



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**

Demonstrațiile zero-knowledge au devenit tot mai populare în ultimii ani datorită progreselor cu privire la aplicabilitatea practică a acestora. Suportul primit din partea cercetătorilor și al companiilor din domeniu au făcut ușoară dezvoltarea de noi protocoale zero-knowledge pe mașini virtuale, în limbaje de programare sau pe sisteme dedicate cu accelerare hardware. Dimensiunea succintă a demonstrațiilor, non-interactivitatea și demonstrarea cunoașterii unei informații fără a o dezvălui fac ca demonstrațiile zero-knowledge să fie o soluție atractivă pentru o gamă largă de probleme. Deși au aceste proprietăți speciale, aceste tehnologii nu au ca scop să înlocuiască protocoalele curente ci să le îmbunătățească, adăugând un nivel nou de securitate și anonimitate. Lucrarea de față își propune să introducă tehnicile folosite în demonstrațiile zero-knowledge de apartenență la mulțimi, prin prezentarea noțiunilor de bază, prin studiul și compararea unor algoritmi existenți și prin propunerea unor îmbunătățiri ale metodelor curente.

## **Abstract**

Zero-knowledge proofs have become increasingly popular in recent years due to the advances in their practical applicability. The support received from researchers and companies in the field has made it easy to develop new zero-knowledge protocols on virtual machines, in programming languages or on dedicated hardware-accelerated systems. The succinctness of proofs, non-interactivity and proving knowledge of information without revealing it make zero-knowledge proofs an attractive solution for a wide range of problems. Even if zero-knowledge proofs have such special properties, these technologies are not intended to replace current protocols but instead they are intended to improve them, adding a new level of security and anonymity. The present paper aims to introduce the techniques used in zero-knowledge membership proofs, by presenting the basic notions, by studying and comparing some existing algorithms, and by proposing some improvements to the current methods.

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>5</b>
1.1	Motivație și obiective . . . . .	5
1.2	Contribuție personală . . . . .	5
1.3	Structura lucrării . . . . .	6
<b>2</b>	<b>Demonstrații zero knowledge</b>	<b>7</b>
2.1	Scurt istoric . . . . .	7
2.2	Preliminarii teoretice . . . . .	8
<b>3</b>	<b>Acumulatori criptografici</b>	<b>9</b>
3.1	Definiții și proprietăți . . . . .	9
3.2	Date private și scalabilitate . . . . .	10
3.3	Arbori hash Merkle . . . . .	11
3.4	Demonstrații de apartenență . . . . .	12
3.5	Verificarea demonstrației . . . . .	13
3.6	Demonstrații de non-apartenență . . . . .	13
3.6.1	Modelul UTXO . . . . .	14
3.6.2	Verificarea în modelul UTXO . . . . .	14
3.6.3	Problema demonstrațiilor de non-apartenență . . . . .	15
3.7	Arbori hash indexați . . . . .	15
3.7.1	Construcția arborelui și algoritmul de inserție . . . . .	16
<b>4</b>	<b>Implementarea protocolului folosind CIRCOM 2.0 și snarkJS</b>	<b>19</b>
4.1	CIRCOM 2.0 . . . . .	19
4.2	snarkJS . . . . .	20
4.3	Circuite utilitare . . . . .	20
4.4	Circuitul LessThan_256BIT_MSBR . . . . .	23
4.5	Circuite secundare . . . . .	25
4.6	Circuitul principal . . . . .	27
4.7	Compilarea și rularea circuitelor . . . . .	30
4.8	Compararea celor doi arbori . . . . .	32

4.9 Exemplu de demonstrație . . . . .	33
<b>5 Concluzii</b>	<b>35</b>
<b>Bibliografie</b>	<b>37</b>

# 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 (ex.: protocolul Zcash pentru Bitcoin și protocolul Tornado Cash pentru Ether), votul electronic descentralizat și anonim (ex.: putem să demonstrăm ca avem dreptul să votăm fără să dezvăluim date personale) sau mai general, folosirea anonimă a unor servicii online (ex.: fără să folosim username/password).

Când privim problema demonstrațiilor de apartenență/non-apartenență apar două întrebări, legate de performanța verificării demonstrațiilor și confidențialitatea datelor, la care dorim să răspundem în această lucrare:

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

### 1.2 Contribuție 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 și non-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* [27], care permit reducerea adâncimii arborelui nullifier-ilor și implicit a numărului de operații necesare pentru demonstrația de non-apartenență. Codul sursă este publicat pe Github [10].

## 1.3 Structura lucrării

Lucrarea este împărțită în 5 capitole:

- Capitolul 1 prezintă motivația studiului acestui domeniu și obiectivele vizate de lucrarea prezentă.
- Capitolul 2 prezintă concepte teoretice necesare pentru înțelegerea subiectului, împreună cu un scurt istoric al demonstrațiilor zero-knowledge.
- Capitolul 3 prezintă noțiunile de acumulator criptografic, arbore Merkle și model UTXO. Sunt prezentate metodele curente de demonstrare și problemele care apar atunci când dorim să trecem într-un mediu zero-knowledge. Sunt introduși de asemenea și arbori hash "indexați", arbori cu proprietăți speciale care îi fac ușor de folosit în demonstrațiile zero-knowledge.
- Capitolul 4 prezintă limbajul folosit pentru scrierea circuitelor, CIRCOM 2.0, împreună cu framework-ul snarkJS folosit pentru generarea și verificarea demonstrațiilor. Sunt prezentate implementările tuturor circuitelor folosite în demonstrațiile zero-knowledge de apartenență/non-apartenență.
- Capitolul 5 prezintă o concluzie cu privire la ideile expuse în această lucrare.

# 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* [14]. Aceștia încercau să 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 [11].

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 [16].

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 [16] [11].

A 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 [16].

## 2.2 Preliminarii teoretice

*Notăție 1.* Vom nota cu  $P$  și  $V$  cele două entități implicate în procesul de demonstrare, *Prover* și *Verifier*.

**Definiție 2.2.1** (ZKP [11]). Dat fiind un sistem de demonstrație  $(P, V)$ , acest sistem este *Zero-Knowledge Proof (ZKP)*, dacă satisface următoarele trei proprietăți:

- **Completitudine:** Un Verifier acceptă întotdeauna o demonstrație pentru o propoziție adevărată.
- **Corectitudine:** Un Verifier onest acceptă o demonstrație mincinosă cu o probabilitate neglijabilă.
- **Zero Knowledge:** După demonstrație  $V$  nu obține nicio informație nouă iar datele folosite de  $P$  rămân confidențiale.

**Definiție 2.2.2** (zk-SNARK [11]). *zero knowledge Succinct Noninteractive Argument of knowledge*, prescurtat *zk-SNARK*, reprezintă o primitivă criptografică care poate să genereze într-un mod eficient un protocol zero knowledge pentru orice problemă sau funcție. Această primitivă 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.

**Ideea de bază:** Se transformă problema (ex.: logaritm discret, 3-colorarea grafului) într-o funcție ale cărei intrări vrem să le ascundem. Apelăm apoi funcția folosind valorile secrete 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 2.2.3** (Funcție hash criptografică [6]). O funcție hash criptografică este o funcție  $h : D \rightarrow R$ , unde domeniul  $D = \{0, 1\}^*$  și codomeniul  $R = \{0, 1\}^n$  pentru  $n \in N$  cu  $n \geq 1$ , care satisface următoarele proprietăți:

- **rezistența la preimage:** 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  $h(x) = h(y)$ .

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



# Capitolul 3

## Acumulatori criptografici

### 3.1 Definiții și proprietăți

**Definiție 3.1.1** (Acumulator criptografic [5]). 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  $x$ .
- $\{0, 1\} \leftarrow Verify(A, \pi_x)$  - acceptă sau respinge demonstrația  $\pi_x$  folosind valoarea hash  $A$ .

