b
UNIVERSITÄT
BERN

Implementing Privacy-Preserving Authentication Flow for U2SSO with OpenID Self-Provider

Anonymous Self-Credentials for OpenID Connect

Bachelor Thesis

Yanis Cedric Berger

from

Bern, Switzerland

Faculty of Science, University of Bern

9. September 2025

Prof. Christian Cachin
Mariarosaria Barbaraci
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Digital identity management faces fundamental privacy and centralization challenges in current federated authentication systems, where users lack control over their digital identities, leading to privacy vulnerabilities and dependence on centralized authorities. While OpenID Connect incorporates some privacy-preserving functionalities, it still relies on a centralized identity provider model. This architecture enables cross-service tracking, creates vendor lock-in, and leaves users vulnerable to data breaches and service discontinuation.

The OpenID Connect community has recognized these limitations and developed the Self-Issued OpenID Provider (SIOP) specification to enable users to act as their own identity providers. However, SIOP faces a fundamental challenge: how can service providers trust self-issued identities without a centralized authority? This trust problem has limited real-world adoption despite SIOP's advantages. The User-Issued Unlinkable Single Sign-On (U2SSO) protocol offers a novel approach that integrates Anonymous Self-Credentials to provide unlinkable pseudonyms across services.

This thesis addresses these limitations by implementing a privacy-preserving Self-Issued OpenID Provider that merges U2SSO's cryptographic primitives with the OpenID Connect standard. A key step during development was the identification and implementation of a deterministic transformation technique that enables derivation of standard ECDSA private keys from the U2SSO master identity, allowing authentication using standard OIDC tokens while preserving privacy and eliminating post-registration U2SSO dependencies. Through the development of a CGo wrapper library and proof-of-concept implementation, this work demonstrates the practical feasibility of integrating self-sovereign identity with industry-standard authentication protocols, achieving efficient ongoing authentication while accepting one-time registration overhead for enhanced privacy.

Contents

Chapter 1

Introduction

Digital identity management has become an important factor in today's digital space. Users increasingly rely on online services, from social media to banking. The traditional approach of maintaining separate credentials for each service has proven both cumbersome and insecure. Single Sign-On (SSO) and Federated Identity Management (FIM) solutions have emerged to address this problem, allowing users to authenticate once with a trusted Identity Provider (IdP) and gain access to multiple services.

However, current federated identity systems suffer from fundamental privacy and centralization issues. Traditional solutions, which typically implement OpenID Connect through centralized identity providers, place digital identities under the control of singular, centralized identity providers, such as Google, Meta, or Microsoft. This centralized architecture creates inherent privacy vulnerabilities through several mechanisms. Identity providers can correlate user activities across all connected services by linking authentication events to the same user account, creating comprehensive behavioral profiles. The OAuth 2.0 [12] authorization flows that underpin OpenID Connect [21] require services to redirect users through the identity provider, giving these providers visibility into which services users access and when. Furthermore, the identity provider controls the authentication tokens and can potentially access or modify the claims being shared with relying parties. This creates an asymmetric power relationship where users must trust identity providers with both their authentication credentials and their activity patterns across the entire federated ecosystem.

Economist Yanis Varoufakis argues in his book "Technofeudalism: What Killed Capitalism" [27] that digital platforms have created a new system, which he calls Technofeudalism, where a few technology giants extract rent from digital fiefdoms rather than competing in traditional markets. Digital identity represents one of the most critical examples of this phenomenon. When users authenticate through "Sign in with Google" or "Login with Facebook", they become digital serfs in these companies' identity fiefdoms.

This dependency relationship enables massive data extraction, something Varoufakis calls "cloud rent", as identity providers track user behavior across the web. The apparent convenience of SSO masks a fundamental transfer of power from users to platform owners. Users lose control over their digital identities while platforms gain unprecedented surveillance capabilities and market power.

Self-Sovereign Identity represents a shift toward user-controlled digital identity. Rather than relying on centralized identity providers, SSI systems enable users to create, manage, and control their own digital identities. The Self-Issued OpenID Provider (SIOP) specification [29] extends the widely adopted OpenID Connect [21] protocol to support self-sovereign identity, allowing users to act as their own identity providers. SIOP eliminates the need for traditional identity providers by enabling users to generate cryptographic proofs [10] of their identity directly from their devices. This approach preserves the familiar OpenID Connect authentication flow that service providers already understand while fundamentally changing the trust model from relying on centralized authorities to cryptographic verification. A Self-Issued OpenID Provider enables a potential pathway toward digital identity liberation, giving users the power to reclaim control and ownership over their digital identities without sacrificing the convenience of single sign-on.

However, implementing SIOP in practice raises significant challenges. How can service providers trust self-issued identities without a centralized authority? In what way can the system prevent Sybil attacks [6] where malicious users create multiple identities? How can users maintain privacy across different services while still enabling legitimate identity verification? These questions have limited the real-world adoption of SIOP despite its privacy and self-sovereign advantages.

This thesis addresses these challenges by implementing a practical SIOP solution using the User-Issued Unlinkable Single Sign-On (U2SSO) system developed by Alupotha et al [1]. The U2SSO protocol provides a cryptographic framework that enables unlinkable authentication across multiple services, leading to the preservation of users' anonymity and Sybil-resistance for relying parties. These two Properties are essential for practical identity systems but difficult to achieve simultaneously.

The core innovation of U2SSO lies in its use of Anonymous Self-Credentials (ASC) [1], which allow users to prove membership in a fixed set of identities without revealing which specific identity they own. This enables service providers to verify that users have legitimate credentials while preserving privacy across services through unlinkable pseudonyms.

By combining SIOP's standardized authentication flows with U2SSO's cryptographic privacy guarantees, this thesis demonstrates how self-sovereign identity can be made practical for real-world usage. The implementation follows the Self-Issued OpenID Provider v2 specification while incorporating U2SSO's privacy-preserving mechanisms, creating a bridge between cryptographic research and industry-standard authentication protocols.

A significant breakthrough emerged during the implementation process: the development of a deterministic key transformation technique that derives U2SSO service-specific secret keys as standard ECDSA private keys. This optimization enables authentication using standard OpenID Connect ID Tokens without requiring any U2SSO protocol elements.

Chapter 2

Background

This chapter provides the necessary background knowledge for understanding the technical foundations and the motivation behind integrating User-Issued Unlinkable Single Sign-On (U2SSO) within the OpenID Connect standard, by implementing a Self-Issued OpenID Provider proposed in the specification draft [29] by the OpenID Federation.

First, we introduce the fundamental concepts needed to understand current single sign-on implementations through OpenID Connect. After examining the current standard, the concept of a Self-Issued OpenID Provider is introduced, quickly examining the key differences as well as what it solves.

The chapter concludes with the cryptographic foundations of the U2SSO system, which are essential to understanding the implementation described in Chapter 3.

2.1 The Digital Identity Problem

Digital identity, the collection of attributes and credentials that uniquely identify users within online systems [26, p. 272], has become a cornerstone of the modern internet. Unlike physical identity verification, digital identity faces unique challenges: persistence across sessions, linkability across services, and dependence on centralized authorities for verification.

Current digital identity solutions suffer from fundamental privacy and centralization issues that compromise user autonomy. When users authenticate through “Sign in with Google” or similar services, they give away control of their digital identities to centralized providers who can track their activities across the web and unilaterally revoke access. Understanding these limitations is essential for understanding why privacy-preserving authentication systems like Self-Issued OpenID Providers are necessary.

2.1.1 Current Federated Identity Limitations and Centralization Concerns

Federated Identity Management (FIM) represents the dominant paradigm for web authentication, where multiple service providers form associations with established identity providers, enabling users to leverage credentials from a provider to access services across the entire federation [3]. While this approach offers convenience through single sign-on capabilities, it creates both technical limitations and broader socioeconomic concerns.

The most visible manifestation of this paradigm is social login, where users authenticate on third-party websites using credentials from established social identity providers. While social logins offer enhanced user convenience by eliminating the need to create distinct credentials for each service, they exemplify the fundamental problems of centralized federated identity [9].

Technical and Operational Limitations

Centralized Control: Users must trust centralized identity providers with both their authentication credentials and their activity patterns. These providers control access to users’ digital identities and can unilaterally revoke access or modify terms of service [16].

Cross-Service Tracking:	Identity providers can correlate user activities across all connected services by linking authentication events to the same user account. The OAuth 2.0 authorization flows require services to redirect users through the identity provider, giving these providers visibility into which services users access and when [16].
Vendor Lock-in:	Once users establish accounts with federated services through a particular identity provider, switching providers becomes costly and complex, creating dependency relationships that limit user choice [4].
Single Points of Failure:	The entire federated ecosystem depends on the continued operation and trustworthiness of centralized identity providers. Service outages or security breaches at these providers can affect access to numerous dependent services [14].

Economic and Power Structure Implications

These technical limitations reflect broader shifts in digital power structures. Economist Yanis Varoufakis argues that digital platforms have created a new system where technology giants extract rent from digital fiefdoms rather than competing in traditional markets [27]. Digital identity represents a critical example of this phenomenon.

When users authenticate through a centralized identity provider they become digital serfs in these companies' identity fiefdoms. This relationship enables what Varoufakis calls "cloud rent", the extraction of value through control of digital infrastructure rather than traditional market competition. Users lose control over their digital identities while platforms accumulate comprehensive behavioral profiles across the web, creating asymmetric power relationships that extend far beyond simple authentication.

This concentration of identity control creates systemic vulnerabilities not just for individual privacy, but for the competitive dynamics of the entire internet economy. Smaller service providers become dependent on identity giants, while users face increasingly limited choices and diminished autonomy over their digital presence.

Privacy and Centralization Concerns

Beyond operational limitations, centralized federated identity creates fundamental privacy vulnerabilities that compromise user autonomy and data protection.

Comprehensive Data Aggregation: Identity providers collect far more information than necessary for authentication, including tracking users' website access patterns [15], device fingerprints [23], location data [5], and behavioral patterns. This data aggregation extends beyond authentication events to include email content, search history, and application usage when users employ integrated services [17].

Informed Consent Challenges: The complexity of federated identity relationships makes meaningful user consent nearly impossible. Users cannot reasonably understand the full scope of data sharing between identity providers and third-party services, violating principles of informed consent central to privacy regulations [28].

Jurisdictional and Regulatory Complications: Users in privacy-protective jurisdictions may find their data processed by identity providers subject to different legal frameworks. This creates regulatory arbitrage where privacy protections can be circumvented through strategic identity provider selection [20].

Profile Permanence and Right to be Forgotten: Centralized identity systems create persistent digital profiles that are difficult to modify or delete. Even when users attempt to exercise rights like data deletion, the interconnected nature of federated systems makes complete data removal practically impossible [19].

2.1.2 The Need for Self-Sovereign Solutions

Self-Sovereign Identity (SSI) represents a paradigm shift toward user-controlled digital identity. Rather than relying on centralized identity providers, SSI systems enable users to create, manage, and control their own digital identities through cryptographic mechanisms.

The Self-Issued OpenID Provider (SIOP) specification [29] extends the widely adopted OpenID Connect [21] protocol to support self-sovereign identity, allowing users to act as their own identity providers. This approach preserves the familiar authentication flows that service providers already understand while fundamentally changing the trust model from relying on centralized authorities to cryptographic verification.

However, implementing SIOP raises significant challenges: How can service providers trust self-issued identities without a centralized authority? How can the system prevent Sybil attacks while preserving user privacy? These questions are difficult to face and have limited real-world adoption, despite SIOP's advantages.

2.2 OpenID Connect

This section provides an overview of the relevant protocols. Unless otherwise cited, all information is drawn from the OpenID Core specification [21] and the Self-Issued OpenID Provider draft [29].

OpenID Connect is an additional layer built on top of the OAuth 2.0 protocol. This extra layer enables authentication, meaning that Clients may verify the identity of an end user provided by an Authorization Server. Additionally, Clients MAY obtain basic profile information about the end user. Information about the authentication performed by the OpenID Provider is returned in the form of an ID Token, a JSON Web Token (JWT) [21].

Authentication vs Authorization

Understanding the distinction between Authentication (AuthN) and Authorization (AuthZ) is essential for analyzing OpenID Connect's privacy implications and the role of Self-Issued OpenID Providers.

Authentication in digital identity systems refers to the process of verifying a claimed identity, answering the question "Who is this user?" When Alice logs into a service using Google as her identity provider, the authentication process confirms that Alice is indeed the owner of her Google account.

The OpenID Connect specification defines authentication as the "process used to achieve sufficient confidence in the binding between the Entity and the presented Identity" [21], emphasizing the trust establishment aspect of the authorization process.

Authorization determines what resources or actions an authenticated user may access, answering "What can this user do?" This includes decisions about which user attributes (email, name, profile) should be shared with the requesting service and what permissions the user grants.

This distinction has important privacy implications: while authentication establishes user identity, the authorization process determines how much personal information flows between identity providers and relying parties. In traditional federated systems, users have limited control over this authorization process, contributing to the privacy problems discussed in Subsection 2.1.1.

Terminology Note: Following OpenID Connect conventions, this thesis uses "Authorization Request" to refer to the initial request from relying party to identity provider, even when the primary purpose is authentication. This reflects the standard protocol terminology where the OAuth 2.0 authorization framework underlies the authentication layer.

Background (OAuth 2.0)

OpenID Connect builds upon OAuth 2.0, an authorization framework that enables third-party applications to obtain limited access to HTTP services through a redirection-based flow [12]. While OAuth 2.0 was designed for authorization rather than authentication, its redirection mechanisms form the foundation for OpenID Connect's identity layer, which create the privacy vulnerabilities that centralized federated identity systems exhibit.

