Privacy-Preserving KYC

CHEN Dezhi

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science (Honours)
in Computer Science

Hong Kong Baptist University

March 27, 2023

Acknowledgements

Soon after submitting this project, my university life is coming to an end. I would like to express my sincere gratitude and appreciation to all the people who have helped me in this project and in my university life.

First of all, I would like to thank my supervisor, Prof. XU Jianliang, for his guidance and support throughout this project. I would also like to thank Mr. ZHANG Yatao, who gave me valuable advice on the implementation of the project.

I am grateful to my family for their love and support, and my partner and friends for their companionship and encouragement during my university life.

Finally, I would like to thank Dr. Elizabeth K.S. LAW for her generous donation to the scholarship that supported my study at Hong Kong Baptist University.

Declaration

I hereby declare that all the work done in this Final Year Project is of my independent effort. I also certify that I have never submitted the idea and product of this Final Year Project for academic or employment credits.

CHEN Dezhi	Date

Contents

Abstract

Know-Your-Customer (KYC) process is a critical step in some businesses to combat crime. Several problems exist in the traditional KYC process and they may threaten users' privacy. A solution concept named *zkKYC* is proposed to address these problems. The solution concept specifies the business requirements for a privacy-preserving KYC system. This project aims to address the challenge of designing a zero-knowledge KYC token generation and verification mechanism, and provide an implementation of a privacy-preserving KYC system based on the zkKYC solution concept, using Self-Sovereign Identity (SSI) via Hyperledger Aries and zero-knowledge proofs with zk-SNARKs via the Circom and SnarkJS libraries.

Introduction

Background

Know-Your-Customer (KYC) process is a critical step in some businesses to combat crimes such as money laundering and terrorist financing. However, the current common KYC process may threaten users' privacy, by requiring more information than necessary, leaking or abusing personal information and so on, while users have to sacrifice their privacy and give up control over their information to meet the compliance requirement. From the businesses' perspective, it is also a challenge for them to keep customers' personal information secure.

When exploring solutions to privacy-preserving KYC, we found an interesting solution concept named *zkKYC*, proposed by Pieter Pauwels in 2021. It describes a "zero-knowledge KYC solution" leveraging self-sovereign identity and zero-knowledge proofs. The paper identifies problems existing in the traditional KYC process and proposes a solution concept to address these problems. It also defines detailed business requirements for such a system.

As a solution concept, the paper did not specify any detail for the implementation. It is also mentioned by the author in the subsequent paper that designing and implementing the zero-knowledge proving system for zkKYC tokens is a valuable challenge to tackle[1]. So, we decided to take on the task of solving the challenge and implementing our privacy-preserving KYC system based on the zkKYC solution concept.

Project Overview

This project consists of two major parts: Self-Sovereign Identity (SSI) and Zero Knowledge Proof (ZKP).

SSI is a decentralized identity framework that allows users to control their own identity information and share it with others only when necessary. It is a promising solution and has been adopted by many organizations. We use SSI to facilitate the issuance, secure storage and verification of credentials that contain the information required for KYC. In our project, we use Hyperledger Aries as the SSI framework.

ZKP is a cryptographic technique that allows a prover to prove a statement to a verifier without revealing any information other than the validity of the statement. We use ZKP to generate the *zkKYC token* and its *validity proof*. This enables the verifier (business) to verify that the zkKYC token contains the information required for KYC without revealing the information itself. In our project, we use zk-SNARKs to achieve the ZKP, using the Circom and SnarkJS libraries.

In the following chapters, we will first identify the problems and the objective of this project. Then, we will introduce the zkKYC solution concept, followed by the system architecture. After that, we will elaborate on the implementation of the SSI part and the ZKP part, respectively, with detailed background knowledge, design thinking and implementation details. After introducing our implementation, we will also review the business requirements specified in the solution concept to discuss how they have been addressed in our implementation. Finally, we will discuss the future work or potentials of this project.

Problems and Objective

Generally speaking, we have the following problems in the current traditional KYC process: - Traditional businesses collect much information for the KYC purpose and clients have poor control over the shared information - Some novel businesses, such as cryptocurrency ones, guarantee privacy and anonymity but lack the KYC process to combat crime.

According to Pauwels (2021), the traditional KYC process has the following problems that threaten the security of users' personal information:

> **Copy Problem:** When users have to share personal information with each regulated entity that they engage with, it is impossible for them to control what these businesses subsequently do with that information. It can be copied, sold, misused, or may indeed be part of a hack or data breach anytime in the future.
> **Bundling Problem:** While AML/CTF regulations usually require only specific data attributes (e.g. name, address, date of birth) of a customer to be verified for KYC purposes, often much more personal data is collected and stored by the regulated entity.
> **Recursive Oversight Problem:** When users have to share personal information with a regulated entity, what governance protections do they have for their information? If data protection regulation does exist in a particular jurisdiction (e.g. GDPR), how is this enforced? How is the regulated entity held accountable for violating data governance obligations? If a regulated entity shares customer identity data with a regulator or other government agency, what privacy protection obligations are they subject to and who holds them to account?

