei atminties sudėtingumai, leidžiantys daryti prielaidas apie optimalų „0/1“ kuprinės problemos algoritmą.

2.2.1. Algoritmų laiko sudėtingumas

Algoritmo laiko sudėtingumas leidžia įvertinti algoritmo operacijų skaičiaus priklausomybę nuo įvesties duomenų kiekio. Tai padeda geriau suprasti ir identifikuoti neefektyvius algoritmus, kurių vykdymo laikas augant duomenų kiekiui gali smarkiai išaugti.

3 lentelėje pateikti analizuojamų algoritmų laiko sudėtingumai. Pagal šiuos duomenis galima pastebėti, kad rekursyvaus algoritmo laiko sudėtingumas eksponentiškai priklauso nuo elementų skaičiaus, tačiau nepriklauso nuo maksimalaus svorio, o likę algoritmai turi tiesioginę priklausomybę nuo elementų skaičiaus ir svorio.

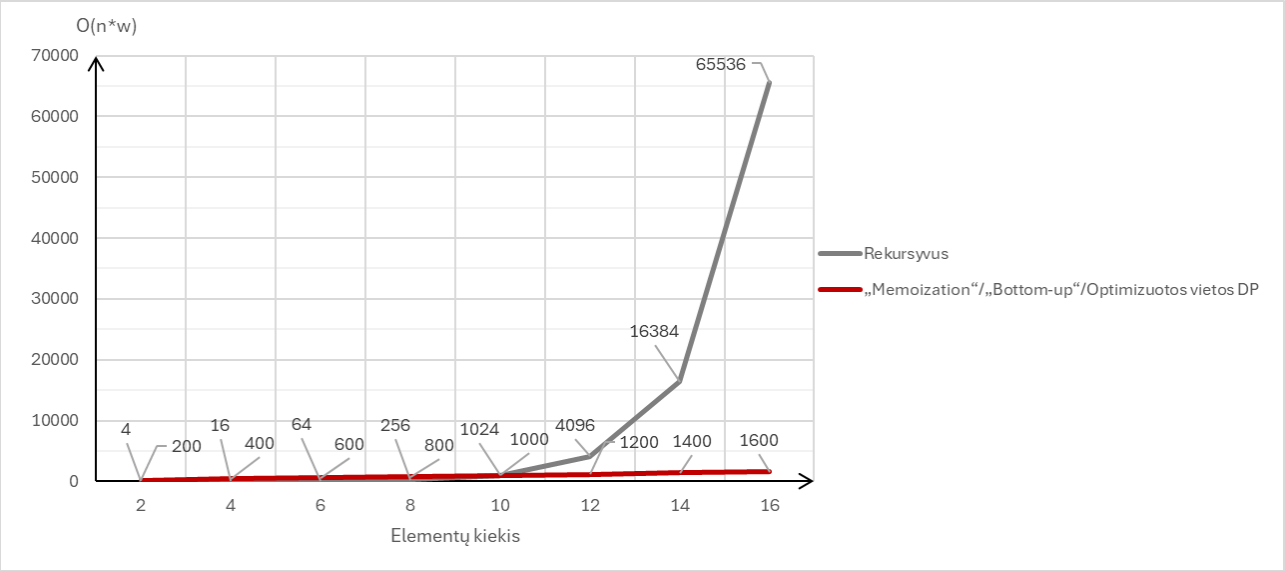
3 lentelė

„0/1” kuprinės problemos algoritmų laiko sudėtingumų palyginimas

Algoritmas	**Laiko sudėtingumas**
Rekursyvus	$O(2^n)$
„Memoization“	$O(n \times w)$
„Bottom-up“	$O(n \times w)$
Optimizuotos vietos dinaminio programavimo	$O(n \times w)$

Šaltinis: sudaryta autoriaus

Remiantis šiuo laiko sudėtingumų palyginimu ir 11 pav. pavaizduota priklausomybe nuo elementų kiekio galima teigti, kad rekursyvus algoritmas yra neoptimalus išmaniųjų sutarčių vykdymui esant dideliam elementų kiekiui. Tuo tarpu, kai elementų yra nedaug, o maksimalus leistinas svoris didelis, rekursyvaus algoritmo laiko sudėtingumas gali būti mažesnis už likusių algoritmų.



Šaltinis: sudaryta autoriaus

11 pav. Algoritmų laiko sudėtingumo priklausomybė nuo elementų kiekio (kai $w = 100$)

Taigi, laiko sudėtingumo atžvilgiu optimalus algoritmas priklauso nuo problemos duomenų kiekio. Esant dideliam elementų kiekiui neoptimalus tik rekursyvus algoritmas, o augant maksimaliam leistinam svoriui krenta kitų algoritmų optimalumas.

Kitame skyrelyje aptariamas algoritmų atminties sudėtingumas.

2.2.2. Algoritmų atminties sudėtingumas

Algoritmai, priklausomai nuo atliekamų operacijų ir duomenų turi skirtingus atminties sudėtingumus. Šis sudėtingumas parodo algoritmui įvykdyti reikalingos atminties priklausomybę nuo įvesties duomenų.

4 lentelėje pateikti algoritmų atminties sudėtingumai. Remiantis šiais duomenimis matoma, kad mažiausiai atminties reikalinga rekursyvaus ir optimizuotos vietos dinaminio programavimo algoritmams. Rekursyvus algoritmas tiesiogiai priklauso nuo elementų kiekio, nes jam įvykdyti reikalingi tik įvesties duomenų – elementų verčių ir svorių – masyvai. Tuo tarpu dinaminio programavimo algoritmas tiesiogiai priklauso nuo maksimalaus leistino svorio, nes algoritmo vykdymo metu skaičiuojamos visos tarpinės vertės nuo 0 iki maksimalaus svorio. Kitų algoritmų atminties sudėtingumas priklauso tiek nuo elementų kiekio, tiek nuo maksimalaus svorio, nes algoritmuose naudojami dvimačiai masyvai.

4 lentelė

„0/1 kuprinės problemos algoritmų atminties sudėtingumų palyginimas

Algoritmas	**Atminties sudėtingumas**
Rekursyvus	$O(n)$

4 lentelės tęsinys

Algoritmas	**Atminties sudėtingumas**
„Memoization“	$O(n \times w) + O(n)$
„Bottom-up“	$O(n \times w)$
Optimizuotos vietos dinaminio programavimo	$O(w)$

Šaltinis: sudaryta autoriaus

Sprendžiant pagal algoritmų atminties sudėtingumus, optimaliausi algoritmai – rekursyvus ir optimizuotos vietos dinaminio programavimo algoritmas. Tačiau EVM atveju svarbu ne tik reikalingas atminties kiekis, bet ir jos tipas – anksčiau minėta atmintis bei saugykla. Kadangi saugyklos operacijos daug kartų brangesnės už atminties, reikalinga įvertinti kurį atminties tipą naudoja pasirinkti algoritmai.

Tiek rekursyvaus, tiek dinaminio programavimo algoritmams pakanka naudoti paprastos atminties tipą. Tačiau, kadangi rekursyvus algoritmas perduoda įvesties duomenis rekursyviai, visi duomenys turi būti kopijuojami, tad nors atminties sudėtingumas ir nėra didelis, tačiau transakcijos vykdymo mokesčiai dėl to išauga. Tuo tarpu dinaminio programavimo algoritmo sprendimui užtenka paprastos atminties masyvo be papildomų atminties kopijavimo mokesčių. „Bottom-up“ algoritmas atminties atžvilgiu atitinka optimizuotos vietos dinaminio programavimo algoritmą, tik su didesniu sudėtingumu. Tuo tarpu „memoizaition“ algoritmo atveju, neužtenka paprasto atminties tipo, nes rekursyvioje funkcijoje duomenys turi būti ne tik perskaitomi, bet ir atnaujinami, todėl reikalinga brangesnė – saugyklos atmintis.

Apibendrinant, išmaniųjų sutarčių kontekste atminties atžvilgiu optimaliausias algoritmas – optimizuotos vietos dinaminio programavimo algoritmas – kurio atminties sudėtingumas tiesiogiai priklauso nuo maksimalaus svorio bei naudojama tik paprasta atmintis.

Kitame poskyryje aptariami alternatyvūs vykdymui „Ethereum“ blokų grandinėje sprendimai, pritaikant minėtuosius algoritmus.

2.3. Alternatyvūs sprendimai

Šiame poskyryje aptariami alternatyvūs sprendimai išmaniųjų sutarčių optimizavimui transakcijų vykdymo mokesčių prasme.

Kadangi dėl „Ethereum“ platformos taikomų mokesčių intensyvūs skaičiavimai išmaniosiose sutartyse yra brangūs, juos galima atlikti taikant egzistuojančias alternatyvas. Kuprinės problemos sprendimo atveju reikalingi duomenys bei pagrindinė išmaniųjų sutarčių logika galėtų būti laikomi „Ethereum“ blokų grandinėje, o algoritmo skaičiavimai atliekami kitur, naudojant orakulų funkcionalumą.

Vienas iš pagrindinių decentralizuotų orakulų tinklų (toliau angl. DON) – „Chainlink“, suteikia galimybę ne tik gauti duomenis, tačiau ir kreiptis į išorines programų sąsajas (angl. API) bei atlikti kitas operacijas už grandinės ribų (Breidenbach et al., 2021). Atskiri mazgų (angl. nodes) operatoriai, kurie valdo „Chainlink“ mazgus, sudaro tinklą, kuris adapterių pagalba leidžia atlikti operacijas už grandinės ribų decentralizuotu būdu. Tolimesniuose skyreliuose aptariamos galimos „Chainlink“ orakulų taikymo alternatyvos šiems skaičiavimams atlikti.

2.3.1. Cross-chain sprendimas

Mokesčių atžvilgiu pagrindinis „Ethereum” blokų grandinės tinklas yra ganėtinai brangus. Taip yra dėl didelės pagrindinio tinklo apkrovos bei techninių jos limitacijų. Tačiau yra kitų su EVM suderinamų grandinių, kurių transakcijų mokesčiai yra daug kartų mažesni nei „Ethereum”. EVM suderinamumas užtikrina tokį patį išmaniųjų sutarčių veikimą, todėl „Ethereum” grandinėje naudojamos išmaniosios sutartys taip pat veiktų ir kitose su EVM suderinamose grandinėse.

„Chainlink CCIP” – grandinių interoperatyvumo protokolas, suteikiantis galimybę išmaniųjų sutarčių, įdiegtų skirtingose grandinėse, tarpusavio komunikacijai. Šis funkcionalumas palaikomas daugumai populiariausių grandinių porų. Viena iš pagrindinių su EVM suderinamų grandinių – „BSC”, kurios vykdymo mokesčiai ženkliai pigesni nei „Ethereum” (vidutinės transakcijų kainos pateiktos 5 lentelėje), todėl ji gali būti naudojama šio cross-chain sprendimo įgyvendinimui.

5 lentelė

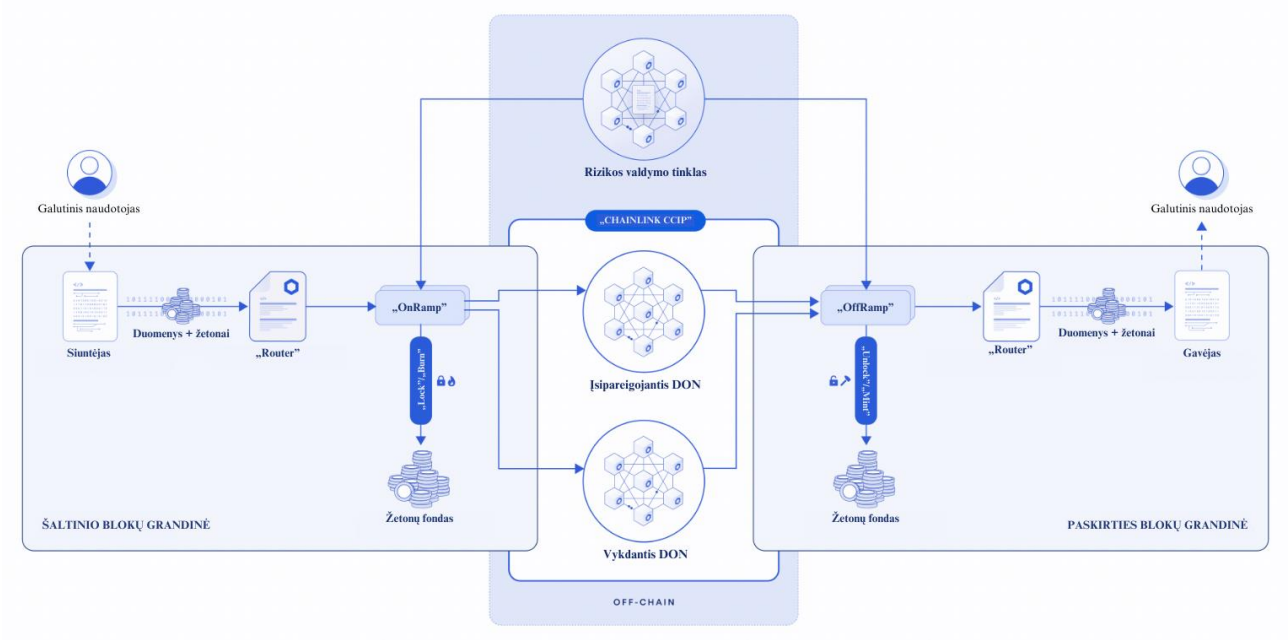
„Ethereum“ ir „BSC“ grandinių transakcijų kainų palyginimas

Grandinė	**„Ethereum”**	**„BSC”**
Vidutinė transakcijos kaina „USD“ (2024-12-14)	0.7815	0.1454
Vidutinė transakcijos kaina „USD“ (2024-12-13)	1.164	0.1411
Vidutinė transakcijos kaina „USD“ (2024-12-12)	1.148	0.2132
Vidutinė transakcijos kaina „USD“ (2024-12-11)	1.088	0.1351
Vidutinė transakcijos kaina „USD“ (2024-12-10)	1.485	0.1376
Vidutinė transakcijos kaina „USD“ (2024-12-09)	1.606	0.1491
Vidutinė transakcijos kaina „USD“ (2024-12-08)	0.8002	0.15

Šaltinis: sudaryta autoriaus remiantis https://ycharts.com/indicators/ethereum_average_transaction_fee ir https://ycharts.com/indicators/binance_smart_chain_average_transaction_fee_es duomenimis

„Chainlink CCIP” protokolas veikia asinchroninių žinučių principu. Aukšto lygio protokolo architektūra pateikta 12 pav. Išmanioji sutartis iš pagrindinės – šiuo atveju „Ethereum” – blokų grandinės per pateiktą „Router” sąsają gali kreiptis į kitą – pavyzdžiui „BSC” – grandinę. Tokiu būdu

galima siųsti duomenis ir žetonus, kuriuos patvirtinęs DON pristatys į paskirtą išmaniąją sutartį kitoje grandinėje. Kadangi žinutės asinchroninės – rezultatas negali būti grąžinamas iš karto, tačiau taip pat gali būti siunčiama CCIP protokolo žinutė su rezultatu, naudojant skirtingus „Router“ sąsajos ir žinutės parametrus.



Šaltinis: https://docs.chain.link/ccip/architecture

12 pav. „Chainlink CCIP“ protokolo architektūra

Nors transakcijos vykdymo mokesčiai už atliekamus skaičiavimus „BSC“ grandinėje ir yra mažesni, remiantis „Chainlink CCIP“ dokumentacija (Chainlink, 2024a), už cross-chain žinučių vykdymą „Chainlink“ orakulų operatoriams yra mokami papildomi mokesčiai, priklausantys nuo naudojamų tinklų, duomenų kiekio ir tipo. Be to, dėl asinchroninio komunikacijos principo ir egzistuojančio CCIP protokolo žinučių uždelsimo, toks skaičiavimo vykdymas laiko atžvilgiu yra lėtesnis už skaičiavimą vienoje išmaniojoje sutartyje.