2.2.1 Core Protocol Flow and Privacy Implications

OpenID Connect involves three actors: the OpenID Provider (OP) that authenticates users, the Relying Party (RP) that requires authentication, and the end user being authenticated. The protocol follows these steps, each creating opportunities for surveillance and tracking:

1. **Authorization Request**: The RP redirects the user to the OP, revealing to the OP which service the user is accessing and when.
2. **Authentication and Authorization**: The OP authenticates the user and can correlate this authentication with the requesting service.
3. **Token Response**: The OP issues an ID Token containing user identity information, establishing ongoing dependency relationships.
4. **UserInfo Request**: The RP may request additional user claims from the OP, enabling further data collection.
5. **Claims Response**: The OP provides user attributes, completing the privacy-compromising information flow.

The protocol flow is illustrated in Figure 2.1, provided by the OpenID specification.

```
+--------+                                   +--------+
|        |                                   |        |
|        |---------(1) AuthN Request-------->|        |
|        |                                   |        |
|        |  +--------+                       |        |
|        |  |        |                       |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                       |        |
|   RP   |  |        |                       |   OP   |
|        |  +--------+                       |        |
|        |                                   |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                   |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                   |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                   |        |
+--------+                                   +--------+
```

Figure 2.1. OpenID Connect Protocol flow.

It is worth noting that steps (1), (2) and (3) constitute the Authentication part of the exchange, whereas steps (4) and (5) comprise the Authorization part.

Authorization Request

In OpenID Connect, the Relying Party (RP) sends an *Authorization Request* to the OpenID Provider (OP) to initiate authentication. For the authentication exchange to succeed, the request MUST contain certain parameters as defined in the OpenID Connect Core specification [21], listed in Table 2.1.

Table 2.1. Baseline OIDC Authorization Request Parameters

Parameter	**Requirement**	**Description / Notes**
`scope`	REQUIRED	Must include the value `openid` to signal an OpenID Connect request.
`response_type`	REQUIRED	OAuth 2.0 response type determining the flow used (e.g., `id_token`, `code`).
`client_id`	REQUIRED	OAuth 2.0 client identifier issued to the RP by the OP.
`redirect_uri`	REQUIRED	URI to which the OP will send the authentication response.
`state`	RECOMMENDED	Opaque value to maintain state between request and callback; mitigates CSRF.
`nonce`	OPTIONAL	Binds the client session to the ID Token to prevent replay attacks.

Upon receiving the request, the Identity Provider initiates the authentication process for the user, typically by requesting the credentials associated with their account.

Authorization Response

After successful user authentication, the OpenID Provider redirects the user back to the relying party's `redirect_uri` with the authentication response. The response contains either an authorization code (Authorization Code Flow) or an ID Token directly (Implicit Flow), along with the state parameter to prevent CSRF attacks.

In case of successful authentication, the response includes:

- code: Authorization code for token exchange (Authorization Code Flow)
- id_token: JWT containing authentication claims (Implicit Flow)
- state: value to mitigate CSRF attacks (all Flows)

Error responses include an error parameter with standardized error codes such as 'access_denied' or 'invalid_request', allowing the relying party to handle authentication failures appropriately.

2.2.2 ID Token and claims

ID Token

An ID Token is a JSON Web Token (JWT). JWTs are compact, URL-safe means of representing claims that are to be transferred between two parties. It is intended for space-constrained environments. A JWT is a String, representing a set of claims as a JSON object encoded in a JWKS, enabling the object to be secured with a digital signature.

A JWT consists of three Base64URL-encoded parts, separated by dots:

1. **Header.** A JSON object indicating the token type and signing algorithm, for example:
2. **Payload.** A JSON object of claims.
3. **Signature.** A digital signature over the header and payload, produced by the OP's private key.

Example JWT:

Encoded Token:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkFsaWNlIn0.
EkN-DOsnsuRjRO6BxXemmJDm3HbxrbRzXglbN2S4sOko
```

Decoded Components:

Header: {`"alg":"RS256","typ":"JWT"`}
Payload: {`"sub":"1234567890","name":"Alice",`}

Claim

A claim is a piece of information asserted about a subject. Claims are represented as a pair, consisting of a claim name and a claim value.

For the Authentication flow defined by OpenID Connect, some claims are REQUIRED:

Table 2.2. Standard Required Claims in OIDC ID Token

Claim	Description
`iss`	Issuer identifier.
`sub`	Subject identifier.
`exp`	Expiration time.
`iat`	Issuance time.
`nonce`	Binds the client session to the ID Token; mitigates replay attacks.

While some claims are REQUIRED, adding more (even custom) claims is OPTIONAL. However, all parties involved in the Authentication exchange MUST agree on the meaning of these claims. This means that code must be in place to handle these additional claims. All claims not understood by the relying party MUST be ignored [21].

2.3 Self-Issued OpenID Provider

2.3.1 Motivation and Scope

Traditional federated identity systems create privacy and centralization concerns by allowing identity providers to track user activities across services and by introducing single points of failure. The Self-Issued OpenID Provider (SIOP) specification addresses these limitations by enabling users to act as their own identity providers, generating identity tokens directly from their devices.

A SIOP aims to achieve the following:

- Eliminate identity provider tracking across services.
- Return ownership of digital identity to users.
- Remove single points of failure.
- Maintain interoperability with existing OpenID Connect infrastructure.

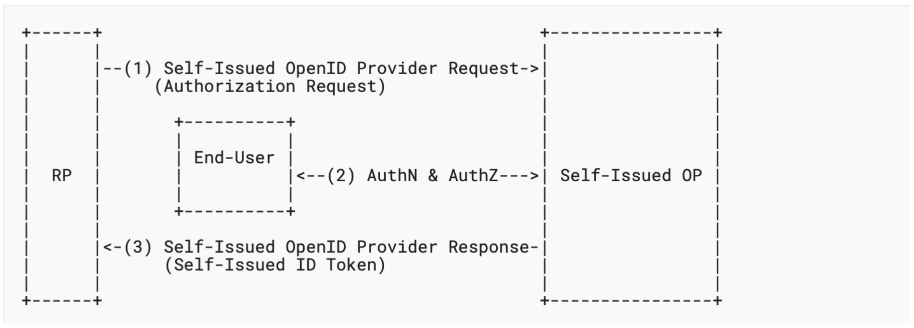
2.3.2 Core Authentication Flow

The SIOP authentication flow follows the same high-level steps as the traditional OP model, but without a centralized identity provider. The RP sends an *Authorization Request* to the SIOP, and the SIOP responds with an *ID Token* containing authentication claims. In the SIOP model:

- The `iss` (issuer) and `sub` (subject) claims **must be identical**, since the subject is also the issuer.

- The `sub` claim must be a *cryptographically verifiable identifier*.

Figure 2.2 illustrates the protocol flow as defined in the SIOP specification.