Această reprezentare permite generarea de demonstrații de apartenență sau non-apartenență, notate  $w_x$  și numite martor (în eng. witness), pentru orice element  $x$  dintr-o mulțime suport  $S$ .

**Definiție 3.1.2.** [4] Acumulatorii pot fi clasificați după tipul de demonstrație suportat:

- **acumulatori pozitivi** - suportă doar demonstrații de apartenență.
- **acumulatori negativi** - suportă doar demonstrații de non-apartenență.
- **acumulatori universali** - suportă ambele tipuri de demonstrații.

**Definiție 3.1.3.** [4] Acumulatorii pot fi clasificați și după metodele de actualizare pe care aceștia le suportă:

- **acumulatori aditivi** - suportă doar introducerea de elemente noi în mulțime.
- **acumulatori substractivi** - suportă doar eliminarea de elemente din mulțime.

- **acumulatori dinamici** - suportă ambele operații descrise mai sus.

**Definiție 3.1.4.** [4] Acumulatorii criptografici pot funcționa într-un mod centralizat, în care acumulatorul se numește *trapdoor-based* și avem o entitate responsabilă pentru actualizarea acumulatorului, sau pot funcționa într-un mod descentralizat, în care acumulatorul se numește *trapdoorless*. 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 actualizeze, să șteargă elemente și să genereze demonstrații într-un mod eficient. Acumulatorii *trapdoorless* permit actualizări publice asupra mulțimii suport, fără a mai fi nevoie de o parte terță de încredere (ex.: în mediul blockchain).

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

## 3.2 Date private și scalabilitate

Din definiția dată mai sus acumulatorii criptografici nu au nicio proprietate 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 actualizat 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 când 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  $x$  și 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  $\pi_x$  și complexitatea în 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 în timp  $O(\log(|S|))$ .

### 3.3 Arbori hash Merkle

**Definiție 3.3.1.** [25] 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.

Funcția hash folosită într-un arbore Merkle este importantă deoarece afectează direct performanța demonstrațiilor. 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 ne dorim o funcție care să fie ușor de scris sub forma unui circuit aritmetic 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* [15], *MiMC* [1], *Vision Mark-32* [3] sau *Rescue* [2].

**Definiție 3.3.2** (Funcția hash Poseidon [15]). Funcția hash Poseidon este o funcție hash criptografică  $POS : F_P^n \rightarrow F_P$  care primește o secvență de lungime  $n \in N, n \geq 1$  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.

În cazul nostru, CIRCOM 2.0 folosește ordinul grupului de puncte generat de curba eliptică *ALT\_BN128* [26].

*Notăție 2.* 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  $S = \{A, B, \dots, F\}$ . Nodurile frunză conțin doar hash-urile  $\{POS(A), POS(B), \dots, POS(F)\}$ , iar nodurile care nu au o valoare setată primesc o valoare *null* notată în figuri cu  $\emptyset$ .

Nodurile cu valoare null permit implementarea eficientă a arborilor hash *sparse* deoarece putem pre-calcula valoarea null pentru fiecare nivel de adâncime. De exemplu în figura 3.1, vom folosi direct valoarea null pentru nivelul 1, în loc să apelăm funcția *POS*.

*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 aceeași poziție.

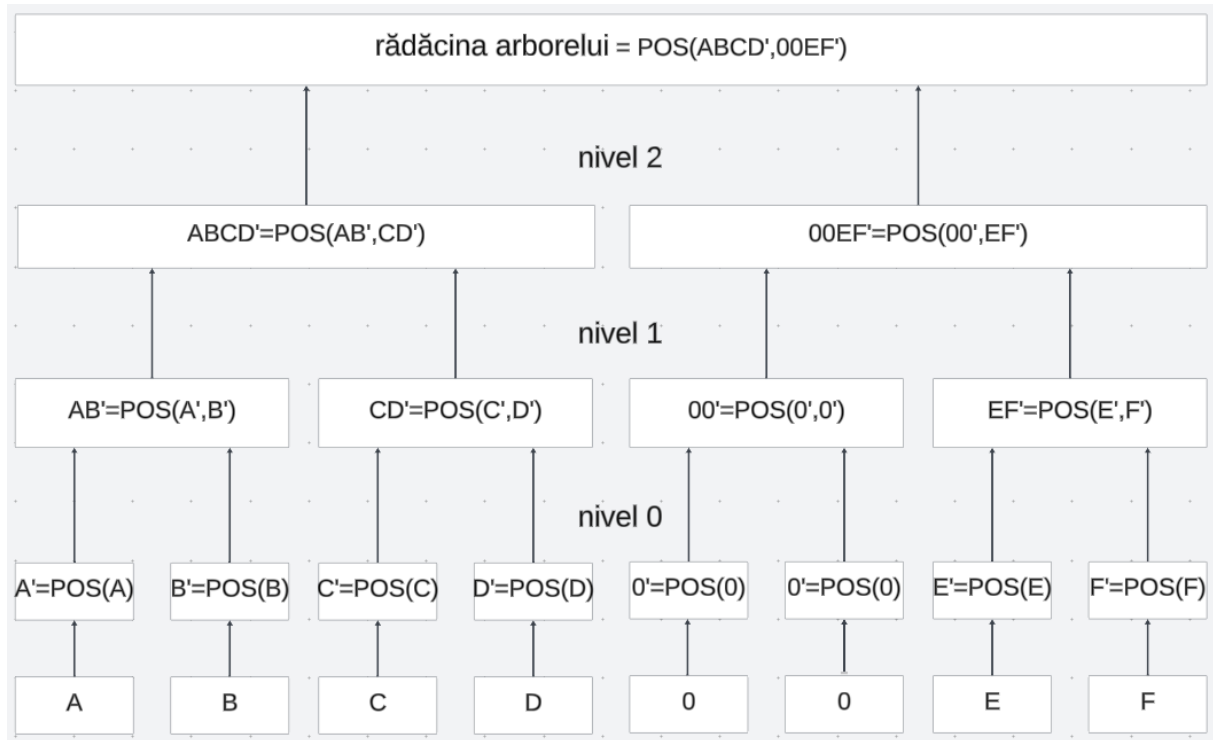


Figura 3.1: Arbore Merkle sparse de adâncime 3

### 3.4 Demonstrații de apartenență

O demonstrație de apartenență în contextul arborilor Merkle presupune parcurgerea corectă a drumului de la nodul frunză cu valoarea  $POS(x)$  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 1.* 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  $C$ .