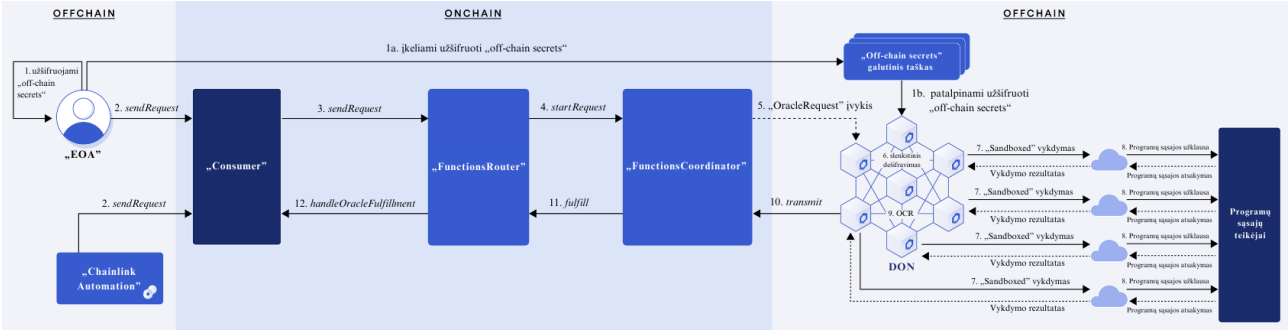
Kitame skyrelyje aptariamas skaičiavimo algoritmo realizavimo off-chain sprendimas.

2.3.2. Off-chain sprendimas

Siekiant sumažinti EVM taikomus išmaniosios sutarties vykdymo mokesčius, algoritmo skaičiavimo logika gali būti iškelta už grandinės ribų, o pati išmanioji sutartis tik laikytų reikalingus duomenis ir įgyvendintų pagrindinę išmaniosios sutarties logiką.

Komunikacija tarp „Ethereum“ išmaniųjų sutarčių bei off-chain skaičiavimų gali būti įgyvendinta naudojant „Chainlink Functions“ funkcionalumą. Remiantis „Chainlink Functions“ dokumentacija (Chainlink, 2024b), šis funkcionalumas leidžia iš blokų grandinės kviesti išorines programų sąsajas ar vykdyti kitą pasirinktinę logiką. Kaip ir „Chainlink CCIP“ atveju, funkcionalumas veikia DON principu, taip užtikrinant decentralizuotą veikimą bei duomenų

teisingumą, o jo architektūra pavaizduota 13 pav. Išmanioji sutartis kreipiasi į „Consumer“ sąsają, o toliau sukuriamas „OracleRequest“ įvykis. Šį įvykį apdoroja DON, izoliuotoje aplinkoje vykdydamas užklausą, įskaitant užklausas į reikalingas programų sąsajas, bei pateikdamas rezultatą atgal į išmaniąją sutartį.



Šaltinis: https://docs.chain.link/chainlink-functions/resources/architecture

13 pav. „Chainlink Functions“ architektūra

Siekiant nesumažinti decentralizuotumo, „Chainlink Functions“ skaičiavimas turėtų būti įgyvendintas ne išorinėje programų sąsajoje, o pačioje funkcijoje, kuri aprašoma naudojantis „JavaScript“ programavimo kalba. Kaip ir cross-chain sprendimo būdu, mokesčiai mokami ne tik už blokų grandinės transakcijos vykdymą, bet ir DON už off-chain logikos vykdymą bei rezultato grąžinimo transakcijos vykdymo mokesčius. Off-chain logikos vykdymo mokestis yra fiksuotas ir priklauso nuo naudojamos blokų grandinės bei tinklo. Kadangi galimas perduoti duomenų kiekis į „Chainlink Functions“ yra limituotas, o duomenų kopijavimas kainuotų papildomus mokesčius, įvesties duomenis „Chainlink Functions“ užklausos vykdymo metu galima gauti iš blokų grandinės, naudojant „Ethereum“ mazgus, pavyzdžiui naudojantis „Infura“ teikiamomis paslaugomis bei „Chainlink Functions Secrets“ funkcionalumu saugumui užtikrinti.

Kitame skyrelyje aptariamos galimos pasiūlytų sprendimų taikymo galimybės bei egzistuojantys sprendimų techniniai apribojimai.

2.4. Praktinės taikymo galimybės

Nors šiame skyriuje nagrinėjama kuprinės problema ir jos variacijos bei egzistuojantys algoritmai, pasiūlyti sprendimai yra bendrinio pobūdžio ir nėra orientuoti tik į konkrečios problemos sprendimą.

Išmaniųjų sutarčių vykdymo mokesčiai „Ethereum“ grandinėje priklauso nuo atliekamų operacijų skaičiaus ir pobūdžio, todėl praktiškai bet kurios problemos sprendimui yra aktualios vykdymo mokesčių sumažinimo galimybės. Pasiūlyti sprendimai turėtų leisti atlikti intensyvius skaičiavimus grandinėje pigiau, o „Chainlink Functions“ atveju taip pat atveriamos papildomos galimybės, tokios kaip išorinių programų sąsajų kvietimas ar grandinėje neprieinamų duomenų

manipuliacijos metodų pritaikymas. Dėl šios priežasties, pastarasis sprendimas galėtų būti taikomas realizuoti išmaniosioms sutartims, kurios:

1. Naudoja sudėtingus algoritmus (pvz. paieškos algoritmai, rikiavimas, grafų teorijos problemos).
2. Reikalauja kompleksiškų duomenų struktūrų, neprieinamų „Ethereum“ grandinėje (pvz. eilės, medžiai, grafai).
3. Reikalauja „Ethereum“ grandinėje neprieinamo funkcionalumo (pvz. sudėtingesnės matematinės operacijos, duomenų šifravimas ar kompresavimas, statistinės ar analitinės operacijos).
4. Reikalauja užklausų į išorines programų sąsajas.
5. Reikalauja apdoroti didelį kiekį duomenų.

Tačiau dėl egzistuojančių techninių ribojimų konkretūs sprendimai gali būti tinkami ne visoms problemoms spręsti. Šie ribojimai aptarti 6 lentelėje. Remiantis pateiktais duomenimis matoma, kad visiems sprendimams taikomas tam tikras išmaniosios sutarties transakcijos vykdymo gas limitas. Visi sprendimai turi įsitekti į „Ethereum“ grandinės šiuo metu nustatytą 30,000,000 gas vienetų limitą blokui. Cross-chain sprendimui taikomas papildomas 140,000,000 gas vienetų limitas „BSC“ grandinės išmaniąjai sutarčiai bei 3,000,000 gas vienetų limitas CCIP protokolo žinutės priėmimui. Off-chain atveju, „Chainlink Functions“ užklausos rezultato priėmimas „Ethereum“ grandinėje apribotas iki 300,000 gas vienetų. CCIP žinutėmis maksimaliai gali būti perduodama 30 kilobaitų duomenų, o „Chainlink Functions“ užklausa taip pat gali užimti tik 30 kilobaitų – įskaitant programinį „JavaScript“ kodą – ir grąžinti tik 256 baitus duomenų. Dėl šios priežasties itin svarbu, kad sprendžiamos problemos rezultatas būtų įmanomas grąžinti arba grandinėje išsaugomas kitu būdu. HTTP užklausos „Chainlink Functions“ sprendimui taip pat yra apribotos URL, užklausos bei atsakymo limitais. Taip pat HTTP užklausos vykdymui skiriamos 9 sekundės, o maksimalus „Chainlink Functions“ užklausos vykdymo laikas yra 10 sekundžių. Visų sprendimų programiniai kodai yra apriboti EVM išmaniosios sutarties kodo apimties limitu – negali viršyti 24 kilobaitų.

6 lentelė

Sprendimų techninių limitų palyginimas

Sprendimas	**On-chain**	**Cross-chain**	**Off-chain**
Gas limitas	30,000,000 gas vienetų „Ethereum“ grandinėje	30,000,000 gas vienetų „Ethereum“ grandinėje, 140,000,000 gas vienetų „BSC“ grandinėje, 3,000,000 gas vienetų CCIP protokolo žinutės priėmimui abejose grandinėse	30,000,000 gas vienetų „Ethereum“ grandinėje, 300,000 gas vienetų „Ethereum“ grandinėje užklausos rezultato priėmimui

6 lentelės tęsinys

Sprendimas	On-chain	Cross-chain	Off-chain
Perduodamo duomenų kiekio limitas	-	30 kilobaitų CCIP protokolo žinutėje	30 kilobaitų „Chainlink Functions“ užklausoje, 256 baitai rezultatui
HTTP užklausų limitas	-	-	5 HTTP užklausos per „Chainlink Functions” užklausą, 9 sekundės HTTP užklausos įvykdymui, 2048 URL simbolių limitas, 30 kilobaitų HTTP užklausos bei 2 megabaitai HTTP atsakymo limitai
Programinio kodo limitas	24 kilobaitai vienai išmaniąjai sutarčiai „Ethereum“ grandinėje	24 kilobaitai vienai išmaniąjai sutarčiai „Ethereum“ ir „BSC“ grandinėse	24 kilobaitai vienai išmaniąjai sutarčiai „Ethereum“ grandinėje

Šaltinis: sudaryta autoriaus remiantis Buterin, V. (2016), Chainlink (2024a, 2024b)

Taigi, prieš taikant konkretų sprendimą svarbu įvertinti, ar dėl jam taikomų techninių apribojimų jis yra tinkamas. Analizuotos „0/1” kuprinės problemos atveju visi sprendimai yra tinkami, tačiau apriboti galimu apdoroti duomenų kiekiu.

Tolimesniame skyriuje nagrinėjamos optimizuotos vietos dinaminio programavimo algoritmo „0/1” kuprinės problemai realizacijos, atliekant skaičiavimus aptartais – on-chain, cross-chain bei off-chain – metodais.

3. EKSPERIMENTINIS „ETHEREUM“ IŠMANIŲJŲ SUTARČIŲ VYKDYMO MOKESČIŲ OPTIMIZAVIMAS

Eksperimentas „Ethereum“ išmaniųjų sutarčių optimizavimui vykdymo mokesčių prasme atliktas išmaniąjai sutarčiai, atitinkančiai „0/1“ kuprinės problemą bei teorinį panaudojimo atvejį, aktualų blokų grandinės kontekste.

Eksperimento objektas

Pasirinktas panaudojimo atvejis – simuliuota DAO, kurios nariai gali teikti investicijos pasiūlymus su tikėtina metine grąža (angl. ROI) – kuprinės problemos elemento verte, bei reikalinga investicijos suma – kuprinės problemos elemento svoriu. Maksimalus leistinas svoris šio panaudojimo atveju yra bendras DAO kapitalas, o problemos tikslas – sulaukus visų investicinių pasiūlymų apskaičiuoti maksimalią investicinę grąžą iš pateiktų pasiūlymų ribotam kapitalui. Nors eksperimentui naudojama simuliuota išmanioji sutartis, toks pat sprendimas gali būti taikomas ir kitų problemų skaičiavimams išmaniosiose sutartyse optimizuoti. Siekiant realizuoti skirtingus skaičiavimo sprendimus, bet išlaikyti tokią pačią likusią išmaniosios sutarties logiką, naudojama abstrakti išmanioji sutartis (angl. abstract contract). Tokia sutartis gali turėti nerealizuotų funkcijų – šiuo atveju skaičiavimo algoritmo – o jos logiką gali paveldėti ir šias funkcijas realizuoti kitos išmaniosios sutartys – pasiūlyti problemos sprendimai.

Eksperimento eiga

Esminės eksperimento dalys, reikalingos rezultatams gauti ir įvertinti:

1. Abstrakčios išmaniosios sutarties, atitinkančios aptartą DAO, realizacija.
2. Techninio pasiūlymo dalyje aptartų „0/1” kuprinės problemos sprendimų – on-chain, cross-chain bei off-chain realizacija abstrakčiai sutarčiai.
3. Realizuotų sprendimų diegimas ir paruošimas vykdymui „Sepolia“ tinkle.
4. Sprendimų testavimas su skirtingais duomenų rinkiniais.
5. Sprendimų testavimo rezultatų palyginimas pagal išmaniųjų sutarčių diegimo mokesčius, vykdymo mokesčius bei kitus papildomus aspektus.

Išmaniųjų sutarčių simuliacija bei testavimas atliktas „Ethereum“ blokų grandinės „Sepolia“ testavimo tinkle, kuris atitinka pagrindinį „Ethereum“ tinklą technologiškai ir palaiko pagrindines integracijas su kitomis reikalingomis sistemomis. Testavimo tinkle naudojami testavimo žetonai. Eksperimentui reikalingi žetonai nemokamai gaunami naudojantis tam skirtais kranais (angl. faucets). „Sepolia“ testavimo tinklų žetonams gauti naudotas kranas https://sepolia-faucet.pk910.de/.

Tolimesniuose poskyriuose aptariamos atskiros atlikto eksperimento dalys.

3.1. Išmaniųjų sutarčių realizavimas

Išmaniosios sutartys, reikalingos eksperimentui atlikti, realizuojamos „Solidity“ programavimo kalba. Eksperimentui naudojama naujausia eksperimento atlikimo metu esanti „Solidity“ kompiliatoriaus versija 0.8.28, o išmaniosios sutartys palaiko kompiliatorių nuo 0.8.20 versijos. Programinis išmaniųjų sutarčių kodas rašomas, kompiliuojamas, diegiamas į grandines ir testuojamas naudojantis „Remix IDE“.

3.1.1. Abstrakti „0/1” kuprinės problemos išmanioji sutartis

Siekiant užtikrinti vieningą struktūrą tarp skirtingų sprendimų ir užtikrinti galimybę palyginti jų diegimo mokesčius, sukurta abstrakti išmanioji sutartis, reprezentuojanti DAO konceptą (7 priedas, 2 įrašas). 7 lentelėje pateiktas realizuotos išmaniosios sutarties funkcijų sąrašas. Šios funkcijos užtikrina bendrinę logiką DAO struktūrai atspindėti bei siūlymų teikimui atlikti ir skaičiavimų rezultatams gauti.

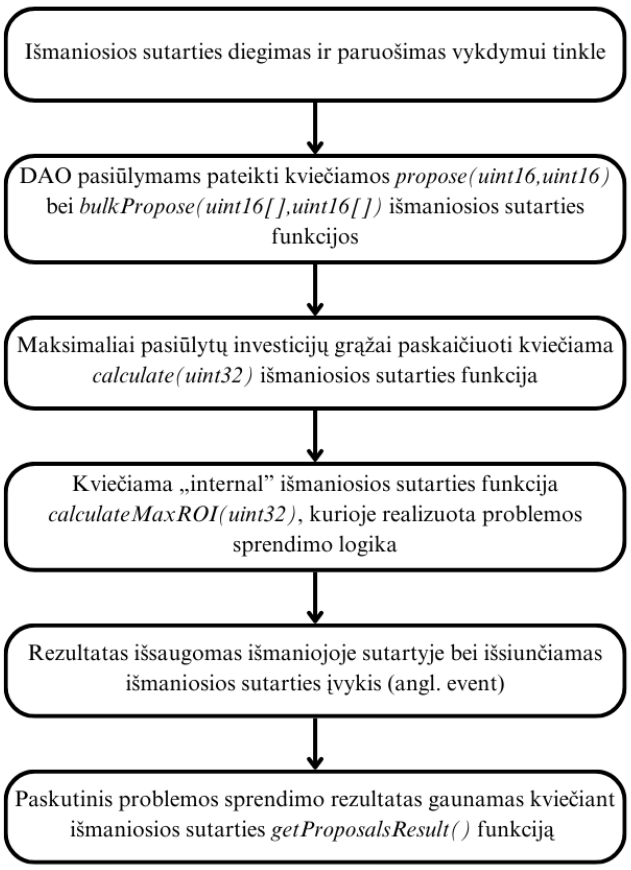
7 lentelė

Abstrakčios DAO išmaniosios sutarties funkcijų sąrašas

Funkcijos parašas (angl. signature)	**Paaiškinimas**
propose(uint16,uint16)	Leidžia pateikti DAO siūlymą. Priima investicijos grąžos (*uint16*) ir reikalingos sumos (*uint16*) argumentus
bulkPropose(uint16[],uint16[])	Leidžia pateikti kelis DAO siūlymus vienoje transakcijoje
calculate(uint32)	Inicijuoja kuprinės problemos rezultato skaičiavimą
getProposal(uint256)	Grąžina DAO siūlymą pagal indeksą
getProposalsCount()	Grąžina DAO siūlymų skaičių
getProposalsResult()	Grąžina paskutinį apskaičiuotą kuprinės problemos rezultatą
owner()	Grąžina kontrakto savininko adresą
resetProposals()	Panaikina DAO siūlymus bei ankstesnius kuprinės problemos rezultatus
calculateMaxROI(uint32)	Virtuali funkcija, kurios realizacija priklausys nuo pasirinkto sprendimo būdo

Šaltinis: sudaryta autoriaus

Pavyzdinė šios išmaniosios sutarties panaudojimo eiga atvaizduota 14 pav. Išmaniosios sutarties pernaudojimui keliems skaičiavimams su skirtingais duomenimis realizuota *resetProposals()* funkcija.



