



CRYPI IDENTITY PROJECT

A Zero-Knowledge Identity Verification Framework
01. July 2025



Auguste Charpentier

Raphaël Gonon

Rémy Le Bohec

Quentin Rataud

1. Zero-Knowledge Proofs Overview

1.1. Definition and Fundamental Properties

A *zero-knowledge proof* (ZKP) is a cryptographic protocol in which one party, the **prover**, convinces another party, the **verifier**, that a given statement is true without disclosing any additional information beyond the validity of the statement. For instance, a party may wish to prove that he knows the answer of a puzzle to another party, without revealing the answer. This can generalize to nearly any equation. A ZKP aims at proving the knowledge of a private solution of these equation (denoted as a **witness**), without revealing any information about the witness in itself.

A secure ZKP satisfies three essential properties:

- **Completeness:** If the statement is true and both parties follow the protocol honestly, the proof will be valid, and thus the verifier always accepts the proof.
- **Soundness:** If the statement is false, no cheating prover can convince the honest verifier, except with negligible probability.
- **Zero-Knowledge:** The verifier learns nothing besides the fact that the statement holds; no extra information about the witness is revealed.

1.2. Sigma-Protocol: Discrete Logarithm Example

We illustrate a simple interactive ZKP, the **Schnorr Sigma-protocol**, for proving knowledge of a discrete logarithm in a prime-order group: let p, q be primes, g a generator of a subgroup of \mathbb{F}_p^\times of order q , and $y = g^w \bmod p$. g, p and y are made public, and the prover knows the secret exponent w and wishes to prove this fact without revealing w .

1. **Commitment:** Prover selects random $r \in \{0, 1, \dots, q-1\}$ and computes $a = g^r \bmod p$ then sends a to the verifier.
2. **Challenge:** Verifier chooses a random challenge e and sends e to the prover.
3. **Response:** Prover computes $z = r + ew \bmod q$ and sends z back. This *converts* the statement into another statement whose solution can be deduced from w , while masking w thanks to random data.
4. **Verification:** Verifier checks that $g^z = a \cdot y^e \bmod p$.

If the equality holds, the verifier accepts; otherwise, rejects.

1.3. Categories of Zero-Knowledge Proof Systems

Traditional ZKPs often require interaction: the verifier issues random **challenges** and the prover responds appropriately, ensuring soundness. However, this comes at a cost of a lot of communication overhead. Many modern systems employ some heuristics, such as the **Fiat-Shamir heuristic** to convert interactive proofs into non-interactive ones by replacing the verifier's random challenges with hashes of the prover's messages, achieving a **non-interactive zero-knowledge proof** (NIZK). Non-Interactive Zero-Knowledge proofs are performed in only two steps:

1. The **prover** sends the whole proof at once;
2. The **verifier** checks its validity.

Modern ZKP frameworks vary along axes of interaction, proof size, and trust assumptions. Here are three commonly used examples today:

- **SNARKs** (Succinct Non-interactive Arguments of Knowledge) Secure under knowledge-of-exponent assumptions and typically require a **trusted setup** for generating public parameters. Produce very small proofs (often \ll 1KB) with fast verification.
- **STARKs** (Scalable Transparent ARguments of Knowledge) Rely only on publicly verifiable randomness (no trusted setup). Proof sizes larger than SNARKs but remain scalable and transparent.
- **Bulletproofs** Non-interactive and do not require a trusted setup, suited for range proofs and arithmetic circuits. Proof sizes grow logarithmically with the size of the statement.

Each system presents trade-offs between setup assumptions, proof size, prover/verifier efficiency, and trust model. Selecting the appropriate ZKP depends on the application's performance and security requirements.

2. Overview of the Identity Project Framework

This section provides a high-level overview of our identity verification framework, detailing the roles of each agent and the sequence of interactions during the authentication process. Through a combination of client-side proof generation, server-side verification, and secure commitment storage, we ensure that sensitive personal data remains private while enabling reliable identity-based access control.

