UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

DEMONSTRAȚII ZERO-KNOWLEDGE DE APARTENENȚĂ LA MULȚIMI

Absolvent
Pârjol Andrei-Nicolae

Coordonator științific
Conf.dr. Ruxandra Olimid

București, iunie 2024

Rezumat

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce vitae eros sit amet sem ornare varius. Duis eget felis eget risus posuere luctus. Integer odio metus, eleifend at nunc vitae, rutrum fermentum leo. Quisque rutrum vitae risus nec porta. Nunc eu orci euismod, ornare risus at, accumsan augue. Ut tincidunt pharetra convallis. Maecenas ut pretium ex. Morbi tellus dui, viverra quis augue at, tincidunt hendrerit orci. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam quis sollicitudin nunc. Sed sollicitudin purus dapibus mi fringilla, nec tincidunt nunc eleifend. Nam ut molestie erat. Integer eros dolor, viverra quis massa at, auctor.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce vitae eros sit amet sem ornare varius. Duis eget felis eget risus posuere luctus. Integer odio metus, eleifend at nunc vitae, rutrum fermentum leo. Quisque rutrum vitae risus nec porta. Nunc eu orci euismod, ornare risus at, accumsan augue. Ut tincidunt pharetra convallis. Maecenas ut pretium ex. Morbi tellus dui, viverra quis augue at, tincidunt hendrerit orci. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam quis sollicitudin nunc. Sed sollicitudin purus dapibus mi fringilla, nec tincidunt nunc eleifend. Nam ut molestie erat. Integer eros dolor, viverra quis massa at, auctor.

Cuprins

Capitolul 1

Introducere

1.1 Motivație și obiective

Obiectivul lucrării de față este acela de a prezenta conceptele și implementările curente pentru protocoalele zk-SNARK folosite în demonstrațiile de apartenență la mulțimi. Deși pot părea abstracte la prima vedere, această ramură de demonstrații (în eng. *zero-knowledge proof of membership*) are o gamă largă de aplicații precum: anonimizarea tranzacțiilor cu criptomonede (de ex. protocolul Zcash pentru Bitcoin și protocolul Tornado Cash pentru Ether), votul electronic descentralizat și anonim (de ex. putem să demonstrăm ca avem dreptul să votăm fără să dezvăluim date personale) sau mai general, folosirea anonimă a unor servicii online (de ex. fără să folosim username/password).

Când privim problema demonstrațiilor de apartenență apar două întrebări legate de performanța verificării demonstrațiilor și confidențialitatea datelor:

- Putem să *verificăm* că un element x face parte dintr-o mulțime S fară să trebuiască să memorăm întreaga mulțime S și cu o complexitate timp $< O(|S|)$?
- Putem să verificăm că elementul x face parte din mulțimea S fără să cunoaștem elementul x ?

1.2 Contribuția personală

Contribuția personală constă în scrierea de librării JavaScript și circuite Circom pentru implementarea arborilor hash Merkle și a procedurilor de generare și verificare a demonstrațiilor de apartenență folosind SNARK-uri. De asemenea se propun și îmbunătățiri, folosind arbori hash "*indexați*" introduși în lucrarea *Transparency Dictionaries with Succinct Proofs of Correct Operation*[22] care permit reducerea adâncimii arborelui și implicit a numărului de operații necesare pentru demonstrație.

1.3 Structura lucrării

Aici voi explica fiecare capitol odata ce este gata

Capitolul 2

Demonstrații zero knowledge

2.1 Scurt istoric

Termenul de *zero knowledge* a fost introdus prima dată în anul 1985 de către cercetătorii Shafi Goldwasser, Silvio Micali și Charles Rackoff de la Institutul de tehnologie din Massachusetts în lucrarea *The knowledge complexity of interactive proof-systems* [12, pp. 291–304]. Aceștia încercau sa rezolve problemele legate de sistemele de demonstrare interactive, sisteme teoretice în care o entitate numită $Prover$ încearcă să convingă o altă entitate numită $Verifier$ că o propoziție matematică este adevărată [9].

Acest tip de sistem este numit interactiv deoarece cele două părți interschimbă mesaje în timpul procesului de demonstrare. Inițial o mare parte din muncă era îndreptată spre asigurarea validității sistemului, adică rezolvarea cazului în care $Prover$-ul avea intenții malițioase și încerca să păcălească $Verifier$-ul în a crede o propoziție falsă ca fiind adevărată [14].

În sistemele de demonstrare interactive este presupus că $Prover$-ul are putere de calcul nelimitată (informal toate problemele sunt fezabile) însă nu este de încredere și $Verifier$-ul are putere de calcul limitată și este onest. Ce au făcut cei trei cercetători a fost să ia în considerare și cazul în care $Verifier$-ul nu este de încredere și s-au întrebat ce informații poate să obțină acesta după o demonstrație. O astfel de scurgere de informații este destul de gravă deoarece din ipoteză folosind aceste sisteme $Verifier$-ul ar putea avea acces la informații pe care în mod normal nu ar fi putut să le calculeze [14] [9].

A fost fost propusă astfel implementarea unui sistem de demonstrații nou, *zero knowledge proof*, în care se demonstrează cunoașterea unei soluții la o problemă în loc de soluție în sine. După terminarea demonstrației $Verifier$-ul nu învață nimic nou în afara faptului că $Prover$-ul cunoaște soluția [14].

2.2 Preliminarii teoretice

Notație 1. Vom nota cu P și V cele două părți implicate în procesul de demonstrare, $Prover$ și $Verifier$.

Definiție 2.2.1. Definim și notăm cu L limbajul formal folosit de P și V pentru a exprima propoziția matematică care urmează să fie demonstrată. Astfel V publică limbajul L iar P demonstrează cunoașterea unui cuvânt $w =< w_1, w_2, ..., w_n >$ astfel încât $w \in L$.

Exemplu 1. Pentru ca P să demonstreze propoziția ”Cunosc factorizarea în 3 termeni pentru elementul 1 din corpul prim F_{13}”, V definește limbajul $L = \{w =< w_1, w_2, ..., w_n > |n = 3, w_i \in F_{13} \forall w_i, w_1 \times w_2 \times w_3 \equiv 1(mod 13)\}$ și P trebuie să trimită un cuvânt x pentru a fi verificat de către V, de exemplu pentru $< 3, 5, 7 >$ avem $3 \times 5 \times 7 \equiv 105 \equiv 1(mod 13)$ deci în acest caz P cunoaște o factorizare corectă.

Definiție 2.2.2 (Semnale private/publice). Cuvântul $w =< w_1, w_2, ..., w_n >$ reprezintă un șir litere ce vor fi folosite ca și semnale în circuitele algebrice folosite de CIRCOM. Aceste semnale pot fi private, caz în care sunt ascunse de $Verifier$ sau pot fi publice caz în care sunt trimise direct $Verifier$-ului.