Šaltinis: sudaryta autoriaus

14 pav. Realizuotos DAO išmaniosios sutarties naudojimo pavyzdys

Abstrakčios išmaniosios sutartys negali būti diegiamos į „Ethereum" blokų grandinę. Tolimesniuose skyreliuose aptariami sukurtą abstrakčią išmaniąją sutartį paveldintys ir skaičiavimus realizuojantys sprendimai.

3.1.2. Algoritmo realizavimas on-chain sprendimo metodu

Šiame skyrelyje analizuojamas sprendimas, kuris bus naudojamas kaip eksperimento optimizacijų rezultatų vertinimo atskaitos taškas – skaičiavimų atlikimas „Ethereum" blokų grandinėje.

Paprasčiausias ir papildomų technologijų panaudojimo nereikalaujantis sprendimas – aprašyti algoritmo logiką pačioje „Ethereum" blokų grandinės tinkle diegiamoje išmaniojoje sutartyje. Remiantis atlikta sprendimo algoritmų analize, pasirinkta realizuoti optimizuotos vietos dinaminio programavimo algoritmą, kurio „Solidity" programinis kodas išmaniojoje sutartyje pavaizduotas 15 pav., o programinis kodas pateiktas 7 priede (3 įrašas).

```
    /// @dev Implements space-optimized dynamic programming algorithm
    function calculateMaxROI(uint32 _totalCapital) internal override {
        Proposal[] memory proposals = suggestedProposals;
        uint32[] memory dynamicROIs = new uint32[](_totalCapital + 1);

        for (uint16 i = 0; i < proposals.length; i++) {
            for (uint32 amount = _totalCapital; amount > 0; amount--) {

                uint32 partialAmount = proposals[i].investmentAmount;
                if (partialAmount <= amount) {
                    uint32 tempROI = dynamicROIs[amount - partialAmount] + proposals[i].estimatedROI;
                    if (tempROI > dynamicROIs[amount]) {
                        dynamicROIs[amount] = tempROI;
                    }
                }

        }

        uint32 result = dynamicROIs[_totalCapital];

        resultBlockNumber = block.number;
        proposalsCount = proposals.length;
        totalCapital = _totalCapital;
        maxROI = result;

        emit MaxROIResult(block.number, proposals.length, _totalCapital, result);
    }
```

Šaltinis: sudaryta autoriaus naudojant *https://carbon.now.sh/* įrankį

15 pav. Algoritmo realizacijos išmaniojoje sutartyje „Solidity“ programinis kodas

Šis sprendimas nereikalauja išorinių integracijų bei skaičiavimams nereikalingos papildomos funkcijos ir saugyklos kintamieji algoritmo veikimui. Aprašytas algoritmas su paveldėta abstrakčia išmaniąja sutartimi naudojantis „Remix IDE“ įdiegtas į „Sepolia“ testavimo tinklą. Diegimo transakcijos informacija pateikta 8 lentelėje. Gas mokestis, pateiktas gas vienetais, už diegimo transakciją mokamas „Sepolia“ tinklo „ETH“ žetonais, gautais anksčiau minėtuoju kranu.

8 lentelė

„Ethereum“ blokų grandinėje realizuoto sprendimo išmaniosios sutarties diegimo transakcija

Transakcijos hash	0x00f1163b5f86e58aa8a181bebf05b6d9ba66bdedefa1e282db1eaccc346c3d29
Išmaniosios sutarties adresas	0x74d991a4a4a294d18261c9b6c2835941c285353e
Transakcijos gas	1,183,462
Diegimo vykdymo gas	1,053,630
Diegiamo kodo gas	129,832

Šaltinis: sudaryta autoriaus remiantis https://sepolia.etherscan.io/ naršyklės duomenimis

Kitame skyrelyje aptariama cross-chain sprendimo „0/1” kuprinės problemos algoritmui realizacija.

3.1.3. Algoritmo realizavimas cross-chain sprendimo metodu

Cross-chain sprendimui įgyvendinti naudojamas aptartas „Chainlink CCIP” protokolas. Algoritmo vykdymui pasirinktas „BSC” blokų grandinės testavimo tinklas dėl mažesnių mokesčių ir

CCIP protokolo suderinamumo su „Sepolia” tinklu. Taip pat abiejų grandinių komunikacija CCIP protokolu palaikoma ir pagrindiniams tinklams.

Sprendimui realizuoti reikalingos 2 dalys:

1. Išmanioji sutartis su skaičiavimo algoritmu „BSC“ grandinėje.
2. Išmanioji sutartis su DAO logika „Ethereum” grandinėje.

Sukurta išmanioji sutartis „BSC” blokų grandinei, galinti priimti CCIP protokolo žinutes iš nurodytos grandinės (7 priedas, 5 įrašas). Priimama duomenų struktūra atitinka DAO išmaniojoje sutartyje teikiamų siūlymų struktūrą. Taip pat ši sutartis pritaikyta apskaičiuotus rezultatus išsiųsti į grandinę, iš kurios buvo gauta skaičiavimo užklausa. Išmaniosios sutarties diegimo transakcijos informacija pateikta 9 lentelėje. Diegiant „BSC” grandinėje gas mokestis parenkamas pagal naudojamo „BSC” grandinės tinklo gas mokestį ir yra mokamas „tBNB” žetonais. Šiuos žetonus nemokamai galima gauti pasinaudojant https://www.bnbchain.org/en/testnet-faucet kranu.

9 lentelė

Cross-chain sprendimo „BSC“ grandinės dalies išmaniosios sutarties diegimo transakcija

Transakcijos hash	0x1a56977fdc23920b74129a66155861eba05ad21fe4d56ccbfd5add8d198c7aff
Išmaniosios sutarties adresas	0x4773060273df3199cda8a6cb847cfa8144f02265
Transakcijos gas	2,553,943
Diegimo vykdymo gas	2,318,611
Diegiamo kodo gas	235,332

Šaltinis: sudaryta autoriaus remiantis https://testnet.bscscan.com/ naršyklės duomenimis

Taip pat realizuota išmanioji sutartis „Ethereum” blokų grandinei „Sepolia” tinkle (7 priedas, 4 įrašas). Išmanioji sutartis paveldi aprašytą abstrakčią DAO sutartį. Ji taip pat turi visą logiką skirtą žinučių siuntimui į kitą blokų grandinę bei rezultato žinučių priėmimui iš tos pačios grandinės, naudojantis CCIP protokolu. Sutarties diegimo transakcijos informacija pateikta 10 lentelėje.

10 lentelė

Cross-chain sprendimo „Ethereum“ grandinės dalies išmaniosios sutarties diegimo transakcija

Transakcijos hash	0xa773ac0bd5e5b8235b99d3d7c19746bc1d5e2bc934b9edc5085a2a805dfe97dd
Išmaniosios sutarties adresas	0x3417f29f8F5a848c541A65606E48e970c8eC3a8C
Transakcijos gas	3,029,973
Diegimo vykdymo gas	2,764,605

10 lentelės tęsinys

Diegiamo kodo gas	265,368

Šaltinis: sudaryta autoriaus remiantis https://sepolia.etherscan.io/ naršyklės duomenimis

Dėl saugumo, priimant CCIP protokolo žinutes abejose išmaniosiose sutartyse yra tikrinamas žinutės siuntėjo grandinės CCIP ID bei siuntėjo adresas. „Ethereum“ grandinės išmaniojoje sutartyje šios reikšmės nustatomos sutarties kūrimo metu, tuo tarpu „BSC“ grandinės išmaniosios sutarties vertės turi būti nustatomos atskiroje transakcijoje, kai jau žinomas „Ethereum“ grandinės sutarties adresas. Transakcijos informacija pateikta 11 lentelėje.

11 lentelė

Cross-chain sprendimo „BSC“ grandinės išmaniosios sutarties konfigūracijos transakcija

Transakcijos hash	0xbe234614e9af17abcf712585ebcdf42863cb35a3bd0a0e669738bcb20e84fa2a
Transakcijos gas	34,699

Šaltinis: sudaryta autoriaus remiantis https://testnet.bscscan.com/ naršyklės duomenimis

Testavimo tinklams skirti konfigūraciniai duomenys randami https://docs.chain.link/ccip/directory/testnet. Sutarčių diegimui naudotos konfigūracinės vertės pateiktos 12 lentelėje. Taip pat reikalingas „LINK“ žetono adresas, nes mokesčiai už CCIP protokolo žinutes mokami šiuo žetonu.

12 lentelė

„Chainlink CCIP“ konfigūraciniai duomenys išmaniosioms sutartims

Tinklas	**„Router“ adresas**	**„LINK“ žetono adresas**
„Sepolia“	0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59	0x779877A7B0D9E8603169DdbD7836e478b4624789
„BSC“ testavimo tinklas	0xE1053aE1857476f36A3C62580FF9b016E8EE8F6f	0x84b9B910527Ad5C03A9Ca831909E21e236EA7b06

Šaltinis: sudaryta autoriaus remiantis tekste minėtu šaltiniu

Šis sprendimas reikalauja daugiau pasiruošimo bei teisingų parametrų parinkimo, siekiant užtikrinti korektišką komunikaciją tarp skirtingose grandinėse įdiegtų sutarčių.

Kitame skyrelyje aptariama off-chain sprendimo realizacija.

3.1.4. Algoritmo realizavimas off-chain sprendimo metodu

Šio sprendimo principas algoritmo skaičiavimą atlikti ne blokų grandinėje, o naudojantis „Chainlink Functions“ teikiamu „JavaScript“ programinio kodo vykdymo funkcionalumu.

Sprendimui įgyvendinti reikalingos 4 dalys:

1. „Chainlink Functions” prenumerata.

2. Išmanioji sutartis su DAO logika „Ethereum" grandinėje.
3. „JavaScript" programinis kodas su problemos algoritmo logika.
4. „JavaScript" programiniam kodui reikalingų užšifruotų kintamųjų (angl. secrets) patalpinimas į „Gist".

Visų pirma, sukurta „Chainlink Functions" prenumerata „Sepolia" tinklui https://functions.chain.link/ puslapyje. Prenumerata papildyta „Sepolia" „LINK" žetonais, kurie naudojami mokesčiams už off-chain skaičiavimus bei rezultato grąžinimą į išmaniąją sutartį. Eksperimentui sukurta ir naudojama prenumerata – https://functions.chain.link/sepolia/3833.

Realizuota išmanioji sutartis su DAO logika „Sepolia" tinkle, siunčianti „Chainlink Functions" užklausas skaičiavimams bei priimanti ir išsauganti apskaičiuotus rezultatus (7 priedas, 6 įrašas). Sutarties diegimo transakcijos informacija pateikta 13 lentelėje.

13 lentelė

Off-chain sprendimo išmaniosios sutarties diegimo transakcija

Transakcijos hash	0xd8d60c3d5ec301062f2a83cd73037792123d03e5c05117eb4120f0f19526e984
Išmaniosios sutarties adresas	0xd9228A889C7bC5901BB344232A172586AaAbA94e
Transakcijos gas	3,684,268
Diegimo vykdymo gas	3,374,076
Diegiamo kodo gas	310,192

Šaltinis: sudaryta autoriaus remiantis https://sepolia.etherscan.io/ naršyklės duomenimis

Sukurtos išmaniosios sutarties adresas pridėtas kaip naudotojas https://functions.chain.link/sepolia/3833 konsolėje, kad išmanioji sutartis galėtų naudoti sukurtos „Chainlink Functions" prenumeratos „LINK" balansą.

Parašytas „JavaScript" programinis kodas, kurį išmanioji sutartis perduos į „Chainlink" DON algoritmo vykdymui (7 priedas, 9 įrašas). „JavaScript" programinis kodas kreipiasi į „Ethereum" mazgą įvesties duomenims gauti. Algoritmo skaičiavimai atliekami tiesiogiai „JavaScript" kode.

Kadangi kreipiamasi į „Ethereum" mazgo programos sąsają naudojantis https://www.infura.io/ teikiamomis paslaugomis, užklausoms reikalingas raktas (angl. API key). Dėl viešo „JavaScript" programinio kodo pobūdžio šis raktas turi būti išsaugotas kaip „off-chain secret". Tai atliekama naudojantis „GitHub" kodo dalijimosi platforma https://gist.github.com/ bei „npm" „@chainlink/functions-toolkit" paketu (7 priedas, 8 įrašas).

Išmaniosios sutarties veikimui reikalinga joje patalpinti algoritmo vykdymo „JavaScript" programinį kodą, užšifruotus „off-chain secrets" bei išmaniosios sutarties, iš kurios „JavaScript"

kodas gaus duomenis adresą. Šių duomenų patalpinimui į išmaniąją sutartį įvykdytos transakcijos informacija pateikta 14 lentelėje.

14 lentelė

Off-chain sprendimo išmaniosios sutarties konfigūravimo transakcija

Transakcijos hash	0x973c4be287868aed46c34c0af7390b287d3dfac4d2ab4181bf90af9d67424da2
Transakcijos gas	1,887,662

Šaltinis: sudaryta autoriaus remiantis https://sepolia.etherscan.io/ naršyklės duomenimis

Visi funkcijoms reikalingi konfigūraciniai duomenys randami https://functions.chain.link/. Sutarčių diegimui bei „off-chain secrets“ patalpinimui naudotos konfigūracinės vertės pateiktos 15 lentelėje. Kaip ir cross-chain sprendimo atveju, „LINK“ žetono adresas reikalingas, nes mokesčiai už „Chainlink Functions“ užklausas mokami šiuo žetonu iš sukurtos „Chainlink Functions“ prenumeratos balanso.

15 lentelė

„Chainlink Functions“ konfigūraciniai duomenys „Sepolia“ tinklo išmaniąjai sutarčiai

„Router“ adresas	**„LINK“ žetono adresas**	**DON ID**	**Prenumeratos ID**
0xb83E47C2bC239B3bf370bc41e1459A34b41238D0	0x779877A7B0D9E8603169DdbD7836e478b4624789	fun-ethereum-sepolia-1/0x66756e2d65746865726575 6d2d7365706f6c69612d31000 00000000000000000	3833

Šaltinis: sudaryta autoriaus remiantis tekste minėtais šaltiniais

„Sepolia“ tinkle įdiegta išmanioji sutartis paruošta off-chain sprendimo vykdymui. Kitame skyrelyje palyginami realizuotų sprendimų paruošimo mokesčiai.

3.1.5. Sprendimų realizacijų paruošimo vykdymui rezultatų palyginimas

Visų realizuotų sprendimų diegimo ir kitų, vykdymo paruošimui reikalingų, transakcijų „Sepolia“ bei „BSC“ testavimo tinkluose sunaudotas gas kiekis pateiktas 16 lentelėje. Remiantis šiais duomenimis galima pastebėti, kad on-chain sprendimas, nereikalaujantis jokių integracijų su išorinėmis sistemomis, diegimo mokesčių prasme yra pigiausias. Šiam sprendimui taip pat nereikalingos papildomos transakcijos vykdymo paruošimui. Cross-chain sprendimas „Sepolia“ tinkle sunaudojo ~2.5 kartų daugiau gas ir papildomai sunaudojo 2,588,642 gas algoritmo sutarties diegimui „BSC“ grandinėje. Tuo tarpu off-chain sprendimo išmaniosios sutarties diegimas kainavo 3,684,268 gas, o „JavaScript“ programinio kodo ir „off-chain secrets“ patalpinimas sutarties saugykloje kainavo papildomus 1,887,662 gas. Bendrai, visiškas off-chain sprendimo išmaniosios sutarties paruošimas vykdymui net ~4.7 kartų brangesnis už on-chain sprendimo sutarties diegimą.