To address these problems, Pauwels (2021) proposed a solution concept named ***zkKYC***, which leverages Self-Sovereign Identity (SSI) and zero-knowledge proofs to achieve the KYC purpose without disclosing any personal information.

While Pauwels (2021) gives comprehensive business requirements and the solution concept, challenges lie in the design and implementation of the zero-knowledge proving system (Pauwels et al., 2022). Therefore, the objective of this project is to study existing SSI technologies, design a zero-knowledge proving mechanism and finally provide an implementation of the zkKYC solution concept.

Solution Overview

In this chapter, we will introduce our solution to the problems using the zkKYC solution concept, and then elaborate on the system architecture of this project.

The zkKYC Solution Concept

The Concept and Roles

zkKYC is a solution concept proposed by Pauwels (2021) for privacy-preserving KYC. It leverages SSI and zero-knowledge proofs to achieve the KYC purpose without disclosing any personal information.

In a typical SSI system, the user has a *wallet* that contains *verifiable credentials* (VCs). A VC is a digital document that contains the information required for KYC, such as eligibility proofs like nationality and age. The zkKYC solution adds a zero-knowledge identity proof on top of the existing SSI system, which is called the *zkKYC token*.

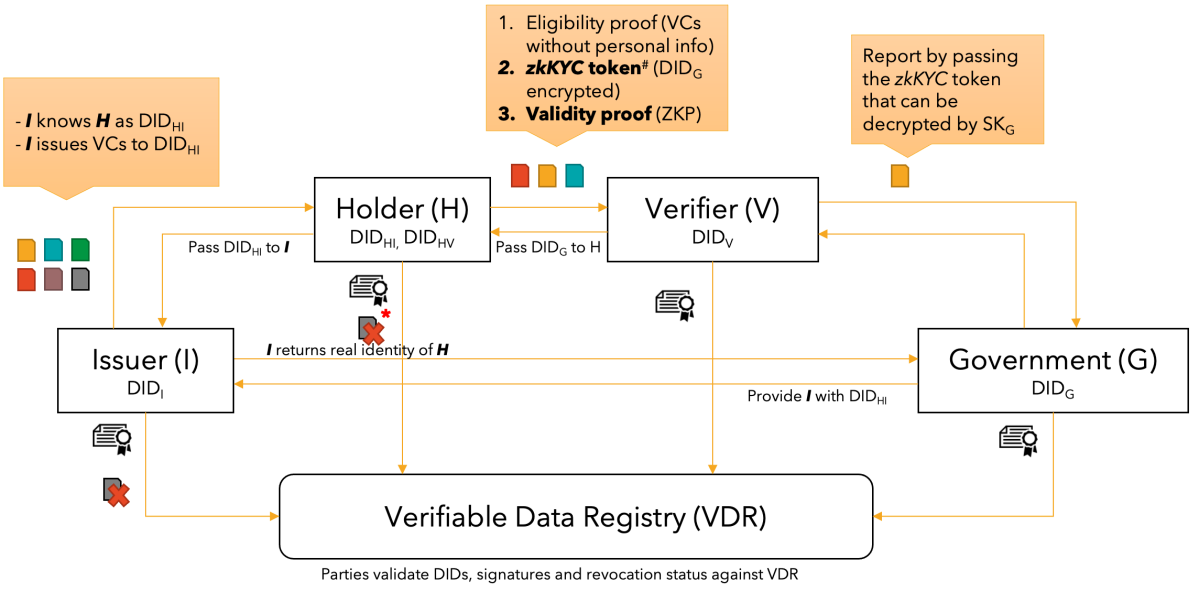


Figure 1: zkKYC Solution Concept

The above figure describes the zkKYC solution concept, where the Issuer issues credentials to the Holder, which the Holder can use to generate presentations of eligibility and identity (zkKYC token) for the Verifier. The Verifier can verify the validity of the information by checking with the information in the Verifiable Data Registry, which we will discuss in detail in the next chapter.

The zkKYC token is generated by the user and contains their identity information, using a zero-knowledge proof system that we designed in this project. The user can then share the zkKYC token with the business (verifier) during the KYC process using the SSI system. The business can then verify the validity of the zkKYC token without revealing any of the information contained in the token. Only the Government can decrypt the token to obtain the user's identity under necessary circumstances.

To summarize the roles involved in the zkKYC solution, we have the following:

- **Issuer:** The Issuer is the entity that controls the real identity of the user. The Issuer issues the Holder with verifiable credentials according to the Holder's information in its database. In reality, this role can be played by the government agency, or some large banks that already have the user's information registered via the traditional KYC process.
- **Holder:** The Holder is the user who holds the verifiable credentials from the Issuer, and may want to register with a new business, or Verifier. The Holder can generate a zkKYC token using the verifiable credentials and share it with the Verifier.
- **Verifier:** The Verifier is the business that wants to verify the identity of the Holder. The Verifier can check the eligibility of the Holder via the credentials, and verify the validity of the zkKYC token without revealing any of the information contained in the token.
- **Government:** The Government is the regulatory authority that can decrypt the zkKYC token to obtain information to retrieve the Holder's real identity under necessary circumstances.