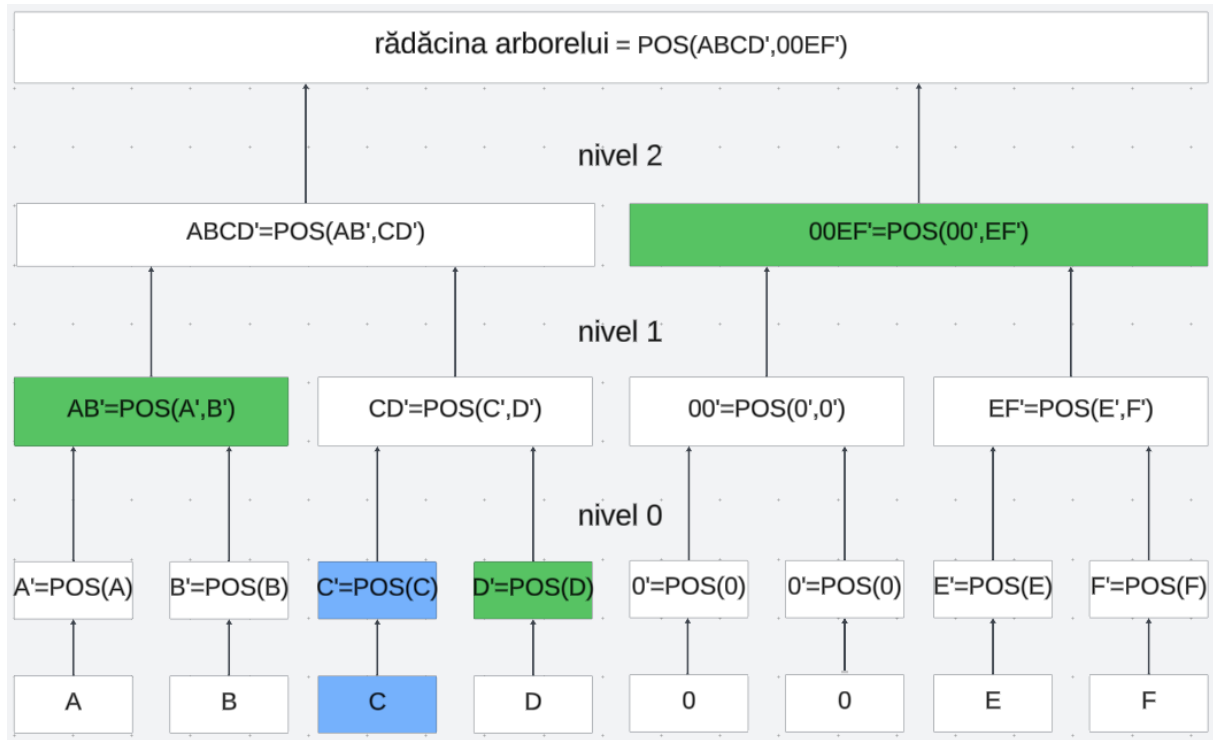


Figura 3.2: Demonstrație pentru nodul cu cheia secretă  $C$

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  $POS(C)$ .

### 3.5 Verificarea demonstrației

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

Dacă rădăcina 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, această metodă de demonstrare nu păstrează datele private deoarece Verifier-ul află poziția hash-ului cheii secrete în arbore.

### 3.6 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.

### 3.6.1 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 și ID-ul arborelui Merkle din care face parte, prin care se indică faptul că nodul a fost ”consumat”. În cazul în care dorim să actualizăm valoarea unui nod, trebuie să anulăm nodul vechi și să 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) [28].

Pentru a demonstra apartenența unui element  $x$  în modelul UTXO trebuie să demonstrăm că  $POS(x)$  face parte din arborele hash-urilor și faptul că nullifier-ul asociat cheii secrete nu se află în arborele nullifier-ilor. După ce o demonstrație este verificată, nullifier-ul este introdus în arborele nullifier-ilor și este astfel anulată cheia secretă pentru care s-a făcut demonstrația.

### 3.6.2 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 [26]. 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-apartenență pentru un anumit nullifier/valoare hash revine la o demonstrație de apartenență 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 nullifier-ilor pentru nullifier

### 3.6.3 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 *null-Tree* are o adâncime foarte mare și necesită executarea funcției hash pentru fiecare nivel. Funcțiile hash reprezintă o operație foarte costisitoare în contextul circuitelor aritmetice 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 așa că vom renunța la structura de arbore Merkle și la folosirea valorilor nullifier pe post de index pentru arborele nullifier-ilor și vom folosi o structură de date îmbunătățită.

## 3.7 Arbori hash indexați

O soluție pentru această problemă de performanță este prezentată în [27] unde se propune idea de *arbore Merkle indexat*, un arbore care ne permite să facem demonstrații zero-knowledge de non-apartenență eficiente.

Acest arbore este dens, are o adâncime substanțial mai mică și nodurile frunză reprezintă *hash commitment*-urile nodurilor unei liste circulare simplu înlanțuite de hash-uri nullifier ordonate în ordine crescătoare.

Lista va conține toate cheiile anulate din arborele hash principal în ordine crescătoare după nullifier-ul folosit iar structura unui nod din listă este compusă din: valoarea nullifier-ului  $nlfr \in F_p$ , valoarea nullifier-ului următor în ordine crescătoare  $nlfr_{next} \in F_p$  și index-ul din listă al următorului nullifier în ordine crescătoare  $i_{next} \in \{0, 2^{depth} - 1\}$ :  $listnode = \{nlfr, i_{next}, nlfr_{next}\}$ . În arborele indexat nullTree sunt introduse doar hash commitment-urile calculate folosind funcția POSEIDON,  $leafnode = POS(listnode)$ .

Regula de inserție pentru un nullifier nou este aceeași cu cea pentru o listă simplu înlanțuit normală, în care creăm un nod nou în listă și schimbăm pointerii pentru a păstra ordinea crescătoare a valorilor hash. Pe lângă inserția în listă va mai trebui să actualizăm și cele două hash commitment-uri din arborele indexat, asociate celor două noduri modificate din listă. Inserția se realizează numai după ce s-a demonstrat că nullifier-ul nu face parte deja din arborele indexat de nullifieri.

Această regulă de inserție și structura ordonată a listei generează o proprietate importantă pe care o vom folosi mai departe: dacă un nod de forma  $listnode = \{nlfr, i_{next}, nlfr_{next}\}$  face parte din listă atunci nu există niciun nullifier care a fost deja folosit în intervalul  $(nlfr, nlfr_{next})$ . Cu această proprietate putem să demonstrăm foarte simplu dacă un nullifier face sau nu parte dintr-un arbore indexat.

### 3.7.1 Construcția arborelui și algoritmul de inserție

Construcția arborelui indexat presupune un nivel adițional folosind un vector, în care vom păstra lista circulară simplu înlanțuită ale cărei noduri sunt compuse din obiecte cu 3 proprietăți: valoarea hash-ului, index-ului hash-ului următor și valoarea hash-ului următor.

Inițial lista conține valorile  $HASH\_MIN = 0$  și  $HASH\_MAX = p - 1$  (unde  $p - 1$  este valoarea "maximă" din câmpul  $F_p$ ) și este compusă din  $LINKED\_LIST = [\{0, 1, p-1\}, \{p-1, 0, 0\}]$ . Riscul de coliziune cu un alt nullifier este neglijabil iar aceste 2 margini, inferioare și superioare, ne reduc câteva cazuri speciale din algoritmul de inserție.

Algoritmul implică căutarea unui nod din listă cu valoarea hash-ului mai mică decât hash-ul inserat ( $nlfr < new\_hash$ ) și valoarea hash-ului următor mai mare decât hash-ul inserat ( $new\_hash < nlfr_{next}$ ).

---

**Algorithm 1** Algoritm pentru inserarea nullifier-ilor în lista arborelui indexat

---

```

1: function INSERTHASHVALUE(nullTree, hashValue)
2:   current_node  $\leftarrow$  nullTree.LinkedList[0]
3:   next_index  $\leftarrow$  current_node.next_index
4:   while next_index  $\neq$  0 do ▷ cât timp nu am parcurs toată lista
5:     current_hash  $\leftarrow$  current_node.hash_value
6:     next_hash  $\leftarrow$  current_node.next_hash_value
7:     if (current_hash < hashValue) && (hashValue < next_hash) then
8:       break
9:     else
10:      current_node  $\leftarrow$  nullTree.LinkedList[next_index]
11:      next_index  $\leftarrow$  current_node.next_index
12:    end if
13:  end while
14:  if next_index = 0 then
15:    throw new Error("Hash already inserted!")
16:  else
17:    new_node = new ListNode()
18:    new_node.hash_value  $\leftarrow$  hashValue
19:    new_node.next_index  $\leftarrow$  next_index
20:    new_node.next_hash_value  $\leftarrow$  current_node.next_hash_value
21:    nullTree.LinkedList.push(new_node)
22:    current_node.next_index  $\leftarrow$  nullTree.LinkedList.length - 1
23:    current_node.next_hash_value  $\leftarrow$  new_node.hash_value
24:  end if
25: end function

```

---

Algoritmul inserează valoarea *hashValue* în listă, altfel aruncă o eroare dacă hash-ul a fost deja inserat.

*Exemplu 2.* În exemplul următor este prezentat un arbore indexat de adâncime 3 (8 noduri frunză) în care inserăm în ordine valorile  $hash \in F_p : 10, 20, 15, 5$ .