16 lentelė

Realizuotų išmaniųjų sutarčių diegimo mokesčių palyginimas

Sprendimas	**On-chain**	**Cross-chain**	**Off-chain**
„Ethereum“ sutarties diegimo transakcijos gas mokestis	1,183,462	3,029,973	3,684,268
„Ethereum“ sutarties paruošimo transakcijos gas mokestis	-	-	1,887,662
Kiti mokesčiai	-	2,588,642 „BSC” grandinės gas	-

Šaltinis: sudaryta autoriaus

Kitame poskyryje atliekamas paruoštų sprendimų išmaniųjų sutarčių testavimas ir transakcijų vykdymo mokesčių palyginimas.

3.2. Realizuotų sprendimų vykdymo testavimas

Eksperimentinio optimizavimo tikslas yra įvertinti skirtingų problemos sprendimų vykdymo mokesčius. Kadangi vykdymo mokesčiai priklauso tiek nuo operacijų skaičiaus, tiek nuo duomenų kiekio, testavimas bus atliekamas su skirtingais duomenų rinkiniais.

3.2.1. Transakcijų vykdymas su testavimo duomenų rinkiniais

Operacijų kiekis algoritme tiesiogiai priklauso nuo algoritmo laiko sudėtingumo. Naudojamo optimizuotos vietos dinaminio programavimo algoritmo laiko sudėtingumas – $O(n \times w)$, kur n – DAO pasiūlymų skaičius (elementų kiekis), o w – bendras kapitalas (maksimalus svoris). Todėl duomenų rinkiniai parinkti taip, kad būtų galima įvertinti sprendimų vykdymo mokesčių priklausomybę nuo laiko sudėtingumo. Taip pat siekiant įvertinti duomenų kiekio įtaką vykdymo mokesčiams, bus palyginami 2 atvejai:

1. Transakcijos vykdymo mokesčių priklausomybė nuo laiko bei atminties sudėtingumų, kai n yra nekintantis (didėjant w).
2. Transakcijos vykdymo mokesčių priklausomybė nuo laiko sudėtingumo, kai w yra nekintantis (didėjant n).

„Chainlink CCIP“ žinutės bei „Chainlink Functions“ užklausos kartu su jų mokesčiais gali būti randamos tam skirtose naršyklėse:

1. „Chainlink CCIP“ žinutės – https://ccip.chain.link/.
2. „Chainlink Functions“ užklausos – https://functions.chain.link/.

Vykdymo mokesčių priklausomybė nuo laiko ir atminties sudėtingumų, kai n nekinta

Į realizuotas išmaniąsias sutartis transakcijomis supildomi testavimo duomenys. Pasirinktas testavimui duomenų kiekis n – 20. Kiekvienai sutarčiai atliekamas skaičiavimas su kintančiu w – 50, 100, 150, 200, 250, 500, 1000, todėl laiko sudėtingumas $O(n \times w)$ yra lygus atitinkamai 1000, 2000,

3000, 4000, 5000, 10000 bei 20000. Gauti sprendimų vykdymo mokesčių rezultatai šiam duomenų rinkiniui pateikti 16 pav., 17 pav., 18 pav.

On-chain sprendimo testavimas vyksta vienos transakcijos būdu, o rezultatas blokų grandinėje išsaugomas iš karto. Šio sprendimo atveju mokesčiai mokami tik už transakcijos „Ethereum“ grandinėje vykdymą „Sepolia“ tinklo „ETH“ žetonais. Vykdymo transakcijų duomenys pateikti 1 priede (1-7 įrašai).

Cross-chain sprendimas susideda iš dvejų žingsnių:

1. Transakcija „Ethereum“ grandinėje, kurios metu duomenys CCIP protokolu siunčiami į „BSC“ grandinę.
2. Transakcija „BSC“ grandinėje, kurios metu apskaičiuojamas rezultatas ir CCIP protokolo žinute grąžinamas į „Ethereum“ grandinę.

Šio sprendimo atveju mokami 3 skirtingi mokesčiai:

1. Transakcijos vykdymo mokestis „Ethereum“ grandinėje „Sepolia“ „ETH“ žetonais.
2. Transakcijos vykdymo mokestis „BSC“ grandinėje „tBNB“ žetonais.
3. Žinučių perdavimo tarp grandinių mokestis testavimo „LINK“ žetonais.

„Ethereum“ grandinės transakcijos pateiktos 2 priede (1-7 įrašai), o „BSC“ grandinės – 3 priede (1-7 įrašai), kartu pateikiami ir CCIP protokolo žinučių mokesčiai.

Off-chain sprendimas taip pat reikalauja tik vienos transakcijos „Ethereum“ grandinėje, kurios metu siunčiama „Chainlink Functions“ užklausa į DON, o apskaičiuotas rezultatas grąžinamas automatiškai kitos transakcijos metu. Šiuo metodu mokami „Ethereum“ grandinės transakcijos vykdymo mokesčiai „Sepolia“ „ETH“ žetonais bei „Chainlink Functions“ užklausos vykdymo mokestis testavimo „LINK“ žetonais. Išmaniosios sutarties transakcijų duomenys pateikti 4 priede (1-7 įrašai), o „Chainlink Functions“ užklausų – 5 priede (1-7 įrašai).

Tokiu pat principu tik su skirtingu duomenų rinkiniu atliekamas eksperimentas su nekintančiu *w* parametru.

Vykdymo mokesčių priklausomybė nuo laiko sudėtingumo, kai *w* nekinta

Šiam testavimo scenarijui pasirinktas *w* – 200. Kiekvienai sutarčiai atliekamas skaičiavimas su *n* – 5, 10, 15, 20, 25, 50, 100, todėl algoritmo laiko sudėtingumai $O(n \times w)$ atitinka ankstesnio eksperimento sudėtingumus. Gauti sprendimų vykdymo mokesčių rezultatai šiam duomenų rinkiniui pateikti 19 pav., 20 pav., 21 pav.

On-chain sprendimo testavimo šiuo duomenų rinkiniais transakcijų duomenys pateikti 1 priede (8-14 įrašai), cross-chain sprendimo – 2 ir 3 prieduose (8-14 įrašai), o off-chain sprendimo – 4 ir 5 prieduose (8-14 įrašai).

Dėl skirtingo transakcijų vykdymo laiko transakcijų vykdymo mokesčiai pateikti gas vienetais, kad neatsirastų netikslumo dėl skirtingų gas mokesčių, priklausančių nuo tinklo būsenos.

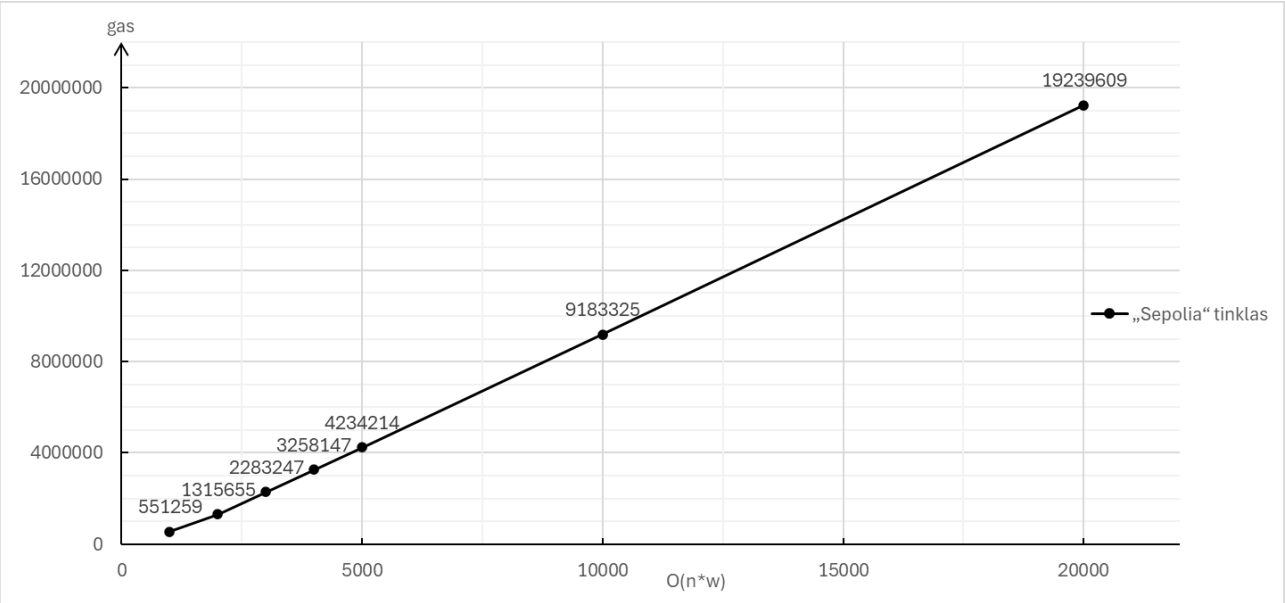
„Chainlink CCIP“ bei „Chainlink Functions“ taikyti mokesčiai pateikti testavimo „LINK“ žetonų išraiška, nes tik tokia informacija pateikiama „Chainlink CCIP“ žinučių bei „Chainlink Functions“ užklausų naršyklėse.

Kitame skyrelyje palyginami bei aptariami gauti sprendimų testavimo eksperimento rezultatai.

3.2.2. Testavimo rezultatai

Remiantis transakcijų duomenimis, gautais iš „Sepolia“ tinklo, „BSC“ testavimo tinklo, „Chainlink CCIP“ žinučių bei „Chainlink Functions“ užklausų naršyklių, sudaryti sprendimų vykdymo mokesčių priklausomybės nuo laiko sudėtingumo grafikai.

16 pav. pateikti on-chain sprendimo testavimo, su nekintančiu elementų skaičiumi rezultatai. Iš grafiko galima pastebėti tiesinės vykdymo mokesčių priklausomybės nuo laiko sudėtingumo tendenciją. Mažiausio laiko sudėtingumo ($O(n * w) = 1000$) atveju vykdymo transakcija sunaudojo 551,259 gas vienetų, o didžiausio sudėtingumo ($O(n * w) = 20000$) – 19,239,609. Tai parodo, kad on-chain sprendimas neefektyvus esant dideliam laiko sudėtingumui, nes gana greitai pasiekiamas „Ethereum“ grandinės gas limitas – 30,000,000 gas vienetų vienam blokui.

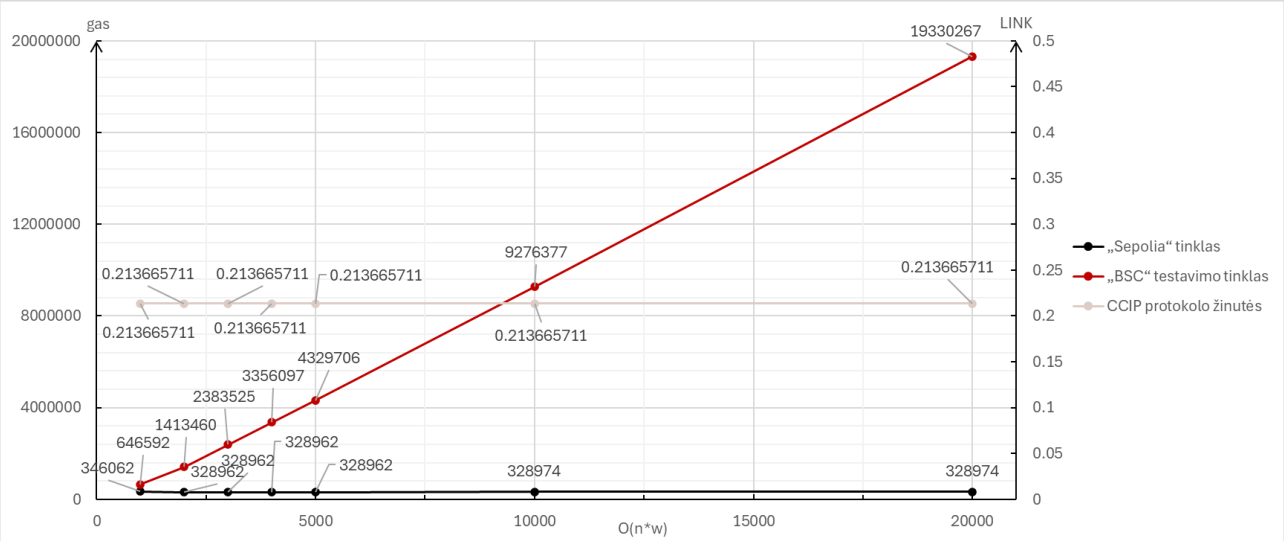


Šaltinis: sudaryta autoriaus

16 pav. On-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai n nekinta)

Cross-chain sprendimo testavimo su nekintančiu elementų skaičiumi rezultatai pavaizduoti 17 pav. Grafike vaizduojami 3 pagrindiniai mokesčiai – „Ethereum“ grandinės transakcijos vykdymo gas vienetai, „BSC“ grandinės transakcijos vykdymo gas vienetai ir „LINK“ mokestis už CCIP protokolo žinutes. „Ethereum“ grandinėje neatliekami skaičiavimai, o tik perduodami duomenys į CCIP protokolo žinutę, todėl grafike matoma, kad gas vienetai nepriklauso nuo algoritmo laiko

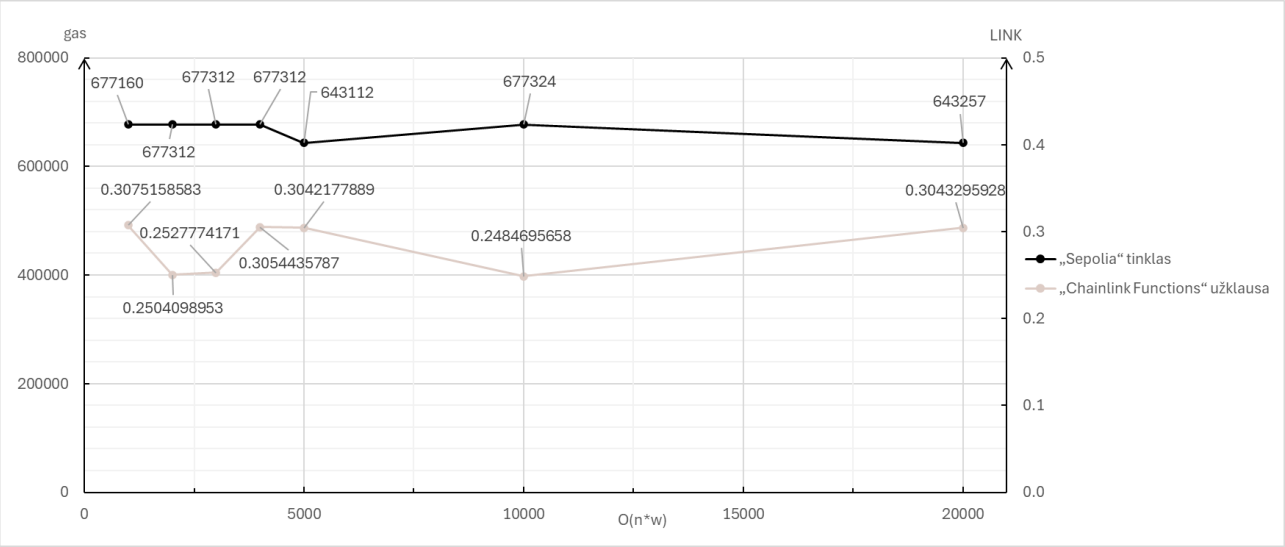
sudėtingumo vertės. Transakcijos vidutiniškai sunaudojo 331,408 gas vienetų nekintačio elementų kiekio ($n = 20$) perdavimui į „BSC“ grandinę. Kadangi elementų kiekis nesikeičia, už CCIP protokolo žinučių perdavimą sumokėti „LINK“ mokesčiai visoms transakcijoms buvo vienodi ir siekė 0.213665711 „LINK“ vienai transakcijai. Tuo tarpu „BSC“ grandinės transakcijos vykdymo mokesčiai, kaip ir on-chain sprendimo atveju tiesiogiai priklauso nuo algoritmo laiko sudėtingumo vertės. Gas vienetai siekė 646,592 bei 19,330,267 atitinkamai mažiausio bei didžiausio laiko sudėtingumo atvejais. Verta paminėti, kad nors šios vertės ir panašios į on-chain sprendimo gautus rezultatus, tačiau „BSC“ grandinėje sumokami mokesčiai, kaip pateikta 5 lentelėje, yra daug kartų mažesni.



