AEQUITAVERSE

Guide d'installation et d'utilisation

Node.js · snarkjs · Preuve ZKP Groth16

R1CS haute entropie · Génération de preuves · Fichiers de sortie

Pour qui est ce guide ?

Utilisateurs finaux sans formation préalable en cryptographie. Windows 10/11 · macOS · Linux Debian/Ubuntu

Windows 10/11	macOS	Linux Debian/Ubuntu
Chapitres 1 à 5	Chapitres 1 à 5	Chapitres 1 à 5
PowerShell recommandé	Terminal.app ou iTerm2	Bash / Zsh
Droits administrateur requis	Mot de passe sudo requis	Droits sudo requis

1. Installer Node.js

Node.js est le moteur d'exécution JavaScript requis pour faire fonctionner snarkjs. Vous devez installer la version LTS (Long Term Support) — actuellement la version 22.x.

> Vérifiez d'abord si Node.js est déjà installé. Ouvrez un terminal et tapez : node --version Si vous voyez v20.x.x ou v22.x.x, passez directement au chapitre 2.

1.1 Windows 10 / 11

1. Ouvrez votre navigateur web et allez sur : https://nodejs.org
2. Cliquez sur le bouton vert marqué LTS (Long Term Support).
3. Téléchargez le fichier .msi (ex : node-v22.x.x-x64.msi).
4. Double-cliquez sur le fichier téléchargé pour lancer l'installateur.
5. Cliquez sur Suivant à chaque étape. Laissez toutes les options par défaut.
6. À l'écran « Tools for Native Modules », cochez la case si elle est proposée.
7. Cliquez sur Installer. Acceptez la demande d'élévation de droits (Oui).
8. Cliquez sur Terminer une fois l'installation complète.

Vérification — ouvrez PowerShell (touche Windows + X → Windows PowerShell) et tapez :

PowerShell

```
node --version
npm --version
```

Résultat attendu :

```
v22.x.x
10.x.x
```

> Si vous voyez "node n'est pas reconnu comme commande", redémarrez votre ordinateur et réessayez.

1.2 macOS

Il existe deux méthodes. La méthode A (recommandée) utilise Homebrew, un gestionnaire de paquets pour macOS.

Méthode A — Via Homebrew (recommandée)

9. Ouvrez Terminal (Applications → Utilitaires → Terminal).
10. Installez Homebrew en collant cette commande et en appuyant sur Entrée :

Terminal macOS

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

11. Suivez les instructions à l'écran. Votre mot de passe de session vous sera demandé.
12. Une fois Homebrew installé, tapez :

Terminal macOS

```
brew install node
```

Méthode B — Via le site officiel

13. Allez sur https://nodejs.org et téléchargez le fichier .pkg (macOS).
14. Double-cliquez le fichier .pkg et suivez les instructions.

Vérification :

Terminal macOS

```
node --version
npm --version
```

1.3 Linux — Debian / Ubuntu

La commande suivante installe Node.js 22 via le dépôt officiel NodeSource. Elle fonctionne sur Debian 11/12 et Ubuntu 20.04/22.04/24.04.

Terminal Linux

```
# Ajouter le dépôt NodeSource
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -

# Installer Node.js
sudo apt-get install -y nodejs

# Vérifier
node --version
npm --version
```

> Si curl n'est pas installé : sudo apt-get install -y curl

1.4 Comparatif des versions requises

Composant	Exigence
Node.js	Version 20 LTS ou 22 LTS (recommandée)
npm	Version 10.x ou supérieure (inclus avec Node.js)
Mémoire RAM	Minimum 4 Go — 8 Go recommandés pour les circuits volumineux
Espace disque	Minimum 2 Go libres pour les fichiers de cérémonie

2. Installer snarkjs

snarkjs est la bibliothèque JavaScript qui permet de générer et vérifier des preuves à divulgation nulle (Zero-Knowledge Proofs) au format Groth16. Elle est développée par l'équipe iden3 et est utilisée par des millions de projets cryptographiques.

> snarkjs s'installe en une seule commande sur tous les systèmes d'exploitation. La commande est identique sur Windows, macOS et Linux.

2.1 Installation globale (recommandée)

L'installation globale rend la commande snarkjs disponible depuis n'importe quel dossier de votre système.