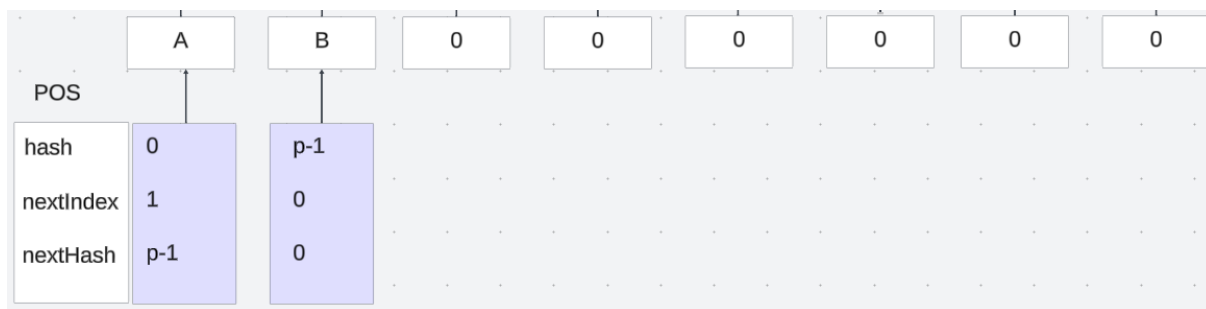


Figura 3.3: Lista inițială

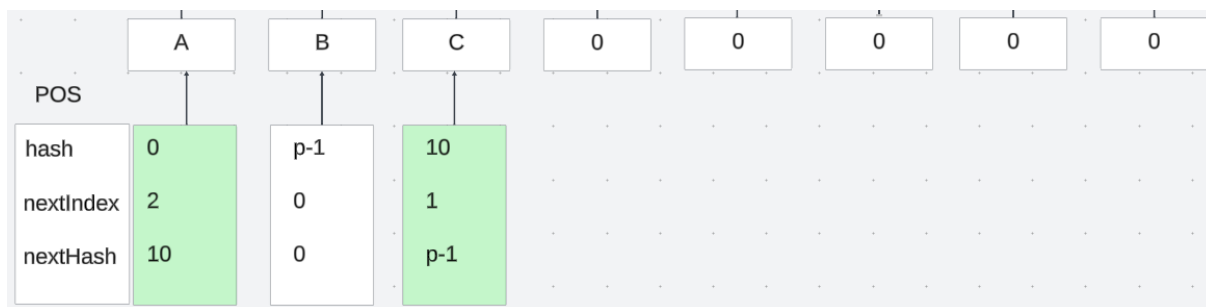


Figura 3.4: Inserăm hash-ul 10

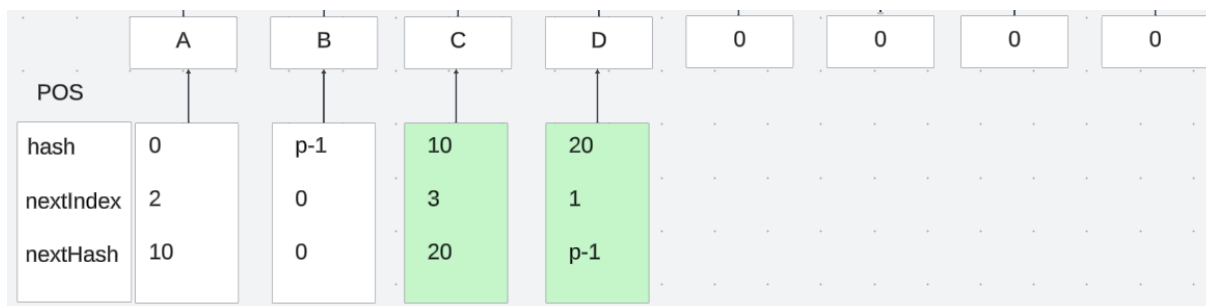


Figura 3.5: Inserăm hash-ul 20

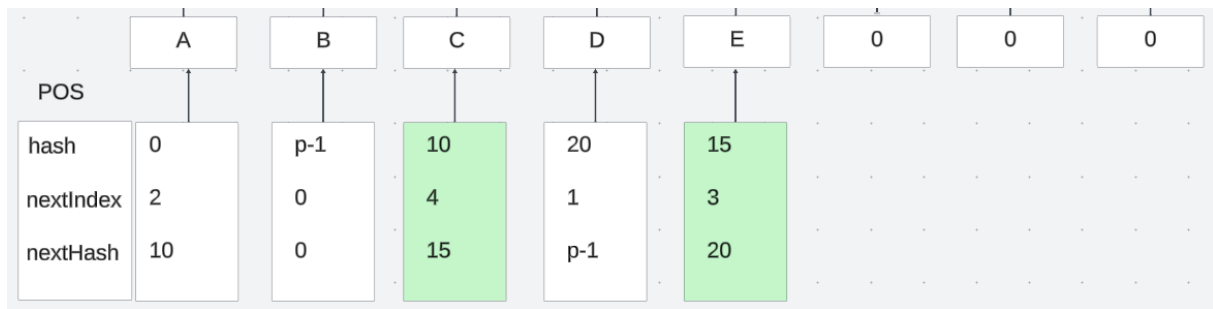


Figura 3.6: Inserăm hash-ul 15

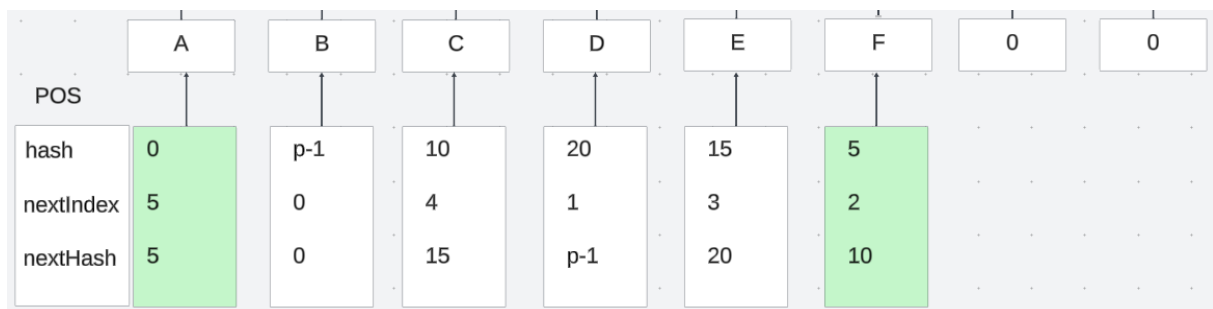


Figura 3.7: Inserăm hash-ul 5

După ce a fost inserat un nod nou în listă trebuie să recalculăm hash commitment-ul atât pentru nodul nou cât și pentru nodul actualizat, adică cele două noduri colorate cu verde din exemplul precedent și apoi să recalculăm rădăcina arborelui indexat.

## 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 Merkle.

### 4.1 CIRCOM 2.0

*CIRCOM* [21][22][23] este un limbaj low-level și un compilator pentru circuite scrise 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 ieșire 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 [26].

Fiecare circuit împreună cu semnalele sale trebuie să fie scris sub forma unei ecuații pătratice, 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 astfel 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 $\text{expr1}=\text{expr2}$
< --	sign< --expr	Atribuie valoarea expresiei din operandul drept
-- >	expr-- >sign	Atribuie valoarea expresiei din operandul stâng
<==	sign<==expr	Echivalent cu ( $\text{sign}< --\text{expr}$ și $\text{sign} === \text{expr}$ )
==>	expr==>sign	Echivalent cu ( $\text{expr}-- >\text{sign}$ și $\text{expr} === \text{sign}$ )

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* [24] 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 [17], PLONK [13] și FFLONK [12].