Šaltinis: sudaryta autoriaus

17 pav. Cross-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai n nekinta)

Skaičiavimo už blokų grandinės ribų sprendimo rezultatai pavaizduoti 18 pav. Matoma, kad nei „Ethereum“ grandinės transakcijos vykdymo mokesčiai, nei už „Chainlink Functions“ užklausą mokami „LINK“ mokesčiai nepriklauso nuo algoritmo laiko sudėtingumo vertės. Pastebimi nežymūs mokesčių svyravimai, galimi dėl skirtingos išmaniosios sutarties bei tinklo būsenos transakcijos vykdymo metu. „Ethereum“ transakcija vidutiniškai sunaudojo 667,541 gas vienetų, o „Chainlink Functions“ vykdymo mokesčiai vidutiniškai siekė ~0.281881 „LINK“ per užklausą.

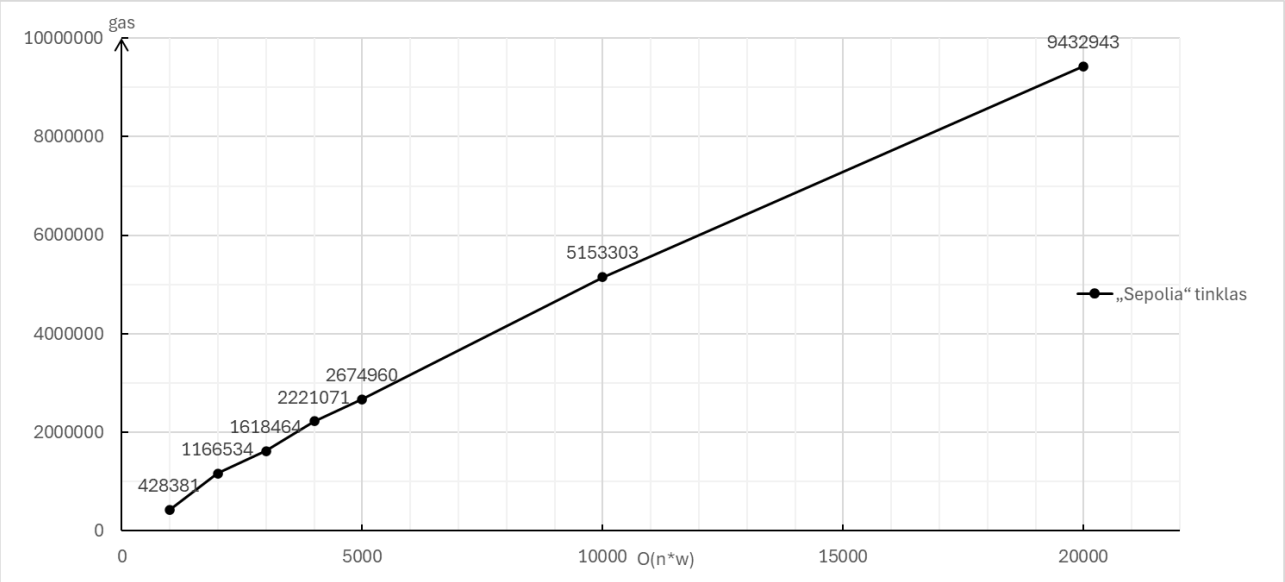


Šaltinis: sudaryta autoriaus

18 pav. Off-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai n nekinta)

Kitas testavimo scenarijus skirtas ištirti vykdymo mokesčių priklausomybę nuo algoritmo laiko sudėtingumo, kai nesikeičia maksimalus elementų svoris – *w*, o auga įvesties duomenų kiekis.

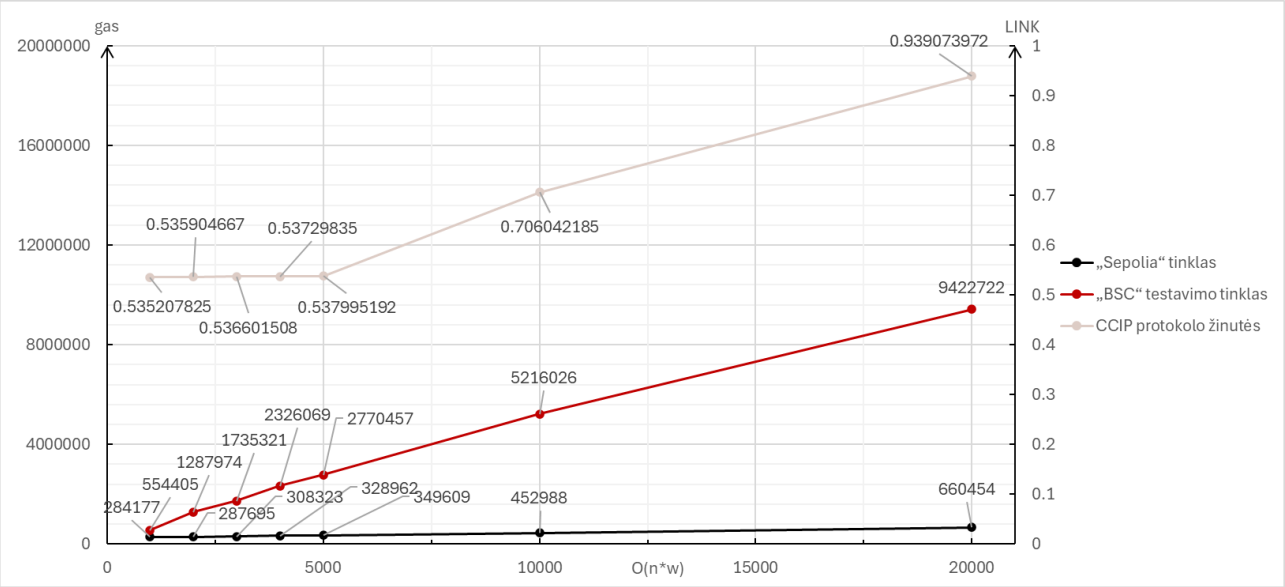
On-chain sprendimo rezultatai šiam scenarijui pateikti 19 pav. Šiuo atveju taip pat pastebima tiesioginė vykdymo mokesčių priklausomybė nuo laiko sudėtingumo – augant sudėtingumui, didėja transakcijos vykdymo mokesčiai. Duomenų rinkinio transakcijų mokesčiai išaugo nuo 428,381 iki 9,432,943 gas vienetų. Visų sudėtingumų atveju vykdymo mokesčiai buvo mažesni už atitinkamo sudėtingumo vykdymo mokesčius su nekintančių elementų skaičiumi. Taip galėjo nutikti dėl skirtingų duomenų įtakos operacijų skaičiui algoritme ar nekintančio atminties sudėtingumo $O(w) = 200$.



Šaltinis: sudaryta autoriaus

19 pav. On-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai w nekinta)

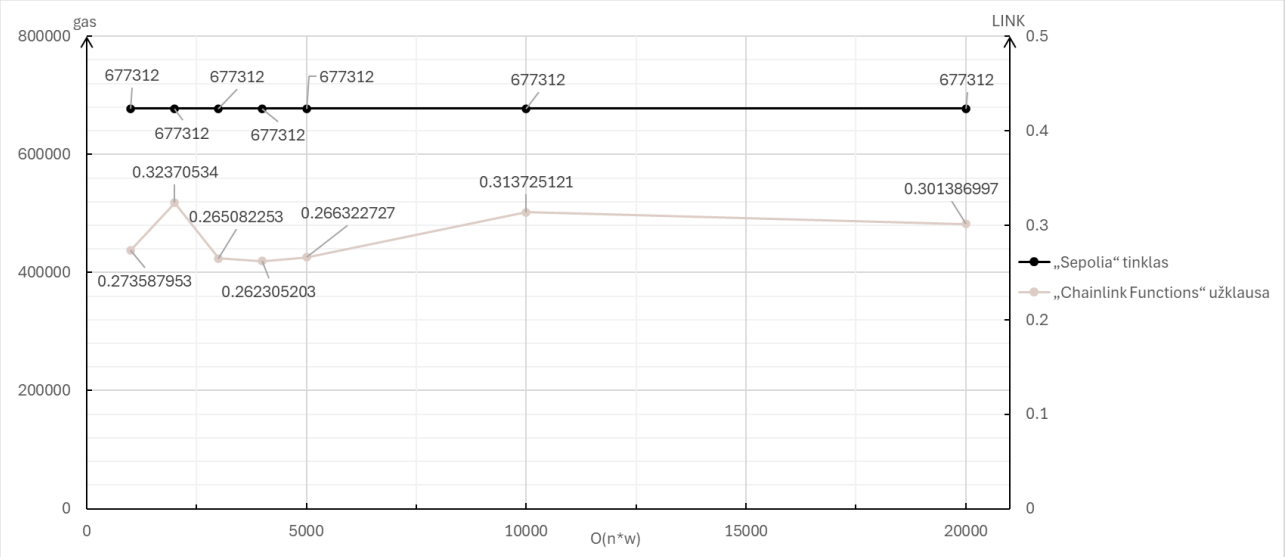
Cross-chain sprendimo atveju mokesčių priklausomybės tendencijos šiek tiek skiriasi nuo pirmojo scenarijaus. Remiantis 20 pav., „Ethereum“ transakcijos gas vienetai didėjant elementų skaičiui atitinkamai augo, nes į CCIP protokolo žinutę turi būti perduodama daugiau duomenų. „LINK“ mokesčiai už CCIP protokolo žinutes didėjant duomenų kiekiui taip pat augo dėl dviejų priežasčių – mokesčiai tiesiogiai priklauso nuo perduodamų duomenų kiekio bei daugiau gas sunaudojama šių duomenų išsaugojimui „BSC“ grandinėje. Mažiausio duomenų kiekio atveju ($n = 5$) šis mokestis siekė 0.535207825 „LINK“, didžiausio ($n = 100$) – 0.939073972 „LINK“. „BSC“ grandinės transakcijos vykdymo mokesčiai taip pat tiesiogiai priklauso nuo laiko sudėtingumo, tačiau kaip ir on-chain sprendimo metodu, sunaudoti gas vienetai žymiai mažesni. Mažiausio laiko sudėtingumo atveju ($O(n * w) = 1000$) siekė 284,177 gas vienetų, o didžiausio ($O(n * w) = 20000$) – 9,422,722 gas vienetų.



Šaltinis: sudaryta autoriaus

20 pav. Cross-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai w nekinta)

Off-chain sprendimo atveju vykdymo mokesčių tendencijos visiškai nesiskyrė nuo pirmojo testavimo scenarijaus rezultatų (21 pav.). „Ethereum“ grandinės transakcijos mokesčiai visais laiko sudėtingumo atvejais buvo lygūs 677,312 gas vienetų, o „Chainlink Functions“ užklausų mokesčiai nežymiai svyravo, vidutiniškai siekdami 0.286587 „LINK“ per užklausą.



Šaltinis: sudaryta autoriaus

21 pav. Off-chain sprendimo vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$, kai w nekinta)

Dar vienas aspektas apart vykdymo mokesčių, padedantis sprendimų efektyvumui įvertinti – greitaveika. 17 lentelėje pateikti sprendimų vykdymo greitaveikos rezultatai. Iš jų galima matyti, kad laiko atžvilgiu greičiausias sprendimas – on-chain. Off-chain sprendimo vykdymas taip pat sąlyginai

greitas – užklausos ir rezultato išsaugojimo transakcijas vidutiniškai skyrė mažiau nei minutė – 4 grandinės blokai, kas „Ethereum“ grandinės standartais trunka apie 48 sekundes. Tuo tarpu cross-chain sprendimo vykdymas užtruko ilgiausiai. Ilgiausiai šio sprendimo eigoje trunka CCIP protokolo žinutės perdavimas iš „Ethereum“ grandinės į „BSC“ grandinę – vidutiniškai apie 20 minučių. Tuo tarpu iš „BSC“ grandinės į „Ethereum“ grandinę rezultatas grąžinamas kelis kartus greičiau – vidutiniškai per 3,5 minutes. Todėl laiko atžvilgiu, esant poreikiui gauti skaičiavimo rezultatus greitai, labiausiai tinkami on-chain bei off-chain sprendimai.

17 lentelė

Sprendimų vykdymo trukmės palyginimas

Sprendimas	**Trukmė**
On-chain	- (iš karto)
Cross-chain sprendimo „Ethereum“-„BSC“ dalis	~1199 s (~20 min)
Cross-chain sprendimo „BSC“-„Ethereum“ dalis	~212 s (~3,5 min)
Off-chain	~48 s (4 „Ethereum” grandinės blokai)

Šaltinis: sudaryta autoriaus remiantis 1-4 prieduose ir paminėtose naršyklėse pateikta informacija

Apibendrinti visų sprendimų rezultatai pateikti 18 lentelėje. Diegimo ir paruošimo vykdymui mokesčių prasme on-chain sprendimas yra pigiausias. Cross-chain išmaniųjų sutarčių diegimas „Ethereum“ grandinėje yra kiek daugiau nei dvigubai brangesnis. Tuo tarpu off-chain sprendimo paruošimas yra beveik 5 kartus brangesnis už on-chain sprendimą, nes į grandinę patalpinamas visas „JavaScript“ programinis kodas, skirtas spręsti problemai. On-chain bei cross-chain sprendimų vykdymo mokesčiai priklauso nuo algoritmo laiko sudėtingumo vertės ($O(n * w)$). Augant abiems laiko sudėtingumo parametrams vykdymo mokesčiai taip pat tiesiogiai didėja. Cross-chain mokesčiai taip pat didėja augant duomenų kiekiui. Tuo tarpu off-chain sprendimo atveju mokesčiai išlieka stabilūs, nepriklausomai nuo duomenų kiekio ir laiko sudėtingumo vertės. On-chain sprendimas įvykdomas iš karto, off-chain sprendimo apdorojimas vidutiniškai užtrunka mažiau nei minutę (4 blokus), o cross-chain sprendimas vidutiniškai trunka ilgiau nei 20 minučių, todėl jis nėra itin tinkamas greitam rezultatų gavimui.

18 lentelė

Sprendimų testavimo rezultatų apibendrinimas

Sprendimas	**On-chain**	**Cross-chain**	**Off-chain**
Diegimo mokesčiai	1,183,462 „Ethereum“ gas	3,064,672 „Ethereum“ gas, 2,588,642 „BSC“ gas	5,571,930 „Ethereum“ gas

18 lentelės tęsinys

Sprendimas	**On-chain**	**Cross-chain**	**Off-chain**
Mokami vykdymo mokesčiai	„Ethereum“ gas	„Ethereum“ gas, „BSC“ gas, CCIP protokolo žinučių perdavimo mokestis	„Ethereum“ gas, „Chainlink Functions“ užklausos vykdymo mokestis
Vykdymo mokesčių priklausomybė nuo laiko sudėtingumo ($O(n * w)$)	Tiesiogiai priklauso	„BSC“ gas priklauso tiesiogiai, kiti nepriklauso	Nepriklauso
Vykdymo mokesčių priklausomybė nuo įvesties duomenų kiekio	Nepriklauso	„Ethereum“ gas ir CCIP protokolo žinučių mokesčiai priklauso tiesiogiai, „BSC“ gas nepriklauso	Nepriklauso
Vykdymo trukmė	Iš karto	> 20 min	< 1 min

Šaltinis: sudaryta autoriaus

Atsižvelgiant į šiuos aspektus, renkantis tinkamą sprendimą svarbu įvertinti įvairius problemos aspektus. Jei rezultatas reikalingas momentiškai, algoritmo laiko sudėtingumas yra nedidelis ar skaičiavimas nėra daugkartinis, galbūt pakankamas sprendimas – on-chain skaičiavimas „Ethereum“ grandinėje. Išaugus duomenų kiekiui ar laiko sudėtingumui, off-chain ar cross-chain sprendimas gali leisti smarkiai sumažinti išmaniosios sutarties skaičiavimo transakcijos vykdymo mokesčius. Cross-chain sprendimas kiek sudėtingesnis, nes reikalinga skaičiavimų užklausų ir rezultatų sinchronizacija tarp grandinių, o taip pat galimas perduodamas duomenų kiekis labiau apribotas nei off-chain atveju. Be to, skaičiavimai „BSC“ grandinėje taip pat limituojami bloko mokesčių limitu, todėl sprendimas nėra tinkamas itin dideliems skaičiavimams. Remiantis vykdymo rezultatais, „Chainlink Functions“ sprendimas yra geresnė alternatyva skaičiavimams „Ethereum“ grandinėje už „Chainlink CCIP“ sprendimą ir leidžia sumažinti „Ethereum“ išmaniųjų sutarčių vykdymo mokesčius.