```
+------+                                           +----------------+
|      |                                           |                |
|      |--(1) Self-Issued OpenID Provider Request->|                |
|      |     (Authorization Request)               |                |
|      |                                           |                |
|      |       +----------+                        |                |
|      |       |          |                        |                |
|      |       | End-User |                        |                |
|  RP  |       |          |<--(2) AuthN & AuthZ--->| Self-Issued OP |
|      |       |          |                        |                |
|      |       +----------+                        |                |
|      |                                           |                |
|      |<-(3) Self-Issued OpenID Provider Response-|                |
|      |      (Self-Issued ID Token)               |                |
|      |                                           |                |
+------+                                           +----------------+
```

Figure 2.2. OpenID Connect SIOP model Protocol flow.

2.3.3 Subject Syntax Types

A *Subject Syntax Type* defines the format of the identifier used in the `sub` claim within ID Tokens issued by a SIOP. Two common types are supported:

- **JWK Thumbprint Subject Syntax Type:** The subject identifier is computed as the thumbprint of the JSON Web Key (JWK) used to sign the ID Token [13]. This allows immediate signature verification without external key lookup.

- **Decentralized Identifier (DID) Subject Syntax Type:** Uses W3C DIDs, supporting advanced features such as key rotation and service endpoints [24] (beyond the scope of this thesis).

In this thesis, the **JWK Thumbprint** syntax type is used for its simplicity and compatibility with U2SSO cryptographic operations.

2.3.4 ID Token Structure in SIOP

While the basic ID Token structure in SIOP is similar to traditional OIDC, it has two key differences:

1. The `iss` and `sub` claims **must be identical**.

2. For the JWK Thumbprint syntax type, the ID Token **must** contain an additional `sub_jwk` claim with the public key used to verify the signature.

2.4 Anonymous Self Credentials and U2SSO

User-issued Unlinkable Single Sign-On (U2SSO) relies on a cryptographic construction called Anonymous Self-Credentials (ASC) to achieve privacy-preserving authentication with Sybil resistance. ASC enables users to prove membership in an authorized set of identities without revealing which specific identity they possess, while simultaneously preventing the creation of multiple accounts with the same service provider.

The ASC construction builds upon several cryptographic primitives that work together to provide these seemingly contradictory properties. At its core, the system uses commitment schemes to hide information while maintaining the ability to prove statements about committed values. Zero-knowledge arguments then allow users to demonstrate knowledge of valid credentials without revealing the credentials themselves. Finally, the ASC framework combines these primitives into a cohesive protocol that supports both anonymous registration and efficient authentication.

This section presents the key cryptographic building blocks underlying the U2SSO protocol. Understanding these primitives and their interplay is essential for analyzing the security and privacy guarantees of the U2SSO system.

2.4.1 Pedersen Commitments

Pedersen commitments [18] form a cornerstone of the privacy-preserving mechanism in the U2SSO system, which enables users to commit to secret values without revealing them and maintaining the ability to prove properties about the committed values.

Definition

Let $\mathbb{G}$ be a cyclic group of prime order q with generators g and h, where the discrete logarithm of h with respect to g is unknown. A Pedersen commitment to a value $m \in \mathbb{Z}_q$ with randomness $r \in \mathbb{Z}_q$ is defined as:

$$C(m, r) = g^m \cdot h^r \tag{2.1}$$

Properties

Pedersen commitments satisfy several cryptographic properties:

Binding Property: Given the computational difficulty of the discrete logarithm problem, it is computationally infeasible for an adversary to find two different pairs (m, r) and (m', r') such that $C(m, r) = C(m', r')$ with non-negligible probability[18].

Hiding Property: The commitment $C(m, r)$ computationally hides the value m. Specifically, for any two messages $m_0, m_1 \in \mathbb{Z}_q$, the distributions $\{C(m_0, r) : r \leftarrow \mathbb{Z}_q\}$ and $\{C(m_1, r) : r \leftarrow \mathbb{Z}_q\}$ are computationally indistinguishable[18].

Homomorphic Property: Pedersen commitments support additive homomorphism:

$$C(m_1, r_1) \cdot C(m_2, r_2) = C(m_1 + m_2, r_1 + r_2) \tag{2.2}$$

Application in U2SSO

In the U2SSO system, Pedersen commitments enable users to commit to their master public keys while participating in zero-knowledge proofs. The homomorphic property is particularly useful for constructing proofs of linear relations between committed values without revealing the underlying secrets. The commitment scheme is instantiated over the secp256k1 elliptic curve, where the group $\mathbb{G}$ is the cyclic group of points on the curve. The generators g and h are chosen such that no party knows the discrete logarithm relationship between them, ensuring the security of the commitment scheme[1].

Multi-Value Commitments

In the CRS-ASC construction, multi-value commitments are employed to allow a single commitment to hide and bind multiple values using a single blinding key. A multi-value commitment for L values $[v_i]_{i=1}^{L}$ with blinding key k is denoted as:

$$C = \text{Com}_{\text{crs}}^{\mathbb{Z}_q, S}([v_i]_{i=1}^{L}, k) \tag{2.3}$$

where crs is the common reference string, $\mathbb{Z}_q$ is the value space, and S is the blinding key space.

2.4.2 Zero-Knowledge Arguments

Zero-knowledge arguments allow a prover to convince a verifier that a statement is true without revealing any additional information beyond the validity of the statement itself [10].

System Components

A zero-knowledge argument system operates through three distinct algorithms:

- **Setup Algorithm**: Establishes the cryptographic parameters and public information required for the proof system, including the common reference string and security parameters.
- **Prover Algorithm**: Takes as input a statement and a witness, and generates a cryptographic proof that demonstrates the truth of the statement without revealing the witness.
- **Verifier Algorithm**: Takes as input a statement and a proof, and outputs whether the proof constitutes valid evidence for the statement's truth.

Security Requirements

Zero-knowledge argument systems must satisfy three fundamental security properties:

Completeness:	An honest prover who knows a valid witness can always convince an honest verifier of a true statement with overwhelming probability.
Soundness:	No computationally bounded dishonest prover can convince an honest verifier of a false statement except with negligible probability.
Zero-Knowledge:	The verifier learns no information about the prover's witness beyond the fact that the statement is true.

Application in U2SSO

In the U2SSO system, cryptographic proofs serve two distinct purposes:

1. **Registration**: Zero-knowledge arguments prove membership in the anonymity set without revealing the specific identity, while preventing multiple registrations through nullifier uniqueness.
2. **Authentication**: After registration, users authenticate using Schnorr signatures with their registered service-specific secret keys. These signatures provide efficient proof of ownership of the registered pseudonym without requiring complex zero-knowledge proofs for each login.

2.4.3 Anonymous Self-Credentials (ASC)

Anonymous Self-Credentials represent the core cryptographic construction enabling privacy-preserving authentication in U2SSO.

Construction Overview (CRS-ASC)

The CRS-ASC system utilizes multi-value Pedersen commitments as master identities (master public keys). Each master identity commits to deterministically derived nullifiers for each service provider, with the master secret key serving as the blinding factor.

In CRS-ASC, the master secret key enables deterministic derivation of service-specific nullifiers, while the master identity is a Pedersen commitment to these nullifiers using the master secret key as the blinding factor.

An ASC system consists of:

- N provers, each possessing a master credential (Φ_j, sk_j) where Φ_j is the master public key and sk_j is the master secret key
- L verifiers, each with a unique verifier identifier $v_l \in V$
- Anonymity set $\Lambda := [\Phi_i]_{i=1}^{N}$ containing all master identities

Master Public Key Generation

For each user j, the master identity is computed as:

$$\Phi_j = g_0^{sk_j} \cdot \prod_{i=1}^{L} g_i^{\text{nul}_{j,i}}$$

where:

- sk_j is the master secret key (blinding factor)
- $\text{nul}_{j,i}$ is the deterministically derived nullifier for service v_i
- $g_0, g_1, \ldots, g_L$ are independent generators from the common reference string

Registration Protocol

To register with service provider l, a user proves in zero-knowledge:

"I know the secret key corresponding to one of the master identities in the anonymity set $\{\Phi_1, \ldots, \Phi_n\}$*, and here is the nullifier* nul_l *that was committed within my master identity for service* l*."*

The binding property of Pedersen commitments ensures that the user must reveal the same nullifier for any subsequent interaction with the same service provider, preventing multiple registrations (Sybil resistance).

2.4.4 Security Analysis

The security properties of the CRS-ASC construction have been formally proven in the original work by Alupotha et al. [1]. Specifically, they demonstrate that CRS-ASC satisfies the following security properties:

Correctness:	Honest provers can always generate valid master credentials and successfully register pseudonyms with service providers.
Robustness:	Honest provers can generate valid proofs even when the anonymity set contains maliciously generated master identities.
Unforgeability:	Adversaries without knowledge of valid secret keys cannot generate valid proofs for registration.
Sybil Resistance:	Each master identity can generate at most one valid nullifier per service provider, preventing multiple registrations.
Anonymity:	Registration proofs do not reveal which specific master identity was used, providing privacy within the anonymity set.
Multi-Verifier Unlinkability:	Registrations with different service providers cannot be linked to the same user, even if service providers collude.

For detailed proofs and security reductions, the reader is encouraged to refer to [1].

Chapter 3

Anonymous Self Credentials for OpenID Connect

This chapter describes a complete proof-of-concept implementation demonstrating the practical feasibility of privacy-preserving authentication through a Self-Issued OpenID Provider. Building upon the cryptographic system U2SSO proposed by Alupotha et al. [1], a working system was developed that integrates U2SSO with the OpenID Connect protocol to enable User-Issued Unlinkable Single Sign-On.

Rather than presenting abstract protocols, this chapter follows a user's complete journey through the system, from initial setup through registration and ongoing authentication. By tracing each step of Alice's interaction with the system, we demonstrate how privacy-preserving authentication can be achieved while maintaining compatibility with existing OpenID Connect infrastructure.

The implementation follows a layered approach, beginning with the development of the "oidcu2sso" library, a CGo wrapper that bridges U2SSO's C/C++ cryptographic library with Go-based OpenID Connect components. This library handles all cryptographic calculations necessary to implement Common Reference String Anonymous Self Credentials (CRS-ASC) while maintaining full compliance with the OpenID Connect protocol.

The complete system consists of three core components: a Self-Issued OpenID Provider that generates privacy-preserving ID Tokens, a relying party that supports both traditional and self-issued authentication flows, and a blockchain-based Identity Registry for decentralized credential management. Certain elements are intentionally simplified to keep the project within the scope of a bachelor thesis, including network transport security, persistent data storage, and production-grade error handling.

The chapter progresses through Alice's complete experience: establishing her master identity, registering with a new service using zero-knowledge proofs, and authenticating efficiently in subsequent visits.

3.1 System Architecture Overview

The OpenID Connect User-Issued Unlinkable Single Sign-On library implements a layered architecture that bridges U2SSO cryptographic primitives with industry-standard authentication protocols. This implementation enables three distinct user journeys: initial setup where users create their master identity, registration flows where users prove membership in the anonymity set to register with new services, and authentication flows where returning users prove ownership of their service-specific credentials.

The system consists of three primary architectural components, as illustrated in Figure 3.1, which work together to support these user journeys seamlessly.

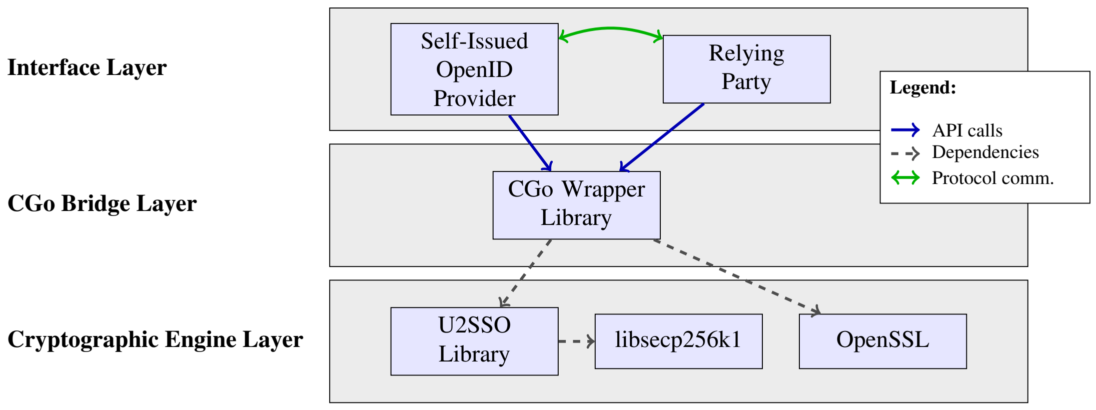


Figure 3.1. System architecture showing component relationships and data flow.

3.1.1 System Components

From Alice's perspective, the system appears as a familiar OpenID Connect flow, she clicks "Sign up with SIOP" and receives an authentication token. Behind this familiar interface, three architectural layers work together to preserve her privacy:

Interface Layer:	Maintains OpenID Connect compatibility, ensuring that services like Service X require no modifications to support privacy-preserving authentication.
CGo Bridge Layer:	Safely integrates complex cryptographic operations with standard web protocols, handling the translation between Alice's high-level authentication requests and low-level zero-knowledge proof generation.
Cryptographic Engine Layer:	Implements the U2SSO protocol that enables Alice to prove her legitimacy without revealing her identity.

This layered approach ensures that while Alice's experience remains familiar, the underlying system provides strong privacy guarantees that traditional federated identity cannot offer.

3.1.2 Protocol Flow Overview

Alice's interaction with the system follows three distinct phases, each building upon the previous to create a complete privacy-preserving identity solution:

Phase 1 - Identity Establishment: Alice creates her master cryptographic identity and joins the anonymity set, establishing her credentials while remaining anonymous among all other users. Alice's U2SIOP then stores the anonymity set for proof generation.

Phase 2 - Service Registration: When Alice encounters a new service, she proves her membership in the legitimate user set without revealing which specific user she is, enabling privacy-preserving account creation. Alice's U2SIOP generates zero-knowledge proofs using the anonymity set cached from Phase 1, while the relying party independently retrieves the current anonymity set from the blockchain-based identity registry to verify Alice's proof.

Phase 3 - Streamlined Authentication: For subsequent visits, Alice authenticates efficiently using service-specific credentials derived from her master identity. This eliminates the need for anonymity set communication, as shown in Figure 3.3.

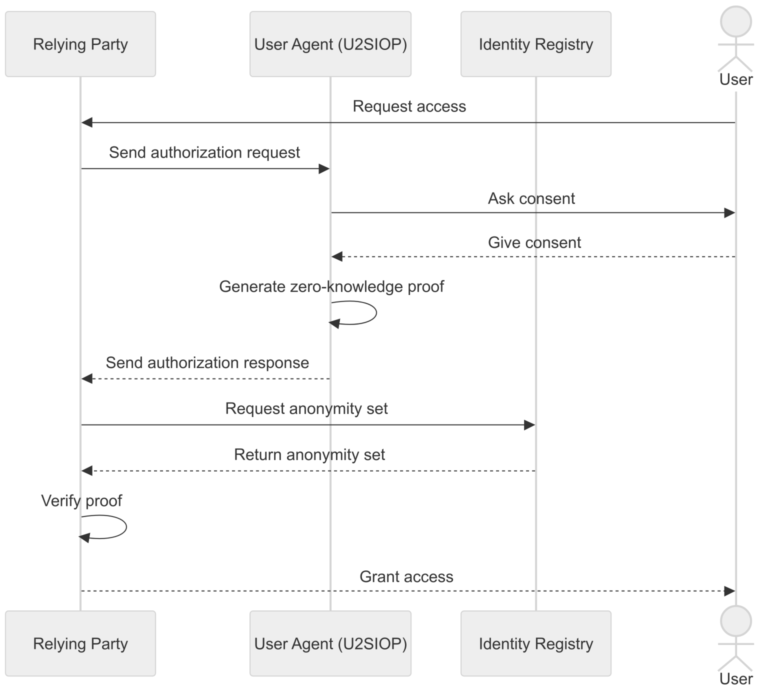


Figure 3.2. Abstract Registration Protocol Flow

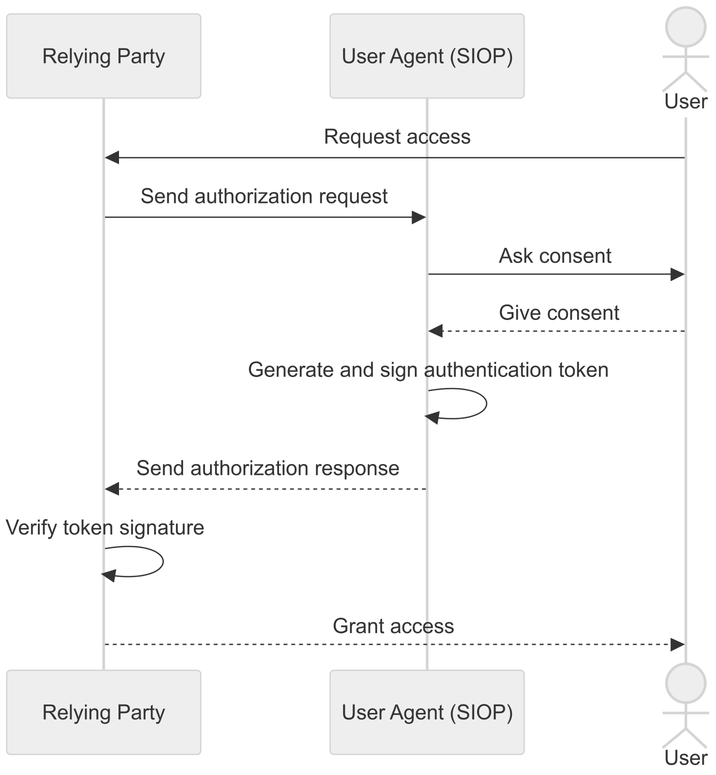


Figure 3.3. Abstract Authentication Protocol Flow

3.1.3 Key Architectural Decisions

Memory Management: The library employs a hybrid approach combining explicit C memory management with Go's garbage collection. This design ensures optimal performance for cryptographic operations while maintaining memory safety at the application level.

Cryptographic Context Isolation: Each cryptographic operation maintains isolated `secp256k1` contexts, ensuring thread safety and preventing cross-contamination between operations. This is particularly important given the sensitive nature of master secret keys and service-specific derivations.

Flow-Specific Optimization: The architecture distinguishes between registration and authentication flows at all levels, from proof generation algorithms to token structures, enabling significant performance optimizations for common authentication scenarios.

These architectural decisions enable Alice to reclaim control over her digital identity while maintaining compatibility with existing infrastructure. The following sections trace her complete journey, beginning with the establishment of her master identity.

3.2 Initial Setup and Master Identity Creation

Before Alice can register with any service or authenticate anywhere, she must first establish her master identity. This one-time setup process creates the cryptographic foundation for all subsequent interactions and represents the user's entry point into the privacy-preserving authentication ecosystem.

Unlike traditional systems where Alice would create accounts with centralized providers like Google or Facebook, this setup makes Alice her own identity provider. The process involves three steps: generating cryptographic credentials that will serve as her identity foundation, ensuring she can recover these credentials if needed, and publishing her public identity to join the global anonymity set.

3.2.1 Master Secret Key Generation and Storage

Alice's Self-Issued Unlinkable OpenID Provider (U2SIOP) generates a cryptographically secure 32-byte master secret key that serves as the foundation for all her future authentication activities. This single key represents Alice's digital identity, compromise of this key would allow an attacker to impersonate Alice across all registered services, making its secure generation and storage paramount.

A c ryptographically secure random number generation provides sufficient entropy while remaining computationally efficient for client-side generation (see Appendix A, Listing A.1). Unlike password-based systems where Alice might choose weak credentials, this approach guarantees cryptographic strength while removing the burden of secure credential selection from Alice.

The generated key is stored locally on Alice's device under her complete control, eliminating the need to trust external parties with her authentication secrets. This storage model addresses a fundamental vulnerability of centralized identity systems where credential compromise at the provider affects millions of users simultaneously.

3.2.2 Credential Recovery

For subsequent uses of her U2SIOP, Alice needs to retrieve her previously generated master secret key. The implementation provides secure retrieval of the stored 32-byte passkey from persistent storage (see Appendix A, Listing A.2). This enables secure reuse of Alice's cryptographic identity whenever it is needed for registration or authentication operations. The function returns both the 32-byte master secret key and a boolean success indicator. The deterministic nature of the system means that Alice's master secret key alone is sufficient to recreate all her service-specific identities.

This approach trades the convenience of centralized account recovery for the security and privacy benefits of self-sovereign identity. Alice gains complete control over her digital identity but must take responsibility for credential management, a fundamental characteristic of self-sovereign systems.

3.2.3 Master Identity Registration

With her master secret key established, Alice's U2SIOP generates the corresponding master public key that will represent her in the global anonymity set. This step transforms Alice from having local cryptographic credentials to becoming part of the global privacy-preserving authentication ecosystem.

The master public key generation represents the most cryptographically sophisticated aspect of the setup process. The implementation integrates current service identifiers (topics) and cryptographically binds unique nullifiers for each registered service to Alice's master identity. This binding ensures that Alice can prove her legitimacy during registration while preventing her from creating multiple accounts with the same service, thus achieving privacy and Sybil resistance.

The cryptographic process involves three essential steps:

1. **Topic Integration**: The current service identifier list establishes which services participate in the ecosystem

2. **Nullifier Binding**: Unique nullifiers for each service identifier are cryptographically bound to Alice's master secret key

3. **Sybil-Resistant Key Generation**: The master public key is generated with embedded nullifier commitments, ensuring single-use per service while preserving anonymity

This nullifier binding mechanism prevents users from creating multiple accounts with the same service while maintaining their position within the legitimate anonymity set.

The complete implementation of this process is detailed in Appendix A, Listing A.3.

Joining the Anonymity Set

Alice's master public key must be published to the blockchain-based identity registry to join the anonymity set. This publication step requires a small gas fee but establishes Alice's position among all legitimate users in the system. The implementation handles the blockchain interaction through smart contract integration detailed in Section 3.6.

The blockchain registry serves two critical functions: it provides a decentralized, tamper-resistant record of all legitimate identities, and it enables service providers to retrieve the current anonymity set for verification purposes. Alice's privacy depends on the size of this anonymity set, larger sets provide stronger anonymity guarantees.

3.3 Registration Flow

With her master identity established, Alice is ready to register with services in the privacy-preserving authentication ecosystem. The registration flow enables Alice to prove she is a legitimate user without revealing her specific identity, while simultaneously preventing her from creating multiple accounts with the same service.

This section traces through Alice's first registration with a new service, which we'll call "Service X". Unlike traditional OAuth flows that rely on centralized identity providers, Alice must now prove her membership in the anonymity set through zero-knowledge cryptographic proofs. The registration involves four key phases that work together to establish Alice's account while preserving her privacy.

3.3.1 User Initiates Registration

When Alice clicks "Sign Up with SIOP" on Service X, she triggers a process that fundamentally differs from traditional OpenID Connect flows. Instead of redirecting to a centralized identity provider like Google, Service X must prepare for direct cryptographic verification of Alice's credentials.

Service X generates a cryptographic challenge that will bind Alice's registration proof to this specific session, preventing replay attacks. The authorization request extends standard OpenID Connect parameters with privacy-preserving requirements: a *proof type* indicating zero-knowledge verification is needed, the cryptographic *challenge*, and a *service identifier*. Otherwise the request parameters are as described in Table 2.1

This establishes the cryptographic context for Alice's subsequent proof generation, ensuring that her registration is bound to this specific session while maintaining compatibility with existing OpenID Connect infrastructure.

Alice's browser is then redirected to her U2SIOP with this authorization request, as shown in Figure 3.4.