PowerShell / Terminal — tous systèmes

```
npm install -g snarkjs
```

> Sur Linux et macOS, si vous obtenez une erreur de permission, utilisez : sudo npm install -g snarkjs

Vérification de l'installation :

```
snarkjs --version
```

Résultat attendu : 0.7.x ou supérieur

2.2 Installer circom (compilateur de circuits)

circom est le compilateur qui transforme votre fichier de circuit (.circom) en fichier R1CS. Il est nécessaire avant d'utiliser snarkjs.

Windows 10/11	macOS	Linux Debian/Ubuntu
# Télécharger le binaire circom # depuis https://github.com/iden3/circom/releases # Fichier : circom-windows-amd64.exe # Renommer en circom.exe # Copier dans C:\Windows\System32\	# Via Homebrew brew install circom # OU via Rust (si Homebrew indisponible) curl --proto "=https" --tlsv1.2 https://sh.rustup.rs | sh cargo install --git https://github.com/iden3/circom	# Via Rust (méthode officielle) curl --proto "=https" --tlsv1.2 https://sh.rustup.rs | sh source ~/.cargo/env cargo install --git https://github.com/iden3/circom

Vérification sur tous les systèmes :

```
circom --version
```

2.3 Installer les dépendances du projet

Dans votre dossier de projet, installez les bibliothèques complémentaires :

PowerShell / Terminal — tous systèmes

```
# Créer un dossier de projet
mkdir mon-projet-zkp
cd mon-projet-zkp

# Initialiser le projet Node.js
npm init -y

# Installer les bibliothèques nécessaires
npm install snarkjs circomlibjs
npm install -D @types/node
```

3. Le circuit R1CS avec haute entropie

Un fichier R1CS (Rank-1 Constraint System) est la représentation mathématique de votre circuit logique. Il encode les règles que votre preuve doit satisfaire. La « haute entropie » signifie que les valeurs secrètes utilisées dans le circuit sont suffisamment aléatoires pour être cryptographiquement sûres.

> Dans le contexte Aequitaverse, notre circuit prouve qu'un DID est actif dans le registre sans révéler lequel. Les entrées privées (le DID, le nonce, le sel) doivent avoir 256 bits d'entropie chacune.

3.1 Le circuit circom complet — did_proof.circom

Créez un fichier nommé did_proof.circom dans votre dossier de projet et copiez le contenu suivant :