Definiție 2.2.3 (ZKP). Dat fiind un sistem de demonstrație (P,V) și un limbaj formal L (astfel încât $x \in L$ să fie echivalent cu x este adevărat), acest sistem este *zero knowledge*, uneori notat cu *ZKP*, dacă satisface următoarele trei proprietăți:

- **Completitudine:** $x \in L$, Pr[V acceptă] = 1 . x este acceptat cu probabilitate 1 atunci când avem un demonstrator onest și o propoziție validă.
- **Corectitudine:** $x \notin L$ Pr[V acceptă] = 1/n , $n \in N$. x este acceptat cu probabilitate redusă/mică atunci când avem o demonstrație mincinosă și un verificator onest.
- **Zero Knowledge:** După demonstrație V nu obține nicio informație nouă iar datele folosite de P rămân confidențiale [9, pagina 3, secțiunea 2.2].

Definiție 2.2.4 (zk-SNARK). În practică vom folosi o primitivă criptografică care poate să genereze într-un mod eficient un protocol zero knowledge pentru orice problemă sau funcție. Această primitivă se numește *zero knowledge Succint Noninteractive Argument of knowledge*, prescurtat *zk-SNARK*, și are următoarele proprietăți aditionale:

- **Succint:** demonstrațiile generate sunt scurte și pot fi verificate rapid.
- **Noninteractive:** nu este necesară comunicarea prin întrebări și răspunsuri dintre $Prover$ și $Verifier$ astfel încât demonstrațiile pot fi generate offline și verificate asincron.
- **ARgument of Knowledge:** se demonstrează cunoașterea unei intrări x pentru o funcție și un rezultat dat [9, pagina 3, secțiunea 2.1 și 2.2].

Ideea de bază: Se transformă problema (ex: logaritm discret, 3-colorarea grafului etc.) într-o funcție a cărei intrări vrem să le ascundem. Executăm apoi funcția folosind criptarea homomorfică și funcția este apoi trecută printr-un procedeu numit "roll up" în care se obține o semnătură scurtă care indică execuția corectă a funcției.

Definiție 2.2.5 (Funcție hash criptografică)**.** O funcție hash criptografică este o funcție $h : D \to R$, unde domeniul $D = \{0,1\}^*$ și codomeniul $R = \{0,1\}^n$ pentru o constantă naturală $n \geq 1$. Această funcție satisface următoarele proprietăți:

- **rezistența la preimagine:** dat fiind o valoare hash $h(x)$ nu este fezabil să găsim valoarea $x \in D$
- **rezistența la coliziune:** nu este fezabil să găsim două valori distincte $x, y \in D$ astfel încât $h(x) = h(y)$ [6][pagina 3].

O funcție hash criptografică trebuie să fie ușor de calculat, folosind un algoritm determinist polinomial.

Capitolul 3

Acumulatori criptografici

Definiție 3.0.1. Un *acumulator criptografic* este o reprezentare compactă a unei mulțimi de elemente sub forma unei valori hash. Formal un acumulator poate fi descris printr-un triplet de 3 algoritmi: $(Acc, Prove, Verify)$ care au următoarele funcționalități:

- $A \leftarrow Acc(S)$ - realizează compresia mulțimii S într-o valoare hash notată cu A.
- $\pi_x \leftarrow Prove(x, S)$ - generează demonstrația de apartenență la mulțimea S pentru elementul x.
- $\{0, 1\} \leftarrow Verify(A, x, \pi_x)$ - acceptă sau respinge demonstrația π_x folosind doar valoare hash A [5].

Notație 2. Spunem că un element x a fost acumulat în A dacă $x \in S$ și valoarea x a fost folosită în generarea valorii hash a acumulatorului. De asemenea spunem că un element x nu a fost acumulat în A dacă $x \notin S$ și valoarea x nu a fost folosită în generarea valorii hash a acumulatorului.

Această reprezentare permite generarea de demonstrații de apartenență, notate w_x și numite martor (în eng. witness), pentru orice element x care a fost acumulat sau demonstrații de non-apartenență pentru orice element care nu a fost acumulat.

Definiție 3.0.2. Acumulatorii pot fi clasificați după tipul de demonstrație suportat, avem astfel:

- **pozitivi** - suportă doar demonstrații de apartenență.
- **negativi** - suportă doar demonstrații de non-apartenență.
- **universali** - suportă ambele tipuri de demonstrații [4].

Definiție 3.0.3. O altă clasificare pentru acumulatori este dată de metodele de actualizare pe care aceștia le suportă, astfel avem acumulatori:

- **aditivi** - suportă doar introducerea de elemente noi în mulțime.

- **substractivi** - suportă doar eliminarea de elemente din mulțime.
- **dinamici** - suportă ambele operații descrise mai sus [4].

Definiție 3.0.4. Dacă avem o entitate (*trusted party*) responsabilă pentru actualizarea acumulatorului acesta se numește *trapdoor-based* altfel acesta se numește *trapdoorless* [4].

Pentru acumulatorii *trapdoor-based*, entitatea responsabilă pentru actualizarea mulțimii suport se numește *managerul acumulatorului*. Acesta deține o cheie secretă (*trapdoor*) și este capabil să adauge, să șteargă elemente și să genereze demonstrații într-un mod eficient [4].

Acumulatorii *trapdoorless* permit actualizări publice asupra mulțimii suport, fără a mai fi nevoie de o parte terță de încredere. Astfel utilizatorii sunt responsabili pentru actualizarea și generarea de demonstrații [4].

Acumulatorii criptografici au numeroase aplicații, cele mai populare fiind : anonymous credentials, group signatures, stocarea datelor in cloud și anonimizarea tranzacțiilor cu criptomonede.

3.1 Date private și scalabilitate

Din definiția dată mai sus acumulatorii criptografici nu au nicio propietate care să păstreze datele private. O entitate malițioasă poate să afle pentru ce element din mulțime s-a făcut o demonstrație sau poate să afle ce element a fost șters sau adăugat în acumulator. Aceste informații pot fi folosite pentru a falsifica demonstrații și pentru a actualiza abuziv mulțimea/acumulatorul.

În practică acumulatorii sunt folosiți în zone în care datele trebuie să rămână private așa că demonstrațiile convenționale sunt înlocuite cu demonstrațiile *zero-knowledge*. Odată cu schimbarea tipului de demonstrație folosit apar și probleme noi pe care le vom discuta și rezolva în continuare : *replay attacks* - folosirea repetată a aceleiași demonstrații, timpi de lucru mari pentru generarea demonstrațiilor și probleme de concurență atunci cand doi sau mai mulți utilizatori încearcă să actualizeze același acumulator în același timp.

Dorim totodată ca acumulatorul să fie scalabil și să ne permită să verificăm apartenența $x \in S$ într-un timp subliniar, fără să reținem toate elementele din S.