```
    /auth?challenge=f04a0db2e9c9ca153e70e0c9bfbcb7bab62ece31dddbabf44a566e9ffa5b9776
    &client_id=6d7e78af064c86eb9b9cb1c3611c9ab60a2f9317e3891891ef31770939f78ef8
    &nonce=GqAHUhVCixburkDccj2eQWcfA8QjkoAj4GjQXkSImhA
    &proof_type=registration
    &redirect_uri=http://localhost:8081/auth/siop/callback
    &response_type=id_token
    &scope=openid
    &service_name=6d7e78af064c86eb9b9cb1c3611c9ab60a2f9317e3891891ef31770939f78ef8
    &state=ScxIaXS14SAA1wZ4wOKBh2UzZyX4Uucb6rpDeuTtoNY
```

Figure 3.4. Registration Authorization Request

3.3.2 SIOP Processes Registration Request

Alice's U2SIOP receives the registration challenge and must now generate cryptographic proofs demonstrating her legitimacy without revealing her identity. This phase represents the core innovation of the system, proving membership in the anonymity set without revealing which specific identity Alice owns.

The U2SIOP processes the request through three sequential cryptographic operations, each building upon the previous to create Alice's privacy-preserving registration response.

Service-Specific Key Derivation

Alice's U2SIOP derives unique cryptographic credentials specifically for Service X, ensuring that Alice's activity cannot be linked across services even if multiple services collude. However, these U2SSO-derived credentials must be transformed into standard ECDSA format to maintain compatibility with existing OpenID Connect infrastructure.

This transformation represents a key innovation enabling privacy-preserving authentication to integrate with industry-standard protocols. The TokenKeyFromDigest function performs this critical bridge:

Listing 3.1. SSK to ECDSA Keypair Transformation

```go
func TokenKeyFromDigest(digest []byte) (*ecdsa.PrivateKey, error) {
    if len(digest) != 32 {
        return nil, fmt.Errorf("digest must be 32 bytes")
    }

    // Deterministic mapping: SSK -> ECDSA private key scalar
    d := new(big.Int).SetBytes(digest)
    curve := elliptic.P256()

    // Ensure d is in valid range [1, N-1] for P-256
    n := curve.Params().N
    d.Mod(d, new(big.Int).Sub(n, big.NewInt(1)))
    d.Add(d, big.NewInt(1))

    // Generate ECDSA keypair
    privateKey := new(ecdsa.PrivateKey)
    privateKey.PublicKey.Curve = curve
    privateKey.D = d
    privateKey.PublicKey.X, privateKey.PublicKey.Y =
        curve.ScalarBaseMult(d.FillBytes(make([]byte, 32)))

    return privateKey, nil
}
```

This transformation implements the mathematical conversion that ensures the same service-specific secret key always produces the same ECDSA keypair, enabling consistent authentication across sessions while maintaining cryptographic isolation between services. The deterministic nature is crucial as Alice must generate identical credentials for each service visit to maintain her established identity.

Zero-Knowledge Proof Generation

With service-specific credentials established, Alice's U2SIOP must now prove that she is a legitimate user without revealing her identity. This requires generating a zero-knowledge proof demonstrating that she possesses the master secret key corresponding to one of the public keys in the anonymity set, without revealing which specific public key she owns.

The registration proof is generated using the `RegistrationProof` function, which implements a zero-knowledge membership proof with nullifiers:

Listing 3.2. Registration Proof Generation

```go
func RegistrationProof(topicList [][]byte, index int, currentm int,
currentN int, serviceName []byte, challenge []byte, mskBytes []byte, idList
[][]byte, spkBytes []byte) (string, string, bool) {
    //... context initialization and topic setup

    //... data preparation and conversion to C structures

    // Zero-knowledge proof generation with nullifier
    proofLen := C.secp256k1_zero_mcom_DBPoE_get_size(&rctx, C.int(currentm))
    proof := make([]C.uint8_t, proofLen)
    nullifier := make([]C.uint8_t, 32)

    if int(C.secp256k1_boquila_prove_DBPoE_memmpk(ctx, &rctx, proofPtr,
           nullifierPtr, mpksPtr, mskPtr, chalPtr, namePtr, name_len, &spk,
           C.int32_t(index), C.int(topic_index), C.int(currentN),
           C.int(currentm))) == 0 {
        return "", "", false
    }

    //... proof verification and result conversion

    return proofHex, nullifierHex, true
}
```

The proof generation process proceeds through several cryptographic steps:

1. **Context Initialization**: Creates secp256k1 cryptographic contexts and retrieves the service identifier index from the list.

2. **Data Preparation**: Converts all input parameters (challenge, service name, anonymity set) from Go types to C structures required by the cryptographic library.

3. **Zero-Knowledge Proof Generation**: Calls the underlying U2SSO function to generate both a membership proof of anonymity set membership and a unique nullifier preventing multiple registrations.

The function returns three critical values: the zero-knowledge proof (as hexadecimal string), the nullifier (as hexadecimal string), and a success boolean indicating whether proof generation completed successfully.

Processing Completion

Upon successful completion of both key derivation and proof generation, Alice's U2SIOP has assembled all cryptographic components necessary for her privacy-preserving registration with Service X. The U2SIOP now possesses:

- Service-specific credentials that identify Alice uniquely within Service X while remaining unlinkable to her other service identities
- A zero-knowledge proof demonstrating her legitimacy without revealing her position in the anonymity set
- A nullifier that prevents multiple registrations while preserving anonymity
- Cryptographic binding to the specific registration session preventing replay attacks

These components will be packaged into a Self-Issued ID Token that maintains full compatibility with OpenID Connect standards while enabling privacy-preserving authentication. The next step involves constructing this token and returning it to Service X for verification and account creation.

3.3.3 ID Token Construction and Response

With cryptographic proof generated, Alice's U2SIOP now packages all necessary information into a Self-Issued ID Token that conforms to the OpenID Connect standard while carrying the privacy-preserving proof data. The registration token contains both standard OIDC claims and specialized cryptographic claims that enable Service X to verify Alice's legitimacy without learning her identity.

JWK Embedding and Token Signing

With the ECDSA keypair generated, Alice's U2SIOP embeds the public key within the ID Token and uses the private key for signing. This process establishes the self-issued identity property while enabling standard JWT verification.

Public Key Embedding: The derived public key is embedded within the ID Token as a JSON Web Key (JWK) containing the P-256 elliptic curve coordinates:

```
    {"sub_jwk": {
      "kty": "EC",
      "crv": "P-256",
      "x": "IwTF1VDLo36segylx2psRTw3GiI9rTz3iFHqwsV37-A",
      "y": "rKjQik_WenpujzNGFdo2P83hzWB_vmm9wOs82XPaSlo",
      "use": "sig",
      "alg": "ES256"
  }}
```

Figure 3.5. Embedded Public Key

Note: CRS-ASC proofs use `secp256k1` internally, while ID Tokens are signed with ES256 on P-256 for OIDC interoperability; the curve split is intentional.

This embedding enables Service X to verify the token signature without requiring prior knowledge of Alice's public key or external key distribution infrastructure.

Subject Identifier Generation: To establish Alice's persistent identity within Service X, the system computes a JWK thumbprint following RFC 7638: [13]

$$\text{Subject} = \text{Base64URL}(\text{SHA-256}(\text{Canonical-JWK})) \tag{3.1}$$

Where the canonical JWK representation includes only the essential members (`crv`, `kty`, `x`, `y`) in lexicographic order, ensuring consistent thumbprint computation across different implementations. This thumbprint serves as both the issuer and subject claims, establishing the self-issued property while providing a stable identifier for Alice's service-specific identity.

Token Signing: Alice's U2SIOP signs the complete token using the ECDSA private key with the ES256 algorithm (ECDSA with P-256 and SHA-256):

$$\text{Signature} = \text{ECDSA-Sign}(\text{SHA-256}(\text{header.payload}), \text{ECDSA-Private-Key}) \tag{3.2}$$

This signature proves that Alice possesses the secret key corresponding to the embedded public key, binds all token claims together to prevent tampering, and enables Service X to verify token authenticity using only the embedded public key. The signing process follows standard JWT conventions, creating the familiar three-part token structure:

```
{BASE64URL(HEADER)}.{BASE64URL(PAYLOAD)}.{BASE64URL(SIGNATURE)}
```

Registration Token Structure

The registration token extends the standard OpenID Connect ID Token with additional claims necessary for privacy-preserving authentication, as shown in Figure 3.6. Unlike traditional ID tokens that rely on trust in centralized identity providers, Alice's self-issued token carries cryptographic proofs that Service X can verify independently. The token structure accommodates both familiar OpenID Connect semantics and the specialized requirements of zero-knowledge authentication.

```json
{
  "iss": "0anCI1EH-LjbpKxACVR8Bk47Jd7cNCaaWwGty8HH8n8",
  "sub": "0anCI1EH-LjbpKxACVR8Bk47Jd7cNCaaWwGty8HH8n8",
  "aud": "6d7e78af064c86eb9b9cb1c3611c9ab60a2f9317e3...",
  "exp": 1721826955,
  "iat": 1721733555,
  "nonce": "bDovRs-jKFaViUAcO2-ZBYHezEmDYrgH-19rdZXko7w",
  "zk_proof": "099f84192def6b4a59e1c7c83ffa6a01f8c9982ead...",
  "nullifier": "a14ae4f22dc724e4dba1f38eef49ca14f40e1bbce4...",
  "currentN": 1002,
  "service_pub_key": "09d73181edc1e588755f1bddaa6ae67fa880...",
  "proof_type": "registration",
  "challenge": "5d8123293c7609e55935986d223600fc9bec...",
  "sub_jwk": {
    "alg": "ES256",
    "crv": "P-256",
    "kty": "EC",
    "use": "sig",
    "x": "4XLc0b2RQokqmnnvObB_CHosMFimvMDIPv4GdVColcA",
    "y": "qrwFXGKeUrmiGL8mMdD4nu1pE7R72N9jbc8fXpiARvQ"
  }
}
```

Figure 3.6. Registration Token Structure

The token structure can be categorized into three distinct claim types that work together to enable privacy-preserving authentication:

Response Transmission

Alice's U2SIOP transmits the completed registration token back to Service X via URL fragment redirection, following the OpenID Connect implicit flow pattern adapted for self-issued tokens. The response includes both the signed ID token and the original state parameter for request-response correlation:

Claim	Type	Description
Modified Standard Claims		
iss	OIDC*	Same value as sub claim, establishing self-issued identity without external providers
sub	OIDC*	JWK thumbprint of embedded public key, enabling cryptographic verification of token authenticity
aud	OIDC	Specifies Service X as intended audience, preventing token misuse if intercepted
nonce	OIDC	Standard replay protection
sub_jwk	SIOP	Alice's service-specific public key as JSON Web Key for signature verification
Cryptographic Proof Claims		
zk_proof	U2SIOP	Zero-knowledge proof demonstrating anonymity set membership without revealing specific identity
nullifier	U2SIOP	Deterministic value binding Alice's master identity to Service X, enabling Sybil resistance
challenge	U2SIOP	Links cryptographic proof to specific registration session, preventing replay attacks
spk	U2SIOP	Alice's service-specific public key for ZK proofs
System Parameter Claims		
currentN	U2SIOP	Anonymity set size at proof generation time, required for verification context reconstruction
proof_type	U2SIOP	Set to registration to distinguish from authentication tokens for appropriate verification

Table 3.1. Registration Token Claims Structure (OIDC = Standard OpenID Connect, OIDC* = Modified Semantics, SIOP = SIOP Extension, U2SIOP = U2SSO Extension)