Business Requirements

Here are the detailed business requirements of the privacy-preserving KYC system specified in the original zkKYC paper:

ID	Business Requirement
BR01	The level of user control, agency and privacy provided and enabled by the self-sovereign identity model MUST be preserved or enhanced. See section 3.2 for details.
BR02	A User SHOULD NOT share personal identifiable information (e.g. name, address, date of birth) when on-boarding at a Business.
BR03	A User MUST prove they meet the criteria defined by the Business or relevant regulator(s) to consume the provided service (e.g. adult, domestic resident, valid driver license for specific vehicle category, verified email address).
BR04	A Business that suspects a specific User of fraud, money laundering or terrorism financing MUST be able to report that User to Government (e.g. regulator).
BR05	A Business that wants to file charges against a specific User due to breach of contract or other dispute MUST be able to report that User to Government (e.g. law enforcement).
BR06	Government (e.g. regulator, law enforcement) MUST be able to identify a reported User based on the information provided and on the ground of reasonable suspicion.
BR07	When a Business reports a User to Government, this MUST NOT be disclosed to the User (i.e. tipping-off).
BR08	A Business SHOULD NOT hold personal identifiable information on a User, unless it is provided to them by Government in context of a reported issue.

We designed our implementation according to these requirements. We will review our system against these requirements after the relevant discussions.

Finally, we strongly recommend you to read the original zkKYC paper by Pauwels (2021) to have a better understanding of this solution.

System Architecture

Ideally, a self-sovereign identity system is decentralized. That means there can be multiple organizations acting as each role. All of them are connected to each other via peer-to-peer credential passing and a decentralized ledger to store the decentralized identity information. We will introduce how SSI works in the next chapter. In this section, we will focus on the system architecture of a single role of the system, as shown in the following figure.

For each role, we have a *frontend UI* for users to interact with. The UI is connected to the *Hyperledger*

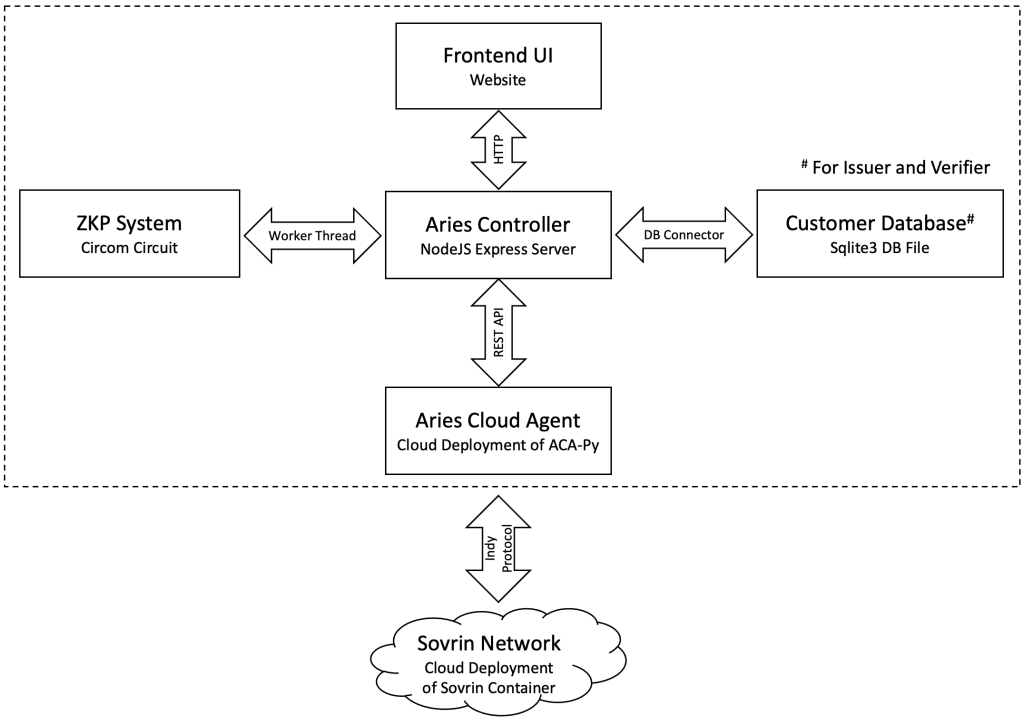


Figure 2: System Architecture for One Role

Aries controller. The frontend UI can be a mobile app or a website. In our project, we have implemented websites as the frontend UI for each role.

The *Aries controller* implements our business logic, such as what to include in the credential, how to generate the token and proof, what to do upon receiving a credential and so on. Thus, it is also connected to the *ZKP system* for zkKYC token generation and verification, and the *database* for retrieving or storing customers' information.