Pentru a realiza cele două obiective trebuie să definim entitățile care participă în procesul de demonstrare :

- **Prover** - cunoaște valoarea secretă x și mulțimea S și are spațiu de memorie și putere de calcul nelimitate. Acesta este responsabil de generarea demonstrației de apartenență.

- **Verifier** - nu cunoaște valoarea secretă x sau mulțimea S și deține spațiu de memorie și putere de calcul limitate. Acesta este responsabil de verificarea demonstrației de apartenență.

Pentru a fi considerate eficiente, dimensiunea acumulatorului A, a demonstrației π_x și complexitatea timp a algoritmului $Verify$ trebuie să fie mai mici decât $|S|$. În continuare sunt prezentați arborii hash Merkle în care algoritmul $Verify$ are o complexitate timp $O(log(|S|))$.

3.2 Arbori hash Merkle

Definiție 3.2.1. Arborii hash Merkle sunt acumulatori criptografici sub forma unor arbori binari în care valoarea fiecărui nod este dată de o funcție hash criptografică care primește ca și intrări valorile copiilor nodului, sau dacă nodul este nod frunză atunci primește valoarea hash a unui element din mulțimea suport [20].

Funcția hash folosită intr-un arbore Merkle este importantă deoarece pentru a actualiza un arbore hash sau pentru a genera și verifica demonstrații trebuie să folosim în mod repetat funcția hash, așa că în contextul zero-knowledge dorim o funcție care să fie ușor de scris sub-forma unui circuit algebric pentru a genera un SNARK cât mai eficient și cu cât mai puține constrângeri. Astfel de funcții hash sunt numite și *"zk-friendly"*, iar câteva funcții folosite în practică sunt : *Poseidon* [13], *MiMC* [1], *Vision Mark-32* [3] sau *Rescue* [2].

Definiție 3.2.2 (Funcția hash Poseidon)**.** Funcția hash Poseidon e este o funcție hash criptografică $POS : F_P^* \rightarrow F_P$ care primește o secvență de lungime arbitrară de elemente din corpul F_P și întoarce un element din corpul F_P. Corpul F_p este un corp prim în care p reprezintă ordinul grupului format din punctele curbei eliptice folosite de către sistemul de demonstrare ZK [13]. În cazul nostru, CIRCOM 2.0 folosește ordinul grupului de puncte generat de curba eliptică ALT_BN128 [21].

Notație 3. În implementarea prezentată în această lucrare vom folosi funcția hash *Poseidon* , prescurtată cu *POS.*

Figura 3.1 prezintă un arbore hash de adâncime 3 în care am acumulat cheile secrete $S = \{A, B, ..., F\}$. Nodurile frunză conțin doar hash-urile $\{POS(A), POS(B), ..., POS(F)\}$, iar nodurile care nu au o valoare setată primesc o valoare *null* notată cu *zeroVal* sau *0.*

Folosirea unei valori *null* prestabilită ne permite să implementăm într-un mod eficient arbori Merkle *sparse* care au o adancime foarte mare însă conțin puține elemente deoarece putem să calculăm valorile null pentru fiecare nivel de adâncime.

Observație. Arborii Merkle au o proprietate adițională care îi face mai puternici decât alți acumulatori criptografici deoarece realizează un *"vector commitment"*: rădăcina arborelui codifică nu numai conținutul mulțimii dar și ordinea în care elementele apar în mulțime și astfel este imposibil pentru un Prover să demonstreze două valori diferite la aceiași poziție.

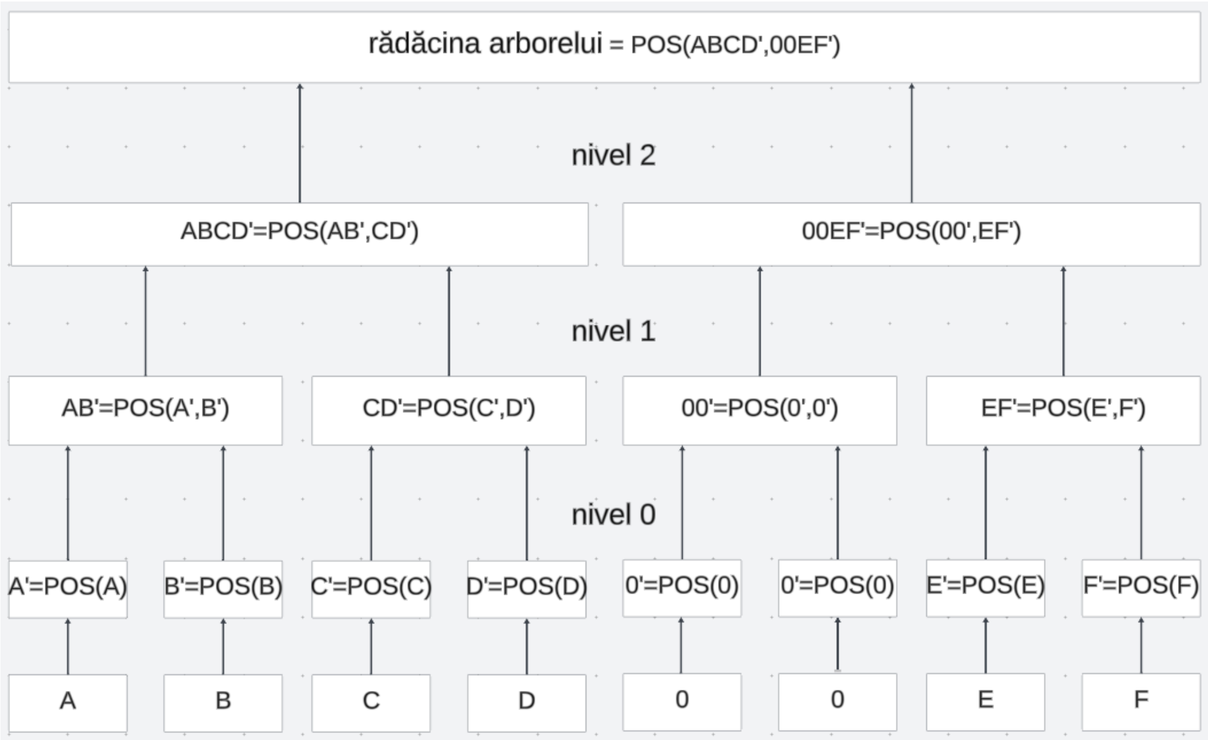

Figura 3.1: Arbore Merkle sparse de adâncime 3

3.3 Demonstrații de apartenență

O demonstrație de apartenență în contextul arborilor Merkle presupune parcurgerea corectă a drumului de la nodul frunză cu valorea $POS(x)$ pentru care se face demonstrația până la rădăcina arborelui.