```
/auth/siop/callback#id_token={SIGNED_JWT_TOKEN}&state={STATE_VALUE}
```

This transmission method ensures that the token reaches Service X securely while maintaining compatibility with existing OpenID Connect client libraries. The fragment-based approach prevents the token from being logged in server access logs and enables client-side processing of the response.

The state parameter enables Service X to correlate this response with Alice's original registration request, ensuring that the response corresponds to the correct user session and preventing cross-site request forgery attacks. Service X can now proceed to verify Alice's zero-knowledge proof and create her account if verification succeeds.

Token Properties and Security Guarantees

The completed registration token provides several important security and privacy properties that enable trustworthy authentication:

Self-Contained Verification: Service X can verify Alice's legitimacy using only the information contained within the token and the publicly available anonymity set, without requiring communication with external identity providers or centralized authorities.

Privacy Preservation: The token reveals only that Alice is a legitimate user (member of the anonymity set) without disclosing her specific identity or enabling correlation with her activities on other services.

Replay Resistance: The combination of challenge binding, nonce protection, and session-specific state ensures that the token cannot be reused across different registration attempts or hijacked by attackers.

Alice's registration token is now complete and in transit to Service X, carrying everything necessary for privacy-preserving account creation while maintaining the familiar OpenID Connect user experience.

3.3.4 RP Verification Process

When Service X receives Alice's registration token, it must verify the zero-knowledge proof and validate the token structure before creating her account. The verification process ensures that Alice is a legitimate user (member of the anonymity set) while maintaining her anonymity and preventing Sybil attacks.

Token Validation and Claim Extraction

Service X first validates the JWT token structure, verifies the signature using the embedded public key, and extracts the cryptographic claims needed for proof verification. The service must retrieve the current anonymity set from the blockchain registry to verify Alice's membership proof.

Zero-Knowledge Proof Verification

The core verification step validates Alice's zero-knowledge proof using the `RegistrationVerify` function:

Listing 3.3. Registration Proof Verification

```
func RegistrationVerify(topicList [][]byte, proofHex string, nullifierHex
string, currentN int, serviceName []byte, challenge []byte, idList
[][]byte, spkBytes []byte) bool {
    //... parameter reconstruction and cryptographic context creation

    //... nullifier processing and format validation

    //... anonymity set transformation and proof data preparation

    // Zero-knowledge verification of anonymity set membership
    verifyResult := int(C.secp256k1_boquila_verify_DBPoE_memmpk(ctx, &rctx,
                        proofPtr, nullifierPtr, mpksPtr, chalPtr, namePtr,
                        name_len, &spk, C.int(topic_index), C.int(currentN),
                        C.int(M)))

    return verifyResult != 0
}
```

The verification process works through several cryptographic steps:

1. **Parameter Reconstruction**: Recreates the same cryptographic context used during Alice's proof generation, ensuring identical verification parameters.
2. **Nullifier Processing**: Decodes Alice's nullifier from hexadecimal format and validates its structure for use in the verification equation.
3. **Anonymity Set Transformation**: Converts the entire anonymity set from the blockchain into the cryptographic format required by the verification algorithm.
4. **Zero-Knowledge Verification**: Calls the underlying U2SSO verification function, which confirms that Alice knows the secret key of one of the anonymity set members without revealing which one.

Account Creation and Session Establishment

If verification succeeds, Service X can confidently create Alice's account knowing that:

- Alice is a legitimate user (member of the anonymity set)
- Alice cannot register multiple accounts (nullifier uniqueness)
- Alice's identity remains private (zero-knowledge proof)

- The registration is bound to this specific session (challenge binding)

Service X stores Alice's service-specific public key and nullifier for future authentication attempts, then establishes an authenticated session. Alice's registration with Service X is now complete, and she can begin using the service while maintaining her privacy and unlinkability from other services.

3.4 Authentication Architecture Evolution

The development of the authentication system underwent significant architectural refinement during implementation. Initially, our approach followed the canonical U2SSO protocol by implementing direct authentication using the protocol's native authentication functions.

3.4.1 Initial U2SSO Authentication Approach

A first implementation utilized U2SSO's built-in authentication mechanisms. This approach required:

- Generation of schnorr signature
- Transmission of signature data alongside standard OpenID Connect tokens
- Relying party verification using U2SSO's `AuthenticationVerify` function

While cryptographically sound, early performance testing revealed several practical limitations:

Computational Overhead: Each authentication required two signatures and verifications, introducing some latency per authentication attempt.

Protocol Complexity: The authentication tokens required additional U2SSO-specific claims, complicating integration with existing OpenID Connect infrastructure.

Network Overhead: Authentication proofs increased token size substantially, impacting transmission efficiency.

3.4.2 Key Improvement: SSK-to-ECDSA Transformation

During the development process, an optimization opportunity emerged: the possibility of deterministically transforming U2SSO's service-specific secret keys (SSK) into standard ECDSA private keys without compromising security guarantees.

Initial investigation involved using Ethereum's `crypto.toECDSA` function to explore this transformation on the secp256k1 curve, demonstrating the concept's viability. However, practical deployment required OpenID Connect compliance, which mandates P-256 elliptic curves for ES256 signatures.

We developed the function Listing 3.1, which performs the mathematically sound transformation from secp256k1-based U2SSO keys to P-256 ECDSA keys, ensuring both OIDC compliance and security preservation.

This breakthrough transformed the authentication architecture. Rather than generating U2SSO authentication proofs for each login attempt, the system could:

1. **Transform once**: Convert the SSK to an ECDSA keypair during initial registration
2. **Authenticate efficiently**: Use standard JWT signature verification for subsequent authentications
3. **Maintain security**: Preserve all U2SSO privacy properties through the mathematical transformation

Mathematical Soundness of the Transformation. The deterministic nature does not compromise security since the input SSK derives from cryptographically secure randomness, and unlinkability is preserved as the transformation operates independently on each service-specific key, maintaining the property that keys for different services remain uncorrelated.

3.4.3 Comparative Analysis and Decision Rationale

The architectural evolution from U2SSO authentication proofs to ECDSA transformation offers several advantages:

Table 3.2. Authentication Architecture Comparison

Aspect	U2SSO Authentication	ECDSA Transformation
Token complexity	High (proof data)	Standard (JWT only)
Infrastructure compatibility	Requires U2SSO libraries	Standard OIDC
Additional claims	✓	×

Security Equivalence: Both approaches provide identical security guarantees, as the ECDSA signature proves possession of the same underlying SSK that would be used in U2SSO authentication proofs.

Practical Impact: This architectural decision transforms privacy-preserving authentication from a specialized cryptographic operation into a standard web authentication flow, dramatically improving adoption feasibility.

Table 3.2 illustrates the improvement which the new key derivation method enables: full OpenID Connect compliance with no additional claims needed to embed within the ID Token. The authentication needs no interaction with the U2SSO system and is now just a regular OIDC authentication.

3.5 Streamlined Authentication Flow

After Alice has successfully registered with Service X, subsequent authentications demonstrate the system's key architectural benefit: leveraging the cryptographic foundations established during registration to enable significantly simplified authentication flows.

3.5.1 Authentication Request Processing

When Alice returns to Service X six months later, the service initiates an authentication request identical to registration but with `proof_type=auth`, signaling that no zero-knowledge proof verification is required.

Alice's U2SIOP recognizes this parameter and processes the request using the same cryptographic components detailed in Section 3.3, but with critical differences:

- No blockchain queries to retrieve the current anonymity set
- No zero-knowledge proof generation or verification
- No nullifier computation

3.5.2 Authentication Token Structure

Authentication tokens use the same JWT signing process as registration but omit all cryptographic proof claims:

```json
{
  "iss": "0anCI1EH-LjbpKxACVR8Bk47Jd7cNCaaWwGty8HH8n8",
  "sub": "0anCI1EH-LjbpKxACVR8Bk47Jd7cNCaaWwGty8HH8n8",
  "aud": "6d7e78af064c86eb9b9cb1c3611c9ab60a2f9317e3891891ef31770939f78ef8",
  "exp": 1721826955,
  "iat": 1721733555,
  "nonce": "WzTRSYw6yuxzX8pzC6ysibmwbfDcM6MRdBsPI-qiTuw",
  "sub_jwk": {
    "kty": "EC",
    "crv": "P-256",
    "x": "IwTF1VDLo36segylx2psRTw3GiI9rTz3iFHqwsV37-A",
    "y": "rKjQik_WenpujzNGFdo2P83hzWB_vmm9wOs82XPaSlo",
    "use": "sig",
    "alg": "ES256"
  }
}
```

Figure 3.7. Authentication Token Structure

Claim	**Registration**	**Authentication**	**Rationale**
`zk_proof`	✓	×	No zero-knowledge proof needed
`nullifier`	✓	×	Sybil resistance already established
`currentN`	✓	×	No anonymity set verification
`service_pub_key`	✓	×	SPK not needed for JWT verification
`proof_type`	✓	×	Token type inferred from claim absence
`challenge`	✓	×	Challenge not used in authentication

Table 3.3. Token Claims Comparison: Registration vs Authentication

The absence of `zk_proof`, `nullifier`, `currentN`, `service_pub_key`, and `proof_type` claims signals to Service X that standard JWT signature verification is sufficient.

Cryptographic Authentication Through JWT Signature

While authentication tokens omit explicit zero-knowledge proofs, they provide cryptographic proof of Alice's credential ownership through the JWT signature itself. The signature demonstrates that Alice possesses the ECDSA private key corresponding to the embedded public key, which in turn proves her ownership of the underlying service-specific secret key.

This signature-based proof establishes credential ownership through a clear chain of cryptographic relationships:

1. A valid JWT signature proves Alice possesses the ECDSA private key
2. The ECDSA private key was deterministically derived from her service-specific secret key (SSK)
3. The SSK was deterministically derived from her master secret key (MSK) and the service identifier
4. Therefore, a valid signature ultimately proves Alice possesses the MSK for this service

Performance Characteristics

The authentication verification process provides significant performance advantages compared to registration verification:

Operation	Registration	Authentication	Performance Impact
JWT Signature Verification	✓	✓	Equivalent
JWK Thumbprint Validation	✓	✓	Equivalent
Blockchain Query	✓	×	Eliminates network latency
Anonymity Set Validation	✓	×	Eliminates O(n) verification
Zero-Knowledge Proof Verification	✓	×	Eliminates heavy cryptography
Nullifier Validation	✓	×	Eliminates anti-Sybil computation

Table 3.4. Verification Operations Comparison

Alice's authentication with Service X is now complete. The streamlined verification process demonstrates how the system achieves practical performance while maintaining the privacy and security properties established during registration.

3.5.3 Security Properties Maintained

Despite the simplified verification process, authentication preserves essential security properties:

Credential Ownership: JWT signature cryptographically proves Alice possesses her service-specific credentials established during registration.

Unlinkability: Alice's service-specific credentials remain isolated from other services.

Session Integrity: Standard OIDC nonce and state parameters prevent replay and CSRF attacks.

The authentication flow successfully demonstrates how privacy-preserving authentication can achieve both strong security guarantees and practical performance for returning users, with the computational complexity frontloaded into the initial registration process.

3.6 Supporting Infrastructure: Blockchain Identity Registry

The privacy-preserving authentication system relies on a Ethereum smart contract that serves as both the identity registry and service identifier management system. This blockchain infrastructure enables the system to maintain a global anonymity set without requiring centralized authorities, while providing the cryptographic context necessary for zero-knowledge proof generation and verification.

3.6.1 Eliminating Trusted Third Parties

The privacy-preserving authentication system requires a mechanism for maintaining the global anonymity set that Alice joined during setup and that Service X queries during verification. Traditional approaches would introduce a centralized registry, exactly the type of trusted third party that the system aims to eliminate.

The blockchain-based identity registry solves this fundamental challenge by providing a decentralized, tamper-resistant record of all legitimate identities in the system. This approach ensures that neither Alice nor Service X must trust any central authority, while maintaining the cryptographic properties required for zero-knowledge proof verification.

Unlike federated identity systems where identity providers control user access, the blockchain registry operates as a public utility that no single party can manipulate or shut down.

This decentralized approach directly enables the three core privacy properties of the system: anonymity (no central authority can link Alice's identity to her position in the set), unlinkability (no coordination between services possible without centralized tracking), and Sybil resistance (cryptographic proofs prevent multiple registrations without centralized verification).

3.6.2 Anonymity Set Management

The registry maintains two essential data structures that enable Alice's privacy-preserving authentication: the identity registry containing master public keys from all legitimate users, and the service service identifier registry listing all participating services.

Identity Registry Structure

Each registered identity consists of a 33-byte master public key split across two uint256 fields to accommodate Ethereum's 32-byte word limitation, along with metadata for revocation and ownership tracking. This structure enables Service X to retrieve the complete anonymity set for verification while preserving user privacy.

Service Topic Management

The service registry maintains cryptographic hashes of participating services rather than plaintext identifiers, enabling efficient proof verification. This design ensures that the service identifier list Alice's master identity commits to remains stable and verifiable.

3.6.3 Implementation Architecture

The implementation demonstrates practical feasibility through a hybrid trust model that balances user autonomy with ecosystem governance. The smart contract provides three categories of operations with different permission levels:

Public Operations: Any user can register their master identity or query the anonymity set, ensuring open access to the privacy-preserving authentication system.

Owner Operations: Users can revoke their own identities if needed, providing essential key lifecycle management.

System Operations: Administrative functions like service identifier registration require contract owner authorization, preventing malicious service injection while maintaining ecosystem integrity.

The Go implementation integrates with this infrastructure through automatically generated type-safe bindings, maintaining local caches of contract state while ensuring consistency through periodic synchronization. The complete smart contract implementation and Go integration details are provided in Appendix B

3.6.4 Deployment Characteristics

Performance analysis confirms the practical feasibility of blockchain-based identity management. Identity registration costs approximately 80,000 gas, representing a one-time setup cost for Alice's lifetime access to privacy-preserving authentication.

The hybrid design minimizes ongoing costs through efficient read operations. Service X retrieves anonymity sets through cost-free view functions, while users pay only for their initial registration and any subsequent identity revocation operations.

This cost structure aligns with the system's usage patterns: expensive operations (registration) occur infrequently, while frequent operations (anonymity set queries) impose no ongoing costs on either users or service providers.

3.6.5 Gas Costs and Access Control

The blockchain infrastructure provides the decentralized foundation necessary for privacy-preserving authentication while maintaining practical deployment characteristics. The hybrid trust model balances user autonomy with ecosystem governance, enabling sustainable operation of the privacy-preserving authentication system.

Gas costs vary by operation complexity:

- **Identity Registration**: 80’000 gas for new storage allocation
- **Topic Registration**: 80’000 gas for new storage allocation
- **Batch Retrieval**: 0 gas (view function)

This cost structure aligns with the system’s usage patterns: expensive operations (registration) occur infrequently, while frequent operations (anonymity set queries) impose no ongoing costs on either users or service providers.

3.7 Discussion of Security and Privacy Properties

This section analyzes how the CGo wrapper library and OpenID Connect integration preserve the security and privacy properties of the underlying U2SSO cryptographic system. Rather than formal proofs, an implementation-focused discussion of how the system maintains the essential security guarantees while bridging U2SSO with standard web authentication protocols is provided.

3.7.1 Preserved Privacy Properties

Registration Anonymity

Alice’s registration with Service X preserves anonymity because her master public key (MPK) is stored in the blockchain anonymity set alongside all other users’ public keys. During registration, Alice generates a zero-knowledge proof that demonstrates she knows the secret key corresponding to one of the MPKs in this set, but the proof cryptographically hides which specific MPK belongs to her.

The CGo wrapper preserves this property by passing Alice’s master secret key to the underlying `secp256k1_boquila_prove_DBPoE_memmpk` function without modification. The OpenID Connect integration maintains anonymity by embedding the proof in standard JWT claims, ensuring that Service X receives only the cryptographic proof and Alice’s service-specific public key, never her position in the anonymity set.

Cross-Service Unlinkability

Alice’s activities across different services remain unlinkable because each service receives a different service-specific public key (SPK), deterministically derived from her master secret key and the specific service identifier. Even if multiple services collude, they cannot correlate Alice’s SPKs to determine they belong to the same user.

The implementation preserves unlinkability through the deterministic service-specific secret key derivation, which uses the underlying U2SSO function `secp256k1_boquila_derive_ssk`. Each service receives a cryptographically isolated identity that appears random and unrelated to Alice’s identities on other services.

Sybil Resistance

Alice cannot register multiple accounts with Service X because each registration generates a unique nullifier deterministically computed from her master secret key and Service X’s identifier. This nullifier acts as a cryptographic “fingerprint” that remains constant across registration attempts while revealing nothing about Alice’s identity.

The nullifier property is preserved because the CGo wrapper directly calls the underlying U2SSO nullifier generation without alteration. Service X stores these nullifiers and rejects any subsequent registration attempts that produce the same nullifier, effectively preventing multiple accounts while maintaining Alice’s anonymity.

3.7.2 Authentication Security Properties

Credential Ownership Proof

During authentication, Alice proves ownership of her service-specific credentials through JWT signature verification rather than zero-knowledge proofs. This works because the `TokenKeyFromDigest` function deterministically derives Alice's ECDSA service-specific secret key from a digest derived from the MSK.

The security property holds because valid JWT signatures require possession of the ECDSA private key, which can only be derived from Alice's original service-specific secret key. An attacker without Alice's SSK cannot generate the corresponding ECDSA keypair and therefore cannot produce valid authentication tokens.

Session Binding and Replay Resistance

Both registration and authentication tokens include challenge and nonce values that bind proofs to specific sessions. Registration challenges are incorporated into the zero-knowledge proof generation, while authentication nonces follow standard OIDC patterns, thus prevening replay attacks.

3.7.3 Security Comparison with Traditional OIDC

Table 3.5 presents a comprehensive comparison between traditional OpenID Connect and the U2SSO-based Self-Issued OpenID Provider across key security and privacy dimensions.

User Authentication: Both systems provide user authentication, demonstrating that U2SIOP maintains core identity functionality while enhancing privacy. This compatibility enables existing relying parties to adopt privacy-preserving authentication without operational changes.

Cross-Service Unlinkability: Traditional OIDC enables cross-service tracking through consistent user identifiers, while U2SIOP provides unlinkability via isolated service-specific identities that cannot be correlated across different services.

Registration Anonymity: Traditional OIDC requires identity disclosure during registration, whereas U2SIOP enables anonymous registration through zero-knowledge proofs of anonymity set membership.

Sybil Resistance: Traditional OIDC relies on centralized provider control for sybil prevention, while U2SIOP achieves sybil resistance through cryptographic nullifiers that prevent multiple registrations without central authority.

Decentralized Trust: Traditional OIDC depends on centralized identity providers, creating trust bottlenecks, while U2SIOP eliminates these dependencies through blockchain-based identity registry and cryptographic proofs.

Identity Provider Tracking: Traditional OIDC providers can track user activities across services, while U2SIOP is resistant to such tracking due to service-specific key isolation.

Single Point of Failure: Traditional OIDC creates single points of failure through centralized providers, while U2SIOP distributes trust across the blockchain network, providing resistance to single point failures.

Property	**Traditional OIDC**	**U2SIOP**
User Authentication	✓	✓
Cross-Service Unlinkability	×	✓
Registration Anonymity	×	✓
Sybil Resistance	×	✓
Decentralized Trust	×	✓
Identity Provider Tracking	Vulnerable	Resistant
Single Point of Failure	Vulnerable	Resistant

Table 3.5. Security and Privacy Property Comparison

Chapter 4

Testing and Benchmarking

To evaluate the Cryptographic library and the proof of concept described in Chapter 3, extensive tests and benchmarking were performed. This chapter shows the results and delivers insights into the feasibility of U2SSO integration into authentication standards, such as OpenID Connect.

The benchmarking and testing were performed locally on a MacBook Pro (2020) with a 2.3 GHz Quad-core Intel Core i7, 16 GB DDR4 RAM, and SSD storage to ensure consistent and controlled testing conditions.

4.1 CGo Wrapper library

This section provides a comprehensive analysis of the Cryptographic wrapper library's functions. The benchmarking reveals insights about the computational efficiency and memory management as well as scalability for each of the core functions. The analysis identifies performance bottlenecks and validates the system's suitability for integration into OpenID Connect.

4.1.1 Results and Analysis

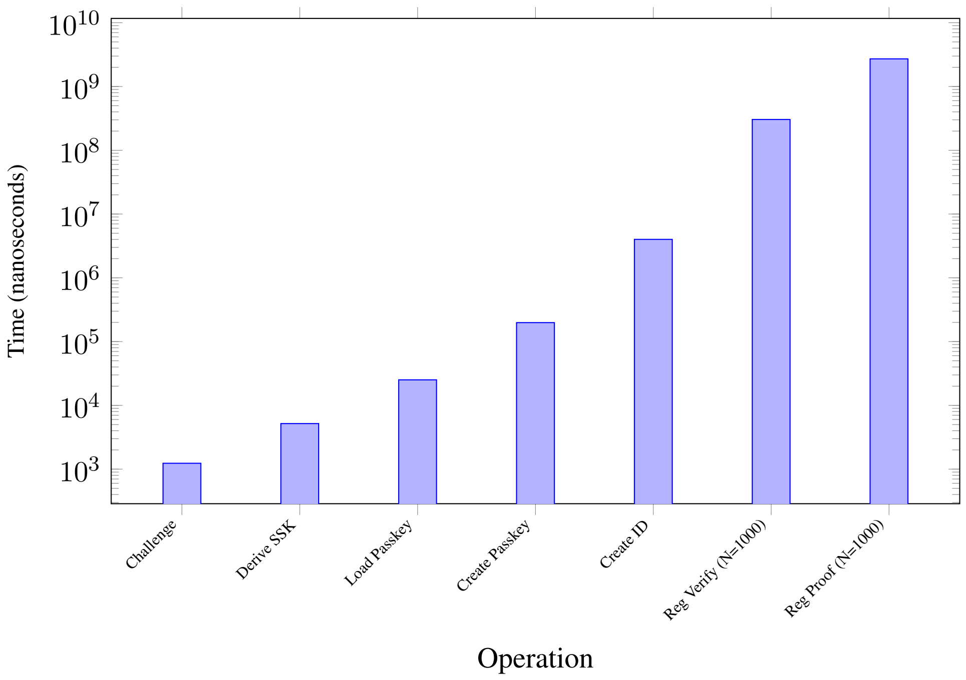


Figure 4.1. U2SSO Operations Performance Comparison

Table 4.1. Basic Cryptographic Operations Performance

Operation	Time (s)
Challenge	1.24×10^{-6}
Derive SSK	5.18×10^{-6}
Load Passkey	2.51×10^{-5}
Create Passkey	1.98×10^{-4}
Create ID	4.01×10^{-3}
Reg Verify (N=1000)	3.04×10^{-1}
Reg Proof (N=1000)	2.71

Figure 4.1 shows that the performance hierarchy of core U2SSO system operations spans several orders of magnitude, from lightweight challenge creation to computationally intensive registration proof generation, which emerges as the system's most expensive operation.

Challenge generation remains the most efficient operation at 1.236 microseconds, making it suitable for high-frequency usage. The SSK derivation follows closely at 5.182 microseconds.

The I/O operations demonstrate moderate computational overhead, with passkey loading at 25.136 microseconds and passkey creation at 198.398 microseconds.

ID creation operations require significantly more computation at 4.007 milliseconds, representing the transition from lightweight cryptographic primitives to more complex identity generation processes.

The registration operations are the most computationally expensive, at almost 3 seconds for anonymity sets with 1000 users. Timing analysis for authentication operations is described in Section 4.2.

Registration Proof Scaling

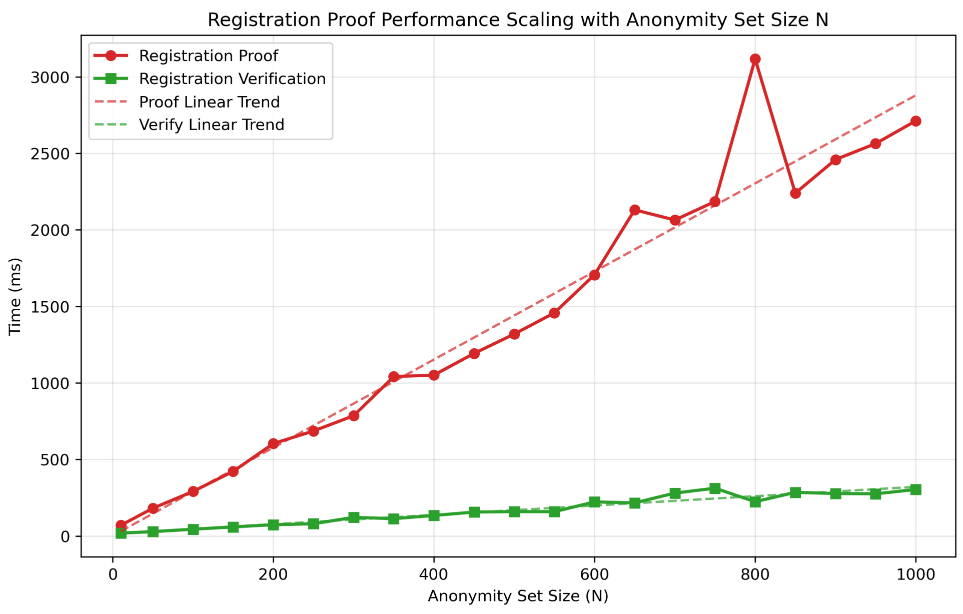


Figure 4.2. Registration Proof scaling

Figure 4.2 and Table 4.2 illustrate the performance impact the anonymity set has on the registration proof calculation. This shows a limitation in the size of the anonymity set. However, since the registration proof is not often performed, the trade-off between additional privacy and performance is acceptable for larger anonymity sets.

The data demonstrates a generally linear scaling relationship between the anonymity set size N and both computational time and memory consumption. As the anonymity set grows from N=10 to N=1000, proof generation time increases from 69.84ms to 2.712 seconds, while verification time scales from 18.82ms to 304.04ms. Some measurement variance is observed, particularly at larger anonymity set sizes, likely due to system-level effects such as garbage collection and thermal management.

Computational Complexity Analysis: The linear trend lines in Figure 4.2 demonstrate that both proof generation and verification exhibit O(N) scaling characteristics. Despite some measurement variance, the overall trend confirms the expected linear relationship between anonymity set size and computational cost. The proof generation process shows a more pronounced scaling impact, with an average increase of approximately 2.6ms per additional user in the anonymity set.

Verification Efficiency: The verification process demonstrates superior efficiency compared to proof generation, maintaining practical performance even at larger anonymity set sizes. While verification times show some variance due to system-level factors, the overall trend remains manageable for service providers.

Measurement Considerations: The observed variance in timing measurements, particularly the outlier at $N = 800$, reflects the challenges of benchmarking cryptographic operations in real-world environments. These variations are attributed to system-level effects including garbage collection cycles, thermal throttling, and background processes, which are representative of actual deployment conditions.

Privacy-Performance Trade-off: Since registration happens only once per service, users can tolerate longer delays for enhanced privacy. Unlike authentication operations that occur frequently, registration delays have minimal impact on overall user experience. The ability to achieve anonymity sets of $N = 1000$ with computation times under 3 seconds provides a favorable trade-off for privacy-conscious deployments, where the privacy benefits significantly outweigh the one-time computational cost.