2.1. Agents and Their Responsibilities

1. Issuing Authority (Authority Site)

- Hosts a web interface where authority agents submit raw identity attributes (first name, last name, date of birth, license category).
- Computes a Poseidon-based cryptographic commitment over the user's identity data and a fresh nonce, then immutably registers this commitment in the **Trusted Storage** (e.g., a blockchain).
- Gives back to the user a virtual identity card (a file named `identity.json`), grouping all of the user's private information. This file should stay on the user's laptop and should never be disclosed. Information contained in this file will be used to generate proofs of knowledge of some of its content.

2. Verifier Interface (Demonstration Server)

- Provides a *Configure Proof* UI where the relying party (e.g., a server that only admits users 18+) sets verification constraints (e.g., date of birth before 2007).
- Constructs an `identity://` URI embedding the verifier's origin and chosen constraints, then redirects the user agent to this URI, such as `identity://verify?dob_before=123456`

3. Identity Client (Client Application)

- Registered as the handler for the `identity://` scheme on the user's machine.
- On invocation, parses the incoming URI, extracts the origin and requested checks, and presents a zenity confirmation dialog listing exactly which attributes will be proven and disclosed to the server.
- Upon user approval, loads the locally stored `identity.json` (or prompts for it), performs non-interactive SNARK proof generation (Groth16 + Poseidon), and re-computes the commitment.

- Packages the zero-knowledge proof, commitment, and echoed constraint parameters into callback query parameters, then invokes the verifier's origin URL (via xdg-open or browser redirect).

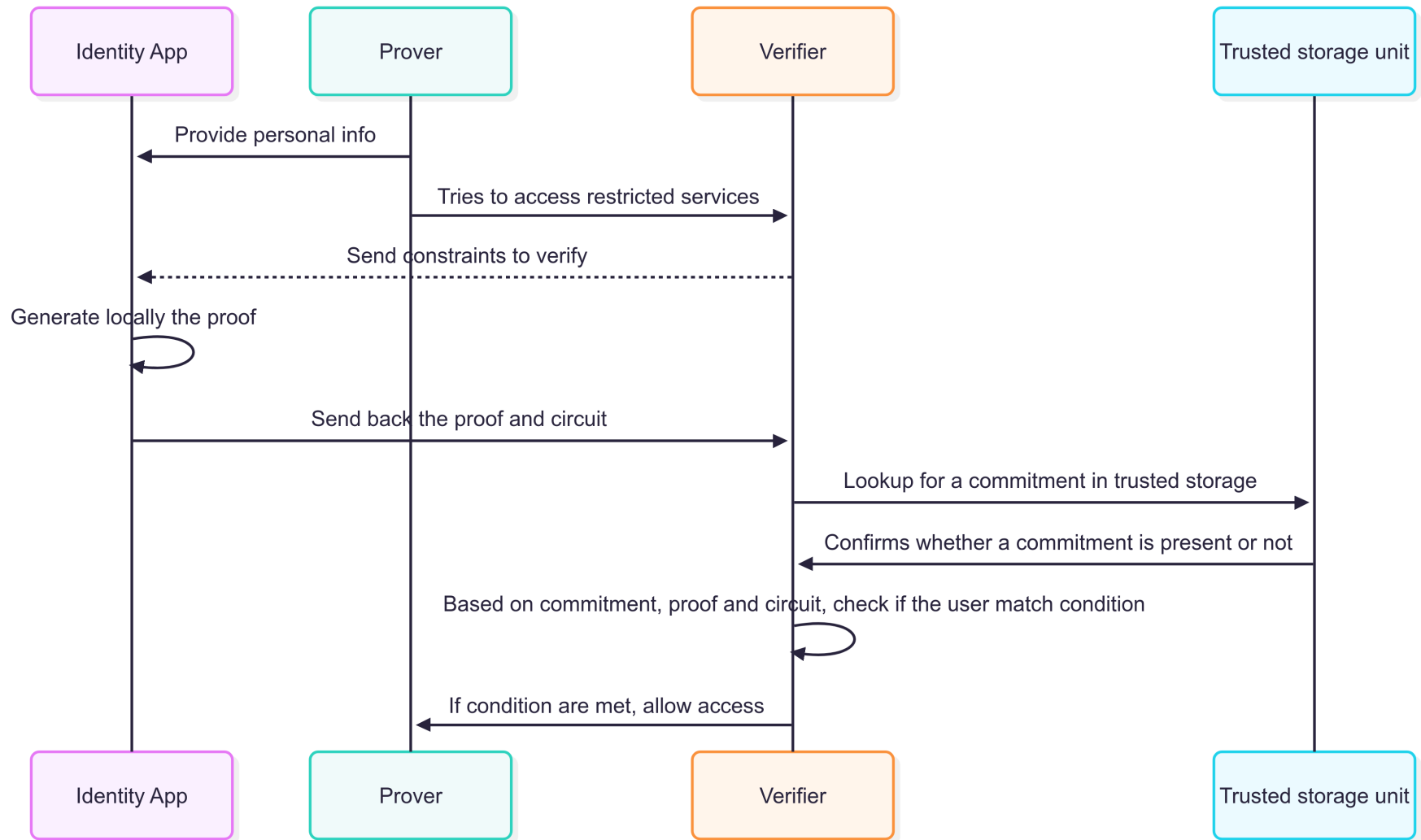
4. Trusted Storage (Blockchain or Commitment Store)

