zkLogin Report: Privacy-Preserving Authentication for Cardano

Eryx Team

December 19, 2025

Contents

1 Introduction

zkLogin is an authentication method for Cardano that allows Web3 users of the Cardano ecosystem to access wallets and dApps with their existing web2 credentials (such as Google, Facebook, Apple, etc.) while preserving self-custody and privacy. Its core innovation lies in combining an **OAuth / OpenID login flow** with a **zero-knowledge proof** and a private component that ensures the user's off-chain identity is not publicly linked to their on-chain address.

This protocol offers a login flow familiar to Web2 users while keeping the power of Web3. This allows for end-user growth in the Cardano community.

In this report, the reader should find a description of the protocol, pros and cons of using zkLogin, design decisions, other possible implementations beyond the scope of this grant, and a description of the security model behind zkLogin.

2 Background and Investigation

The original protocol was presented in the paper zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials. The report will make an overview of the background needed to make sense of zkLogin and a future section will enumerate the details that make this Cardano implementation different from the original protocol.

2.1 OpenID Connect and JWT Format

OpenID Connect (OIDC) is a web2 authentication protocol built on top of the OAuth 2.0 framework. It involves 3 main actors:

- An **OpenID Provider (OIDP)**
- An **end user**
- A **third-party application** (wallet or dApp)

The **OpenID provider** is an authority (like Google, Apple, Facebook, Microsoft, etc.) that provides an authentication mechanism for the user. A **third-party application** may allow a login mechanism via one of these providers.

OpenID uses an ID Token which is a JSON Web Token (or **JWT**) that holds claims about the authenticated user. They are formed by:

- **Header:** contains the type of the token, the algorithm used to digitally sign its content (for example, RS256) and the public key to check the signature. This public key must be published by the provider.
- **Payload:** holds claims about the authenticated user, including:
  - `sub`: identifies the user within the OIDP context.
  - `iss`: universally identifies the OIDP.

  - `aud`: identifies the third-party app within the OIDP context.
  - `nonce`: an arbitrary value sent in the authentication request, returned untouched, useful for integrity checks (takes part in the signature body).
  - Other fields: issue time, expiration time, name and email of the user, etc.

- **Signature:** the signature of the Payload, generated by the OIDP's private key and the signature algorithm specified in the header.

3 Protocol Overview

zkLogin is designed to provide a Two Factor Authentication method for a non-traditional Cardano address. This address is a **validator's hash address**, not one derived from a private key, but this is transparent for the user. The user will have its own address, which can be sent to anyone and use it to receive tokens, but the way those tokens are unlocked lies in the core of the zkLogin protocol, which this document will detail in the following sections.

3.1 Involved Actors

The protocol involves many actors:

- The **Users**: end users who own a zkLogin address.
- An **OpenID Provider (OIDP)**: an authority that supports the OpenID Connect protocol.
- A **Wallet (or dApp)**: implementing the zkLogin protocol, with a clear frontend/backend separation. The backend is optional and there should be a clear separation between the frontend and the backend so that the server has no way to create transactions on behalf of the user.
- The **Cardano Blockchain**: where transactions will occur.
- A **ZK proof service**: the user's device or an external service with the ability to create Groth16 Zero Knowledge proofs.
- A **Salt service**: this service must hold some user's secret and it can be implemented in the user's device or rely on an external service.

3.2 App-Flow

Assuming Google as the OpenID provider, a zkLogin use case should have the following steps:

1. The user opens the wallet app (wApp).

2. The wApp frontend creates a set of **ephemeral credentials** for the new session.
3. The user logs into the wApp with Google credentials, obtaining a signed JWT from Google with the user data.
4. The wApp requests access to the **secret salt** (from the salt service or the user itself).
5. The wApp derives the user address and governing script, so now it can show the account funds and history of transactions.
6. Given the derived address, the user now is able to deposit their assets in that specific address.
7. User begins a transfer of funds from the wallet to any Cardano address. The spending is then controlled by the ephemeral secret data, the salt and the Google authentication credentials.
8. A request is made to the **ZK proof service** to generate a proof ensuring certain statements (detailed later in the report).
9. The wApp frontend signs the transaction with one of the ephemeral credentials.
10. The transaction is submitted and succeeds if valid.