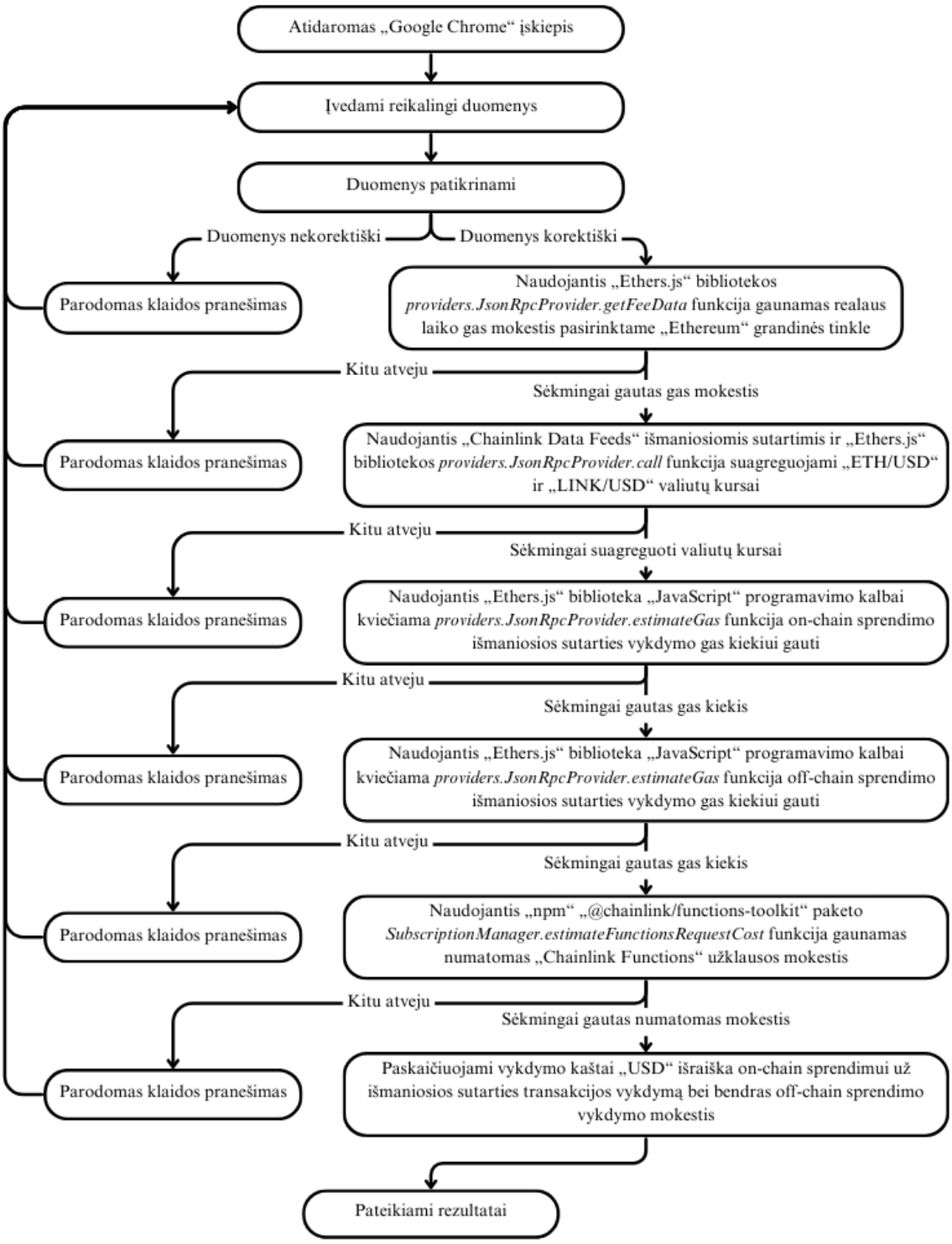
Be to, realizuotas off-chain sprendimas padidino eksperimentui naudotos „Ethereum“ išmaniosios sutarties efektyvumą. Esminis skirtumas yra tai, kad skaičiavimai šiuo sprendimo būdu nėra apriboti „Ethereum“ grandinės bloko mokesčių limitu. Šis metodas iš esmės suteikia galimybę realizuoti itin kompleksiškas išmaniąsias sutartis „Ethereum“ blokų grandinėje, nes intensyvūs skaičiavimai naudojant „Chainlink Functions“ užklausas gali būti atliekami už grandinės ribų. Dėka DON veikimo principo neprarandamas ir decentralizuotumo principas, taip išlaikant aukštus išmaniųjų sutarčių saugumo standartus.

Kadangi esant nedideliam duomenų kiekiui ar laiko sudėtingumui mokesčiai už transakcijos vykdymą skaičiuojant „Ethereum“ grandinėje gali būti mažesni už bendrus off-chain sprendimo mokesčius, tinkamesnis sprendimas galėtų būti pasirenkamas dinamiškai, priklausomai nuo duomenų kiekio ir kitų vykdymo mokesčiams aktualių aspektų. Kitame poskyryje aptariamas sukurtas prototipas, leidžiantis realiu laiku palyginti on-chain bei off-chain sprendimų vykdymo mokesčius.

3.3. Prototipas dinaminiam „Ethereum“ išmaniųjų sutarčių sprendimų vykdymo mokesčių įvertinimui

Remiantis realizuotų sprendimų techniniais aspektais sukurtas „Google Chrome“ įskiepis, leidžiantis realiu laiku palyginti numatomus on-chain bei off-chain sprendimų vykdymo mokesčius.

Įskiepio veikimo diagrama pateikta 22 pav. Įskiepio veikimui naudojama „Ethers.js“ biblioteka ir „npm“ „@chainlink/functions-toolkit“ paketas „JavaScript“ programavimo kalbai. Be to, naudojamos „Chainlink Data Feeds“ išmaniosios sutartys valiutų kursams gauti (reikalingi adresai pasiekiami naršyklėje adresu https://docs.chain.link/data-feeds/price-feeds/addresses?network=ethereum). Numatomi vykdymo mokesčių rezultatai pateikiami „USD“ išraiška.



Šaltinis: sudaryta autoriaus

22 pav. Realizuoto „Google Chrome“ įskiepio veikimo diagrama

Siekiant naudotis įskiepiu reikalingos „Ethereum“ grandinėje įdiegtos išmaniosios sutartys bei naudojimui turi būti paruošta „Chainlink Functions“ prenumerata. Šių sprendimų pavyzdžiai aprašyti 3.1.2 bei 3.1.4 skyreliuose. Įskiepio naudojimosi vadovas pateiktas 6 priede.

Testuojant eksperimento off-chain sprendimo realizaciją bei sukurtą įskiepį pastebėta, kad numatomas „Chainlink Functions“ užklausos mokestis tiek minėtoje „Chainlink Functions“ užklausų

naršyklėje, tiek įskiepyje yra didesnis nei realus mokestis, nuskaitytas po užklausos išpildymo. Remiantis „Chainlink Functions“ dokumentacija (Chainlink, 2024b) tai gali nutikti dėl 2 priežasčių:

1. Užklausos metu nurodomas didesnis užklausos rezultato grąžinimo funkcijos gas limitas nei iš tikro sunaudojama gas rezultatui grąžinti.
2. „Chainlink Functions“ užklausos metu rezervuoja prenumeratos „LINK“ balansą skaičiuojant pagal didesnį nei realų tuo momentu gas mokestį. Taip apsisaugoma, kad išaugus gas mokesčiui tarp užklausos išsiuntimo ir rezultato grąžinimo, rezervuotų „LINK“ žetonų užtektų užklausos išpildymui įvykdyti.

Remdamasis gautais rezultatais naudotojas gali realiu metu įvertinti ir pasirinkti sprendimą su mažesniais vykdymo mokesčiais.

IŠVADOS

1. Išnagrinėjus „Ethereum“ blokų grandinės išmaniųjų sutarčių technines specifikacijas išsiaiškinta, kad neefektyviai parašytas išmaniųjų sutarčių programinis kodas lemia didesnius vykdymo mokesčius. Pagrindinės problemos susijusios su nekorektišku atminties ar saugyklos naudojimu cikluose, pertekliniais skaičiavimais ar neoptimaliais palyginimais, o visos atliekamos operacijos apmokestinamos tam tikru gas vienetų kiekiu.
2. Aptarti egzistuojantys išmaniųjų sutarčių programinio kodo optimizavimo metodai bei išnagrinėti našieji kompiuterijos metodai, kurie galėtų padėti sumažinti išmaniųjų sutarčių vykdymo mokesčius bei padidinti efektyvumą. Teoriškai išanalizuoti tradicinio programavimo kompiuterijos metodai – godieji algoritmai, lygiagretus programavimas, dinaminis programavimas – bei išmaniųjų sutarčių kontekste egzistuojantys metodai – orakulai. Išsiaiškinta, kad skaičiavimo orakulai galėtų būti pritaikyti skaičiavimų atlikimui už blokų grandinės ribų, o skaičiavimams realizuoti pritaikomi minėtieji tradicinio programavimo metodai – godieji ir dinaminio programavimo algoritmai.
3. Išanalizuota pavyzdinė „0/1“ kuprinės optimizacijos problema, aptartas jos aktualumas išmaniųjų sutarčių kontekste ir palyginti jos sprendimui taikomi algoritmai – rekursyvusis, „memoization“, „bottom-up“ bei optimizuotos vietos dinaminio programavimo algoritmas. Aptarus EVM technines specifikacijas įvertinta, kad optimizuotos vietos dinaminio programavimo algoritmas yra tinkamiausias realizacijai išmaniosiose sutartyse vykdymo mokesčių prasme. Pasiūlytos alternatyvos išmaniųjų sutarčių optimizacijai atlikti. „Chainlink CCIP“ protokolo panaudojimas su „BSC“ grandine leistų skaičiavimus vykdyti pigesnėje „BSC“ grandinėje. Pasiūlytas „Chainlink Functions“ sprendimas, kuris algoritmo skaičiavimus vykdytų už grandinės ribų.
4. Aptartas „0/1“ kuprinės problemos pritaikymo išmaniųjų sutarčių kontekste pavyzdys – DAO. Eksperimentui atlikti sukurta jos koncepciją atitinkanti išmanioji sutartis bei realizuoti 3 skaičiavimų sprendimai – on-chain skaičiavimas optimizuotos vietos dinaminio programavimo algoritmu „Ethereum“ grandinėje, cross-chain skaičiavimas naudojantis „Chainlink CCIP“ protokolu „BSC“ grandinėje ir „Chainlink Functions“ pritaikymas skaičiavimams už grandinės ribų. Visi sprendimai realizuoti „Ethereum“ „Sepolia“ testavimo tinkle. Išmaniųjų sutarčių diegimo mokesčių prasme, pigiausias sprendimas – on-chain, kurio diegimo transakcija sunaudojo 1,183,462 „Ethereum“ gas vienetų. Cross-chain sprendimo diegimas buvo ~2.5 kartų brangesnis „Ethereum“ grandinėje – 3,064,672 gas vienetų – bei kainavo papildomus 2,588,642 „BSC“ grandinės gas vienetų. Off-chain sprendimo diegimas buvo beveik 5 kartus brangesnis už on-chain ir sunaudojo 5,571,930 „Ethereum“ grandinės gas vienetų, nes sprendimo paruošimo mokesčiai priklauso nuo algoritmui reikalingo „JavaScript“ programinio kodo apimties.

5. Realizuotiems sprendimams atliktas eksperimentinis testavimas su skirtingais duomenų rinkiniais. Jais ištirta realizuotų sprendimų vykdymo mokesčių priklausomybė nuo algoritmo laiko sudėtingumo vertės ($O(n * w)$ nuo 1000 iki 20,000) ir nuo duomenų kiekio – kai n (elementų kiekis) nesikeičia, bei kai didėja nuo 5 iki 100. Pastebėta, kad on-chain sprendimo vykdymo mokesčiai tiesiogiai priklauso nuo algoritmo sudėtingumo vertės ir išaugo nuo 551,259/428,381 iki 19,239,609/9,432,943 gas vienetų minėtiems laiko sudėtingumams su skirtingais duomenimis. Cross-chain sprendimo atveju, „BSC“ grandinės sutarties vykdymo mokesčiai taip pat priklauso nuo laiko sudėtingumo vertės ir keitėsi nuo 646,592/554,405 iki 19,330,267/9,422,722 „BSC“ grandinės gas vienetų. Be to, cross-chain sprendimo „Ethereum“ išmaniosios sutarties ir CCIP protokolo žinučių mokesčiai priklauso nuo duomenų kiekio. Tik off-chain sprendimo vykdymo mokesčiai nepriklausė nuo abiejų rodiklių ir siekė apie ~670,000 „Ethereum“ grandinės gas vienetų bei ~0.28 „LINK“ per užklausą. Greitaveikos prasme on-chain sprendimas yra greičiausias – įvykdomas iš karto, cross-chain sprendimo vykdymas užtrunka virš 20 minučių dėl CCIP protokolo žinučių perdavimo tarp grandinių, o off-chain sprendimas vidutiniškai įvykdytas per greičiau nei 1 minutę. Nuspręsta, kad „Chainlink Functions“ pritaikymas išmaniosiose sutartyse yra geresnė alternatyva skaičiavimams on-chain, nei „Chainlink CCIP“ naudojimas bei gali padidinti „Ethereum“ išmaniųjų sutarčių efektyvumą ir sumažinti vykdymo mokesčius.

SIŪLYMAI

1. „Ethereum“ išmaniosioms sutartims, kuriose reikalinga atlikti intensyvius skaičiavimus siūloma naudoti „Chainlink Functions“ platformą, taip sumažinant vykdymo transakcijų mokesčius ir padidinant išmaniųjų sutarčių efektyvumą. Prieš taikant sprendimą svarbu įvertinti aptartus platformos techninius limitus ir sutarties naudojimo principus.
2. Siekiant nepermokėti už realizuotų „Ethereum“ išmaniųjų sutarčių transakcijų vykdymo mokesčius rekomenduojama naudoti sukurtą „Google Chrome“ įskiepį. Įskiepis leidžia realiu laiku palyginti numatomus on-chain bei off-chain sprendimo su „Chainlink Functions“ integracija vykdymo mokesčius, priklausomai nuo esamos tinklo būsenos ir mokesčių žetonų vertės.

LITERATŪROS SĄRAŠAS

1. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., ir Rubio, A. (2020). *GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts.* doi:10.1007/978-3-030-45237-7_7

2. Ao, M., Li, C., Li, Y., Shao, Z., Sun, J., Wang, G., ir Yang, W. (2024). *Adaptive Parallel Scheduling Scheme for Smart Contract.* doi:10.3390/math12091347

3. Assi, M., ir Haraty, R. A. (2018). *A Survey of the Knapsack Problem.*

4. Bartoletti, M., Benetollo, L., Bugliesi, M., Crafa, S., Pettinau, R., Pinna, A., . . . Zunino, R. (2024). *Smart Contract Languages: a comparative analysis.*

5. Bellman, R. (1957). *DYNAMIC PROGRAMMING.*

6. Beniiche, A. (2020). *A Study of Blockchain Oracles.*

7. Breidenbach, L., Cachin, C., Chan, B., Coventry, A., Ellis, S., Juels, A., . . . Zhang, F. (2021). Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks.

8. Buterin, V. (2014). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* Prieiga per https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

9. Buterin, V. (2016). EIP-170: Contract code size limit.

10. Cacchiani, V., Iori, M., Locatelli, A., ir Martello, S. (2022). *Knapsack problems — An overview of recent advances. Part I: Single knapsack problems.* Computers & Operations Research.