Funcția $Prove(x, S)$ generează vectorii de lungime $\lfloor log(|S|) \rfloor$: *siblings* - vectorul cu fiecare nod frate din fiecare nivel și *path* - vectorul care codifică poziția nodului curent în funcția hash pentru a calcula nodul următor. În implementare folosim 0 pentru stânga și 1 pentru dreapta. Pe lângă cei doi vectori care atestă că elementul face parte din mulțimea S la poziția indicată, un *Prover* mai trebuie să demonstreze și faptul că știe preimaginea valorii $POS(x)$ prin funcția hash aleasă, folosind un circuit zk-SNARK simplu.

Exemplu 2. În Figura 3.2 de mai jos sunt colorate cu verde toate nodurile care fac parte din vectorul $siblings_C$ pentru demonstrația de apartenență a elementului C.

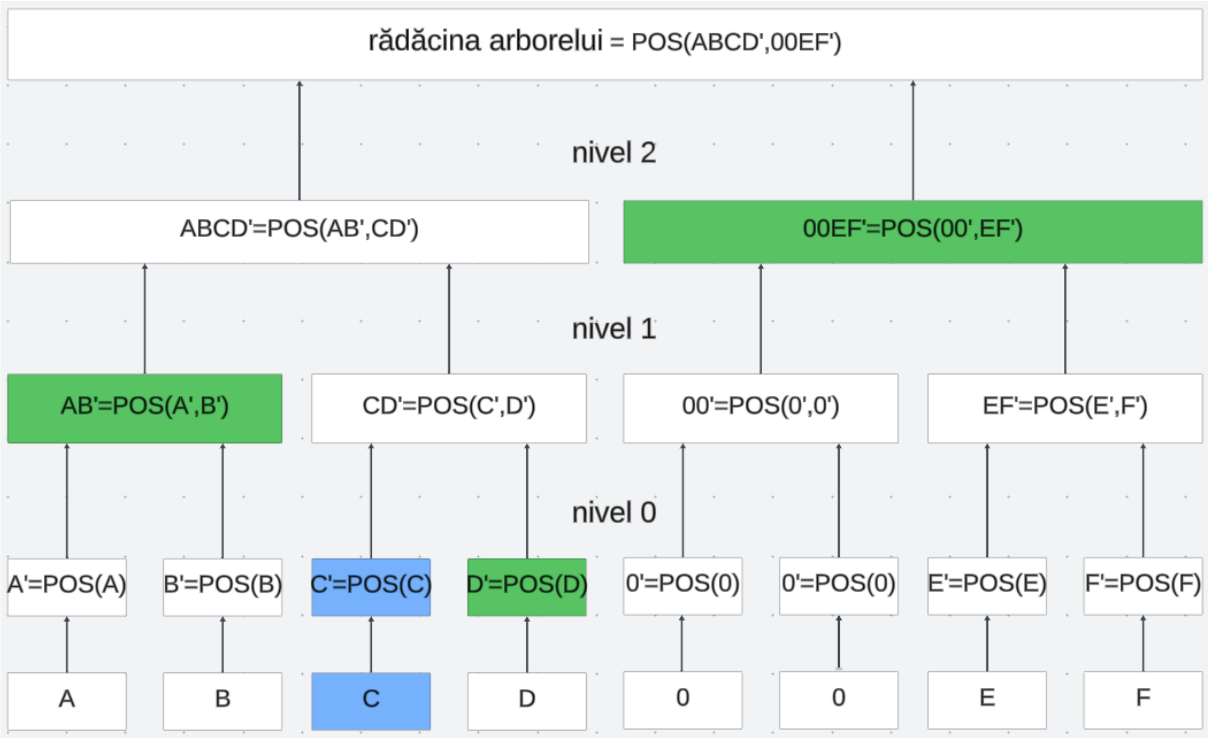

Figura 3.2: Demonstrație pentru nodul cu cheia secretă C

În acest exemplu , $Prove(x, S)$ generează vectorii $siblings_C = [D', AB', 00EF']$, $path_C = [0, 1, 0]$ împreună cu o demonstrație zK-SNARK pentru preimaginea $POS(C)$.

3.4 Verificarea demonstrației

Verificarea unei demonstrații Merkle presupune verificarea preimaginii și recalcularea rădăcinii arborelui folosind vectorii *siblings* și *path* și funcția publică hash.

Dacă rădăcinia calculată este egală cu rădăcina arborelui Merkle atunci demonstrația este acceptată, altfel este respinsă.

Verifier-ul poate să evalueze o astfel de demonstrație într-un timp logaritmic și fără să salveze în memorie întreg arborele. Deși eficientă, acestă metodă de demonstrare nu păstrează datele private deoarece Verifier-ul află poziția hash-ului cheii secrete în arbore și semnătura acesteia în cazul în care nu folosim demonstrații zero knowledge.

3.5 Demonstrații de non-apartenență

Potrivit definiției de mai sus un arbore hash Merkle este un acumulator criptografic *dinamic*, care suportă operațiile de inserare, actualizare și ștergere, însă în practică putem să executăm doar operația de inserare într-un mod sigur deoarece actualizarea sau ștergerea duc la scurgeri de informații cu privire la nodurile care au fost actualizate/șterse.

3.5.1 Modelul UTXO

Pentru a elimina un nod dintr-un arbore Merkle acesta trebuie *anulat* folosind un *nullifier*. Un nullifier este un hash-commitment compus din cheia secretă a nodului si ID-ul arborelui Merkle din care face parte, prin care se indică faptul că nodul a fost "consumat". În cazul in care dorim să actualizăm valoarea unui nod, trebuie sa anulăm nodul vechi și sa inserăm un nod cu valoarea nouă.

Valoarea unui nullifier nu trebuie să dezvăluie ce nod anulează din arborele hash și este salvată într-un alt arbore Merkle sparse numit *"Nullifier tree"*. Acest model cu doi arbori hash se numește UTXO Model (eng. *Unspent Transactions Outputs*) și este folosit de exemplu în gestionarea tranzacțiilor cu criptomonede. Modelul UTXO este util deoarece rezolvă o problemă de securitate în ceea ce privește demonstrațiile uzuale: replay attacks - folosirea repetată a aceleiași demonstrații [23].

Pentru a demonstra apartenența unui element x în modelul UTXO trebuie să demonstrăm că $POS(x)$ face parte din arborele hash-urilor și faptul ca nullifier-ul asociat cheii secrete nu se află in arborele nullifier-ilor.

3.5.2 Verificarea în modelul UTXO

Arborele nullifier-ilor *nullTree* trebuie să fie un arbore Merkle sparse cu suficiente noduri frunză pentru a cuprinde toate valorile posibile pentru funcția hash folosită. În cazul nostru, funcția hash *POSEIDON* generează valori în câmpul prim Z_p, p fiind numărul de elemente (ordinul) câmpului generat de curba eliptică ALT_BN128 [21]. Numărul p se află între $2^{253} \leq p \leq 2^{254}$, așa că avem nevoie de un arbore hash cu 254 de nivele pentru a putea acomoda toate valorile din codomeniul funcției hash POS.