4 Adversarial Model

The protocol relies on partially trusting external agents, none of which are fully trusted. The design ensures no single external agent has enough information to act on the user's behalf.

- The **wallet** backend (optional, ideal implementations don't really have a backend).
- The **OpenID provider** has authentication data but no salt, cannot derive the address.
- The **Salt service** has the salt but no OpenID login data.
- The **ZK proof service** never has the session ephemeral secret key (eph_sk) to sign transactions.

The **application frontend** is trusted during the session. Ephemeral credentials expire (max_epoch), mitigating long-term security risks.

5 Protocol in Detail

5.1 Main Properties

First, let's start by enumerating the main properties that make this protocol different from a traditional scheme: **security** and **unlinkability**.

- **Security**: refers to the unforgeability of the scheme. An adversary cannot sign transactions on behalf of the user. Also, zkLogin supports expiration for its credentials.
- **Unlinkability**: no third party (except the application frontend, controlled by the user) is able to link the user zkLogin address with the OpenID Connect user credentials. In other words, on-chain and off-chain identity cannot be linked by any of the involved parties.

5.2 Session Credentials

It was mentioned in the protocol overview that some ephemeral credentials must be created by the application frontend when the user wants access to its account. These credentials are:

- an ephemeral public and private key-pair (`eph_pk` and `eph_sk`).
- a random value (`rand`).
- an expiration time. This is a way to set an expiration time for the session, and must be expressed in blockchain time. For Cardano, this can be a slot in an epoch, but we'll call it `max_epoch` for convention.

These values are used to compute a `nonce` value with the formula

$$\text{nonce} = H(\text{eph\_pk}, \text{rand}, \text{max\_epoch}).$$

The `eph_sk` is not used to compute the `nonce`. Instead, it's used to sign the transactions in this session.

These credentials are only valid for a single session, and are no longer valid when the `max_epoch` has passed in the Cardano blockchain.

5.3 Interaction with OpenID Provider

Once the session credentials are created, we can start the interaction with the OpenID Provider. For the development of the protocol, we will support Google as our authenticator by default. Recall from the JWT section that when requesting Google for authentication credentials we can send a `nonce` value that will return untouched in the JWT. Not surprisingly, this `nonce` will be the `nonce` computed from the session credentials.

Once the authentication process is completed, the application frontend will receive a JWT with an `iss`, `sub`, `aud` and `nonce` as described above. This JWT is encoded in base64 and signed by Google, which is a key point for the protocol security.

5.4 zkLoginId

Each user will have a **zkLoginId** linked to it's address. This is not the Cardano address itself, but a value from which the address can be derived. Let's explain how we generate this value. The OpenID flow has 3 values that we'll use for this purpose and that were mentioned earlier. These are the subject identifier (**sub**), the app's audience (**aud**) and the OIDP's identifier (**iss**).

If we were to build the zkLoginId from these 3 values, the user off-chain identity (Google credentials) would be directly linked to the on-chain identity (zkLoginId). To avoid this, the protocol uses an additional **salt**. The zkLoginId is what is called a **stable identifier** (or **stid**).

In other words, we compute

$$\text{zkLoginId} = H(\text{iss}, \text{aud}, \text{sub}, \text{salt}).$$

Remember that the `sub`, `aud` and `iss` values are obtained by the frontend when the user authenticates with the OpenID Provider, and the `salt` is retrieved by the user itself or by the communication with some salt service.

5.5 The validator script

At the beginning of this report it was stated that a zkLogin address is not a regular Cardano address, in the sense that it's not derived from a private key. Instead, it's a validator's hash address. Now we must address what is it that the validator is doing, because there lies the core of the protocol. Let's do a quick summary. At this point, the application frontend holds the following data:

Stable $\rightarrow$ Doesn't change over sessions.