11. Chainlink. (2024a). *Chainlink CCIP documentation.* Prieiga per https://docs.chain.link/ccip

12. Chainlink. (2024b). *Chainlink Functions documentation.* Prieiga per https://docs.chain.link/chainlink-functions

13. Chaliasos, S., Gervais, A., ir Livshits, B. (2022). *A study of inline assembly in solidity smart contracts.* doi:10.1145/3563328

14. Chen, J., Chen, T., Feng, Y., Li, X., Li, Z., Xiao, X., . . . Zhou, H. (2020). *GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts.* doi:10.1109/TETC.2020.2979019

15. Chen, L., Gao, Z., Lu, Y., Shah, N., Shi, W., ir Xu, L. (2017). *Scalable Blockchain Based Smart Contract Execution.* doi:10.1109/ICPADS.2017.00054

16. Chen, T., Li, X., Luo, X., ir Zhang, X. (2017). *Under-Optimized Smart Contracts Devour Your Money.* doi:10.1109/SANER.2017.7884650

17. Chen, W., Chen, X., Dai, H.-N., Imran, M., Weng, J., Xie, S., ir Zheng, Z. (2019). *An Overview on Smart Contracts: Challenges, Advances and Platforms.*

18. Curtis, S. (2003). *The classification of greedy algorithms.* doi:10.1016/j.scico.2003.09.001

19. de Bruin, A., Kindervater, G. A., ir Trienekens, H. W. (1995). *Asynchronous Parallel Branch and Bound.*

20. Dickerson, T., Gazzillo, P., Herlihy, M., ir Koskinen, E. (2017). *Adding Concurrency to Smart Contracts.* doi:10.1145/3087801.3087835

21. Doss, R., Pan, L., ir Praitheeshan, P. (2020). *Security Evaluation of Smart Contract-Based On-chain Ethereum Wallets.* doi:10.1007/978-3-030-65745-1_2

22. He, M., Qin, B., Song, L., Xia, S., Yoshida, N., Yu, T., ir Zhang, Y. (2024). *How to Save My Gas Fees: Understanding and Detecting Real-world Gas Issues in Solidity Programs.*

23. He, S., Lü, X., Pei, Y. W., Qian, S., Wu, T., Xu, Y., ir Zhai, X. (2024). *Overview of improved dynamic programming algorithm for optimizing energy distribution of hybrid electric vehicles.* doi:10.1016/j.epsr.2024.110372

24. Hristakeva, M., ir Shrestha, D. (2005). *Different Approaches to Solve the 0/1 Knapsack Problem.*

25. Ilyas, A. (2022). Dynamic Programming: An Overview and Implementation.

26. Kaur, K., Tomar, S., ir Tripathi, M. (2022). *Gas Fee Reduction by Detecting Loop Fusible Patterns in Ethereum Smart Contract.* doi:10.1109/ANTS56424.2022.10227770

27. Kumar, V., Migdalas, A., ir Toraldo, G. (2003). *Nonlinear optimization and parallel computing.* doi:10.1016/S0167-8191(03)00013-9

28. Oliva, G. A., Hassan, A. E., ir Jiang, Z. M. (2020). *An exploratory study of smart contracts in the Ethereum blockchain platform.* doi:10.1007/s10664-019-09796-5

29. OWASP. (2023a). *Vulnerability: Denial Of Service.* Prieiga per https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC06-denial-of-service-attacks.html

30. OWASP. (2023b). *Vulnerability: Gas Limit Vulnerabilities.* Prieiga per https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC09-gas-limit-vulnerabilities.html

31. Rust, J. (2006). *Dynamic Programming.*

32. Schnabel, R. B. (1984). *PARALLEL COMPUTING IN OPTIMIZATION.*

33. Sherry, Y., ir Thompson, N. (2021). *How Fast Do Algorithms Improve?* Proceedings of the IEEE.

34. Szabo, N. (1994). Smart Contracts.

35. The Solidity Authors. (2024). *Solidity.* Prieiga per https://docs.soliditylang.org/en/v0.8.28/

36. Wood, G. (2024). ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER.

37. Yakovenko, A. (2017). *Solana: A new architecture for a high performance blockchain v0.8.13.*

PRIEDAI

1 PRIEDAS

ON-CHAIN SPRENDIMO TESTAVIMO TRANSAKCIJŲ DUOMENYS

Detalesnė transakcijų informacija pagal hash pateikta https://sepolia.etherscan.io/.

Nr.	n	w	Transakcijos hash	Gas
1	20	50	0x1a989b2226308f9b63d472e0dbbd71ce010ae30bf35de25459a50fa7192ac2b8	551,259
2	20	100	0xa809226638235c83f72c1ca97188a35ae992a8b71225a3c5ecc36d118451dbe6	1,315,655
3	20	150	0xc0416bd15bdd22e133ba5a867d4014d0735bb5146453f8d1da480f5019aa5532	2,283,247
4	20	200	0x0d218834ece30a0639dd04229bf0b41aca6fb0598fb0fabfdae5f31b9a03cf3a	3,258,147
5	20	250	0x9c56884bb11dde2e9644a78185cfb7ab6d7cc02e0bb98d8ead5b6ac9958b5f15	4,234,214
6	20	500	0x3da788545cc0cae7a3543e207b73681d181d97882eb5a322c0c7f82531fcd8ec	9,183,325
7	20	1000	0x14364e51899bc02bb325f7ba6120c7da4705f7dd71969d179ae89a6fbf097c1e	19,239,609
8	5	200	0xb35f05553e4b303c4ea702da47229ef0d693b273ac55a348402bb14fb89860d9	428,381
9	10	200	0x6e858ffdf6edda8280b28f43d61c5bb734e00dc5968d0d1ac5153fd7dff7a737	1,166,534
10	15	200	0x5eb144444dbaaa2c5aa5afb6c760198beee0cb8db681af85f25926e6223d6ee8	1,618,464
11	20	200	0xc38a8efe6deb4bb42b6e578e08d483aedb81c837345ac08245b4c37fa3e2364b	2,221,071
12	25	200	0x1d4f3df321591bb8c1685e0becc9b238a35f783c09380c1939274346790dd8c7	2,674,960
13	50	200	0xcc6c1faaf7d3e7d6c3267521ca51fe6502c33a1ed218d0676546d5909721d885	5,153,303
14	100	200	0x09221ed23517e42aea9ae015e172ba472d14ec324b88c7f2bcec829223c0d9d2	9,432,943

CROSS-CHAIN SPRENDIMO TESTAVIMO „ETHEREUM“ GRANDINĖS TRANSAKCIJŲ DUOMENYS

Detalesnė transakcijų ir „Chainlink CCIP“ žinučių informacija pagal hash pateikta https://sepolia.etherscan.io/ ir https://ccip.chain.link/ naršyklėse.

Nr.	n	w	Transakcijos hash	Gas	„LINK“ mokestis
1	20	50	0x38973fb27bc015b92c58c8a4fce898534fa3b77ccbf4620bc2c8d2d4e5fd7ef6	346,062	0.179750025
2	20	100	0xaa0828abca1b33c44b43dc8dcfe223c4560e64371ffaa0b1b4906ea544d0245a	328,962	0.179750025
3	20	150	0xafddb2f830c51ebb8ef6ab3a394a600ed5bce959158d7fcc2124f1ce97af7c49	328,962	0.179750025
4	20	200	0xbb8189b35858842478a109bb364afe2ed0372c47bcbd60767dea03fd657fd838	328,962	0.179750025
5	20	250	0x49d47243fa6c6107317faa3a84fab3bd03e541e4e5a368c7f713524b87aba94b	328,962	0.179750025
6	20	500	0x8fa61e1b0125e26a0ee21fc84c8b792895dcacb0f9d43985057b896836c24d36	328,974	0.179750025
7	20	1000	0x30f068042ebba9271748f3b89d83ad434586510c5d49663f8c9cc56f64bbd73c	328,974	0.179750025
8	5	200	0x82eeabb79c0795e43bf8890fafd56370b1ddfed023970d49acc4e883427d07e5	284,177	0.169745697
9	10	200	0x66856a0d4c317819ed28ad5b5d0edbf3176bc8bc726ef3641bfb5c0486972b1d	287,695	0.170442539
10	15	200	0x0bd2e4efb1c73f70b1bb897b419482c347271a5faf1e79ab03a8a97d806cb177	308,323	0.171139381
11	20	200	0xb7d3ce77eeb8ee61a2eb0d7456e4dba6da6f42304094db21eff93f50a4c20e1c	328,962	0.171836223
12	25	200	0xdcc6ceefb845861b376e844089f7bbc9125027c1fa3b4b75e5fab7c850237b79	349,609	0.172533064
13	50	200	0x6c68314a7b2a2ce66c3995470ff1a3ff15f6f554e1491c2144161911d0926786	452,988	0.340580057
14	100	200	0xaba420bd56526de6320966565b9fdeb602d91df21c240e36a02b2a78a17ebc87	660,454	0.573611845

CROSS-CHAIN SPRENDIMO TESTAVIMO „BSC“ GRANDINĖS TRANSAKCIJŲ DUOMENYS

Detalesnė transakcijų ir „Chainlink CCIP“ žinučių informacija pagal hash pateikta https://sepolia.etherscan.io/ ir https://ccip.chain.link/ naršyklėse.

Nr.	n	w	Transakcijos hash	Gas	„LINK“ mokestis
1	20	50	0x6ed04168e32a59a8baef591306056a59a93c87614a69f2ae248654163d4579db	646,592	0.033915686
2	20	100	0x319b42bd1764e2ee71983a5f7dfebef1023a9293506d4215295fcf4794dc75d9	1,413,460	0.033915686
3	20	150	0x5590efd76d931f0f0a2e5485067f6702ed7fb58dc10776c2eb1ca99c38cdac50	2,383,525	0.033915686
4	20	200	0xd9b6020cbb5976e9e63a4a0836bbdb078919fa5ff1045e8831db68a5adc5fbf5	3,356,097	0.033915686
5	20	250	0x0b96d2e574928ad531e0a4478e301d4d39252ea8d27783e2e1299240c0bb7758	4,329,706	0.033915686
6	20	500	0xcb5c52cbdde6f43a962295c7270869a87c62dc9898d637eb24fa49c77ecd50a9	9,276,377	0.033915686
7	20	1000	0x88ee824ce6094b53e9ab3e5b62aa27e0e718e0a3d15ad2b1c8a26ead794d0c1e	19,330,267	0.033915686
8	5	200	0x2c0da2614194c971725ee2a956066f570be96446abdf0936d7605c8549573f06	554,405	0.365462128
9	10	200	0x4b913f026491de58135f7ff90ae5a471335befe3a53fbfec45464dde879f536c	1,287,974	0.365462128
10	15	200	0xcbef054f38fcf887265a6bd00e2e29676d775c6b04d5bca0f862b235ac00a6b9	1,735,321	0.365462128
11	20	200	0xb54a7d5179f4923af584db5a5ea9b7ffa74034738562655af04307529fa24224	2,326,069	0.365462128
12	25	200	0xc8d53fb1aeee1a80201376cfb708102093a04d7a21f9c307292fdd836798a231	2,770,457	0.365462128
13	50	200	0xa71da5d9977df6c0db1fc8cc37579a934dfd7779f12debd9dc8337627f6daafe	5,216,026	0.365462128
14	100	200	0x599c46c69a1982ff26b2f8490f0f3ae027fc0d7026d8d7572168d64f110074d9	9,422,722	0.365462128

OFF-CHAIN SPRENDIMO TESTAVIMO TRANSAKCIJŲ DUOMENYS

Detalesnė transakcijų informacija pagal hash pateikta https://sepolia.etherscan.io/.

Nr.	n	w	Transakcijos hash	Gas	„LINK“ mokestis
1	20	50	0xebae237e142825cd607e4fbd25b4d78b932bd292384f1c068e4d90795f50684a	677,160	0.307515858
2	20	100	0x0d160820936a8adf93c94dac2f5da10ecc0f8743a8452776b73d214705073ec9	677,312	0.250409895
3	20	150	0x50b70985742cb922c925e1258b7dc12fd3c8d4b4edb23fc3bb415673ea930ec6	677,312	0.252777417
4	20	200	0xdecf532b3fe8e1c7dbbcb43f7048c7f76cf21eacb541e4b253697c189f641d2e	677,312	0.305443579
5	20	250	0xf8ce3c2c655141db36710695d44a353a60479a0ce3fd15e38be87025f02dd9c5	643,112	0.304217789
6	20	500	0x46f245170dde467d7181db462e8c1dd0b0bc081694ea79d60d674f4461f02040	677,324	0.248469566
7	20	1000	0x8b8704ebb8823bc8d81296483c89f895c0dd4e2fae261ee4f87995d9f2c25399	643,257	0.304329593
8	5	200	0xce4ad20cc1a843f145126127b7a67b00b21d45549230116ebf63dc01fe06029f	677,312	0.273587953
9	10	200	0x7bdc337bed7cf065569eb31a4a07c21194785064e36d1b19dab3a3b428d76ad4	677,312	0.32370534
10	15	200	0x2bff3019ec4001a0626943d918c16a555f28732fc6d7ce7d0edf2d5a3200811f	677,312	0.265082253
11	20	200	0x07fa617ce28ca7092ce71624d8aecd1b16e4f80b2725b6afd49a800d8779d1b9	677,312	0.262305203
12	25	200	0xbd2a1a9151e239f2cd66ad99d66deb6da919b066fbaad907fa56b28bb757d1d2	677,312	0.266322727
13	50	200	0xdaa1d55a394436cb1c3f0dae5479d26e30ea2aba69b0f58a2b5783ddd61c48a6	677,312	0.313725121
14	100	200	0xe8acd4a70de5df2b6b3b9d00a38f3aced56e7647c8d5838ef0954080c355ddf6	677,312	0.301386997

OFF-CHAIN SPRENDIMO TESTAVIMO „CHAINLINK FUNCTIONS“ UŽKLAUSŲ DUOMENYS

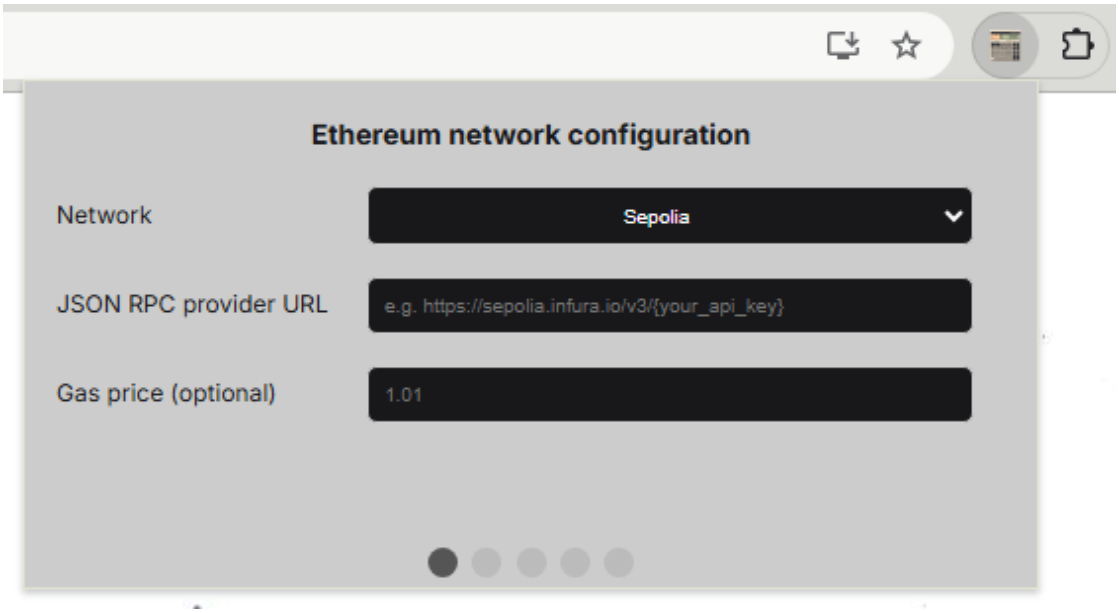
Užklausų numeriai atitinka 4 priede pateiktų transakcijų numerius. Detalesnė „Chainlink Functions“ užklausų informacija pagal užklausos ID pateikta https://functions.chain.link/sepolia/3833 naršyklėje.