In our project, we implemented the *Aries controller* using NodeJS. The *ZKP system* is built using the *zkSNARK* libraries Circom and SnarkJS. The *database* is simply SQLite3 DB files.

Then, the Aries controller is connected to the *Hyperledger Aries Framework* via HTTP REST API provided by the Aries Cloud Agent (ACA-Py). We have deployed the ACA-Py on a remote server, one for each of the Issuer, Holder and Verifier. ACA-Py will handle the communication with the other agents, and perform verification using the information stored in the distributed ledger of the Sovrin Network, which acts as the Verifiable Data Registry (VDR) in our system.

Self-Sovereign Identity (SSI) Using Hyperledger Aries

Introduction to SSI

What is Self-Sovereign Identity?

Self-sovereign identity (SSI) is a term to describe a new approach to identity administration that allows individuals to control the use of their personal information[2], [3]. SSI systems are decentralized, meaning that they do not rely on a central authority to manage identity information.

In traditional identity systems, the identity provider (e.g. government, corporation) is responsible for managing the identity information of individuals. The identity provider is also responsible for verifying the identity of individuals. In contrast, SSI systems allow individuals to manage their own identity information and verify their identity themselves. This is achieved by using cryptographic keys to sign information and verify the authenticity of the information.

Besides authentication, SSI systems also allow individuals to prove their eligibility for certain services. For example, an individual can prove that they are over 18 years old to open an investment account, by presenting a *verifiable credential (VC)* issued by a trusted authority and stored in his or her digital wallet. The verifier can then be assured about the eligibility of the individual without reaching out to the original issuer, or knowing any unnecessary information.

In this system, the credential issuer decides which attributes to include in the credential, the holder decides which attributes to disclose to the verifier, and the verifier decides whether to accept the credential based on the disclosed information and the credibility of the issuer. The possibility of selective disclosure of information helps address the *Bundling Problem* as we discussed in the previous chapter.

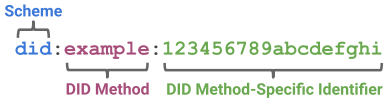
Key Components in SSI

Technically, SSI systems are built on top of a set of standards[4], [5]. In this subsection, we will discuss the three key components, namely *Decentralized Identifier (DID)*, *Verifiable Credential (VC)* and *Verifiable Data Registry (VDR)*.

Decentralized Identifier (DID)

The decentralized identifier (DID) is a globally unique identifier the subject. The subject can be individuals and organizations. The word "decentralized" also describes the practice that the DID is not managed by a central authority, but generated by the subject itself.

A DID is a string of characters that consists of three parts: the DID scheme, the DID method and the method-specific identifier, as illustrated in the following example: All DIDs start with the string `did:`.



The DID method suggests how the DID can be resolved to a *DID Document*, such as `did:ethr` using the Ethereum blockchain, or `did:sov` using the Sovrin network. Each DID method has its own rules for generating and resolving the method-specific identifier.

Resolved DID documents are JSON objects that contain the public keys and service endpoints of the subject. The DID document can be used to verify the authenticity of the DID and related signatures using the public keys, or to communicate with the subject using the service endpoints.

Verifier Credential (VC)

The Verifiable Credential (VC) is a JSON object that contains a set of attributes that are relevant to the subject. It is issued by the issuer and stored in the holder's wallet. The holder can then present the VC to the verifier to prove their eligibility for certain services. Depending on the DID method, it may be possible for the issuer to revoke issued VCs.

The credentials are verifiable in the way that the verifier can verify the validity of the signatures via the public keys in the issuer's DID document, which is recorded in the *Verifiable Data Registry*.

The signing scheme used in the VC is designed to support selective disclosure, for example, the CL signature for JSON-LD credentials. Some schemes also support predicate proofs, which allow the holder to prove that a certain attribute is within a certain range, such as age. For example, the BBS+ signature.

Finally, with a verifiable credential, the holder can select a subset of its attributes and generate a VC presentation for the verifier.

Verifiable Data Registry (VDR)

A verifiable data registry is where the DID documents, VC schemas and the revocation list are stored. Ideally, the VDR is a decentralized system that is not controlled by a single entity. The VDR can be a blockchain, such as Ethereum, or a distributed ledger, such as Sovrin, which is an instance of Hyperledger Indy. In our project, we use Sovrin as the VDR.