- Maintains a public ledger of all valid identity commitments submitted by the Issuing Authority.
- Is queried both at registration time (to record new commitments) and at verification time (to ensure the proof's commitment was indeed issued by the trusted authority).

2.2. Interaction Sequence

1. **Access Request** The End-User Client requests a protected resource from the Verifier Server. The server's policy requires proof of being over 18.
2. **Policy Encoding and Redirect** The Verifier Server constructs an `identity://verify?origin=<callback>&dob_before=<threshold>` URI encoding the age check and redirects the client to it.
3. **Protocol Handling and User Consent** The user's system launches the Identity Client Application (via the `identity://` handler). The application parses the parameters, displays a Zenity (or CLI) prompt summarizing the checks (e.g., "Confirm your birth date is before YYYY-MM-DD"), and requests user approval.
4. **Proof Generation** Upon confirmation, the Identity Client:
 - Loads `identity.json` to obtain private data (dob, license, nonce, expiration, names).
 - Performs local sanity checks (expiration date vs. today, matching requested name/license).
 - Recomputes the Poseidon commitment and builds the ZK circuit with public inputs (today, flags, and values).
 - Executes Groth16 to generate a succinct proof.
5. **Callback and Commitment Push** The Identity Client appends proof, commitment, and other inputs to the original callback URI and opens it (via xdg-open).
6. **Proof Verification** The Verifier Server receives the callback request:
 - Extracts proof and commitment from URI parameters.
 - Queries Trusted Storage (blockchain) to confirm the commitment was registered.
 - Invokes the Identity Client verify function (Groth16 verification against the circuit's public inputs).
 - If both storage check and proof verification succeed, the server grants access.

Overview of the Identity Project Framework



3. The Identity ZKP Circuit

3.1. Circuit Purpose and High-Level Structure

Our circuit is designed once and for all to support all verification policies (age checks, name checks, license checks, ...) without needing per-use-case recompilation. It takes as private inputs the full set of identity attributes stored in the user's `identity.json`:

- `dob` (date-of-birth, days-since-CE)
- `license` (license category as integer)
- `nonce` (fresh random 128-bit value)
- `expiration` (ID expiration date, days-since-CE)
- `first_name` (string)
- `last_name` (string)¹

And it takes as public inputs:

- `today` (current date, days-since-CE)
- `commitment` (Poseidon hash of all private fields)
- A series of flag/value pairs for each optional check:
 - `req_fname_flag`, `req_fname_val`
 - `req_lname_flag`, `req_lname_val`
 - `dob_before_flag`, `dob_before_val`
 - `dob_after_flag`, `dob_after_val`
 - `dob_equal_flag`, `dob_equal_val`
 - `req_license_flag`, `req_license_val`

All elements are encoded as points over the elliptic curve BN254.