Putem să folosim valorile hash / nullifier pe post de index în vectorul nodurilor frunză și să codificăm valorile 0 pentru nefolosit și 1 pentru folosit.

Demonstrația de non-aparteneță pentru un anumit nullifier/valoare hash revine la o demonstrație de apartenența a elementului "0" la poziția nullifier-ului. Structura de arbore este necesară deoarece vrem ca $Verifier$-ul să poată să verifice non-apartenența fără să memoreze toate elementele din vectorul nodurilor frunză.

Structura sparse a arborilor hash ne permite deci o verificare ușoară a non apartenenței iar demonstrația nu dezvăluie informații noi pentru Verifier deoarece funcția care leagă un nullifier de cheia pe care o anulează este o funcție hash criptografică.

Demonstrația zero knowledge completă va trebui să conțină următoarele:

- demonstrația de apartenență în arborele hash pentru cheia secretă
- demonstrația că nullifier-ul a fost calculat corect
- demonstrația de non-apartenență în arborele nullTree pentru nullifier

3.5.3 Problema demonstrațiilor de non-apartenență

Deși demonstrația de non-apartenență este simplă și are un timp constant aceasta nu poate să fie folosită în mod eficient în SNARK-uri deoarece arborele nullifier-ilor *nullTree* are o adâncime foarte mare și necesită executarea funcției hash pentru fiecare nivel. Funcțiile hash reprezintă o operație foarte costisitoare în contextul circuitelor algebrice folosite în SNARK-uri și deși există funcții hash optimizate pentru acest mediu, precum funcția hash $POSEIDON$, faptul că trebuie să apelăm funcția de 254-ori pentru a demonstra non-apartenența unui nullifier va duce la probleme de scalabilitate.

Capitolul 4

Implementarea protocolului folosind Circom 2.0 și snarkJS

În această secțiune sunt prezentate limbajul Circom, biblioteca snarkJS și implementările tuturor circuitelor folosite în protocolul de demonstrare a apartenenței folosind arbori indexați și arbori hash.

4.1 Circom 2.0

Circom [16][17][18] este un limbaj low-level și un compilator pentru circuite scris în *Rust*. Acesta este folosit împreună cu librăria *snarkJS* pentru a genera și verifica demonstrații zero-knowledge eficiente.

Limbajul lucrează cu un singur tip de date, numit semnal(eng. signal) care poate avea diverse nivele de acces: public/privat și care poate fi folosit în diverse părți ale unui circuit: semnal de intrare, semnal intermediar și semnal de ieșire. Dacă nu este specificat altfel atunci semnalele de intrare sunt private. Semnalele intermediare sunt private și nu pot fi modificate iar semnalele de iesire sunt publice și nu pot fi modificate. Semnalele pot lua valori doar din câmpul finit F_p unde p este ordinul grupului generat de curba eliptică ALT_BN128 [21].

Fiecare circuit împreună cu semnalele sale trebuie să fie scris sub forma unui Quadratic Arithmetic Program (QAP), adică fiecare semnal intermediar sau de ieșire are forma:

$signalOUT === (A * signalINTER1 + B) * (C * signalINTER2 + D)$ unde $A, B, C, D \in F_P$.

Pe lângă operatorii aritmetici clasici, Circom folosește și operatori care vor fi folosiți la compilare pentru generarea de constrângeri asupra semnalelor. Avem astel operatorul de constrângere "===" , operatorul de atribuire fără constrângere (la stânga/dreapta) $< --/-- >$ și operatorul de atribuire împreună cu constrângere (la stânga/dreapta) <==/==>.

Operator	Forma	Descriere
===	expr1 === expr2	Creează o constrângere nouă expr1=expr2
< −−	sign< −−expr	Atribuie valoarea expresiei din operandul drept
−− >	expr−− >sign	Atribuie valoarea expresiei din operandul stâng
<==	sign<==expr	Echivalent cu (sign< −−expr și sign === expr)
==>	expr==>sign	Echivalent cu (expr−− >sign și expr === sign)

Tabela 4.1: Operatorii specifici limbajului Circom

Scrierea circuitelor, notate cu *Template* atunci când sunt definite și cu *Component* atunci când sunt instanțiate, permite o dezvoltare modulară deoarece semnalele de ieșire ale unui circuit pot fi redirecționate către semnalele de intrare ale unui alt circuit.

Odată scrise, circuitele pot fi construite și compilate în fișiere *.r1cs (formatul sistemelor de constrângeri, Rank 1 Constraint System).

4.2 snarkJS

snarkJS [19] este o bibliotecă JavaScript care face parte din ecosistemul Circom. Este responsabilă de generarea *trusted setup*-ului și a cheilor de verificare și demonstrare pentru diverse protocoale SNARK precum: GROTH16 [15], PLONK [11] și FFLONK [10].

Folosind funcțiile din biblioteca snarkJS putem să generăm și să verificăm demonstrații zero-knowledge în browser/server sau într-un mediu decentralizat precum Ethereum EVM folosind contractele solidity generate de către circom.

4.3 Circuite utilitare

Circuitele utilitare prezentate mai jos nu au legătură directă cu protocolul de demonstrare a apartenenței însă aduc un nivel ridicat de modularitate și separă funcționalitatea codului. Acestea rezolvă o problemă care apare în Circom atunci când încercăm să comparăm valori foarte mari (apropiate de ordinul câmpului F_p).

Un dezavantaj al limbajului Circom este faptul că semnalele pot lua valori doar în câmpul F_p și nu pot avea valorile *true/false* deci nu putem folosi operatorii de comparație ($<, >, =, \leq, \geq$) cu operanzi semnale.

Librăria standard Circom implementează două circuite *"LessThan" și "GreaterThan"* însă acestea sunt limitate la valori $\leq 2^{252}$, deoarece operațiile de comparare nu sunt atât de comune, și nu acoperă toate valorile posibile din codomeniul F_p al funcției hash POSEIDON, valori hash pe care dorim să le comparăm.

Pentru a putea compara toate valorile din F_p, vom extinde circuitul *"LessThan"*. Acesta va transforma întâi operanzii în reprezentări binare pe 256 biți (folosim reprezentarea cu cel mai important bit la dreapta) și va compara fiecare bit pe rând folosind circuitul *"LessThan"* original. Cum în Circom nu avem instrucțiuni de salt (precum *break/jump to*) trebuie să parcurgem toate cele două secvențe de biți așa că pentru a evita apelurile excesive la circuitul *"LessThan"*, vom folosi *chunk*-uri de câte n biți (în implementare folosim 4 chunk-uri de 64 biți). În continuare sunt prezentate toate circuitele folosite pentru a extinde circuitul original.