Typical SSI Workflow

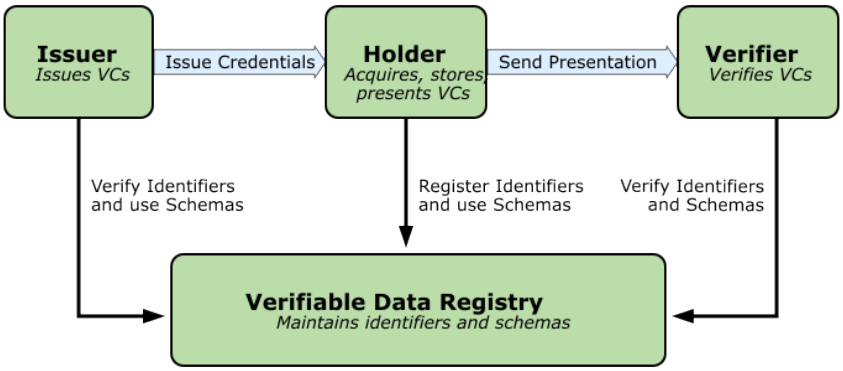


Figure 3: Typical SSI Workflow

As you can see, the zkKYC solution is an adaption of the SSI workflow. In the typical SSI workflow, parties register their DIDs and credential schemas on the VDR. The issuer then issues a VC to the holder, and the holder presents the VC to the verifier. The verifier can then verify the authenticity of the VC using the public keys in the issuer's DID document, which is recorded in the VDR.

Hyperledger SSI Framework

Relevant Hyperledger Projects

Hyperledger Foundation is a non-profit organization that incubates and develops open source blockchain projects. In the SSI space, there are three relevant Hyperledger projects. They are *Hyperledger Aries*, *Hyperledger Indy* and *Hyperledger Ursa*.

The following figure illustrates the relationship between these projects.

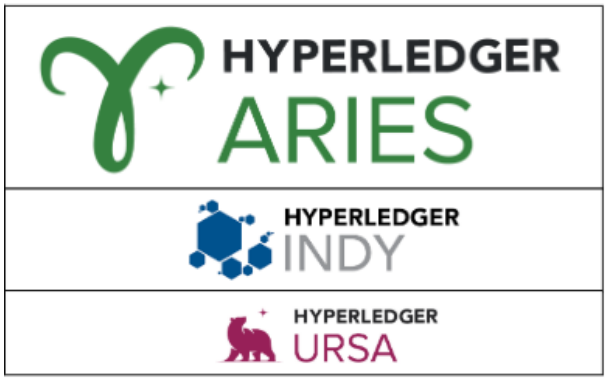


Figure 4: Hyperledger SSI Projects

The lowest layer is the *Hyperledger Ursa* project, which provides cryptographic primitives for the other projects. The *Hyperledger Indy* project provides a distributed ledger for storing DID documents and VCs. For instance, the Sovrin Network we use in our project is developed based on the Hyperledger Indy project. The *Hyperledger Aries* project provides a set of APIs for interacting with the distributed ledger and communicating with other agents.

Hyperledger Aries

As a developer of SSI applications, we write programs using the Hyperledger Aries framework[6]. The Aries framework provides a set of APIs that allow developers to interact with other SSI agents through the underlying SSI protocols such as *Connection Protocol*, *Credential Exchange Protocol* and *Present Proof Protocol*. There are also several implementations of Aries agents, such as *Aries Cloud Agent Python (ACA-Py)* and *Aries Framework Go*.

In our project, we use ACA-Py as the agent implementation. The following figure illustrates the architecture of ACA-Py[7].

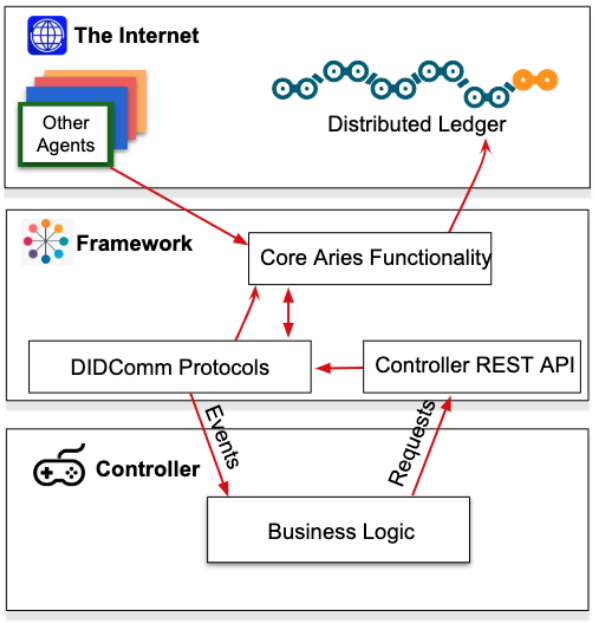


Figure 5: Hyperledger Aries Architecture

To develop an SSI application with Hyperledger Aries, we need to implement our business logic in the *controller* of the agent. From there, we define which credentials to issue, which attributes to disclose and which credentials to accept, and so on. With ACA-Py, we can implement the controller in any

programming language and interact with the framework through the REST API.

Implementing zkKYC with Hyperledger Aries

In our project, we have four roles, namely *Issuer*, *Holder*, *Verifier* and *Government*. We have implemented agent controllers for Issuer, Holder and Verifier to demonstrate the SSI workflow of the zkKYC system for issuing and verifying the credentials of eligibility and zkKYC token.

The Government may also have its own agent controller of an SSI agent to communicate with the Issuer and Verifier. For simplicity, we did not implement the Government agent because what matters from the Government is just its key pair for encrypting and decrypting the *zkKYC token*, which we will demonstrate by direct inputs during encryption and decryption.