- `sub` (identifies the user in the context of the OIDP)
- `aud` (identifies the application in the context of the OIDP)
- `iss` (identifies the OIDP)
- `salt` (secret value that unlinks the OIDP credentials from the zkLoginId)
- zkLoginId $= H(\text{iss}, \text{aud}, \text{sub}, \text{salt})$

Ephemeral $\rightarrow$ Different for each session.

- `[eph_pk, eph_sk]` (public and private key-pair)
- `rand` (a random session value)
- `max_epoch` (expiration time for the session)
- nonce $= H(\text{eph\_pk}, \text{rand}, \text{max\_epoch})$
- The complete and signed `JWT` from Google. Note that this is considered ephemeral because it contains the `nonce` value, so it will be different for each session although `sub`, `aud` and `iss` do not change over time.

A different script means a different address, and since we want all users to have an unique address, the protocol must make each validator script different. The way it's achieved is by **engraving the `zkLoginId` in the validator's source code**. This makes all the validators different thus making all the addresses different, which is what we were looking for. But now let's see which conditions should hold for an application to use such address.

The validator script must verify the association between the `eph_pk` and the hardcoded zkLoginId. These values are not directly linked, but they are linked in a specific session through the JWT. We can think of a "chain of associations":

1. The transaction `tx` that includes the validator and consumes an UTxO with the validator's address is signed with the `eph_sk`.
2. The `tx` redeemer must contain the `eph_pk` and validate the transaction signature. At this point we've only proved that the `eph_sk` and the `eph_pk` correspond to each other.
3. The `nonce` is composed of the `eph_pk`, `rand` and `max_epoch`, so the validator should also check that the nonce satisfies $\text{nonce} = H(\text{eph\_pk}, \text{rand}, \text{max\_epoch})$. This `nonce` comes from the JWT.
4. The validator should check that the `nonce` is present in the `JWT`. However, the provenance of this value is uncertain until it's signed by an OpenID Provider.
5. The OpenID provider signature should be checked in the `JWT` against an official public key of the provider. Once the validator checked that, it's sure that the rest of the `JWT` is also correct.
6. The last step is to verify that the written `zkLoginId` is composed of the data from the `JWT` and the provided salt, in other words, check the formula $\text{zkLoginId} = H(\text{iss}, \text{aud}, \text{sub}, \text{salt})$.
7. The `max_epoch` is valid with respect to the current blockchain time (isn't expired).

Once all the steps are completed, the validator can be sure that the transaction was signed by someone who

8. has access to the `salt`
9. is able to obtain a valid `JWT` with the credentials associated to the address (in other words, someone that is able to authenticate successfully with the OpenID Provider)

This would be enough if it wasn't for the visibility of the transaction: all the information used to verify the connection is public for everyone to see, and that is something that breaks both security and unlinkability. This is why we need to use ZK Proofs.

5.6 Adapting the validator script to use zkProofs

Most of the conditions the validator needs to check contain private inputs and are expensive computations to perform in a blockchain (like the hash function computation or the signature verification). These are the main reasons to use zk proofs as the main tool to preserve privacy and reduce the on-chain computation.

The protocol designs a circuit using the Groth16 proving system with the following inputs:

Public:

- `eph_pk`
- `OIDP pk`
- `zkLoginId`
- `max_epoch`

Private:

- `JWT` (including `sub`, `aud`, `iss` and `nonce`)
- `rand`
- `salt`

The circuit must assert the following conditions:

- zkLoginId $= H(\text{iss}, \text{aud}, \text{sub}, \text{salt})$
- nonce $= H(\text{eph\_pk}, \text{rand}, \text{max\_epoch})$
- The values for `sub`, `aud` and `iss` used to compute the `zkLoginId` are the ones present in the `JWT`
- The signature of the `JWT` validates with the publicly listed `OIDP pk`.

In other words, steps from 3. to 6. from the previous section are not performed by the validator itself but embedded in a zk proof that the Aiken script must verify. Since there are Groth16 validators implemented in Aiken, this is something possible to do.

6 Wallet responsibilities and decisions

The novelty of this project is the implementation of the base circuits and Aiken validators for the zkLogin protocol, but there are many things that a wallet or dApp frontend must do in order to integrate it. Also, there are many decisions to be made that affect security and integrity of the system. This section will list those responsibilities and decisions to be made by a willing application.