3.2. Commitment Verification

1. We recompute a nested Poseidon hash over the private inputs, chaining five calls of capacity-3 absorption:

$$\text{commitment} = H(H(H(H(\text{dob} \parallel \text{license}) \parallel H(\text{exp} \parallel \text{nonce})) \parallel \text{firstname}) \parallel \text{lastname})$$

This construction arises from Poseidon's arity-3 design and still ensures full preimage resistance and collision resistance under standard cryptographic assumptions. By only publishing commitment on-chain, no personal data is ever revealed.

¹For improved privacy and capabilities, we decided to make the identity expiration date, first name and last name private data. The circuit allows to generate proofs over the name if authentication by name is needed.

3.3. Mandatory Expiration Check

We always enforce:

$$\text{expiration} > \text{today}$$

so that expired identity cards cannot produce valid proofs. This check is hardwired to prevent reuse of stale identity data.

3.4. Conditional Attribute Checks

Each optional check uses a gated equality (or comparison) pattern of the form

$$(\text{constraint_value} - \text{witness_value}) \times \text{flag} = 0$$

Thus:

- If $\text{flag} = 1$, the circuit forces the corresponding check to hold.
- If $\text{flag} = 0$, the term is multiplied by zero, and therefore the check is disabled.

This pattern scales to all name, date, and license checks, giving us a single, compact, and versatile circuit.

4. Choice of Libraries

This chapter presents and justifies the main cryptographic primitives and software libraries chosen for our identity verification framework. We first introduce the high-level proof model (SNARK), then describe why we selected the Groth16 instantiation, motivate the choice of the Poseidon hash, and finally explain our adoption of the Arkworks Rust ecosystem.

4.1. SNARKs as the Foundational Proof System

We employ a **Succinct Non-Interactive Argument of Knowledge (SNARK)** as the core proof primitive. A SNARK offers:

- **Succinctness:** proof sizes remain constant, regardless of circuit complexity.
- **Non-Interactivity:** a single proof message suffices—no back-and-forth protocol is needed.
- **Argument-of-Knowledge:** sound under standard cryptographic assumptions, assuring the verifier that the prover indeed knows a valid witness.

These properties align perfectly with our requirements: lightweight proofs easily embedded in URLs and fast client-side generation, while preserving strong zero-knowledge guarantees.

4.2. Groth16: Our SNARK Instantiation

We chose **Groth16** as the concrete SNARK scheme for its proven efficiency and widespread adoption. Especially, Groth16 features the following characteristics:

- **Proof Size:** exactly three group elements (two in G_1 and one in G_2), totaling 256 bytes regardless of circuit depth.
- **Verification Time:** typically a few tenth of a second using optimized pairings on BN254.
- **Setup:** requires a one-time, trusted-setup ceremony producing a proving key and a verifying key.

Groth16's succinct proofs and fast verification make it ideal for client-side proof generation and server-side checks in web environments. Its trusted-setup requirement is mitigated by distributing pre-generated keys and allowing users to replace them if desired.

Groth16's workflow resembles this:

1. `Setup()` \rightarrow `(ProvingKey, VerifyingKey)`

2. `Prove(ProvingKey, PrivateInputs, PublicInputs) → Proof`
3. `Verify(VerifyingKey, PublicInputs, Proof) → {true, false}`

This three-stage workflow cleanly separates parameter generation, proof creation, and verification, matching our architecture's client and server roles.

4.3. Poseidon: ZKP-Friendly Hashing

For all commitment computations, we use the **Poseidon** hash function, specifically designed for zero-knowledge proof systems.

- **ZKP Efficiency:** Poseidon's arity-3 sponge construction maps directly to low-degree polynomial constraints, minimizing circuit size.
- **Cryptographic Security:** instantiations over BN254 satisfy collision and preimage resistance under standard assumptions.

Poseidon's design drastically reduces the number of R1CS constraints compared to generic hashes such as SHA256, accelerating proof generation.

4.4. Arkworks-RS: The Rust ZKP Ecosystem

We build our circuits and runtime on the **Arkworks** library suite in Rust, that we discovered through hands-on challenges like CryptoHack's "Ticket Maestro" exercise.