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 (aproprate 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. 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]

```

1 template Num2Bits(n){
2     signal input in;
3     signal output out[n];
4
5     var value = 0;
6     var pow = 1;
7     for(var i=0;i<n;i++){
8         out[i] <-- (in>>i)&1;
9         out[i]*(1-out[i]) === 0;
10        value += out[i]*pow;
11        pow += pow;
12    }
13    value === in;}

```

Circuitul **Num2Bits( $n$ )**[7] face parte din librăria standard și este folosit pentru a genera secvența binară de lungime  $n$  asociată numărului trimis prin semnalul de intrare **in**. Avem  $n$  semnale de ieșire corespunzătoare 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]

```

1 template Bits2Num(n){
2     signal input in[n];
3     signal output out;
4
5     var value = 0;
6     var pow = 1;
7     for(var i=0;i<n;i++){
8         in[i]*(1-in[i]) === 0;
9         value += pow*in[i];
10        pow +=pow;
11    }
12    out <== value;}

```

**Bits2Num(n)**[7] face parte din librăria standard și 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]

```

1 template LessThan(){
2     signal input in[2];
3     signal output out;
4
5     component n2b = Num2Bits(253);
6
7     n2b.in <== in[0] + (1 << 252) - in[1];
8     out <== 1 - n2b.out[252];
9 }

```

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]

```

1 template isZero(){
2     signal input in;
3     signal output out;
4
5     signal inv <-- in == 0 ? 0 : 1/in;
6     out <== 1 - in*inv;
7     out*in === 0;
8 }

```

**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]

```

1 include "../isZero.circom";
2
3 template isEqual(){
4     signal input in[2];
5     signal output out;
6     component checkZero = isZero();
7     checkZero.in <== in[0] - in[1];
8     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.circom"

```

1 include "./Num2Bits.circom";
2 include "./Bits2Num.circom";
3 include "./LessThan.circom";
4 include "./isEqual.circom";
5
6 template LessThan_256BIT_MSBR(){
7     signal input in[2];
8     signal inter1[4];
9     signal inter2[4];
10    signal output out;
11    component n2b = Num2Bits(256);
12    component b2n[4];
13    n2b.in <== in[0];
14    for(var i=0;i<4;i++){
15        b2n[i] = Bits2Num(64);
16        for(var j=0;j<64;j++){
17            b2n[i].in[j] <== n2b.out[j+(64*i)];
18        }
19        inter1[i] <== b2n[i].out;
20    }
21    component n2b_2 = Num2Bits(256);
22    component b2n_2[4];
23    n2b_2.in <== in[1];
24    for(var i=0;i<4;i++){
25        b2n_2[i] = Bits2Num(64);
26        for(var j=0;j<64;j++){
27            b2n_2[i].in[j] <== n2b_2.out[j+(64*i)];
28        }
29        inter2[i] <== b2n_2[i].out;
30    }

```

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 ținute în semnalele **inter1** și **inter2**.

”LessThan\_256BIT\_MSBR.circom”

```

1  signal interBINcomp1[4];
2  signal interBINcomp2[4];
3  component interLT[4];
4  component isEq[4];
5
6  for(var i=0;i<4;i++){
7      interLT[i] = LessThan();
8      isEq[i] = isEqual();
9      interLT[i].in[0]<==inter2[i];
10     interLT[i].in[1]<==inter1[i];
11     isEq[i].in[0]<==inter2[i];
12     isEq[i].in[1]<==inter1[i];
13     interBINcomp1[i]<== interLT[i].out;
14     interBINcomp2[i]<== (1-interLT[i].out)-isEq[i].out;
15 }
16
17 signal num[2];
18 component b2n_f[2];
19 b2n_f[0]=Bits2Num(4);
20 b2n_f[0].in[0]<== interBINcomp1[0];
21 b2n_f[0].in[1]<== interBINcomp1[1];
22 b2n_f[0].in[2]<== interBINcomp1[2];
23 b2n_f[0].in[3]<== interBINcomp1[3];
24 b2n_f[1]=Bits2Num(4);
25 b2n_f[1].in[0]<== interBINcomp2[0];
26 b2n_f[1].in[1]<== interBINcomp2[1];
27 b2n_f[1].in[2]<== interBINcomp2[2];
28 b2n_f[1].in[3]<== interBINcomp2[3];

```

Fiecare segment din cele două secvențe **inter1** și **inter2** este comparat iar dacă un segment este mai mare atunci îl înlocuim cu 1, altfel îl înlocuim cu 0. Putem astfel să reducem numărul de apeluri la circuitul standard **LessThan**.



”LessThan\_256BIT\_MSBR.circom”

```

1  num[0] <== b2n_f[0].out;
2  num[1] <== b2n_f[1].out;
3
4  component LTF = LessThan();
5  LTF.in[0] <== num[0];
6  LTF.in[1] <== num[1];
7  out <== LTF.out;
8 }

```

La sfârșit semnalele **interBINcomp1** și **interBINcomp2** ajung să conțină 2 secvențe de 4 biți, care sunt convertite înapoi 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 această secțiune sunt prezentate circuitele secundare create pentru a realiza demonstrații în arbori hash normali și indexați.

Listing 4.7: ”Selector.circom”

```

1 template Selector(){
2     signal input switcher;
3     signal input in[2];
4     signal int[4];
5     signal output out[2];
6     0 == (switcher)*(1-switcher);
7     int[0] <== in[0]*(1-switcher);
8     int[1] <== in[1]*switcher;
9     out[0] <== int[0] + int[1];
10
11     int[2] <== in[1]*(1-switcher);
12     int[3] <== in[0]*switcher;
13     out[1] <== int[2] + int[3];
14
15 }

```

**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 aceeași 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"

```
1 include "../node_modules/circomlib/circuits/poseidon.circom";
2 include "../Selector.circom";
3
4 template HashTreeLevel(){
5     signal input in[2];
6     signal input position;
7     signal output out;
8
9     component poseidon = Poseidon(2);
10    component selector = Selector();
11
12    selector.in[0] <== in[0];
13    selector.in[1] <== in[1];
14    selector.switcher <== position;
15
16    selector.out[0] ==> poseidon.inputs[0];
17    selector.out[1] ==> poseidon.inputs[1];
18
19    poseidon.out ==> out;
20
21 }
```

**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 circuit este folosit exclusiv în demonstrațiile de apartenență în arbori hash simpli, în timp ce demonstrațiile de non-apartență în arbori indexați necesită verificări și constrângeri adiționale. Acestea sunt tratate într-un singur circuit, circuitul principal descris mai jos, pentru a ne asigura că valorile primite sunt conforme. De exemplu dacă nu folosim același circuit atunci o entitate malițioasă care cunoaște două elemente secrete  $x$  și  $y$  din mulțimea  $S$  poate să demonstreze apartenența lui  $x$  și să anuleze elementul  $y$ . Într-un singur circuit acest lucru nu este posibil deoarece valoarea folosită la calculul nullifier-ului este aceeași cu cea folosită la demonstrația de apartenență.

## 4.6 Circuitul principal

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

Listing 4.9: "MainProof.circom - Demonstrația de apartenență la arborele hash"

```
1 include "../node_modules/circomlib/circuits/poseidon.circom";
2 include "../HashTreeLevel.circom";
3 include "../LessThan_256BIT_MSBR.circom";
4
5 template MainProof(depth){
6     signal input sk;
7     signal input siblingsPk[depth];
8     signal input path[depth];
9     signal input nullifierHash;
10    signal input lowLeafHashValue;
11    signal input lowHash;
12    signal input nextIndex;
13    signal input highHash;
14    signal input nullifierTreeSiblingsPk[depth];
15    signal input nullifierTreePath[depth];
16    signal input root;
17    signal input nullifierRoot;
18    signal input nodeTreeID;
19    signal input nullifierTreeID;
20
21    signal intermed[depth+1];
22    component levelChecker[depth];
23    component poseidon = Poseidon(1);
24    poseidon.inputs[0] <== sk;
25    poseidon.out ==> intermed[0];
26    for(var i=0;i<depth;i++){
27        levelChecker[i] = HashTreeLevel();
28        levelChecker[i].in[0] <== intermed[i];
29        levelChecker[i].in[1] <== siblingsPk[i];
30        levelChecker[i].position <== path[i];
31        levelChecker[i].out ==> intermed[i+1];
32    }
33    intermed[depth] === root;
34
```

**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.

”MainProof.circom - Calculul corect al nullifier-ului”