6.1 General responsibilities

The application must:

- Create the ephemeral credentials when the user wishes to start a new session: an ephemeral public and private key-pair (`eph_pk` and `eph_sk`) and a random value (`rand`). The application must ensure the secure storage of those credentials in the user device (the frontend) throughout the session, and its deletion when the session expires in such a way that set of credentials is never used again.
- It should be able to fetch the Cardano blockchain to get the current time, to set an according expiration time. The expiration time must be reasonable for a user session, not too long, not too short, but that's a design decision of the application.
- Based on those values, it should compute the `nonce` (and also store it throughout the session).
- It should be registered as a Google Client in the OpenID Connect protocol and allow authentication using this method. The authentication request must be done with the `nonce` from the previous step. Once the credentials are fetched, it should decode and parse them to get the `iss`, `aud` and `sub` values.
- It should obtain the user salt (more on salt management later).
- Based on the salt and the Google credentials, it should compute the `zkLoginId`. With this `zkLoginId` it should create the Aiken script that will govern the transactions from this address. The user address should be derived from that script.
- It should create or request the creation of the zk proof (more on proof creation later).
- Lastly, it should create the transaction. The transaction should consume an UTxO governed by the script, and should include the `max_epoch`, `zk proof` and `eph_pk` in the redeemer.

6.2 Salt service

The salt management has some variations. It's treated as something external to the protocol, in the sense that the protocol does not dictate how it should be done. However, this report will mention some possibilities for managing the user secret salt.

- **Managed by the user**: this is the most traditional approach. It has the advantage that no third party will ever have access to the secret salt, but the disadvantage is that this salt can be lost like any other secret.

- **Managed by the user device**: this is a sophistication of the previous option. The key can be locked in the user device and be kept safe by the operating system security rings. This, however, links the salt to a specific device. If this device were lost or stolen, the responsibility to remember the salt would fall on the user again.
- **Managed by a salt service**: this option relies on an untrusted third party to hold the salt. It should have its own authentication method (ideally different from the OIDC already use for zkLogin), store the user salt and retrieve it whenever needed. This option is not as unsafe as it seems, since the salt alone isn't enough to create a transaction in the name of the user. This is different than storing a secret key, since the secret key alone can be used to sign transactions.
- **Using a custom backend**: this is no different from the salt service, except the salt service is the backend of the application itself.

The ideal method will be the result of a trade-off between security and user experience.

6.3 Proof generation

The proof generation is a key point of the protocol, however there are some variations with respect to where the proof can be created.

- **On the user's device**: this option is by far the most secure, but has the disadvantage of using many resources that the device might not have. The proof generation could be slow which would make the overall user experience worse.
- **Use an external service**: this option can speed up the proof generation time by still preserving the security, since the `eph_sk` used to sign transactions is hidden. However, this breaks the unlinkability property for the zk proof service since it will have access to the `JWT` and the `salt`.
- **Using a custom backend**: this is no different from the previous option, except that the proof is generated by the application itself.

7 Advantages and disadvantages of zkLogin

The most important advantage of using zkLogin is that no third party will have a way of using the user address if this protocol is implemented according to the security model, but the user doesn't need to be the keeper of the secret keys.

In a traditional model, the user has 2 options:

1. safeguard the private key associated with his address (self-managed, non-custodial wallets).

2. trust a 3rd party to do it for him (custodial wallets).

If you choose the second option, a compromised or corrupted third party could have full control over the user's address. While choosing the first option may seem the best, handling and securing private keys is often error-prone, and many users end up losing their credentials and consequently, their assets. Some implementations of the zkLogin protocol prevent both scenarios. In particular, if the user chooses to use a salt service it wouldn't have to safeguard any data, but the salt alone isn't enough to control the wallet.

The downside of using zkLogin is that your address is linked to a specific OpenID provider and wallet/dApp, since the address is derived from the `sub`, `aud` and `iss` fields of the JWT. Some variants of the protocol prevent the dependency on the wallet by using only the `sub` and `iss` fields to compute the `zkLoginId`, but that is not explored in this project.