Listing 4.1: "Num2Bits.circom" [7]

```
template Num2Bits(n){
    signal input in;
    signal output out[n];

    var value = 0;
    var pow = 1;
    for(var i=0;i<n;i++){
        out[i] <-- (in>>i)&1;
        out[i]*(1-out[i]) === 0;
        value += out[i]*pow;
        pow += pow;
    }

    value === in;}
```

Circuitul **Num2Bits(n)**[7] este folosit pentru a genera secvența binară de lungime n asociată numarului trimis prin semnalul de intrare **in**. Avem n semnale de ieșire corespunzatoare fiecărui bit , cel mai important fiind pe poziția **out[255]** (Most significant bit right). În implementare vom folosi $n = 256$ pentru a putea acoperi toate elementele din F_p.

Listing 4.2: "Bits2Num.circom" [7]

```
template Bits2Num(n){
    signal input in[n];
    signal output out;

    var value = 0;
    var pow = 1;
    for(var i=0;i<n;i++){
        in[i]*(1-in[i]) === 0;
        value += pow*in[i];
        pow +=pow;
    }
    out <== value;
}
```

Bits2Num(n)[7] este folosit pentru a calcula elementul din F_p echivalent cu secvența binară trimisă prin cele n semnalele de intrare **in[n]**. Rezultatul este calculat modulo p și trimis prin semnalul de ieșire **out**.

Listing 4.3: "LessThan" [8]

```
template LessThan(){
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(253);

    n2b.in <== in[0]+(1<<252)-in[1];
    out <== 1-n2b.out[252];
}
```

Circuitul **LessThan** face parte din librăria standard Circom [8]. Acesta primește 2 semnale de intrare **in[0]** și **in[1]** și returnează prin semnalul de ieșire **out**: 1 dacă $in[0] < in[1]$ și 0 altfel.

Listing 4.4: "isZero.circom" [8]

```
template isZero(){
    signal input in;
    signal output out;
```

```
    signal inv <-- in == 0 ? 0 : 1/in;
    out <== 1 - in*inv;
    out*in === 0;
}
```

isZero face parte din librăria standard Circom [8]. Acesta primește un singrul semnal de intrare **in** și returnează prin semnalul de ieșire **out**: 1 dacă semnalul de intrare este 0 și 0 altfel.

Listing 4.5: ”isEqual.circom” [8]

```
include "./isZero.circom";

template isEqual(){
    signal input in[2];
    signal output out;
    component checkZero = isZero();
    checkZero.in<== in[0]-in[1];
    out<==checkZero.out;
}
```

Circuitul **isEqual** face parte din librăria standard Circom [8]. Acesta primește două semnale de intrare **in[0]** și **in[1]** , calculează diferența dintre cele două pe care apoi o trimite ca și semnal de intrare într-un circuit ”isZero”. Semnalul de ieșire este preluat de la semnalul de ieșire al circuitului isZero folosit.

4.4 Circuitul LessThan_256BIT_MSBR

Listing 4.6: "LessThan_256BIT_MSBR"

```
include "./Num2Bits.circom";
include "./Bits2Num.circom";
include "./LessThan.circom";
include "./isEqual.circom";

template LessThan_256BIT_MSBR(){
    signal input in[2];
    signal inter1[4];
    signal inter2[4];
    signal output out;
    component n2b = Num2Bits(256);
    component b2n[4] ;
    n2b.in <== in[0];
    for(var i=0;i<4;i++){
        b2n[i] = Bits2Num(64);
        for(var j=0;j<64;j++){
            b2n[i].in[j]<==n2b.out[j+(64*i)];
        }
        inter1[i]<==b2n[i].out;
    }
    component n2b_2 = Num2Bits(256);
    component b2n_2[4] ;
    n2b_2.in <== in[1];
    for(var i=0;i<4;i++){
        b2n_2[i] = Bits2Num(64);
        for(var j=0;j<64;j++){
            b2n_2[i].in[j]<==n2b_2.out[j+(64*i)];
        }
        inter2[i]<==b2n_2[i].out;
    }

    signal interBINcomp1[4];
    signal interBINcomp2[4];
    component interLT[4];
    component isEq[4];
```

```
        for(var i=0;i<4;i++){
            interLT[i] = LessThan();
            isEq[i] = isEqual();
            interLT[i].in[0]<==inter2[i];
            interLT[i].in[1]<==inter1[i];
            isEq[i].in[0]<==inter2[i];
            isEq[i].in[1]<==inter1[i];
            interBINcomp1[i]<== interLT[i].out;
            interBINcomp2[i]<== (1-interLT[i].out)-isEq[i].out;
        }

        signal num[2];
        component b2n_f[2];
        b2n_f[0]=Bits2Num(4);
        b2n_f[0].in[0]<== interBINcomp1[0];
        b2n_f[0].in[1]<== interBINcomp1[1];
        b2n_f[0].in[2]<== interBINcomp1[2];
        b2n_f[0].in[3]<== interBINcomp1[3];
        b2n_f[1]=Bits2Num(4);
        b2n_f[1].in[0]<== interBINcomp2[0];
        b2n_f[1].in[1]<== interBINcomp2[1];
        b2n_f[1].in[2]<== interBINcomp2[2];
        b2n_f[1].in[3]<== interBINcomp2[3];

        num[0]<== b2n_f[0].out;
        num[1]<== b2n_f[1].out;

        component LTF = LessThan();
        LTF.in[0]<==num[0];
        LTF.in[1]<==num[1];
        out<==LTF.out;
}
```

Circuitul **LessThan_256BIT_MSBR** folosește toate cele 5 circuite prezentate mai sus și extinde circuitul *LessThan* deoarece permite compararea tuturor valorilor din F_p. Pentru a trece de limitarea circuitului *LessThan* din librăria standard, vom reprezenta valorile semnalelor de intrare **in[0]** și **in[1]** pe 256 biți și vom compara pe rând 4 segmente de 64 biți din fiecare secvență. Dacă un segment este mai mare atunci îl înlocuim cu 1, altfel îl înlocuim cu 0. La sfârșit semnalele de intrare ajung să conțină 2 secvențe de 4 biți, care sunt convertite inapoi la elemente din F_p folosind circuitul *Bits2Num* și comparate cu *LessThan* pentru a returna rezultatul final: 1 dacă $in[0] < in[1]$ și 0 altfel.

4.5 Circuite secundare

În acestă secțiune sunt prezentate circuitele secundare create pentru a realiza demonstrații în arbori hash normali și indexați.