- **Comprehensive Toolkit:** includes implementations of Groth16, Poseidon, R1CS gadgets, and serialization utilities.
- **Performance:** optimized for pairing-based operations on BN254, matching our curve choice.
- **Community & Audit:** actively maintained by teams from Zcash and Ethereum Foundation, ensuring ongoing security reviews.

We used the following features :

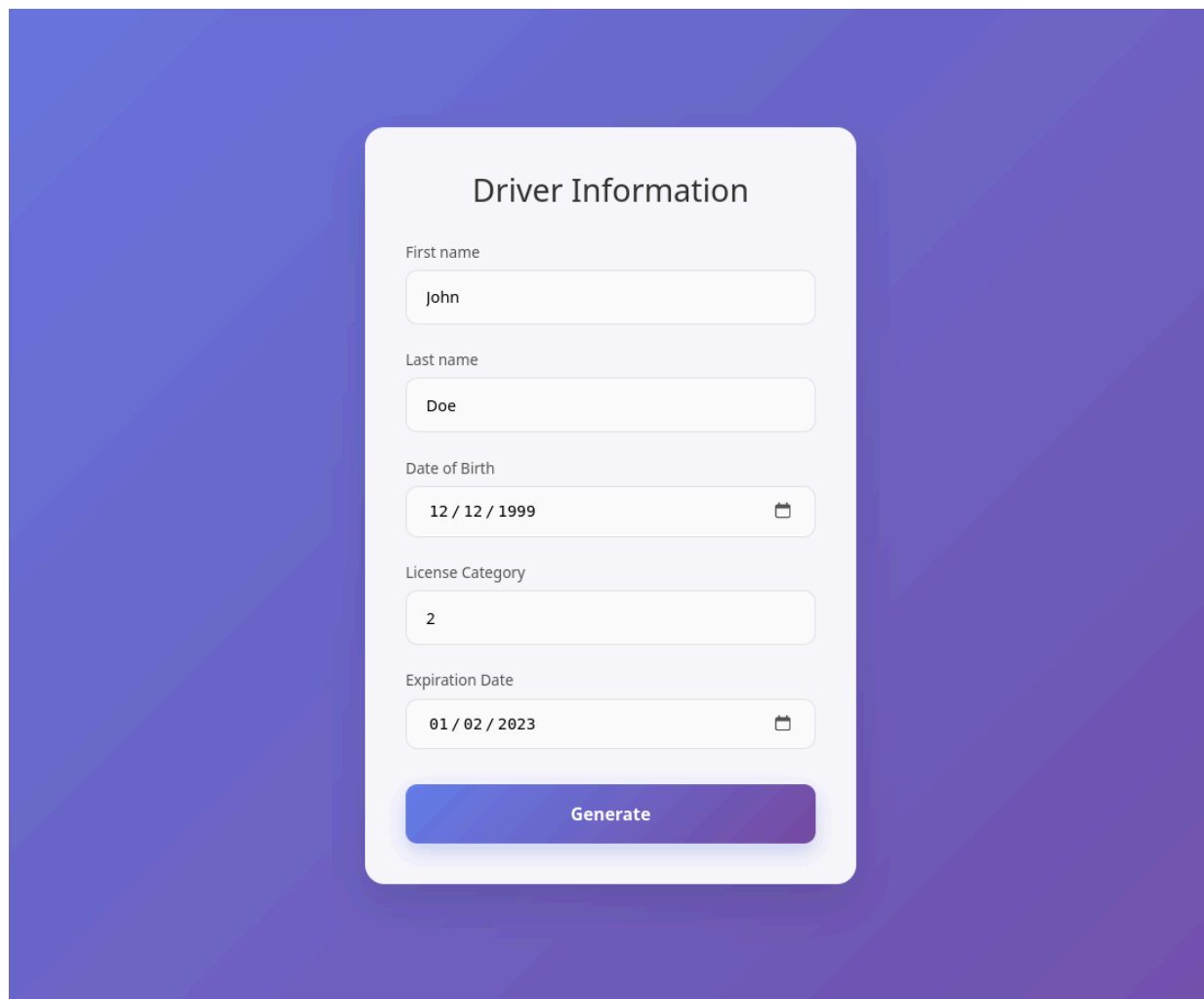
- `ark-groth16` — Groth16 SNARK paradigm
- `ark-ff` / `ark-serialize` — finite-field and (de)serialization primitives
- `ark-rlcs-std` — high-level arithmetic and comparison gadgets
- `arkworks-native-gadgets` & `arkworks-rlcs-gadgets` — Poseidon hash in native and constraint form