Table 4.2. Registration Proof Performance Scaling Analysis

N	Proof Time (ms)	Verification Time (ms)	Memory Usage (KB)
10	70	18.8	6.97
50	181	28.4	8.38
100	291	44.4	10.0
150	422	59.4	11.9
200	604	73.7	13.3
250	686	80.1	15.9
300	786	122	16.6
350	1040	114	18.6
400	1050	134	19.9
450	1190	157	22.6
500	1320	159	24.6
550	1460	159	24.6
600	1710	223	26.6
650	2130	217	27.9
700	2060	281	30.6
750	2190	312	33.3
800	3120	224	33.3
850	2240	286	34.6
900	2460	277	38.6
950	2560	276	38.6
1000	2710	304	46.6

4.2 Proof of Concept

To evaluate the performance of the Self-Issued OpenID Provider, extensive benchmarking was performed on the system. This section presents performance analysis of the developed U2SIOP, examining both the registration and the authentication flow.

The performance characteristics of such a U2SIOP implementation are particularly important due to the additional computational overhead introduced by the zero-knowledge proofs. Unlike traditional OpenID Connect flows, which primarily involve token exchange and validation, a U2SIOP system must generate cryptographic proofs to demonstrate identity. This adds computational complexity, which may impact user experience.

4.2.1 Test Environment

The performance evaluation was conducted in a controlled local development environment to ensure consistent and reproducible results. The test configuration included:

- **U2SIOP Server:** Running on localhost:8080
- **RP Server:** Running on localhost:8081
- **Blockchain:** RPC Server: http://127.0.0.1:7545
- **Network Configuration:** Local loopback interface (minimal network latency)
- **Hardware:** MacBook Pro (2020), 2.3 GHz Quad-core Intel Core i7 processor, 16 GB 3733 MHz DDR4 memory, 1 TB SSD storage
- **Anonymity Set:** 1’002 active IDs as well as eight services registered to the system

This controlled environment eliminates external network variables and provides baseline performance metrics. However it does neglect any network overhead that would occur in a real-world scenario, as all exchange happens on a local machine.

4.2.2 Test Scenarios

Two scenarios were evaluated:

1. **Registration Flow:** Simulates new user registration, requiring initial credential establishment
2. **Authentication Flow:** Simulates existing user authentication using previously established credentials

Each scenario was tested with 100 iterations to assess performance consistency. Separate timing measurements were conducted for individual components to isolate performance characteristics and identify main sources of computational overhead.

The anonymity set was populated using an automated setup script that generates user identities. The script creates individual passkey files for each user, derives unique identities using the active service identifier list from the smart contract, and registers these identities to the anonymity set.

4.2.3 Metrics Collection

The benchmark tool collected the following performance metrics for each test iteration:

- **Round Trip Time:** Complete request-response cycle time
- **Token Generation Time:** Time required for U2SIOP to generate cryptographic and the token
- **Token Validation Time:** Time required for the RP to validate the token and verify the proof
- **Blockchain Operation Time:** Isolated blockchain contract interaction time

4.2.4 Results

Round Trip Time Analysis

Table 4.3. U2SIOP Round Trip Time Performance Analysis (100 iterations)

Scenario	Avg Time (ms)	Min Time (ms)	Max Time (ms)
Registration	18’500	15’700	25’600
Authentication	5.5	2.3	37

The results demonstrate consistent performance with authentication operations completing significantly faster than registration operations.
Key findings from the overall performance analysis:

- **Authentication operations** average 5.5 ms
- **Registration operations** average 18’500 ms (≈ 18.5 s)
- **Zero failure rate** across all 200 test iterations, demonstrating excellent reliability

Component Performance Analysis

The breakdown of individual components within each flow type reveals the computational distribution within the U2SIOP system.

Registration Flow Component Breakdown

Table 4.4. Registration Flow Component Performance Breakdown (100 iterations)

Avg Round Trip (ms)	Avg Proof Generation (ms)	Avg Token Validation (ms)
18’500	3’100	15’400

Registration flow shows significantly higher computational requirements:

- **Proof Generation:** Averages 3’100 ms (16.8% of total time)
- **Token Validation:** Averages 15’400 ms (83.2% of total time)

Authentication Flow Component Breakdown

Table 4.5. Authentication Flow Component Performance Breakdown (100 iterations)

Avg Round Trip (ms)	Avg Proof Generation (ms)	Avg Token Validation (ms)
5.5	4.4	1.1

Authentication Flow performance demonstrates efficient processing:

- **Proof Generation:** Averages 4.4 ms (80% of total time)
- **Token Validation:** Averages 1.1 ms (20% of total time)

Blockchain Operations Performance

To identify the specific performance bottleneck, blockchain operations were benchmarked in isolation:

Table 4.6. Isolated Blockchain Operations Performance (100 runs)

Component	Avg Time (ms)	Percentage of Blockchain Time
ID List Retrieval	14'100	99.5%
Topics Retrieval	66	0.5%
Total Blockchain	**≈ 14'200**	**100.0%**

Critical findings from blockchain analysis:

- **ID List Retrieval dominates** ID List retrieval at 99.5% of blockchain time
- **Blockchain operations account** for 76.8% of total registration time (14'200 ms out of 18'500 ms)

4.2.5 Performance Bottleneck Analysis

The detailed benchmarking reveals the true performance bottleneck in the U2SIOP implementation. Token validation time (83.1% of total registration time) primarily consists of blockchain operations rather than cryptographic computation:

Registration Flow Detailed Breakdown:

- **Token Generation:** 3.1 s (16.8% of total time)
- **Blockchain Operations:** 14.2 s (76.7% of total time)
  - ID List Retrieval: 14.1 s (76.3% of total time)
  - Topics Retrieval: 66 ms (0.4% of total time)
- **Other Protocol Overhead:** 1.2 s (JWT parsing, validation, etc. making up 6.5% of total time)

The analysis demonstrates that the blockchain operations, specifically retrieving the complete anonymity set (1'002 active IDs) from the smart contract, represent the primary performance bottleneck rather than cryptographic computation.

4.2.6 Conclusion

The performance evaluation reveals several key insights relevant to the practical feasibility of a Self-Issued OpenID Provider:

User Experience: Authentication response times around 5.5 ms fall within excellent limits for web applications. Registration times of 18.5 seconds exceed acceptable user experience thresholds and represent the primary barrier to practical deployment. However, the proposed caching optimization Subsection 4.2.7 demonstrates a clear path to reducing registration times to 4-5 seconds for subsequent operations.

Computational Overhead: The dramatic increase in proof generation time for registration operations reflects the substantial difference in cryptographic complexity between authenticating with existing credentials versus establishing new identities. However, the isolated benchmark analysis reveals that blockchain operations (76.7%) represent the primary bottleneck rather than pure cryptographic computation (16.9%).

Scalability Considerations: The sub-millisecond pure authentication proof generation time (when isolated, averaging 0.09 ms) demonstrates excellent scalability potential for cryptographic operations, while the blockchain operations present the primary scaling challenge for registration flows.

Overall, this evaluation demonstrates the feasibility of such a Self-Issued OpenID Provider, though with notable performance considerations. The response times for both registration and authentication, while higher than traditional authentication systems, remain within acceptable timeframes for privacy-focused applications. A SIOP implementation as described in[29] and Chapter 3 can deliver the enhanced privacy and security promised by decentralized identity, with a clear performance trade-off that users may find acceptable for the privacy benefits provided.

4.2.7 Performance Optimization Opportunities

The benchmarking results reveal a significant optimization opportunity that could dramatically improve the practical viability of the U2SIOP implementation. While the zero-knowledge proof generation performs efficiently (3.1 seconds), the primary performance bottleneck lies in the retrieval of anonymity set data from the blockchain smart contract.

Current Performance Bottleneck Analysis

As demonstrated in the detailed breakdown above, token validation accounts for 83.1% of the total registration time (15.4 seconds out of 18.5 seconds), with blockchain operations representing the overwhelming majority of this overhead (76.7% of total time). The current implementation fetches the complete anonymity set (1’002 active IDs) as well as all service identifiers from the blockchain for every single verification operation. This results in multiple expensive RPC calls to the Ethereum client, substantial data transfer, and blockchain transaction overhead.

Proposed Optimization: Anonymity Set Caching with Identifiers

A significant performance improvement could be achieved by implementing **anonymity set caching with unique identifiers**. Since anonymity sets are immutable once established, this optimization strategy would include:

1. **Anonymity Set Identification:** Each anonymity set configuration receives a unique identifier (hash of the set contents or deployment block number)
2. **Local Caching:** Relying Parties cache anonymity sets locally, indexed by their unique identifiers
3. **Cache-First Retrieval:** Before fetching from the blockchain, RPs check whether the required anonymity set is already available in their local cache

The implementation would require minimal enhancements to the smart contract architecture to expose anonymity set identifiers efficiently. Relying Parties would maintain local caches of anonymity sets with associated identifiers, performing lightweight identifier lookups before each verification and only retrieving anonymity set data from the blockchain when encountering a previously unseen identifier. Since anonymity sets remain fixed after creation, cached data never becomes stale, eliminating the need for complex invalidation mechanisms.

Expected Performance Impact

This optimization leverages the fact that anonymity sets are immutable once established, and Relying Parties will frequently encounter the same anonymity sets across multiple verification operations. The expected performance improvements are:

- **Current Performance:** 18.5 seconds per registration (regardless of repetition)

- **With Caching:** First encounter with an anonymity set $\sim$18.5 seconds, subsequent operations using the same anonymity set $\sim$4 to 5 seconds
- **Performance Improvement:** Approximately 77% reduction in registration time for cache hits

This estimation assumes cached operations would eliminate the 14.2-second blockchain retrieval overhead while maintaining the 3.1-second ZK proof generation and adding minimal cache validation overhead.

The optimization maintains all privacy and security guarantees while addressing the primary practical barrier to deployment.

Chapter 5

Conclusion

This thesis has successfully demonstrated the practical feasibility of implementing privacy-preserving authentication through the integration of the User-Issued Unlinkable Single Sign-On protocol with the Self-Issued OpenID Provider specification. By bridging the ongoing cryptographic research provided by Alupotha et al. with industry standard authentication protocols, this thesis provides a concrete pathway toward decentralized digital identity management that preserves user privacy while maintaining familiar authentication flows, already understood by relying parties and end users.

The primary work of this thesis is the development of a complete Self-Issued Unlinkable OpenID Provider implementation, which achieves both privacy preservation and Sybil resistance through cryptographic guarantees rather than trusted centralized authorities. Such a U2SIOP enables users to act as their own identity providers and achieves unlinkability across different services through deterministically derived pseudonyms, addressing a fundamental flaw of traditional federated identity systems.

A particularly significant contribution emerged from the collaborative development process: the identification and implementation of a deterministic SSK-to-ECDSA transformation technique. This demonstrates how privacy-preserving protocols can be optimized for practical deployment without security compromise. The transformation eliminates per-authentication proof generation overhead while maintaining all U2SSO privacy properties, achieving authentication times that make privacy-preserving authentication competitive with traditional systems.

The CGo wrapper library provides a clean interface between the complex cryptographic computations required by U2SSO and the standardized authentication flows of OpenID Connect. Additionally, it provides all functionalities necessary to communicate and work with the Identity Registry. Finally, an implementation of the Identity Registry is provided in the form of a Solidity smart contract, allowing users to publish their master public keys to the anonymity set and relying parties to publish their service names to the service identifier list.

The implemented U2SIOP, as well as a relying party which accepts both Self-Issued ID Tokens and traditional OpenID Connect ID tokens, serves as a proof of concept for the feasibility and readiness for real-world implementation of such a decentralized identity management system.

The performance evaluation demonstrates that privacy-preserving authentication can be achieved within acceptable latency bounds. Authentication averaging 5.47 ms fall well within user experience expectations and would remain responsive even when accounting for real-world network latency. However, registration operations present a more complex performance profile, averaging 18.5 seconds due primarily to anonymity set retrieval from the blockchain (14.2). This bottleneck could be addressed through anonymity set caching mechanisms or by optimizing the balance between anonymity set size and retrieval efficiency. Such optimizations would significantly reduce registration times but would require careful consideration of the fundamental privacy-performance tradeoff: smaller anonymity sets enable faster registration at the cost of reduced anonymity guarantees, while larger sets provide stronger privacy but impose greater computational overhead. The zero failure rate across all test iterations validates the reliability of the underlying cryptographic implementations.

The implementation proves that the apparent trade-off between privacy and convenience in digital identity systems is not inevitable. Through maintaining compatibility with existing OpenID Connect infrastructure, while adding cryptographic guarantees, the system provides a path to privacy-preserving authentication without the need to completely rebuild identity infrastructure.

The blockchain-based identity registry eliminates the need for trusted central authorities and ensures Sybil resistance. This decentralized approach addresses the systemic risks identified in Varoufakis's critique of digital platform feudalism. It paves the way forward to becoming more independent from technology giants for basic digital services and could allow users to reclaim some power over their own data, without sacrificing the convenience provided by single sign-on.

Some limitations must be acknowledged in this implementation. The anonymity set size directly impacts both privacy guarantees and performance. Therein lies an important trade-off. As the anonymity set increases, so does the proof size and computation time, resulting in significant performance degradation

This performance impact has both inherent and implementation-specific components. The proof generation overhead is inherent to the U2SSO protocol: larger anonymity sets require more complex zero-knowledge proofs, making for an unavoidable cryptographic trade-off. However, the 14.2-second blockchain retrieval bottleneck, which constitutes most of the 18.5-second registration time, is implementation-specific. Alternative Identity Registry architectures could potentially eliminate this retrieval overhead while maintaining identical privacy guarantees.

Nevertheless, this may represent an acceptable performance trade-off, as anonymity set size only impacts registration operations, which occur once per service. The one-time nature of this overhead means that substantial registration delays may be tolerable in exchange for ongoing privacy benefits during frequent authentication operations.

An additional limitation lies in the creation of the master credentials. Since the nullifier is precomputed for each service registered to the system, a user may use these credentials only for relying parties registered within the service identifier list at the master public key creation event. There is currently no way to update the master credentials to new services joining the system without revoking the public identity and generating a new one, which would generate additional gas fees when writing to the blockchain.

Additionally, the system relies on the discrete logarithm assumption, making it vulnerable to quantum attacks. With the continued advancements in quantum computing long-term security guarantees can only be assured by migration to post-quantum cryptographic primitives.

Several directions for future research emerge from this thesis. Integration with other emerging standards, such as Verifiable Credentials [25] and selective disclosure [7] could prove valuable to a Self-Issued Identity provider and further enhance privacy or even enable attribute-based authentication. Improvements to scalability through more efficient zero-knowledge systems could reduce performance degradation.

This thesis demonstrates that self-sovereign digital identity is not merely a theoretical ideal but can be implemented with existing cryptographic techniques. Through integration of the U2SSO protocol with OpenID Connect standards, this thesis provides a foundation for privacy-preserving authentication systems.

The proof of concept shows that users can reclaim control over their digital identities without sacrificing the convenience that has allowed digital landlords to extract rent in the form of personal data from the user. The successful integration suggests that the barriers to widespread adoption of such privacy-preserving identity systems are not impossible to conquer, with the challenges moving forward not lying in technical limitations. Such systems could prove a pathway to the eventual reclamation of digital identities and enable true digital self-sovereignty in the digital space.

Appendix A

Cryptographic Library Functions

A.1 Master Identity Functions

Listing A.1. Master Secret Key Generation