```

1   component poseidonNullifier = Poseidon(3);
2   poseidonNullifier.inputs[0] <== sk;
3   poseidonNullifier.inputs[1] <== nodeTreeID;
4   poseidonNullifier.inputs[2] <== nullifierTreeID;
5   poseidonNullifier.out === nullifierHash;

```

**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 element din mulțimea de elemente atunci când funcția hash folosită este criptografică.

”MainProof.circom - Demonstrația că hash-ul nullifier-ului se află în intervalul deschis lowHash-highHash”

```

1   //is higher than the nullifier
2   component LT256[2];
3   LT256[0] = LessThan_256BIT_MSBR();
4   LT256[0].in[0] <== lowHash;
5   LT256[0].in[1] <== nullifierHash;
6   LT256[0].out === 1;
7
8   LT256[1] = LessThan_256BIT_MSBR();
9   LT256[1].in[0] <== nullifierHash;
10  LT256[1].in[1] <== highHash;
11  LT256[1].out === 1;

```

**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**.

”MainProof.circom - Calculul corect al nodului frunză din arborele nullifier-ilor”

```

1
2   component poseidonLowLeafHashValue = Poseidon(3);
3   poseidonLowLeafHashValue.inputs[0] <== lowHash;
4   poseidonLowLeafHashValue.inputs[1] <== nextIndex;
5   poseidonLowLeafHashValue.inputs[2] <== highHash;
6   poseidonLowLeafHashValue.out == lowLeafHashValue;

```

**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 lista ordonată 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 listă.

”MainProof.circom - Demonstrația de non-apartenență la arborele nullifier-ilor”

```

1   signal intermedNULL[depth+1];
2
3   component levelCheckerNullifier[depth];
4   lowLeafHashValue ==> intermedNULL[0];
5
6   for(var i=0;i<depth;i++){
7     levelCheckerNullifier[i] = HashTreeLevel();
8     levelCheckerNullifier[i].in[0] <== intermedNULL[i];
9     levelCheckerNullifier[i].in[1] <== nullifierTreeSiblingsPk[
10    i];
11    levelCheckerNullifier[i].position <== nullifierTreePath[i];
12    levelCheckerNullifier[i].out ==> intermedNULL[i+1];
13  }
14  intermedNULL[depth] == nullifierRoot;
15 }
16 component main {public [root,nullifierRoot,nodeTreeID,
    nullifierTreeID,
    nullifierHash]} = MainProof(20);

```

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

## 4.7 Compilarea și rularea circuitelor

Pentru a putea compila circuitele va trebui să instalăm în ordine limbajul Rust [20], compilatorul CIRCOM 2.0 [18] și Node.js [19].

Codul sursă care include librăriile Javascript scrise pentru implementarea arborilor hash Merkle, a arborilor indexați cât și a metodelor de generare și verificare a demonstrațiilor împreună cu circuitele descrise mai sus se află pe Github [10].

Odată descărcat, trebuie să instalăm toate modulele Node de care codul este dependent rulând comanda *npm install* în folderul principal al repository-ului.

```
1 dynamic-hash-tree-main>npm install
2
3 added 108 packages, and audited 109 packages in 3s
4
5 34 packages are looking for funding
6   run `npm fund` for details
7
8 found 0 vulnerabilities
```

Circuitul principal este compilat și salvat folosind comanda *circom* în folderul "*circuits*":

Listing 4.10: "Compilarea circuitului principal"

```
1 \circuits> circom MainProof.circom --wasm --r1cs --sym -o ./
   build
2 template instances: 220
3 non-linear constraints: 15143
4 linear constraints: 0
5 public inputs: 5
6 private inputs: 85
7 public outputs: 0
8 wires: 14136
9 labels: 38529
10 Written successfully: ./build/MainProof.r1cs
11 Written successfully: ./build/MainProof.sym
12 Written successfully: ./build/MainProof_js/MainProof.wasm
13 Everything went okay
```

Comanda indică modul de calcul al circuitului (*-wasm* folosind fișiere WebAssembly), modul de reprezentare al contrângerilor (*-r1cs* Rank 1 constraint system), generarea de simboluri folosite la debug (*-sym*) și folderul de output: *./build*. Pentru exemplul de mai sus a fost folosit circuitul "*MainProof.circom*" în care adâncimea arborilor folosiți a fost

setată direct în circuit, la 20 de nivele, folosind parametrul *depth*.

După ce am terminat compilarea circuitului putem să începem trusted setup-ul care constă în generarea unui fișier \*.ptau în care fiecare entitate care participă la procesul de demonstrare poate contribui în diverse faze. Modulul snarkJS [24] prezintă în mod detaliat pașii necesari pentru generarea fișierului însă protocolul PLONK folosit în acest exemplu nu necesită un setup specific și putem să descărcăm un fișier \*.ptau deja generat cu condiția ca acesta să fie suficient de mare pentru a cuprinde toate constrângerile circuitului. Vom descărca fișierul *powersOfTau28\_hez\_final\_17.ptau* pe care îl vom pune în folderul *circuits/ptau* deoarece circuitul compilat are aproximativ 120 000 de constrângeri.

Vom genera apoi cele 2 chei (Proving Key și Verifying Key) folosite de către snarkJS în generarea și verificarea SNARK-urilor pentru un anumit circuit:

Listing 4.11: "Cheia de demonstrare folosită de SnarkJS"

```
1 circuits> snarkjs plonk setup build/MainProof.r1cs
2 ptau/powersOfTau28_hez_final_17.ptau keys/MainProof_PK.zkey
3 [INFO] snarkJS: Reading r1cs
4 [INFO] snarkJS: Plonk constraints: 120467
5 [INFO] snarkJS: Setup Finished
```

Listing 4.12: "Cheia de verificare folosită de SnarkJS"