8 Technical Details

8.1 Validator overview

Let's see the data that should be present in the redeemer for the protocol to work:

Listing 1: Transaction Redeemer format

```
type ZkLoginRedeemer {
    max_epoch: Int,
    proof: Groth16Proof,
    eph_pk: ByteArray,
}
```

The validator body and its main functions look as follows:

Listing 2: Validator body

```
Validator ZKLogin {
    spend(
        _datum: Option<Data>,
        redeemer: ZkLoginRedeemer,
        _utxo: OutputReference,
        self: Transaction) {
    verifyGroth16(
        redeemer.proof,
        redeemer.eph_pk,
        redeemer.max_epoch,
        zkLoginID
    );
    verify_tx_is_signed(
        self.extra_signatories,
        redeemer.eph_pk
    );
```

```
    verify_timestamp(redeemer.max_epoch)
  }
}
```

An overview of the circuit looks like the following:

Listing 3: Circom circuit overview

```
template ZkLogin {
    // Public
    signal input eph_pk;
    signal input OIDP_pk;
    signal input jwt_signature;
    signal input zkLoginId;
    signal input nonce;
    signal input max_epoch;

    //Private
    signal input JWT;
    signal input rand;
    signal input salt;
    signal input sub;
    signal input aud;
    signal input iss;

    // JWT parsing
    component parser = JWTParser;
    parser.jwt <== JWT;
    parser.sub <== sub;
    parser.aud <== aud;
    parser.iss <== iss;
    parser.nonce <== nonce;

    // zkLoginId derivation check
    component id_derivation = ZkLoginIdDerivation;
    id_derivation.jwt <== JWT;
    id_derivation.salt <== salt;
    zkLoginId === id_derivation.out;

    // nonce derivation check
    component nonce_derivation = NonceDerivation;
    nonce_derivation.eph_pk <== eph_pk;
    nonce_derivation.rand <== rand;
    nonce_derivation.max_epoch <== max_epoch;
    nonce_derivation.out === nonce;

    // OIDP signature verification
    component verificator = JWTSignatureVerification;
    verificator.OIDP_pk <== OIDP_pk;
    verificator.jwt_signature <== jwt_signature;
    verificator.jwt <== JWT;
```

```
44 }
```

8.2 Signature Algorithms

Let's explore the detail of the signature algorithms that will be used in the protocol. The signature algorithms in the application frontend must match the ones demonstrated by the zk circuits. Otherwise, the zk proof verification will fail, and funds will not be unlocked from the account.

8.2.1 Google signature

For this instance of the protocol, we will use Google as our authenticator and OIDP. The Google signature is used to sign the JWT and thus guarantee its authenticity. The algorithm used is RS256 with the SHA256 hash function. It's an asymmetric algorithm that uses a pair of RSA private and public keys.

8.2.2 Cardano signature

The Cardano signature is used in the context of signing transactions. In zkLogin, the transactions must be signed with the `eph_sk` that corresponds with the `eph_pk` that will take part in the redeemer. The algorithm used in this case is **Ed25519**, and it's the standard algorithm for signing transactions in Cardano.

8.3 Hash Functions

The zkLogin protocol uses hash functions to compute the `nonce` and the `zkLoginId`. The algorithm used is the PoseidonBN254 hash function. Poseidon is a function that is called zk-friendly because it doesn't use too much bitwise arithmetic, which is usually expensive to represent in arithmetic equations. The Poseidon version used in the circuits must match exactly the one in the offchain components, or the verification will fail.

8.4 Why Groth16?

The selection of **Groth16** as the primary proving system for the zkLogin protocol on Cardano, over alternatives like Plonk and systems based on FRI (e.g., Plonky2, Risc0), is driven by two critical factors: its efficiency with Cardano's existing primitives and its compliance with transaction size limits.