Listing 4.7: "Selector.circom"

```
template Selector(){
    signal input switcher;
    signal input in[2];
    signal int[4];
    signal output out[2];
    0 === (switcher)*(1-switcher);
    int[0] <== in[0]*(1-switcher);
    int[1] <== in[1]*switcher;
    out[0] <== int[0] + int[1];

    int[2] <== in[1]*(1-switcher);
    int[3] <== in[0]*switcher;
    out[1] <== int[2] + int[3];

}
```

Selector este folosit pentru a seta semnalele de intrare ale funcției hash POSEIDON pentru fiecare nivel din arbore. Circuitul primește 3 semnale de intrare, **in[0]**, **in[1]**, **switcher** și returnează semnalele inversate dacă $switcher = 1$, altfel returnează aceleași semnale. În generarea demonstrațiilor, pentru fiecare vector de hash-uri *"siblings"* este generat și un vector de aceiași dimensiune, numit *path*, ce conține codificările poziției (0-stânga și 1-dreapta) hash-ului curent pentru a calcula următorul hash. Circuitul este apelat întotdeauna cu hash-ul curent prin **in[0]** și cu hash-ul sibling prin **in[1]**.

Listing 4.8: "HashTreeLevel.circom"

```
include "../node_modules/circomlib/circuits/poseidon.circom";
include "./Selector.circom";

template HashTreeLevel(){
    signal input in[2];
    signal input position;
    signal output out;
```

```
    component poseidon = Poseidon(2);
    component selector = Selector();

    selector.in[0] <== in[0];
    selector.in[1] <== in[1];
    selector.switcher <== position;

    selector.out[0] ==> poseidon.inputs[0];
    selector.out[1] ==> poseidon.inputs[1];

    poseidon.out ==> out;

}
```

HashTreeLevel este folosit pentru a calcula următorul nivel dintr-un arbore Merkle dat. Circuitul primește hash-ul din nivelul anterior împreună cu hash-ul sibling din nivelul curent și ordinea în care apar în apelul funcției hash POSEIDON. Acest ciruit este folosit exclusiv în demonstrațiile de apartenență în arbori hash simpli, în timp ce demonstrațile de apartență în arbori indexați au o structură similară dar necesită mai multe verificări și constrângeri și sunt tratate direct în circuitul principal.

4.6 Circuitul principal

Listing 4.9: "MainProof.circom"

```
include "../node_modules/circomlib/circuits/poseidon.circom";
include "./HashTreeLevel.circom";
include "./LessThan_256BIT_MSBR.circom";

template MainProof(depth){
    signal input sk;
    signal input siblingsPk[depth];
    signal input path[depth];
    signal input nullifierHash;
    signal input lowLeafHashValue;
    signal input lowHash;
    signal input nextIndex;
    signal input highHash;
    signal input nullifierTreeSiblingsPk[depth];
    signal input nullifierTreePath[depth];
    signal input root;
    signal input nullifierRoot;
    signal input nodeTreeID;
    signal input nullifierTreeID;

    //proof of membership
    signal intermed[depth+1];
    component levelChecker[depth];
    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== sk;
    poseidon.out ==> intermed[0];
    for(var i=0;i<depth;i++){
        levelChecker[i] = HashTreeLevel();
        levelChecker[i].in[0] <== intermed[i];
        levelChecker[i].in[1] <== siblingsPk[i];
        levelChecker[i].position <== path[i];
        levelChecker[i].out ==> intermed[i+1];
    }
    intermed[depth] === root;
    //proof of nullifier
    component poseidonNullifier = Poseidon(3);
```

```
poseidonNullifier.inputs[0]<==sk;
poseidonNullifier.inputs[1]<==nodeTreeID;
poseidonNullifier.inputs[2]<==nullifierTreeID;
poseidonNullifier.out === nullifierHash;

// proof that lowhash is lower than nullifier and highhash
//is higher than the nullifier
component LT256[2] ;
LT256[0] = LessThan_256BIT_MSBR();
LT256[0].in[0]<==lowHash;
LT256[0].in[1]<==nullifierHash;
LT256[0].out === 1;

LT256[1] = LessThan_256BIT_MSBR();
LT256[1].in[0]<==nullifierHash;
LT256[1].in[1]<==highHash;
LT256[1].out === 1;

//proof that the 3 items lowhash,next index and highhash ,
//hash into the leaf node of the nullfier
component poseidonLowLeafHashValue = Poseidon(3);
poseidonLowLeafHashValue.inputs[0] <== lowHash;
poseidonLowLeafHashValue.inputs[1] <== nextIndex;
poseidonLowLeafHashValue.inputs[2] <== highHash;
poseidonLowLeafHashValue.out === lowLeafHashValue;

//proof of membership for the lowLeafHashValue same as
//proof of non - membership for the nullifier
signal intermedNULL[depth+1];

component levelCheckerNullifier[depth];
lowLeafHashValue ==> intermedNULL[0];

for(var i=0;i<depth;i++){
    levelCheckerNullifier[i] = HashTreeLevel();
    levelCheckerNullifier[i].in[0] <== intermedNULL[i];
    levelCheckerNullifier[i].in[1] <== nullifierTreeSiblingsPk[i];
    levelCheckerNullifier[i].position <== nullifierTreePath[i];
    levelCheckerNullifier[i].out ==> intermedNULL[i+1];
```

```
    }
    intermedNULL[depth] === nullifierRoot;
}
component main {public [root,nullifierRoot,nodeTreeID,nullifierTreeID,
nullifierHash]} = MainProof(16);
```

Circuitul principal folosește toate circuitele descrise mai sus. Acesta este responsabil pentru constrângerile a cinci demonstrații:

1. **demonstrația de apartenență la arborele hash** - această parte folosește semnalele de intrare private **sk**, **siblingsPk[depth]**, **path[depth]** și semnalul public **root**. Cheia secretă **sk** reprezintă preimaginea hash-ului pentru care se face demonstrația și pornind de la aceasta se calculează fiecare nivel din arbore folosind circuitul *HashTreeLevel*. La sfârșit verificăm că ultimul hash obținut este egal cu rădăcina arborelui din semnalul **root**. Semanele **sk**, **path[depth]** și **siblingsPk[depth]** sunt private deoarece oricare din ele dezvăluie informații Verifier-ului despre nodul țintă al demonstrației. Semnalul **root** este public deoarece hash-ul rădăcinii și hash-urile nodurilor frunză sunt publice prin definiție.

2. **calculul corect al nullifier-ului** - nullifier-ul este un *hash commitment* între cheia secretă **sk** (semnal privat), ID-ul arborelui hash **nodeTreeID** (semnal public) și ID-ul arborelui indexat de nullifieri **nullifierTreeID** (semnal public) calculat folosind funcția POSEIDON astfel :

   $nullifierHash = POSEIDON([sk, nodeTreeID, nullifierTreeID])$ Circuitul principal se asigură astfel că semnalul public **nullifierHash** este calculat corect. Hash-ul nullifier-ului poate să fie public deoarece acesta nu poate să fie asociat cu niciun elemnent din mulțimea de elemente atunci când funcția hash folosită este criptografică.