did_proof.circom — copier intégralement

```
pragma circom 2.1.6;

include "node_modules/circomlibjs/src/poseidon.circom";
include "node_modules/circomlibjs/src/comparators.circom";
include "node_modules/circomlibjs/src/mux1.circom";

// Sous-circuit : vérification d'un chemin Merkle
// Prouve qu'une valeur appartient à un arbre sans révéler laquelle
template MerkleProof(depth) {
    signal input  leaf;
    signal input  pathElements[depth];
    signal input  pathIndices[depth];
    signal output root;

    component hashers[depth];
    signal currentHash[depth + 1];
    currentHash[0] <== leaf;

    for (var i = 0; i < depth; i++) {
        hashers[i] = Poseidon(2);
        // Entrée gauche ou droite selon pathIndices[i]
        var left  = pathIndices[i] == 0 ? currentHash[i] : pathElements[i];
        var right = pathIndices[i] == 0 ? pathElements[i] : currentHash[i];
        hashers[i].inputs[0] <== left;
        hashers[i].inputs[1] <== right;
        currentHash[i + 1] <== hashers[i].out;
    }
    root <== currentHash[depth];
}

// Circuit principal DID Proof
// merkleDepth = 20 supporte jusqu'à 1 048 576 DID
```

```
template DIDProof(merkleDepth) {

    // === ENTRÉES PUBLIQUES (visibles par le vérificateur) ===
    signal input merkleRoot;      // Racine de l'arbre des DID actifs
    signal input commitmentDID;   // hash(did_id + nonce_prive)
    signal input nullifier;       // hash(did_id + avatar_salt)
    signal input timestampLimite; // Expiration de la preuve

    // === ENTRÉES PRIVÉES (jamais révélées) ===
    // HAUTE ENTROPIE REQUISE : chacune doit être 256 bits aléatoires
    signal input didId;           // L'identifiant DID secret
    signal input noncePrive;      // Nonce aléatoire lié au DID
    signal input avatarSalt;      // Sel unique pour cet avatar
    signal input timestampPreuve; // Heure de génération
    signal input merklePathElements[merkleDepth];
    signal input merklePathIndices[merkleDepth];
    signal input statutDID;       // 1 = ACTIF

    // === COMPOSANTS ===
    component poseidonCommit = Poseidon(2);
    component poseidonNull   = Poseidon(2);
    component poseidonLeaf   = Poseidon(2);
    component merkleVerif    = MerkleProof(merkleDepth);
    component checkActif     = IsEqual();
    component checkTimestamp = LessThan(64);

    // === CONTRAINTE C1 : commitment correct ===
    poseidonCommit.inputs[0] <== didId;
    poseidonCommit.inputs[1] <== noncePrive;
    poseidonCommit.out       === commitmentDID;

    // === CONTRAINTE C2 : nullifier correct ===
    poseidonNull.inputs[0] <== didId;
    poseidonNull.inputs[1] <== avatarSalt;
    poseidonNull.out       === nullifier;

    // === CONTRAINTE C3 : DID dans l'arbre Merkle ===
    poseidonLeaf.inputs[0] <== didId;
    poseidonLeaf.inputs[1] <== statutDID;
    merkleVerif.leaf                <== poseidonLeaf.out;
    merkleVerif.pathElements        <== merklePathElements;
    merkleVerif.pathIndices         <== merklePathIndices;
    merkleVerif.root                === merkleRoot;

    // === CONTRAINTE C4 : statut ACTIF ===
    checkActif.in[0] <== statutDID;
    checkActif.in[1] <== 1;
    checkActif.out   === 1;

    // === CONTRAINTE C5 : preuve récente (< 10 minutes) ===
    checkTimestamp.in[0] <== timestampPreuve;
```

```
    checkTimestamp.in[1] <== timestampLimite;
    checkTimestamp.out   === 1;
}

// Point d'entrée — signaux publics déclarés explicitement
component main {public [merkleRoot, commitmentDID, nullifier, timestampLimite]}
    = DIDProof(20);
```

3.2 Générer les entrées avec haute entropie

Créez un fichier generate_inputs.js. Ce script génère des valeurs secrètes cryptographiquement sûres (256 bits d'entropie réelle via crypto.randomBytes) et les formate pour snarkjs.