Nr.	Užklausos ID	„LINK“ mokestis
1	0xebae237e142825cd607e4fbd25b4d78b932bd292384f1c068e4d90795f50684a	0.307515858
2	0x0d160820936a8adf93c94dac2f5da10ecc0f8743a8452776b73d214705073ec9	0.250409895
3	0x50b70985742cb922c925e1258b7dc12fd3c8d4b4edb23fc3bb415673ea930ec6	0.252777417
4	0xdecf532b3fe8e1c7dbbcb43f7048c7f76cf21eacb541e4b253697c189f641d2e	0.305443579
5	0xf8ce3c2c655141db36710695d44a353a60479a0ce3fd15e38be87025f02dd9c5	0.304217789
6	0x46f245170dde467d7181db462e8c1dd0b0bc081694ea79d60d674f4461f02040	0.248469566
7	0x8b8704ebb8823bc8d81296483c89f895c0dd4e2fae261ee4f87995d9f2c25399	0.304329593
8	0xce4ad20cc1a843f145126127b7a67b00b21d45549230116ebf63dc01fe06029f	0.273587953
9	0x7bdc337bed7cf065569eb31a4a07c21194785064e36d1b19dab3a3b428d76ad4	0.32370534
10	0x2bff3019ec4001a0626943d918c16a555f28732fc6d7ce7d0edf2d5a3200811f	0.265082253
11	0x07fa617ce28ca7092ce71624d8aecd1b16e4f80b2725b6afd49a800d8779d1b9	0.262305203
12	0xbd2a1a9151e239f2cd66ad99d66deb6da919b066fbaad907fa56b28bb757d1d2	0.266322727
13	0xdaa1d55a394436cb1c3f0dae5479d26e30ea2aba69b0f58a2b5783ddd61c48a6	0.313725121
14	0xe8acd4a70de5df2b6b3b9d00a38f3aced56e7647c8d5838ef0954080c355ddf6	0.301386997

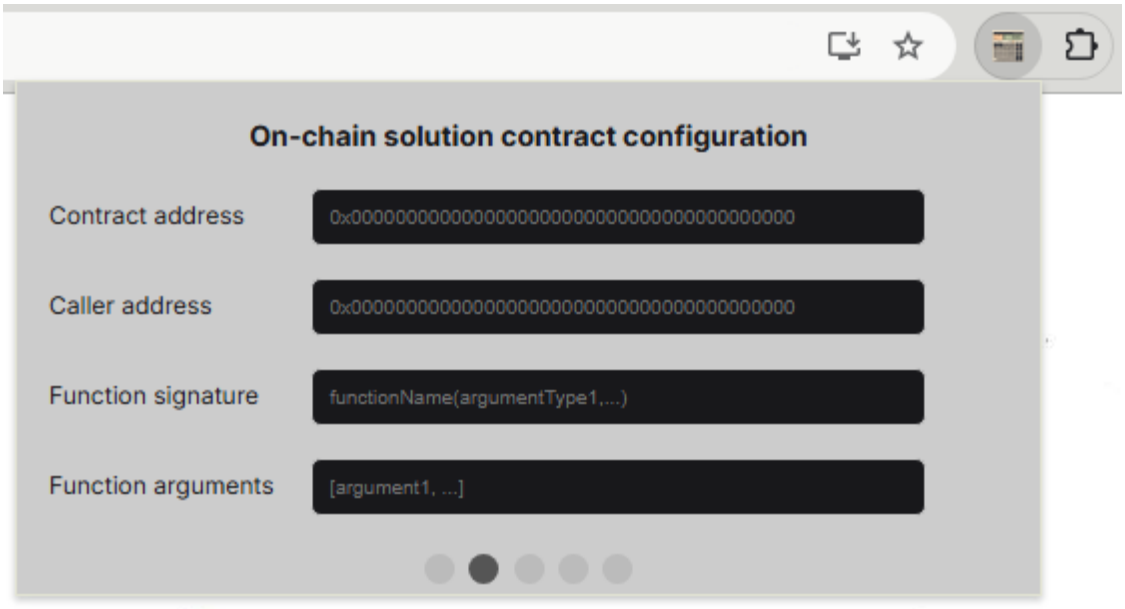
„GOOGLE CHROME" ĮSKIEPIO NAUDOJIMO VADOVAS

Įskiepis gali būti įdiegiamas „Google Chrome" įskiepių valdymo lange. Pasirenkama kelti nesupakuotus failus ir pridedamas 7 priede aprašytas *extension/dist* aplankas (15 įrašas). Atlikus šiuos veiksmus įskiepis yra pridedamas į naršyklę.

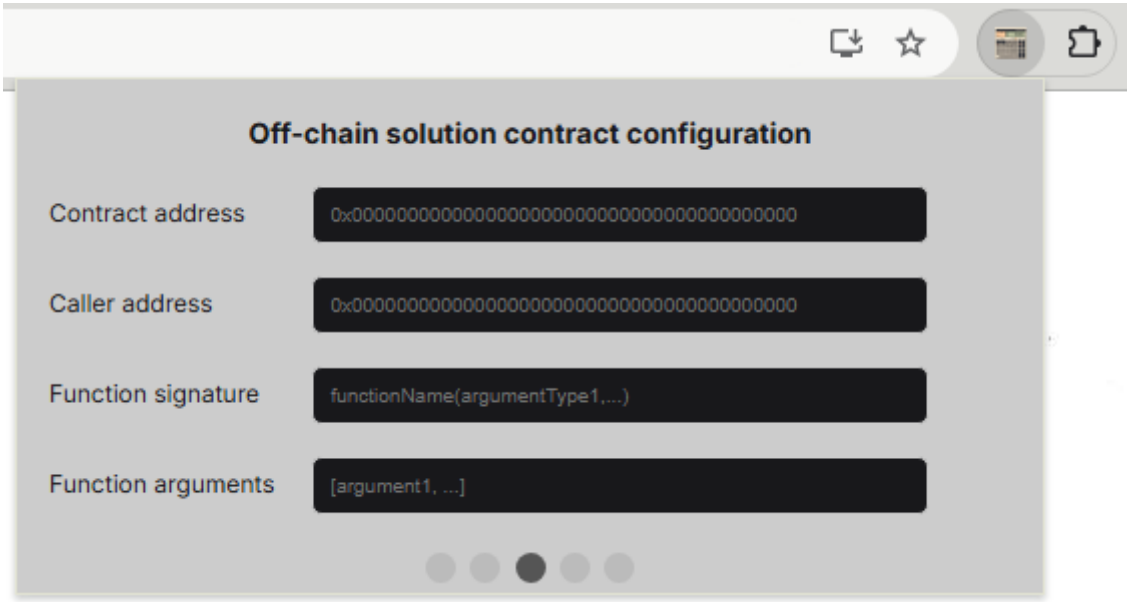
Žemiau pateikta realizuoto „Google Chrome" įskiepio naudotojo sąsaja. Ją sudaro 5 skiltys, iš kurių pirmoji – naudojamo „Ethereum" tinklo konfigūracijos skiltis.



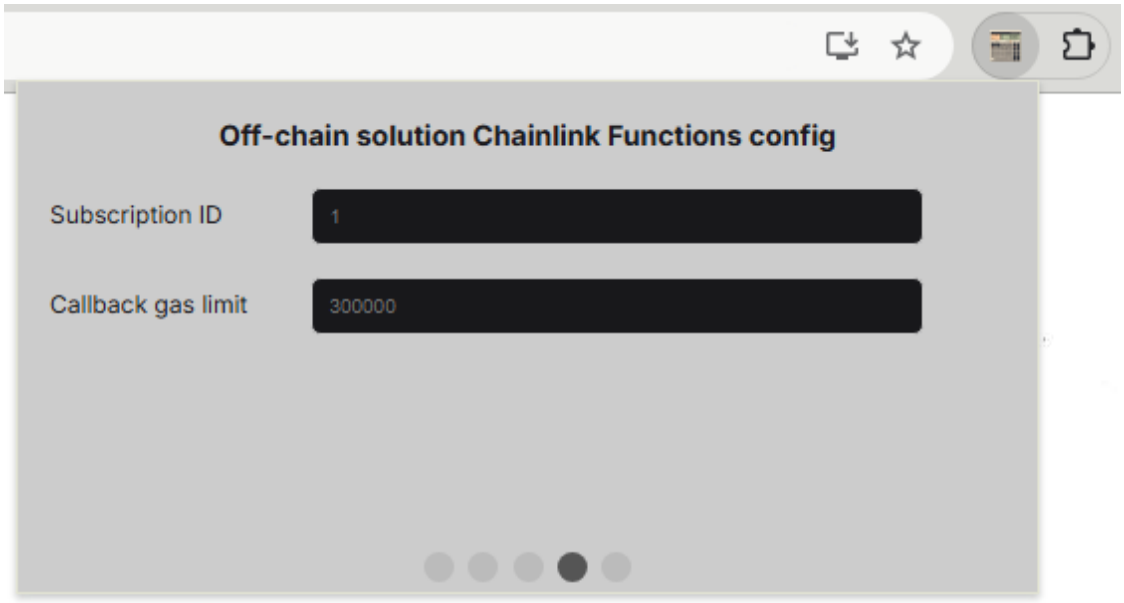
Toliau naudotojas turi įvesti įdiegtos on-chain sprendimo „Ethereum" išmaniosios sutarties duomenis.



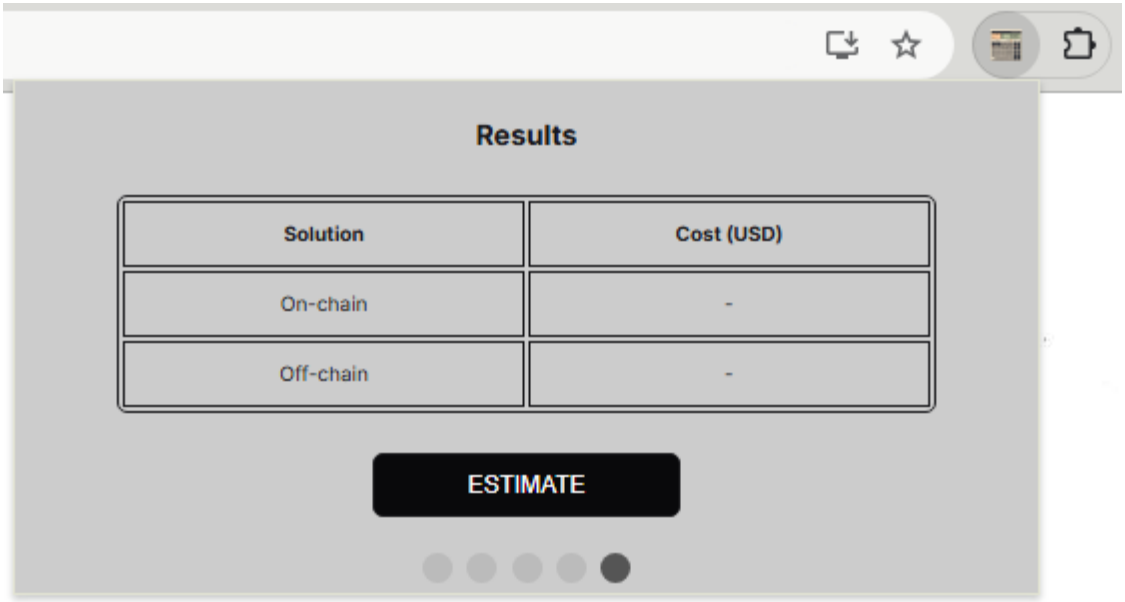
Off-chain sprendimo konfigūravimas sudarytas iš 2 skilčių, pirmiausia suvedami off-chain sprendimo „Ethereum" išmaniosios sutarties duomenys.



Taip pat reikalingi ir su „Chainlink Functions“ užklausos vykdymu susiję parametrai.



Paskutinioji skiltis skirta rezultatams. Norėdamas gauti numatomus sprendimų vykdymo mokesčius naudotojas turi užpildyti įvesties laukus ir paspausti skaičiavimo (*ESTIMATE*) mygtuką. Tuomet skaičiuojami numatomi sprendimų vykdymo mokesčiai bei rezultatų lentelėje pateikiamos šių mokesčių „USD“ išraiškos. Nekorektiškų įvesčių ar nenumatytos klaidos atveju gaunamas klaidos pranešimas bei pažymimos skiltys su neteisingomis įvestimis. Daugiau informacijos apie klaidą galima rasti naršyklės konsolėje (angl. console).



Aptartose naudotojo sąsajos skiltyse esančių įvesčių laukų aprašai paaiškinti žemiau pateiktoje lentelėje.

Skiltis	**Įvesties laukas**	**Reikšmė**
Ethereum network configuration	Network	Naudojamas „Ethereum“ tinklas
Ethereum network configuration	JSON RPC provider URL	JSON RPC paslaugų teikėjo („Ethereum“ mazgo) nuoroda
Ethereum network configuration	Gas price (optional)	Gas mokestis gwei formatu, nenurodžius naudojamas iš mazgo gautas gas mokestis
On-chain solution contract configuration	Contract address	On-chain sprendimo išmaniosios sutarties adresas
On-chain solution contract configuration	Caller address	On-chain sprendimo išmaniosios sutarties funkcijos kvietėjo adresas
On-chain solution contract configuration	Function signature	On-chain sprendimo išmaniosios sutarties funkcijos parašas (angl. signature)
On-chain solution contract configuration	Function arguments	On-chain sprendimo išmaniosios sutarties funkcijos argumentai
Off-chain solution contract configuration	Contract address	Off-chain sprendimo išmaniosios sutarties adresas
Off-chain solution contract configuration	Caller address	Off-chain sprendimo išmaniosios sutarties funkcijos kvietėjo adresas
Off-chain solution contract configuration	Function signature	Off-chain sprendimo išmaniosios sutarties funkcijos parašas (angl. signature)
Off-chain solution contract configuration	Function arguments	Off-chain sprendimo išmaniosios sutarties funkcijos argumentai
Off-chain solution Chainlink Functions config	Subscription ID	„Chainlink Functions“ prenumeratos ID
Off-chain solution Chainlink Functions config	Callback gas limit	„Chainlink Functions“ užklausos rezultato grąžinimo funkcijos gas limitas

EKSPERIMENTO REALIZACIJOS PROGRAMINIO KODO APRAŠAS

Visas programinis kodas, naudotas eksperimentinės dalies išmaniųjų sutarčių realizacijoms bei „Google Chrome“ įskiepiui sukurti, pateiktas „GitHub“ programinio kodo talpykloje, pasiekiamoje adresu https://github.com/ignasbudreika/magistras. Toliau pateikti šių programinio kodo failų aprašai.

Nr.	Aplankas	Failas	Aprašas
1.	contracts		Realizuotų išmaniųjų sutarčių „Solidity“ programiniai kodai
2.	contracts	DAO.sol	Abstrakti DAO koncepto išmanioji sutartis „Ethereum“ grandinei
3.	contracts	OnChainDAO.sol	On-chain sprendimo išmanioji sutartis „Ethereum“ grandinei
4.	contracts	CrossChainDAO.sol	Cross-chain sprendimo išmanioji sutartis „Ethereum“ grandinei
5.	contracts	CrossChainKnapsack.sol	Cross-chain sprendimo išmanioji sutartis „BSC“ grandinei
6.	contracts	OffChainDAO.sol	Off-chain sprendimo išmanioji sutartis „Ethereum“ grandinei
7.	functions		Off-chain sprendimo „JavaScript“ programinis kodas
8.	functions	secrets.js	„Off-chain secrets“ užšifravimas, patalpinimas į „Gist“ bei nuorodos užšifravimas
9.	functions	offchain.js	Off-chain sprendimo skaičiavimas „Chainlink“ orakuluose
10.	extension		Realizuoto „Google Chrome“ įskiepio programinis kodas
11.	extension	vite.config.js	„Vite“ konfigūracija
12.	extension	index.html	Įskiepio HTML programinis kodas
13.	extension/public	manifest.json	Įskiepio konfigūracija
13.	extension/src	index.js	Įskiepio įvesčių validacijos bei numatomų mokesčių skaičiavimo logika
14.	extension/src	index.css	Įskiepio naudotojo sąsajos stiliai
15.	extension/dist		Įskiepio programos paketas, naudojamas įskiepio diegimui į naršyklę