```
1 \circuits> snarkjs zkey export verificationkey
2 keys/MainProof_PK.zkey keys/MainProof_VK.json
3 [INFO] snarkJS: EXPORT VERIFICATION KEY STARTED
4 [INFO] snarkJS: > Detected protocol: plonk
5 [INFO] snarkJS: EXPORT VERIFICATION KEY FINISHED
```

## 4.8 Compararea celor doi arbori

În această secțiune sunt prezentate și comparate rezultatele obținute la compilarea și rularea circuitelor în care folosim arbori Merkle și arbori hash indexați pentru arborele nullifier-ilor.

În primul caz, circuitul *HashTreeProof.circom*, vom folosi un arbore Merkle sparse pentru arborele nullifier-ilor, cu adâncimea de 254 nivele pentru a acomoda toate valorile din codomeniul funcției hash POSEIDON. Vom folosi aceste valori pe post de index și vom codifica 0 pentru nefolosit și 1 pentru folosit. Demonstrația de non-apartenență pentru o anumită valoare  $x$  se rezumă deci la a demonstra apartenența valorii 0 la index-ul  $x$ .

În al doilea caz vom folosi un arbore indexat pentru arborele nullifier-ilor, în care demonstrația de non-apartenență revine la a demonstra existența unui nod specific care indică faptul că valoarea nullifier-ului nu a fost încă introdusă. Adâncimea arborelui indexat este setată la 20 de nivele, acesta fiind o structură de date densă spre deosebire de structura sparse din primul caz, și este același circuit *MainProof.circom* folosit în exemplul din secțiunea următoare în care folosim pe post de mulțime suport mulțimea tuturor cuvintelor din limba engleză.

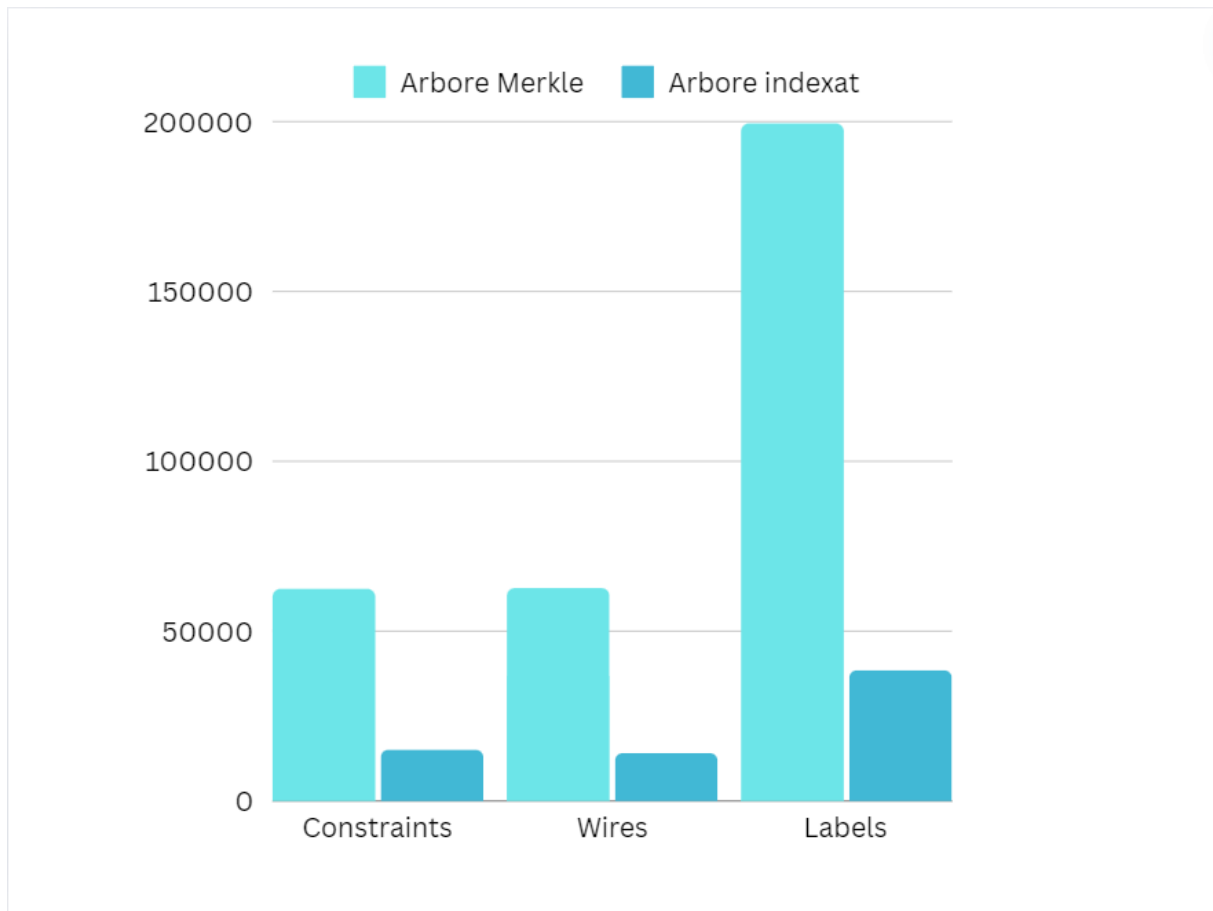


Figura 4.1: Comparația metricelor de compilare pentru cei doi arbori



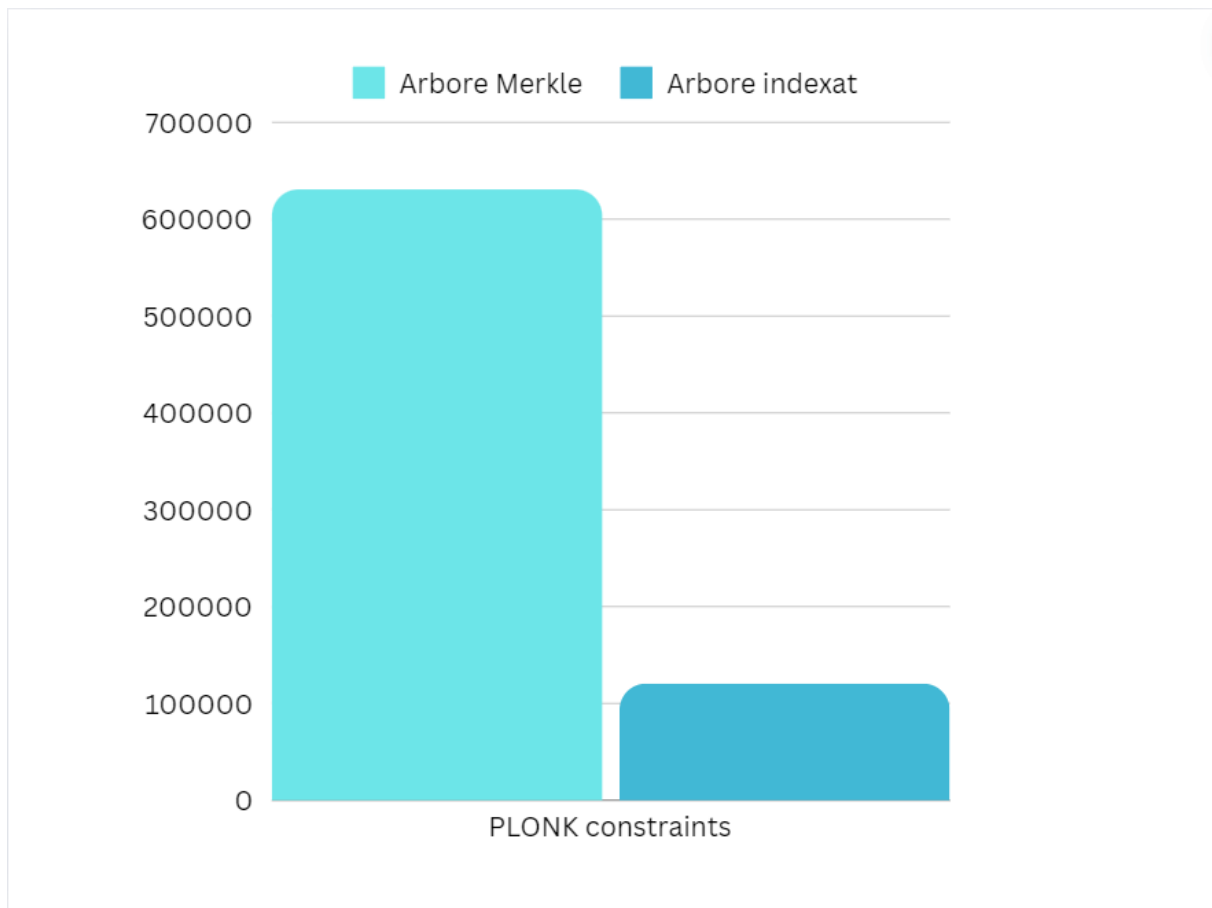


Figura 4.2: Comparația numărului de constrângeri pentru protocolul PLONK

Figura 4.1 prezintă numărul de noduri (eng. "Labels") și numărul de muchii (eng. "Wires") necesare pentru a genera fiecare circuit aritmetic în parte. De asemenea sunt prezentate și numărul de constrângeri (eng. "Constraints") necesare pentru a genera sistemele de constrângeri r1cs asociate circuitelor. Aceste metrice au un efect direct proporțional asupra timpului de demonstrare și verificare și putem observa că folosirea arborilor indexați pentru arborele nullifier-ilor este mai performantă decât un arbore Merkle normal.

Acest lucru reiese și din Figura 4.2 în care sunt prezentate numărul de constrângeri PLONK generate de *snarkJS* atunci când sunt create cheile de demonstrare și verificare.

## 4.9 Exemplu de demonstrație

În fișierul *dynamic-hash-tree-main/tests/demo.js* vom găsi un exemplu de demonstrație, în care folosim un arbore hash ce conține hash-urile POSEIDON a tuturor cuvintelor din limba engleză (aproximativ 370 000) preluate dintr-un repository Github public [9]. Arborele hash și arborele nullifier-ilor sunt inițializați și populați pentru ca ulterior aceștia să fie folosiți în demonstrații de către cele două entități:

**Entitatea "Prover"** este responsabilă cu calculul tuturor valorilor pentru semnalele de intrare ale circuitului având ca scop demonstrarea apartenenței unui cuvânt din mulțime folosind arborii din memorie și cheia de demonstrare generată mai sus.