generate_inputs.js — copier intégralement

```
const crypto = require('crypto');
const { buildPoseidon } = require('circomlibjs');
const fs = require('fs');

// ── Générer un bigint aléatoire 256 bits (haute entropie) ──────
// crypto.randomBytes utilise le générateur cryptographique du système
// d'exploitation (CryptGenRandom sur Windows, /dev/urandom sur Linux/macOS)
// C'est le seul générateur acceptable pour des secrets cryptographiques.
function randomBigInt256() {
    const bytes = crypto.randomBytes(32); // 32 octets = 256 bits
    return BigInt('0x' + bytes.toString('hex'));
}

// ── Réduire modulo le champ BN254 utilisé par circom ────────────
// snarkjs et circom travaillent dans le champ premier BN254.
// Toutes les valeurs doivent être inférieures à cette constante.
const BN254_FIELD_PRIME = BigInt(

'21888242871839275222246405745257275088548364400416034343698204186575808495617'
);

function randomField() {
    // Générer jusqu'à ce que la valeur soit dans le champ
    // (rejet si >= prime — rarissime, environ 1 fois sur 2^256)
    let val;
    do {
        val = randomBigInt256();
    } while (val >= BN254_FIELD_PRIME);
    return val;
}

async function main() {
    const poseidon = await buildPoseidon();

    // ── Valeurs secrètes — haute entropie garantie ──────────────
    const didId      = randomField(); // Votre DID privé simulé
```

```
    const noncePrive = randomField(); // Nonce lié au DID
    const avatarSalt = randomField(); // Sel unique pour cet avatar

    console.log('DID privé       :', didId.toString());
    console.log('Nonce privé     :', noncePrive.toString());
    console.log('Avatar salt     :', avatarSalt.toString());
    console.log('--- Gardez ces valeurs SECRÈTES et LOCALES ---');

    // ── Calculer les valeurs publiques via Poseidon ──────────────
    // Poseidon est une fonction de hachage optimisée pour les circuits ZKP
    const commitment = poseidon.F.toString(
        poseidon([didId, noncePrive])
    );
    const nullifier = poseidon.F.toString(
        poseidon([didId, avatarSalt])
    );

    // ── Simuler un arbre Merkle simple (profondeur 20) ──────────
    // En production, cet arbre vient du registre blockchain
    const statutDID = 1n; // 1 = ACTIF
    const feuille   = poseidon.F.toString(
        poseidon([didId, statutDID])
    );

    // Chemin Merkle factice pour la démonstration
    // En production : récupéré depuis le noeud blockchain
    const merklePathElements = Array(20).fill(0n).map(() => randomField());
    const merklePathIndices  = Array(20).fill(0n).map(() =>
BigInt(Math.round(Math.random())));

    // Calculer la racine Merkle (simplifiée pour la démo)
    let currentHash = BigInt(feuille);
    for (let i = 0; i < 20; i++) {
        const [left, right] = merklePathIndices[i] === 0n
            ? [currentHash, merklePathElements[i]]
            : [merklePathElements[i], currentHash];
        currentHash = BigInt(poseidon.F.toString(poseidon([left, right])));
    }
    const merkleRoot = currentHash;

    // ── Timestamps ──────────────────────────────────────────────
    const maintenant     = BigInt(Math.floor(Date.now() / 1000));
    const timestampLimite = maintenant + 600n; // +10 minutes

    // ── Assembler les entrées du circuit ────────────────────────
    const inputs = {
        // Entrées publiques
        merkleRoot:           merkleRoot.toString(),
        commitmentDID:        commitment,
        nullifier:            nullifier,
        timestampLimite:      timestampLimite.toString(),
```

```
        // Entrées privées (NE JAMAIS PARTAGER)
        didId:                didId.toString(),
        noncePrive:           noncePrive.toString(),
        avatarSalt:           avatarSalt.toString(),
        timestampPreuve:      maintenant.toString(),
        merklePathElements:   merklePathElements.map(e => e.toString()),
        merklePathIndices:    merklePathIndices.map(e => e.toString()),
        statutDID:            '1',
    };

    // ── Sauvegarder dans input.json ──────────────────────────
    fs.writeFileSync('input.json', JSON.stringify(inputs, null, 2));
    console.log('');
    console.log('✓ Fichier input.json généré avec succès.');
    console.log('✓ Entropie : 256 bits par valeur secrète.');
    console.log('✓ Source entropie : crypto.randomBytes (CSPRNG système).');
    console.log('');
    console.log('ATTENTION : input.json contient vos valeurs secrètes.');
    console.log('Ne le partagez JAMAIS. Supprimez-le après usage.');
}

main().catch(console.error);
```

> La haute entropie est garantie par crypto.randomBytes, qui appelle directement le générateur cryptographique du système d'exploitation (Windows : BCryptGenRandom, Linux/macOS : /dev/urandom). Ne remplacez jamais cet appel par Math.random() — Math.random() n'est PAS cryptographiquement sûr.

4. Compiler le circuit et générer la preuve

Cette étape transforme votre fichier circom en R1CS, prépare les clés cryptographiques, et génère votre première preuve ZKP.

> Toutes les commandes ci-dessous sont à exécuter dans votre dossier de projet (mon-projet-zkp), dans l'ordre exact indiqué. Une étape manquée rend les suivantes impossibles.

4.1 Étape 1 — Compiler le circuit

La compilation transforme votre fichier .circom en trois fichiers : R1CS (contraintes), WASM (calculateur de witness) et fichier de symboles de débogage.

Windows 10/11

```
circom did_proof.circom `
  --r1cs `
  --wasm `
  --sym `
  --output .
```

macOS

```
circom did_proof.circom \
  --r1cs \
  --wasm \
  --sym \
  --output .
```

Linux Debian/Ubuntu

```
circom did_proof.circom \
  --r1cs \
  --wasm \
  --sym \
  --output .
```

Fichiers créés après compilation :

Fichier	Description
did_proof.r1cs	Le circuit en format Rank-1 Constraint System — entrée principale de snarkjs
did_proof_js/	Dossier contenant le calculateur de witness (generate_witness.js + did_proof.wasm)
did_proof.sym	Symboles de débogage — utile si le circuit échoue

4.2 Étape 2 — Télécharger les paramètres Powers of Tau

Les paramètres Powers of Tau sont le résultat d'une cérémonie de confiance multi-parties. Ce fichier est public et identique pour tous.