Environment Setup

Hyperledger Aries Cloud Agent

In our project we use the *Hyperledger Aries Cloud Agent (ACA-Py)* as the SSI agent. We have deployed the three ACA-Py agents on the same machine with different ports. The Issuer agent is running on port 8020, the Holder agent is running on port 8030, and the Verifier agent is running on port 8040.

For the demonstration purpose, we let the each agent connect to an in-memory Indy wallet. The wallet is a secure storage for the agent's keys, credentials and information of established connections. The wallet is encrypted with a master key, which is generated by and managed by the agent.

Sovrin Network

Sovrin is a distributed ledger for identity built on top of Hyperledger Indy. It is a permissioned network, which means that only authorized entities can join the network. In our project, we deployed a local Sovrin network using a container running four validator nodes. Then, we can register new DIDs to our local Sovrin network for development and demonstration.

Major Components

In this subsection, we will introduce major components of the roles in our project. For each component, we will start from the frontend user interface to describe the operations that the user can perform, and then for each operation, we will describe the corresponding agent controller's work in the back-end.

Issuer Site

The Issuer Site simulates the process of customer real identity registration with the Issuer, and the process of verifiable credential request and issuance.

First, the customer (holder) needs to fill in the registration form and create a profile with the Issuer. This step demonstrates the traditional KYC process performed by the Issuer, before it can issue verifiable credentials to the customer. The customer's profile is stored in the Issuer's database.

Then, before requesting verifiable credentials, the customer needs to setup a connection between the Issuer and his/her wallet agent. To do this, he/she needs to login the Issuer's customer portal and get an invitation URL from the Issuer. Then, the customer pastes this URL into the Holder's Wallet agent. The Holder's Wallet agent will then setup a connection with the Issuer's agent in the backend.

During the connection setup, the Holder's agent will generate a dedicated DID, DID_{HI}, for the connection. The issuer will then bind this DID with the customer's real identity in its database. Then, the holder can login the Issuer's customer portal to request verifiable credentials for eligibility proof or *zkKYC*.

The eligibility proof VC contains the customer's personal information such as age and nationality, which may be required by the verifier. When it is requested, the Issuer will issue a VC with the customer's personal information and send it via the connection with the holder's wallet agent.

The zkKYC credential is a special credential we designed in this project for the zkKYC system. It contains the Issuer's DID, the Holder's DID (DID_{HI}), and the Issuer's signature on the tuple of these two DIDs. This credential will be sent to the Holder in the same way as the other VC.

Holder's Wallet

This is the controller for the Holder's wallet agent.

As mentioned above, the first function of the Holder's wallet agent is to setup a connection. It can accept an invitation URL from either the Issuer or the Verifier and setup connections to them.

The second function of the Holder's wallet agent is to accept and store the verifiable credentials issued by the Issuer. The holder can see a list of credentials stored in the wallet on the frontend.

When registering a business with the Verifier, the Holder will be required to present a verifiable credential for eligibility proof. Upon getting the presentation request, the Holder can select the credential from the wallet and generate a presentation for the Verifier. The presentation process will be handled by the Holder's wallet agent.

The generation and presentation of the zkKYC token, however, works differently from the standard SSI procedures. We need to be aware that what the Holder presents to the Verifier in this step is not the zkKYC credential itself, but a *zkKYC token*, containing the information in the original credential from the Issuer AND information about the connection with this specific Verifier, along with its *validity proof*.

In fact, this step is achieved by generating the zkKYC token and proof with the ZKP system, which we will discuss in the next chapter, and then *issuing* a new verifiable credential to the Verifier. Here, we let the Holder to issue the credential, because the content is actually generated by the Holder. Upon receiving this credential, the Verifier can further verify the validity of the zkKYC token through our ZKP system.

Verifier Site

Similar to the Issuer Site, the Verifier Site simulates the process of business registration, but instead of conducting traditional KYC process, the verifier simply requests for presentations of verifiable credentials we discussed above.

As already mentioned in the previous subsections, the Holder needs to setup a connection, by first going to the Verifier's customer portal and getting an invitation URL. Similarly, the Holder will have a dedicated DID DID_{HV} for the connection.

The credential requests are also initiated from the customer portal by clicking on the *Credential Request* button. The Verifier will then send a credential request to the Holder's wallet agent via the connection, and automatically complete the process when the Holder's wallet agent in the backend.

The Verifier can later retrieve the zkKYC token of a certain DID_{HV} and report it to the Government when it is required.

Government's Portal

The Government's portal is a web application that allows the Government to decrypt the reported zkKYC tokens to get the information about the Holder.

The information contains the Verifier's DID, the Holder's dedicated DID for the Verifier, the Holder's dedicated DID for the Issuer and the Issuer's DID. Then, the Government can check if this token is for the reporting Verifier (to avoid copying/abusing), and ask the Issuer to retrieve the Holder's real identity from its database using the Holder's DID_{HI}.