3. **demonstrația că hash-ul nullifier-ului se află în intervalul** $(lowHash, highHash)$. Folosind circuitul *LessThan 256BIT MSBR* ne asigurăm că nullifier-ul se află între semnalele private **lowHash** și **highHash**.

4. **calculul corect al nodului frunză din arborele nullifier-ilor** - fiecare nod frunză din arborele indexat al nullifier-ilor este un *hash commitment* al fiecărui nod din lanțul ordonat de forma $\{lowHash, nextIndex, highHash\}$(toate semnalele private), notat în circuit cu semnalul privat **lowLeafHashValue**. Ne asigurăm astfel că valorile **lowHash** și **highHash** sunt valide și se respectă ordinea din lanț.

5. **demonstrația de non-apartenență la arborele nullifier-ilor** - se demonstrează că valoarea *hash commitment-ului* **lowLeafHashValue** face parte din arborele indexat și deci valorea nullifier-ului nu a fost introdusă încă în arbore.

În total după compilarea circuitului pentru o adâncime de 16 nivele, obținem un R1CS cu următoarele proprietăți :

```
snarkjs r1cs info build/MainProof.r1cs
[INFO]  snarkJS: Curve: bn-128
[INFO]  snarkJS: # of Wires: 12168
[INFO]  snarkJS: # of Constraints: 13183
[INFO]  snarkJS: # of Private Inputs: 69
[INFO]  snarkJS: # of Public Inputs: 5
[INFO]  snarkJS: # of Labels: 32265
[INFO]  snarkJS: # of Outputs: 0
```

Bibliografie

[1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy și Tyge Tiessen, *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*, Cryptology ePrint Archive, Paper 2016/492, https://eprint.iacr.org/2016/492, 2016, URL: https://eprint.iacr.org/2016/492.

[2] Tomer Ashur, Al Kindi, Willi Meier, Alan Szepieniec și Bobbin Threadbare, *Rescue-Prime Optimized*, Cryptology ePrint Archive, Paper 2022/1577, https://eprint.iacr.org/2022/1577, 2022, URL: https://eprint.iacr.org/2022/1577.

[3] Tomer Ashur, Mohammad Mahzoun, Jim Posen și Danilo Šijačić, *Vision Mark-32: ZK-Friendly Hash Function Over Binary Tower Fields*, Cryptology ePrint Archive, Paper 2024/633, https://eprint.iacr.org/2024/633, 2024, URL: https://eprint.iacr.org/2024/633.

[4] Foteini Baldimtsi, Ioanna Karantaidou și Srinivasan Raghuraman, *Oblivious Accumulators*, Cryptology ePrint Archive, Paper 2023/1001, https://eprint.iacr.org/2023/1001, 2023, URL: https://eprint.iacr.org/2023/1001.

[5] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan și Dimitris Kolonelos, *Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular*, Cryptology ePrint Archive, Paper 2019/1255, https://eprint.iacr.org/2019/1255, 2019, DOI: 10.1007/s10623-023-01245-1, URL: https://eprint.iacr.org/2019/1255.

[6] Lily Chen, Dustin Moody, Andrew Regenscheid și Angela Robinson, *Digital Signature Standard (DSS)*, en, 2023-02-02 05:02:00 2023, DOI: https://doi.org/10.6028/NIST.FIPS.186-5, URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=935202.

[7] *CIRCOM 2.0 bitify*, https://github.com/iden3/circomlib/blob/master/circuits/bitify.circom, Ultima accesare: 2024-05-19.

[8] *CIRCOM 2.0 comparators*, https://github.com/iden3/circomlib/blob/master/circuits/comparators.circom, Ultima accesare: 2024-05-19.

[9] Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic și Arthur Gervais, *Do You Need a Zero Knowledge Proof?*, Cryptology ePrint Archive, Paper 2024/050, https://eprint.iacr.org/2024/050, 2024, URL: https://eprint.iacr.org/2024/050.

[10] Ariel Gabizon și Zachary J. Williamson, *fflonk: a Fast-Fourier inspired verifier efficient version of PlonK*, Cryptology ePrint Archive, Paper 2021/1167, https://eprint.iacr.org/2021/1167, 2021, URL: https://eprint.iacr.org/2021/1167.

[11] Ariel Gabizon, Zachary J. Williamson și Oana Ciobotaru, *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*, Cryptology ePrint Archive, Paper 2019/953, https://eprint.iacr.org/2019/953, 2019, URL: https://eprint.iacr.org/2019/953.

[12] Shafi Goldwasser, Silvio Micali și Chales Rackoff, „The knowledge complexity of interactive proof-systems”, în Oct. 2019, ISBN: 9781450372664, DOI: 10.1145/3335741.3335750.

[13] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy și Markus Schofnegger, *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*, Cryptology ePrint Archive, Paper 2019/458, https://eprint.iacr.org/2019/458, 2019, URL: https://eprint.iacr.org/2019/458.

[14] Green și Blaze, *Zero Knowledge Proofs: An illustrated primer – A Few Thoughts on Cryptographic Engineering*, URL: https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/, (accessed: 26.04.2024).

[15] Jens Groth, *On the Size of Pairing-based Non-interactive Arguments*, Cryptology ePrint Archive, Paper 2016/260, https://eprint.iacr.org/2016/260, 2016, URL: https://eprint.iacr.org/2016/260.

[16] *Official Circom 2.0 Documentation*, https://docs.circom.io/, Ultima accesare: 2024-05-19.

[17] *Official Circom 2.0 Github repository*, https://github.com/iden3/circom, Ultima accesare: 2024-05-19.

[18] *Official Iden3 Documentation*, https://docs.iden3.io/circom-snarkjs/, Ultima accesare: 2024-05-19.

[19] *Official snarkJS Github repository*, https://github.com/iden3/snarkjs/blob/master/README.md, Ultima accesare: 2024-05-19.

[20] Charalampos Papamanthou, Shravan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets și Stjepan Golemac, *Reckle Trees: Updatable Merkle Batch Proofs with Applications*, Cryptology ePrint Archive, Paper 2024/493, https://eprint.iacr.org/2024/493, 2024, URL: https://eprint.iacr.org/2024/493.

[21] Christian Reitwiessner, *EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128*, URL: https://eips.ethereum.org/EIPS/eip-196, (accessed: 02.05.2024).

[22] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno și Srinath Setty, *Transparency Dictionaries with Succinct Proofs of Correct Operation*, Cryptology ePrint Archive, Paper 2021/1263, https://eprint.iacr.org/2021/1263, 2021, URL: https://eprint.iacr.org/2021/1263.

[23] Dionysis Zindros, *Lecture 10: Accounts Model and Merkle Trees*, Stanford, Spring online lecture, https://web.stanford.edu/class/ee374/lec_notes/lec10.pdf, 2022, URL: https://web.stanford.edu/class/ee374/lec_notes/lec10.pdf.