PowerShell / Terminal — tous systèmes (remplacer \ par ` sur Windows PowerShell)

```
# Télécharger les paramètres pour un circuit de moins de 2^20 contraintes
# Fichier d'environ 700 Mo — le téléchargement peut prendre quelques minutes
curl -L https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_20.ptau
\
     -o pot20_final.ptau

# Vérifier le téléchargement (hash SHA256 attendu)
snarkjs powersoftau verify pot20_final.ptau
```

> Sur Windows PowerShell, remplacez le \ de fin de ligne par ` (accent grave) pour la continuation de ligne. Ou écrivez la commande sur une seule ligne sans le signe de continuation.

4.3 Étape 3 — Préparer la clé de prouveur (ceremony setup)

PowerShell / Terminal — tous systèmes

```
# Phase 2 de la cérémonie — spécifique à votre circuit
snarkjs groth16 setup did_proof.r1cs pot20_final.ptau did_proof_0000.zkey

# Contribuer à la cérémonie (ajouter votre entropie)
# Une phrase secrète vous sera demandée — tapez n'importe quel texte long
snarkjs zkey contribute did_proof_0000.zkey did_proof_final.zkey \
    --name="Mon identité anonyme" -v

# Exporter la clé de vérification publique
snarkjs zkey export verificationkey did_proof_final.zkey verification_key.json
```

Lorsque vous contribuez à la cérémonie, snarkjs vous demandera d'entrer une phrase secrète. Tapez une phrase longue et aléatoire — par exemple une suite de mots sans signification. Cette contribution personnelle renforce la sécurité cryptographique du fichier .zkey final.

4.4 Étape 4 — Générer les entrées

PowerShell / Terminal — tous systèmes

```
# Générer les valeurs secrètes avec haute entropie
node generate_inputs.js

# Résultat attendu :
# DID privé       : 1847392847392...
# Nonce privé     : 9384729384729...
# Avatar salt     : 2847293847293...
# --- Gardez ces valeurs SECRÈTES et LOCALES ---
# ✓ Fichier input.json généré avec succès.
# ✓ Entropie : 256 bits par valeur secrète.
```

4.5 Étape 5 — Calculer le witness

Le witness est l'ensemble des valeurs intermédiaires calculées par le circuit à partir de vos entrées. Il est nécessaire pour générer la preuve.

Windows 10/11	macOS	Linux Debian/Ubuntu

`node did_proof_js\generate_witness.js `
 	
 `did_proof_js\did_proof.wasm `
 `  input.json `
 `  witness.wtns`
	`node did_proof_js/generate_witness.js \`
	
	 `did_proof_js/did_proof.wasm \`
	 `  input.json \`
	 `  witness.wtns`
		`node did_proof_js/generate_witness.js \`
		
		 `did_proof_js/did_proof.wasm \`
		 `  input.json \`
		 `  witness.wtns`

4.6 Étape 6 — Générer la preuve Groth16

PowerShell / Terminal — tous systèmes

```
# Générer la preuve — opération la plus longue (~30 secondes à 3 minutes)
# selon la puissance de votre machine
snarkjs groth16 prove did_proof_final.zkey witness.wtns proof.json public.json

# Résultat attendu : deux fichiers créés
#   proof.json   — la preuve cryptographique
#   public.json  — les signaux publics associés
```

4.7 Étape 7 — Vérifier la preuve localement

PowerShell / Terminal — tous systèmes

```
# Vérifier que la preuve est valide avant de la soumettre
snarkjs groth16 verify verification_key.json public.json proof.json

# Résultat attendu :
# [INFO]  snarkJS: OK!
```

> Si vous voyez "INVALID" au lieu de "OK!", vérifiez que vos entrées dans input.json satisfont bien toutes les contraintes du circuit. Relancez node generate_inputs.js pour régénérer des entrées cohérentes.

5. Comprendre les fichiers de sortie

snarkjs produit plusieurs fichiers. Il est essentiel de savoir lesquels sont publics, lesquels sont privés, et que faire avec chacun.