5. Outcomes and Future Directions

5.1. Practical Results in Deployment

In practice, our identity verification framework has proven to be both responsive and user-friendly. The complete proof generation process—covering local checks, circuit instantiation, and Groth16 proof creation—takes on the order of one second on typical consumer hardware. Similarly, verifier-side proof verification completes in approximately one second, making real-time authentication feasible for most web and mobile scenarios.

Here is the site used by the authority to capture user information:



The image shows a web form titled "Driver Information" centered on a purple gradient background. The form is a white rounded rectangle containing several input fields and a button. The fields are labeled "First name", "Last name", "Date of Birth", "License Category", and "Expiration Date". The "First name" field contains "John", "Last name" contains "Doe", "Date of Birth" contains "12 / 12 / 1999", "License Category" contains "2", and "Expiration Date" contains "01 / 02 / 2023". Each date field has a small calendar icon to its right. At the bottom of the form is a blue button with the text "Generate".

After clicking on generate the information is sent to the blockchain and the `identity.json` file is downloaded to the user's computer.

Driver Information

First name

John

Last name

Doe

Date of Birth

12 / 12 / 1999

License Category

2

Expiration Date

01 / 02 / 2023

Generate

Generated Output:

```
{  "first_name": "John",  "last_name": "Doe",  "dob": 730100,  "license": 2,  "expiration": 738522,  "nonce": "XXXXXXXXXXXXXXXXXXXX"}
```

Here is the page for the verifier to indicate the constraints they wish to check:

Identity Verification

Configure what information you'd like to prove

First Name

e.g. Alice

Last Name

e.g. Smith

License Number

2

Born After

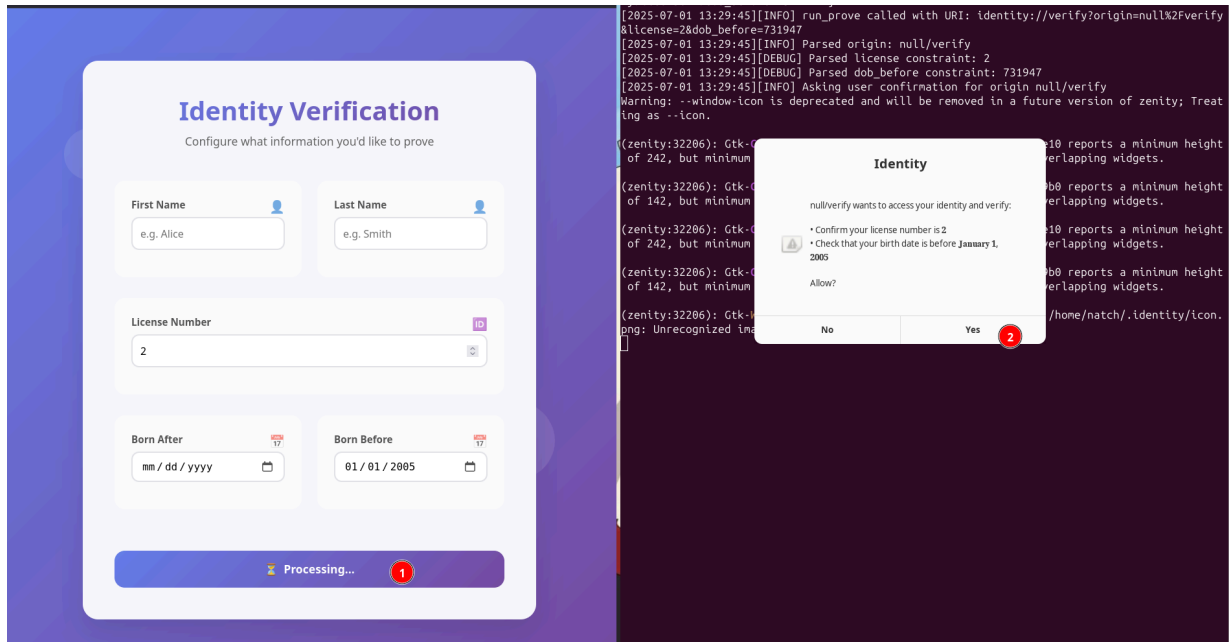
mm / dd / yyyy

Born Before

01 / 01 / 2005

Validate Identity

When you click on `Validate Identity`, the Identity application is called up to perform the verification process:



Once the verification procedure has been validated, we are taken to a page that confirms whether or not the constraints have been verified.

✓ Identity validated

- **Date of birth before:** 2005-01-01
- **License number:** 2

[← Back](#)

The user experience flows smoothly:

- **Issuance:** Users obtain their `identity.json` via the Authority Site in a single click.
- **Proof Configuration:** The Demonstration Server’s “Configure Proof” UI guides relying parties to assemble their constraints with clear icons and validation steps.
- **Consent Prompt:** The Identity Client’s Zenity dialog succinctly lists only the requested attributes (e.g., “Confirm your birth date is before YYYY-MM-DD”), ensuring informed consent.
- **Redirect & Callback:** Redirects to the `identity://` handler and back to the verifier occur in under half a second, with minimal browser prompts.

On the server side, integrating the verification flow into an Express.js endpoint proved straightforward: a single HTTP request validates both the commitment’s presence in trusted storage and the SNARK proof. Administrators will find the setup simple, requiring only placement of the proving/verifying keys in a known directory and running standard Node.js services.

5.2. Security Assumptions and Mitigations

Our framework relies on several critical security assumptions:

1. **Cryptographic Security of Poseidon** We assume that Poseidon is preimage- and collision-resistant. Its arity-3 sponge construction is chained as

$$\text{commitment} = H(H(H(H(\text{dob} \parallel \text{license}) \parallel H(\text{exp} \parallel \text{nonce})) \parallel \text{firstname}) \parallel \text{lastname})$$

Under standard algebraic hash assumptions, this construction preserves data confidentiality.

2. **Nonce Entropy**

The inclusion of a fresh nonce prevents brute-force linkage attacks: without the nonce, an attacker knowing most identity fields could iterate remaining possibilities until matching a public commitment. Our implementation generates a 128-bit nonce following ANSSI recommendations, ensuring the search space is sufficiently large. A weaker or predictable nonce would critically endanger privacy.

3. **Trusted Setup for SNARK**

Groth16 requires a trusted setup to generate the proving and verifying keys. If the setup were maliciously backdoored (which we cannot prove it is not), the key distributor could forge proofs for arbitrary identities. To mitigate:

- We distribute an example key-set bundled in the application, but we allow users to replace them.
- We recommend an optional multi-party computation (MPC) ceremony: stakeholders jointly generate keys so no single party controls the trapdoor. Our client simply loads keys from `$HOME/.identity`, making it trivial to swap in MPC-generated parameters.

By explicitly documenting these assumptions and providing mechanisms (e.g., key replacement, nonce verification) to strengthen them, we maintain a clear security model and reduce hidden risks.

5.3. Recommended Future Improvements

To further enhance robustness and usability, we propose mainly:

- **Transparent SNARK Variants**

Explore STARKs or Plonk: these eliminate trusted setup at the expense of larger proofs or slightly higher verification cost. A transparent protocol could increase trust for highly regulated deployments.

- **Decentralized Commitment Storage**

Instead of relying on a centralized database, publish all public commitments to a decentralized blockchain (e.g., Ethereum, a proof-of-stake chain, or a specialized layer-2 solution). This approach removes single points of failure, increases transparency, and

leverages the blockchain's inherent tamper-resistance to ensure commitments cannot be retroactively altered. This approach would also be a step forward Identity being a completely zero-trust framework.