ZKP Integration

As mentioned in earlier subsections, the zkKYC token and its validity proof are generated by the ZKP system, which is the ZKP part of this project. The ZKP system packaged into a NodeJS module named **zkkyc-js**. It provides a set of APIs for Issuer to sign the zkKYC token, for Verifier to parse and verify the zkKYC token, and for Government to decrypt the zkKYC token.

We integrate **zkkyc-js** into the SSI backend by letting the corresponding agents call the APIs via a *worker thread* of NodeJS. The worker thread is a separate thread that can be used to perform time-consuming tasks without blocking the main thread. This is necessary because the zero-knowledge proof generation and verification are very slow in comparison to the other operations in the SSI backend, and thus we have this implementation to avoid blocking the main thread.

The computation results are passed back to the main thread via callbacks, and then subsequently passed to the database or to the frontend.

Zero Knowledge Proof (ZKP) with zk-SNARKs

Generating the *zkKYC token* and the *validity proof* is an unaddressed challenge in the zkKYC paper. In this section, we will discuss our approach to solving this problem using zero knowledge proof with *zk-SNARKs*.

zk-SNARK Fundamentals

In this section, we will introduce the fundamentals of zk-SNARKs, and how we can implement zk-SNARKs in our project.

What are zk-SNARKs?

zk-SNARKs stands for *Zero Knowledge Succinct Non-interactive ARgument of Knowledge*. It is a type of zero knowledge proof that allows a prover to convince a verifier that a statement is true, without revealing any information beyond the statement itself. The statement is usually a mathematical expression, and the prover can prove that the statement is true by providing a *witness* that satisfies the statement.

There are many zk-SNARK schemes, or proving systems. To understand how zk-SNARKs achieve zero knowledge proof, we will use a polynomial-based proving system as an example to demonstrate the mechanism[8]

Knowledge as Polynomial

There is an advantageous property of polynomials. Two non-equal polynomials of degree at most d can intersect at no more than d points. So, when an x is randomly chosen from N numbers, the probability of choosing an intersecting point is $\frac{d}{N}$, which is negligible when N is significantly larger than d.

Therefore, if the prover responds with a correct evaluation of the polynomial when given a random position x, it is highly confident that the prover does know the coefficients of the polynomial.

Thanks to this property, we can express knowledge as coefficients of a polynomial, and easily prove that we know the coefficients.

Factorization

There are usually some constraints on the knowledge (polynomial) that the prover claims to know. For example, the prover claims that he knows a polynomial of degree d,

$p(x) = c_0x^0 + c_1x^1 + \ldots + c_dx^d$,

and it has roots $x = 1$ and $x = 2$.

Factorization can help us extract these constraints. In the above example, because *p(x)* has roots $x = 1$ and $x = 2$., we know that *p(x)* can be factorized into

$p(x) \ = \ t(x)\ h(x)$,

where $t(x) = (x - 1)(x - 2)$ and is called the *target polynomial*, and $h(x)$ is the quotient of $p(x)/t(x)$.

So far, the prover and the verifier can try proving in the following protocol:

- Verifier chooses a random number s, and computes $t_s = t(s)$. Then, Verifier passes s to Prover
- Prover obtains $h(x) = \frac{p(x)}{t(x)}$ and computes $p_s = p(s)$ and $h_s = h(s)$. Then, Prover passes p_s and h_s to Verifier
- Verifier verifies $p_s = t_s \cdot h_s$

However, the proof is valid only if both Verifier and Prover honestly follow this protocol. The prover can easily cheat by first choosing an arbitrary value as h_s, and then obtain an p_s by multiply the arbitrary h_s by t_s.

Homomorphic Encryption

The flaw of the above protocol is caused by passing a plaintext s for polynomial evaluation, allowing dishonest provers to fabricate an equation. This problem can be solved by homomorphic encryption.

A simple homomorphic encryption is

$E(x) = g^x \pmod n$,

where g is a natural number base, and $mod\ n$ makes it difficult to solve $y = E(x)$ for x, due to the high complexity of the discrete logarithm problem.

Here are two operations supported by $E(x)$:

- Addition of two encrypted values a and b:

$$g^{a+b} = g^a \cdot g^b = E(a) \cdot E(b)$$

- Multiplication of an encrypted value a by an unencrypted value m:

$$(g^a m) = (g^a)^m = (E(a))^m$$

Now, we can encrypt the random value s chosen by Verifier before passing to Prover. As $E(x)$ does not support exponential operation encrypted values, the verifier needs to prepare encrypted values of $s^0, s^1, \ldots, s^d$. The original protocol can be updated as follows:

- Verifier chooses a random number s, and computes $t_s = t(s)$.
- Verifier computes $E_{s^i} = E\left(s^i\right)$ for i in $0,\ 1,\ \ldots,\ d$ and passes E_{s^i} to Prover
- Prover obtains $h(x) = \frac{p(x)}{t(x)}$
- Prover computes $E_{p_s} = E\left(p(s)\right) = g^{p(s)} = g^{c_0 s^0 + c_1 s^1 + \ldots + c_d s^d}$

$$= \left(g^{s^0}\right)^{c_0} \cdot \left(g^{s^1}\right)^{c_1} \cdot \ldots \cdot \left(g^{s^d}\right)^{c_d} = \prod_{i=0}^{d} \left(E_{s^i}\right)^{c_i}$$