5.1 Vue d'ensemble de tous les fichiers

Fichier	Description et niveau de confidentialité
did_proof.r1cs	Le circuit compilé. Partageable — ne contient aucun secret. Utilisé pour la cérémonie.
did_proof.wasm	Calculateur de witness. Partageable. Doit être distribué aux utilisateurs finaux.
pot20_final.ptau	Paramètres Powers of Tau. Public. Identique pour tous les projets.
did_proof_0000.zkey	Clé intermédiaire avant contribution. À supprimer après la cérémonie.
did_proof_final.zkey	Clé de prouveur finale. PRIVÉE — garder en sécurité sur le serveur. Ne jamais partager.
verification_key.json	Clé de vérification. PUBLIC — à publier sur la blockchain et distribuer aux vérificateurs.
input.json	Vos entrées secrètes. TRÈS PRIVÉ — supprimer après génération du witness.
witness.wtns	Le witness calculé. PRIVÉ — supprimer après génération de la preuve.
proof.json	La preuve ZKP. PUBLIC — à transmettre au vérificateur (smart contract).
public.json	Les signaux publics. PUBLIC — à transmettre avec proof.json.

5.2 Contenu de proof.json — la preuve cryptographique

Le fichier proof.json contient trois éléments mathématiques de la courbe BN254. Il est entièrement public et ne contient aucun secret.

proof.json — exemple de sortie (valeurs fictives pour illustration)

```
{
  "pi_a": [
    "738192837465102938475610293847561029384756102938",
    "102938475610293847561029384756102938475610293847561029",
    "1"
  ],
  "pi_b": [
    [
      "984756102938475610293847561029384756102938475610293847561029384",
      "561029384756102938475610293847561029384756102938475610293847561"
    ],
```

```
    [
      "29384756102938475610293847561029384756102938475610293",
      "84756102938475610293847561029384756102938475610293847"
    ],
    ["1", "0"]
  ],
  "pi_c": [
    "475610293847561029384756102938475610293847561029384756",
    "102938475610293847561029384756102938475610293847561029",
    "1"
  ],
  "protocol": "groth16",
  "curve": "bn128"
}
```

5.3 Contenu de public.json — les signaux publics

Le fichier public.json contient les valeurs que le vérificateur connaît. Ces valeurs sont calculées dans input.json et constituent la partie visible de votre preuve.

public.json — exemple de sortie

```
[
  "847293847561029384756102938475610293847561029384756102938475610293",
  "561029384756102938475610293847561029384756102938475610293847561029",
  "293847561029384756102938475610293847561029384756102938475610293847",
  "1710000000"
]

// Ces quatre valeurs correspondent dans l'ordre à :
// [0] merkleRoot      — racine de l'arbre des DID actifs
// [1] commitmentDID   — hash(did_id + nonce_prive)
// [2] nullifier       — hash(did_id + avatar_salt)
// [3] timestampLimite — expiration de la preuve
```

5.4 Que faire avec ces fichiers ?

Fichier(s)	Action à effectuer
proof.json + public.json	Soumettre au smart contract DIDVerifier sur la blockchain Aequitaverse. Commande : voir section 5.5.
verification_key.json	Publier une seule fois sur la blockchain au déploiement du contrat. Ne change jamais.
did_proof_final.zkey	Garder sur un serveur sécurisé. Nécessaire pour que les utilisateurs génèrent leurs preuves.
input.json	SUPPRIMER immédiatement après avoir généré witness.wtns. Ne jamais stocker.

Fichier(s)	Action à effectuer
witness.wtns	SUPPRIMER immédiatement après avoir généré proof.json. Ne jamais stocker.

5.5 Soumettre la preuve via la ligne de commande

Pour soumettre votre preuve au smart contract Aequitaverse, utilisez la commande d'export Solidity de snarkjs puis l'interface Hardhat ou Foundry.