```
1 console.log("Starting ZK Proof Generation");
2 var start = new Date().getTime();
3 const { proof, publicSignals } = await snarkjs.plonk.
fullProve(
4 inputs,
5 "../circuits/build/MainProof_js/MainProof.wasm",
6 "../circuits/keys/MainProof_PK.zkey");
7 var time_now = new Date().getTime();
8 console.log(`Proof and public signals generated in
9 ${time_now-start} mseconds`);
10
11 OUTPUT:
12 Starting ZK Proof Generation
13 Proof and public signals generated in 76587 mseconds
```

**Entitatea "Verifier"** este responsabilă cu verificarea SNARK-ului generat folosind valorile  $\{ proof, publicSignals \}$  trimise de către Prover și cheia de verificare generată mai sus. După fiecare demonstrație corectă, valoarea nullifier-ului folosit este introdusă în arborele indexat al nullifier-ilor pentru a evita refolosirea demonstrațiilor.

```
1 const vKey = JSON.parse(fs.readFileSync(
2 "../circuits/keys/MainProof_VK.json"));
3 const res = await snarkjs.plonk.verify(vKey,publicSignals,
proof);
4 console.log("Proof result: ");
5 console.log(res);
6 var end = new Date().getTime();
7 console.log(`Verify time : ${end-time_now} mseconds`);
8 OUTPUT:
9 Proof result:true
10 Verify time : 31 mseconds
```

# Capitolul 5

## Concluzii

În lucrarea de față am prezentat noțiunile principale care stau la baza demonstrațiilor zero-knowledge de apartenență la mulțimi și problemele care apar în diverse implementări. Am prezentat de ce pentru fiecare tip de demonstrație (apartenență/non-apartenență) avem nevoie de structuri de date diferite și am arătat de ce arborii indexați sunt mai performanți decât arborii hash Merkle în cazul demonstrațiilor de non-apartenență.

Folosirea arborilor indexați ne permite să generăm și să verificăm demonstrații de non-apartenență într-un timp de lucru redus față de arborii Merkle. Acest lucru este posibil deoarece arborii indexați reduc apelurile costisitoare la funcțiile hash și le înlocuiesc cu operații de comparare care sunt mult mai simple.

Deși acest tip de demonstrații este folosit cel mai mult în mediul blockchain pentru a anonimiza tranzacțiile utilizator, există și alte aplicații interesante precum:

- Self-sovereign identity: o modalitate de a utiliza cărți de identitate astfel încât să nu fie dezvăluită identitatea posesorului. De exemplu putem astfel să votăm electronic și să accesăm servicii online care necesită anumite condiții precum vârsta mai mare de 18 ani sau domiciliul într-un anumit județ fără să ne dezvăluim identitatea.
- În domeniul bancar: o bancă poate să genereze o mulțime cu toți clienții care sunt solvenți, pentru ca apoi fiecare în parte să poată demonstra ca este solvent folosind o demonstrație de apartenență. Și în domeniul administrativ în același mod, o companie poate să demonstreze că a plătit la timp toate taxele fără să trebuiască să dezvăluie tot istoricul tranzacțiilor.
- Forum-uri/canale de comunicație anonime: Putem să formăm mulțimi de utilizatori autorizați iar pentru a trimite un mesaj acesta trebuie să fie urmat de o demonstrație de apartenență. Cum demonstrația este zero-knowledge, mesajul nu poate să fie atribuit direct niciunui utilizator din mulțime.

Acest tip de demonstrații sunt deja folosite cu succes în practică iar circuitele și librăriile implementate în CIRCOM și JavaScript din această lucrare reprezintă un punct de start în studierea și verificarea tehnicilor de demonstrație zero-knowledge de apartenență/non-apartenență.

# Bibliografie

- [1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy și Tyge Tiessen, „MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity”, în *Advances in Cryptology – ASIACRYPT 2016*, ed. de Jung Hee Cheon și Tsuyoshi Takagi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 191–219, ISBN: 978-3-662-53887-6.
- [2] Tomer Ashur, Al Kindi, Willi Meier, Alan Szeponiec 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”, în *Public-Key Cryptography – PKC 2024*, ed. de Qiang Tang și Vanessa Teague, Cham: Springer Nature Switzerland, 2024, pp. 99–131, ISBN: 978-3-031-57722-2.
- [5] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan și Dimitris Kolonelos, „Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular”, în *Financial Cryptography and Data Security*, ed. de Nikita Borisov și Claudia Diaz, Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 393–414, ISBN: 978-3-662-64322-8.
- [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](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] *Codul sursă al dataset-ului "english words"*, <https://github.com/dwyl/english-words>, Ultima accesare: 2024-06-11.
- [10] *Codul sursă asociat lucrării*, <https://github.com/andreiparjol/fac/dynamic-hash-tree>, Ultima accesare: 2024-06-11.
- [11] 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>.
- [12] 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>.
- [13] 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>.
- [14] Shafi Goldwasser, Silvio Micali și Chales Rackoff, „The knowledge complexity of interactive proof-systems”, în Oct. 2019, ISBN: 9781450372664, DOI: [10.1145/3335741.3335750](https://doi.org/10.1145/3335741.3335750).
- [15] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy și Markus Schofnegger, „Poseidon: A New Hash Function for Zero-Knowledge Proof Systems”, în *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 519–535, ISBN: 978-1-939133-24-3, URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/grassi>.
- [16] 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/>, Ultima accesare: 2024-04-24.
- [17] Jens Groth, „On the Size of Pairing-Based Non-interactive Arguments”, în *Advances in Cryptology – EUROCRYPT 2016*, ed. de Marc Fischlin și Jean-Sébastien Coron, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326, ISBN: 978-3-662-49896-5.
- [18] *Install Circom 2.0*, <https://docs.circom.io/getting-started/installation/>, Ultima accesare: 2024-05-30.
- [19] *Install Node.js*, <https://nodejs.org/en>, Ultima accesare: 2024-05-30.
- [20] *Install Rust*, <https://www.rust-lang.org/tools/install>, Ultima accesare: 2024-05-30.

- [21] *Official Circom 2.0 Documentation*, <https://docs.circom.io/>, Ultima accesare: 2024-05-19.
- [22] *Official Circom 2.0 Github repository*, <https://github.com/iden3/circom>, Ultima accesare: 2024-05-19.
- [23] *Official Iden3 Documentation*, <https://docs.iden3.io/circom-snarkjs/>, Ultima accesare: 2024-05-19.
- [24] *Official snarkJS Github repository*, <https://github.com/iden3/snarkjs/blob/master/README.md>, Ultima accesare: 2024-05-19.
- [25] 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>.
- [26] 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>, Ultima accesare: 2024-05-02.
- [27] 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>.
- [28] Dionysis Zindros, *Lecture 10: Accounts Model and Merkle Trees*, Stanford, Spring online lecture, 2022, URL: [https://web.stanford.edu/class/ee374/lec\\_notes/lec10.pdf](https://web.stanford.edu/class/ee374/lec_notes/lec10.pdf), Ultima accesare: 2024-05-19.