```go
func CreatePasskey(filename string) {
    msk := make([]C.uint8_t, 32)
    mskPtr := (*C.uint8_t)(unsafe.Pointer(&msk[0]))
    C.RAND_bytes(mskPtr, 32)
    mskBytes := C.GoBytes(unsafe.Pointer(mskPtr), 32)
    file, err := os.OpenFile(filename, os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        log.Fatal(err)}
    defer func() {
        if err := file.Close(); err != nil {
            log.Fatal(err)}}()
    if _, err := file.Write(mskBytes); err != nil {
        panic(err)}}
```

Listing A.2. Master Secret Key Retrieval

```go
  func LoadPasskey(filename string) ([]byte, bool, error) {
      mskBytes := make([]byte, 32)
      file, err := os.Open(filename)
      if err != nil {return nil, false, err}
      defer func() {
          if err := file.Close(); err != nil {
              log.Fatal(err)}}()
      n, err := file.Read(mskBytes)
      if err != nil && err != io.EOF {
          log.Fatal(err)}
      if n == 0 {
          return nil, false, nil}
      return mskBytes, true, nil}
```

Listing A.3. Master Public Key Generation

```go
    func CreateID(topicList [][]byte, mskBytes []byte) []byte {
  msk := make([]C.uint8_t, 32)
  mskPtr := (*C.uint8_t)(unsafe.Pointer(&msk[0]))
  for i := 0; i < 32; i++ {
      msk[i] = C.uint8_t(mskBytes[i])
  }

  ctx := C.secp256k1_context_create(C.SECP256K1_CONTEXT_SIGN |
  C.SECP256K1_CONTEXT_VERIFY)

```

```
    gen_seed := (*C.uint8_t)(C.malloc(C.sizeof_uint8_t * 32))
    defer C.free(unsafe.Pointer(gen_seed))
    C.memset(unsafe.Pointer(gen_seed), C.int(gen_seed_fix), 32) // 11

    topicsPtr, topicsSize := convertTopicsToC(topicList)
    defer C.free(unsafe.Pointer(topicsPtr))

    rctx := C.secp256k1_ringcip_DBPoE_context_create(
        ctx,
        C.int(10),
        C.int(N),
        C.int(M),
        gen_seed,
        topicsPtr,
        topicsSize)

    mpk := make([]C.pk_t, 1)
    if C.secp256k1_boquila_gen_DBPoE_mpk(ctx, &rctx, &mpk[0], mskPtr) == 0 {
        fmt.Println("mpk creation failed")
    }
    mpkBytes := C.GoBytes(unsafe.Pointer(&mpk[0]),
    C.int(unsafe.Sizeof(mpk[0])))
    return mpkBytes
}
```

Listing A.4. Challenge Generation for Registration

```
        func CreateChallenge() []byte {
            chal := make([]C.uint8_t, 32)
            chalPtr := (*C.uint8_t)(unsafe.Pointer(&chal[0]))
            C.RAND_bytes(chalPtr, 32)
            chalBytes := C.GoBytes(unsafe.Pointer(chalPtr), 32)
            return chalBytes}
```

Appendix B

Smart Contract Implementation

The complete `OidcU2SSO` smart contract implementation:

Listing B.1. Smart Contract Implementation

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract OidcU2SSO {
    struct ID {
        uint256 id;
        uint256 id33;
        bool active;
        address owner;
    }

    struct Topic {
        bytes32 topicHash;
        bool active;
        address owner;
    }

    address private _owner;
    ID[] public idList;
    Topic[] public topicList;
    uint256 public nextIdIndex;
    uint256 public nextTopicIndex;

    constructor() {
        _owner = msg.sender;
        nextIdIndex = 0;
        nextTopicIndex = 0;
    }

    // ========== ID Management Functions ==========
    function addID(uint256 _id, uint256 _id33) public returns (uint256) {
        idList.push(ID(_id, _id33, true, msg.sender));
        nextIdIndex = nextIdIndex + 1;
        return nextIdIndex - 1;
    }

    function revokeID(uint256 _index) public {
        require(_index < idList.length, "Index out of bounds");
        ID storage id = idList[_index];
        require(msg.sender == id.owner || msg.sender == _owner, "Not
        authorized");
        id.active = false;
    }

    function getIDs(uint256 _index) public view returns (uint256, uint256) {
        require(_index < idList.length, "Index out of bounds");
        ID storage id = idList[_index];
        return (id.id, id.id33);
    }

    function getState(uint256 _index) public view returns (bool) {
        require(_index < idList.length, "Index out of bounds");
        ID storage id = idList[_index];
        return id.active;
    }

    function getIDSize() public view returns (uint256) {
        return nextIdIndex;
    }

    function getIDIndex(uint256 _id, uint256 _id33) public view returns
    (int256) {
        for (uint256 i = 0; i < nextIdIndex; i++) {
            if ((idList[i].id == _id) && (idList[i].id33 == _id33)) {
                return int256(i);
            }
        }
        return -1;
    }

    function addTopic(bytes32 _topicHash) public returns (uint256) {
        require(msg.sender == _owner, "Only owner can add topics");
        topicList.push(Topic(_topicHash, true, msg.sender));
        nextTopicIndex = nextTopicIndex + 1;
        return nextTopicIndex - 1;
    }

    function revokeTopic(uint256 _index) public {
        require(_index < topicList.length, "Index out of bounds");
        Topic storage topic = topicList[_index];
        require(msg.sender == topic.owner || msg.sender == _owner, "Not
        authorized");
        topic.active = false;
    }

    function getTopic(uint256 _index) public view returns (bytes32) {
        require(_index < topicList.length, "Index out of bounds");
        Topic storage topic = topicList[_index];
        return topic.topicHash;
    }

    function getTopicState(uint256 _index) public view returns (bool) {
        require(_index < topicList.length, "Index out of bounds");
        Topic storage topic = topicList[_index];
        return topic.active;
    }

    function getTopicSize() public view returns (uint256) {
        return nextTopicIndex;
    }

    function getTopicIndex(bytes32 _topicHash) public view returns (int256)
    {
        for (uint256 i = 0; i < nextTopicIndex; i++) {
            if (topicList[i].topicHash == _topicHash) {
                return int256(i);
            }
        }
        return -1;
    }

    function getAllActiveTopics() public view returns (bytes32[] memory) {
        uint256 activeCount = 0;

        // First pass: count active topics
        for (uint256 i = 0; i < nextTopicIndex; i++) {
            if (topicList[i].active) {
                activeCount++;
            }
        }

        // Second pass: collect active topics
        bytes32[] memory activeTopics = new bytes32[](activeCount);
        uint256 currentIndex = 0;

        for (uint256 i = 0; i < nextTopicIndex; i++) {
            if (topicList[i].active) {
                activeTopics[currentIndex] = topicList[i].topicHash;
                currentIndex++;
            }
        }

        return activeTopics;
    }
}
```

Bibliography

[1] J. Alupotha, M. Barbaraci, I. Kaklamanis, A. Rawat, C. Cachin, and F. Zhang, “Anonymous self-credentials and their application to single-sign-on,” *IACR Cryptol. ePrint Arch.*, p. 618, 2025.

[2] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels,” Request for Comments 2119, RFC Editor, Mar. 1997.

[3] D. W. Chadwick, “Federated identity management,” in *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures* (A. Aldini, G. Barthe, and R. Gorrieri, eds.), pp. 96–120, Berlin, Heidelberg: Springer Berlin Heidelberg, Aug. 2009.

[4] Cloudflare, “What is vendor lock-in? — vendor lock-in and cloud computing,” 2024.

[5] Y. Dimova, T. V. Goethem, and W. Joosen, “Everybody’s looking for SSOmething: A large-scale evaluation on the privacy of OAuth authentication on the web,” *Proceedings on Privacy Enhancing Technologies*, vol. 2023, pp. 452–467, Oct. 2023.

[6] J. R. Douceur, “The sybil attack,” in *Peer-to-Peer Systems*, pp. 251–260, Springer, 2002.

[7] D. Fett, K. Yasuda, and B. Campbell, “Selective Disclosure for JWTs (SD-JWT),” Internet-Draft draft-ietf-oauth-selective-disclosure-jwt-22, Internet Engineering Task Force, May 2025. Work in Progress.

[8] A. Fiat and A. Shamir, “How to prove yourself: practical solutions to identification and signature problems,” in *Advances in Cryptology – CRYPTO ’86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, Aug. 1986.

[9] R. Gafni and D. Nissim, “To social login or not login? exploring factors affecting the decision,” *Issues in Informing Science and Information Technology*, vol. 11, pp. 57–72, Jan. 2014.

[10] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” *SIAM Journal on Computing*, vol. 18, pp. 186–208, 1989.

[11] F. Hao, “Schnorr Non-interactive Zero-Knowledge Proof,” Request for Comments 8235, RFC Editor, Sept. 2017.

[12] D. Hardt, “The OAuth 2.0 Authorization Framework,” Request for Comments 6749, RFC Editor, Oct. 2012.

[13] M. B. Jones and N. Sakimura, “JSON Web Key (JWK) Thumbprint.” RFC 7638, Sept. 2015.

[14] W. Li and C. J. Mitchell, “Security issues in OAuth 2.0 SSO implementations,” in *Information Security* (S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. M. Yiu, eds.), (Cham), pp. 529–541, Springer International Publishing, 2014.

[15] W. Li and C. J. Mitchell, “User access privacy in OAuth 2.0 and OpenID connect,” in *Proceedings of 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 664–672, IEEE Press, 2020.

[16] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations." RFC 6819, Jan. 2013.

[17] S. G. Morkonda, S. Chiasson, and P. C. van Oorschot, "Empirical analysis and privacy implications in oauth-based single sign-on systems," in *WPES '21: Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society, Virtual Event, Korea, 15 November 2021*, pp. 195–208, ACM, 2021.

[18] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology – CRYPTO '91* (J. Feigenbaum, ed.), (Berlin, Heidelberg), pp. 129–140, Springer Berlin Heidelberg, 1992.

[19] E. Politou, E. Alepis, and C. Patsakis, "Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions," *Journal of Cybersecurity*, vol. 4, Mar. 2018.

[20] E. Pollman, "Tech, regulatory arbitrage, and limits," *European Business Organization Law Review*, vol. 20, p. 567, 2019.

[21] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID connect core 1.0 incorporating errata set 2," Final Errata Set 2, OpenID Foundation, Dec. 2023.

[22] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, pp. 161–174, Jan. 1991.

[23] J. Spooren, D. Preuveneers, and W. Joosen, "Mobile device fingerprinting considered harmful for risk-based authentication," in *Proceedings of the Eighth European Workshop on System Security, EuroSec 2015, Bordeaux, France, April 21, 2015* (J. Caballero and M. Polychronakis, eds.), pp. 6:1–6:6, ACM, 2015.

[24] M. Sporny, A. Guy, M. Sabadello, and D. Reed, "Decentralized identifiers (DIDs) v1.0," W3C recommendation, World Wide Web Consortium (W3C), July 2022. Available at `https://www.w3.org/TR/did-1.0/`.

[25] O. Terbu, T. Lodderstedt, K. Yasuda, D. Fett, and J. . Heenan, "OpenID for Verifiable Presentations 1.0," standards track draft, OpenID Foundation, July 2023.

[26] H. C. A. van Tilborg and S. Jajodia, eds., *Encyclopedia of Cryptography and Security*. New York, NY: Springer, 2 ed., 2011.

[27] Y. Varoufakis, *Technofeudalism: What Killed Capitalism*. Melville House, 2024.

[28] N. Wang, H. Xu, and J. Grossklags, "Third-party apps on facebook: privacy and the illusion of control," in *Proceedings of the 5th ACM Symposium on Computer Human Interaction for Management of Information Technology*, CHIMIT '11, (New York, NY, USA), pp. 4:1–4:10, Association for Computing Machinery, 2011.

[29] K. Yasuda, M. Jones, and T. Lodderstedt, "Self-Issued OpenID Provider v2 – draft 13," Standards Track Draft OpenID Connect Self-Issued v2.1.0 Draft 13, OpenID Foundation, Nov. 2023.

[30] Zitadel, "OIDC: Openid connect client and server library," 2024. Accessed: 21/07/2025.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum	Unterschrift