The choice of Groth16 is highly synergistic with the cryptographic capabilities currently available in PlutusTx, Cardano's smart contract language. Cardano introduced built-ins for elliptic curve operations and pairings over **BLS12-381** specifically to reduce the cost of on-chain verification. Groth16, which utilizes the KZG polynomial commitment scheme over the BLS12-381 curve, is perfectly suited to leverage these primitives. The complexity of the Groth16 verifier is relatively low and is independent of the circuit size (apart from the

number of public inputs). Verification involves fast operations supported directly by Plutus built-ins, such as four Miller loops and a final verification step, which is key to maintaining low computational costs and fast execution times on the blockchain.

The defining factor that eliminates most other proving systems, particularly those based on **FRI** (such as Plonky2, Plonky3, Risc0, and others), is the stringent transaction size limit imposed by the Cardano network. A Groth16 proof is exceptionally compact: a compressed proof over BLS12-381 requires only **192 bytes**. This is far below the current maximum allowed transaction size on Cardano, which is set to `16,384 bytes (16 KB)`. In contrast, FRI-based proofs require a large number of Merkle authentication paths, meaning they inherently result in significantly larger proof sizes, often starting around 17 KB even for small circuits and reaching 500 KB or more for useful programs. Because FRI-based systems would easily exceed the 16 KB transaction limit, Groth16's minimal proof size makes it one of the only viable systems for on-chain verification in this environment.

9 Difference with previous implementations

This protocol was implemented, tested and used in a productive environment in the SUI blockchain. However, the implementation differs in one key aspect. SUI blockchain has zkLogin embedded in its core protocol, allowing the `zkLoginId` to be the address itself. To do this, the SUI protocol has to include the zk proof validation in the nodes natively.

In our cardano implementation, we had to find a workaround to this, since we do not intend to change the core Cardano protocol to support zkLogin. Instead, an easier way is to embed the `zkLoginId` in an Aiken validator, use the validator's hash as the user address and use the Aiken validator to check the zk proof.

10 Future Contributions

The present implementation of zkLogin for Cardano establishes the fundamental components of the protocol, enabling an end-to-end authentication flow based on OpenID credentials, ephemeral session keys, salts, and Groth16 zero-knowledge proofs. However, this work also opens several avenues for expansion and refinement. This section outlines future contributions that can strengthen interoperability, security, performance, and ecosystem adoption.

10.1 Multi-Provider Support

The current system integrates Google as the initial OpenID Provider due to its mature infrastructure and reliable OIDC implementation. A key future contribution is supporting additional providers such as Meta, Apple, X, Microsoft, GitHub, and enterprise identity platforms.

This requires handling:

- different JWT claim formats,
- multiple signature algorithms (ES256, RS256, PS256)
- distinct JWK rotation mechanisms

Extending the protocol to multiple providers is essential to broad adoption and a fully trustless, interoperable authentication ecosystem.

10.2 Multi-Device and Recovery Mechanisms

Since zkLogin replaces long-term private keys with salts and OIDC authentication, account persistence depends on the user's ability to:

1. authenticate to the same OpenID account
2. recover or re-access their salt.

Future work includes:

- salt recovery flows
- device migration strategies
- optional multi-device key-sharing schemes
- secure storage in device hardware (TPM, Secure Enclave)

These improvements aim to avoid single points of failure while maintaining privacy guarantees.

10.3 Handling OIDC Public Key Rotation

OpenID Providers periodically rotate their public keys, and long-lived validators must maintain compatibility with these rotations. Future work should explore:

- automatic caching of JWK sets with expiration policies
- circuit-level support for multiple accepted provider keys

Correct handling of key rotation is critical for maintaining protocol security in production environments.

10.4 Preserving Staking Rewards in zkLogin-based Addresses

In the current zkLogin design, user identities are bound to script-based payment credentials, which could difficult participation in Cardano's staking mechanism. In this direction, future improvements should explore mechanisms that allow zkLogin-controlled funds to accumulate and withdraw rewards without compromising the non-custodial and privacy-preserving properties of the protocol. Possible directions include:

- The use of hybrid addresses combining a zkLogin-based payment script with a conventional stake key
- Controlled withdrawal of rewards via stake scripts enforcing the same identity constraints as the zkLogin validator

Enabling staking compatibility would significantly improve capital efficiency and user experience, while preserving the security guarantees of zkLogin-based authentication.