PowerShell / Terminal — tous systèmes

```
# Exporter les paramètres de la preuve au format Solidity (calldata)
snarkjs groth16 exportsolidityverifier verification_key.json verifier.sol

# Exporter les arguments d'appel du contrat
snarkjs zkey export soliditycalldata public.json proof.json

# Résultat : une ligne au format Solidity prête à copier-coller
# dans l'interface du contrat DIDVerifier ou dans un script Hardhat.

# Exemple de résultat (valeurs fictives) :
# ["0x1a2b...","0x3c4d..."],
# [["0x5e6f...","0x7a8b..."],["0x9c0d...","0x1e2f..."]],
# ["0x3a4b...","0x5c6d..."],
# ["0x7e8f...","0x9a0b...","0x1c2d...","0x3e4f..."]
```

5.6 Résumé des commandes — tableau de référence rapide

Opération	Commande
Compiler le circuit	circom did_proof.circom --r1cs --wasm --sym --output .
Télécharger Powers of Tau	curl -L [URL ptau] -o pot20_final.ptau
Setup cérémonie	snarkjs groth16 setup did_proof.r1cs pot20_final.ptau did_proof_0000.zkey
Contribuer à la cérémonie	snarkjs zkey contribute did_proof_0000.zkey did_proof_final.zkey
Exporter clé vérification	snarkjs zkey export verificationkey did_proof_final.zkey verification_key.json
Générer les entrées	node generate_inputs.js
Calculer le witness	node did_proof_js/generate_witness.js did_proof_js/did_proof.wasm input.json witness.wtns
Générer la preuve	snarkjs groth16 prove did_proof_final.zkey witness.wtns proof.json public.json
Vérifier la preuve	snarkjs groth16 verify verification_key.json public.json proof.json

Opération	Commande
Exporter pour Solidity	snarkjs zkey export soliditycalldata public.json proof.json

6. Dépannage — problèmes courants

Problème	Solution
"node" n'est pas reconnu	Windows : redémarrez PowerShell après installation de Node.js. macOS/Linux : vérifiez que /usr/local/bin est dans votre PATH (echo $PATH).
"permission denied" lors de npm install -g	Linux/macOS : ajoutez sudo devant la commande. Windows : lancez PowerShell en administrateur.
circom : "command not found"	Vérifiez que Rust et cargo sont installés (cargo --version). Relancez cargo install --git https://github.com/iden3/circom.
Erreur "INVALID" lors de la vérification	Vos entrées ne satisfont pas les contraintes. Relancez node generate_inputs.js et recommencez depuis l'étape 4.5.
Erreur de mémoire lors de la génération	Le circuit est très large. Augmentez la mémoire Node.js : NODE_OPTIONS=--max-old-space-size=8192 snarkjs groth16 prove ...
Téléchargement ptau très lent	Le fichier fait ~700 Mo. Utilisez un réseau câblé. Vous pouvez reprendre un téléchargement interrompu avec curl -C -.
La cérémonie prend trop de temps	Normal — le setup peut prendre 5 à 15 minutes selon votre machine. Ne pas interrompre le processus.
input.json : "Error: Assert Failed"	Vos valeurs secrètes ne correspondent pas aux valeurs publiques. Régénérez avec node generate_inputs.js sans modifier manuellement les valeurs.

6.1 Obtenir de l'aide

- Documentation officielle snarkjs : https://github.com/iden3/snarkjs
- Documentation circom : https://docs.circom.io
- Forum Aequitaverse (territoire Civitas) : votre administrateur système dispose du lien d'accès
- En cas d'urgence : contactez l'équipe technique via votre avatar Aequitaverse certifié

Annexe — Règles de sécurité obligatoires

> Ces règles ne sont pas optionnelles. Leur non-respect peut compromettre l'ensemble du système de preuves Aequitaverse et invalider votre DID.

15. Ne partagez JAMAIS les fichiers input.json, witness.wtns, et did_proof_final.zkey. Ces fichiers contiennent ou permettent de reconstruire vos secrets cryptographiques.
16. Supprimez input.json immédiatement après avoir généré witness.wtns. La commande est : del input.json (Windows) ou rm input.json (macOS/Linux).

17. Supprimez witness.wtns immédiatement après avoir généré proof.json.
18. Stockez did_proof_final.zkey sur un support chiffré (chiffrement de disque activé). Sur Windows : BitLocker. Sur macOS : FileVault. Sur Linux : LUKS.
19. Ne générez jamais vos secrets avec Math.random() ou tout autre générateur non cryptographique. Utilisez uniquement crypto.randomBytes comme montré dans generate_inputs.js.
20. Vérifiez toujours localement votre preuve (snarkjs groth16 verify) avant de la soumettre à la blockchain.
21. Renouvelez vos entrées (avatar_salt) pour chaque nouvel avatar. Un même salt réutilisé pour deux avatars permet de les corréler — ce qui détruirait votre anonymat.