- Prover computes E_{h_s} in a similar way
- Prover passes E_{p_s} and E_{h_s} to Verifier
- Verifier verifies $E_{p_s} = \left(E_{h_s}\right)^{t_s} \ \Rightarrow\ g^{p(s)} = \left(g^{h(s)}\right)^{t(s)} \ \Rightarrow\ p(s) = h(s)\ t(s)$

This updated protocol prevents the prover from getting plaintext s and directly computing $t(s)$. However, this does not completely eliminate the possibility of cheating. Prover can still fabricate an equation as follows:

- Compute $g^{t(s)}$ using E_{s^i}
- Find a random value r, let $z_h = g^r$, $z_p = \left(g^{t(s)}\right)^r$, so that $z_p = z_h^{t(s)}$
- Now, E_{p_s} and E_{h_s} can be forged by z_p and z_h respectively

Knowledge-of-Exponent Assumption (KEA)

To address the problem above, we need to enforce the prover to obtain the values via the polynomials $p(x)$ and $h(x)$. Note that after applying the homomorphic encryption, the coefficients of polynomials are used to *exponentiate* the encrypted values E_{s_i}. That is why KEA can help.

The problem KEA addresses is described as follows:

Alice has a value a. She needs to pass a to Bob and asks Bob to perform an exponentiation a^v, where the exponent v is *only known by Bob*. How can Alice ensure that Bob has performed the exponentiation for a with some v, instead of returning some other values?

The solution is to prepare a *shifted* value $a^{'} = a^{\alpha} \mod n$, where α is chosen and only known by Alice. Then, Alice passes $a^{'}$ along with a, and asks Bob to perform the same exponential operation on a, $a^{'}$ to obtain b, $b^{'}$. Finally, Alice checks if

$$b^{\alpha} = (a^{v})^{\alpha} = a^{\alpha v} = a^{'v} = b^{'}$$

Since Bob cannot obtain α except by unfeasible brute-forcing, the only way for Bob to produce $\left(b, b^{'}\right)$ that satisfies the above equation is to perform the exponentiation.

Therefore, in addition to steps in the previous protocol, Verifier will prepare shifted values $E^{'}_{s^i} = g^{\alpha s^i}$ to be passed to Prover along with E_{s^i}, and Prover will return E_{p_s} and $E^{'}_{p_s}$. Finally, if Verifier checks that if $(E_{p_s})^{\alpha} = E^{'}_{p_s}$, it proves that the prover has correctly performed the homomorphic-encrypted calculation.

Achieving Zero-Knowledge

So far, we have achieved sound and complete proving. But values passed from Prover to Verifier, $E_{p_s} = g^{p(s)}$, $E_{h_s} = g^{h(s)}$ and $E^{'}_{p_s} = g^{\alpha p(s)}$ contains some knowledge about the polynomial $p(x)$ that can be extracted by Verifier.

Currently, the verifier verifies that these values satisfy

$$E_{p_s} = (E_{h_s})^{t(s)}, (E_{p_s})^{\alpha} = E^{'}_{p_s}$$

It is obvious that if the prover exponentiates E_{p_s}, E_{h_s} and $E^{'}_{p_s}$ to the power δ, the equations still hold. So, the simple solution is to select a secret random δ, calculate $E_{zk-p_s} = (E_{p_s})^{\delta}$, $E_{zkh_s} = (E_{h_s})^{\delta}$ and $E_{zkp^{'}_s} = \left(E^{'}_{p_s}\right)^{\delta} \ (mod\ n)$ and pass the E_{zk} values to the verifier.

There is a sequence diagram on the next page showing the protocol we discussed above.

Until now, we have achieved sound and complete zero-knowledge proving. Full zkSNARK also involves another two properties: succinctness and non-interactivity, but we will not cover those details in this report to avoid further distraction from our main topic.

Choosing a Proving System

Groth16 and PLONK are two popular zk-SNARK proving systems.

Groth16[9] is based on pairing-based cryptography and uses a construction called *bilinear pairing*. Groth16 is efficient and features succinct proofs and reasonably fast performance, and it has been widely adopted in applications such as privacy cryptocurrencies like Zcash[10], [11].

A major drawback of Groth16 is that it requires a trusted setup for each different circuit. A trusted setup is a process that generates the parameters for the proving system[12]. It can be analogous to the process of generating the encrypted powers of s to get E_{s^i} in the previous section.

The trusted setup is a one-time process that is expensive and time-consuming, and must be performed by trusted parties using secure Multi-Party Computation (MPC).

PLONK[13], on the other hand, is a more recently proposed, polynomial-based zk-SNARK proving system that leverages the *polynomial commitment scheme*[14].

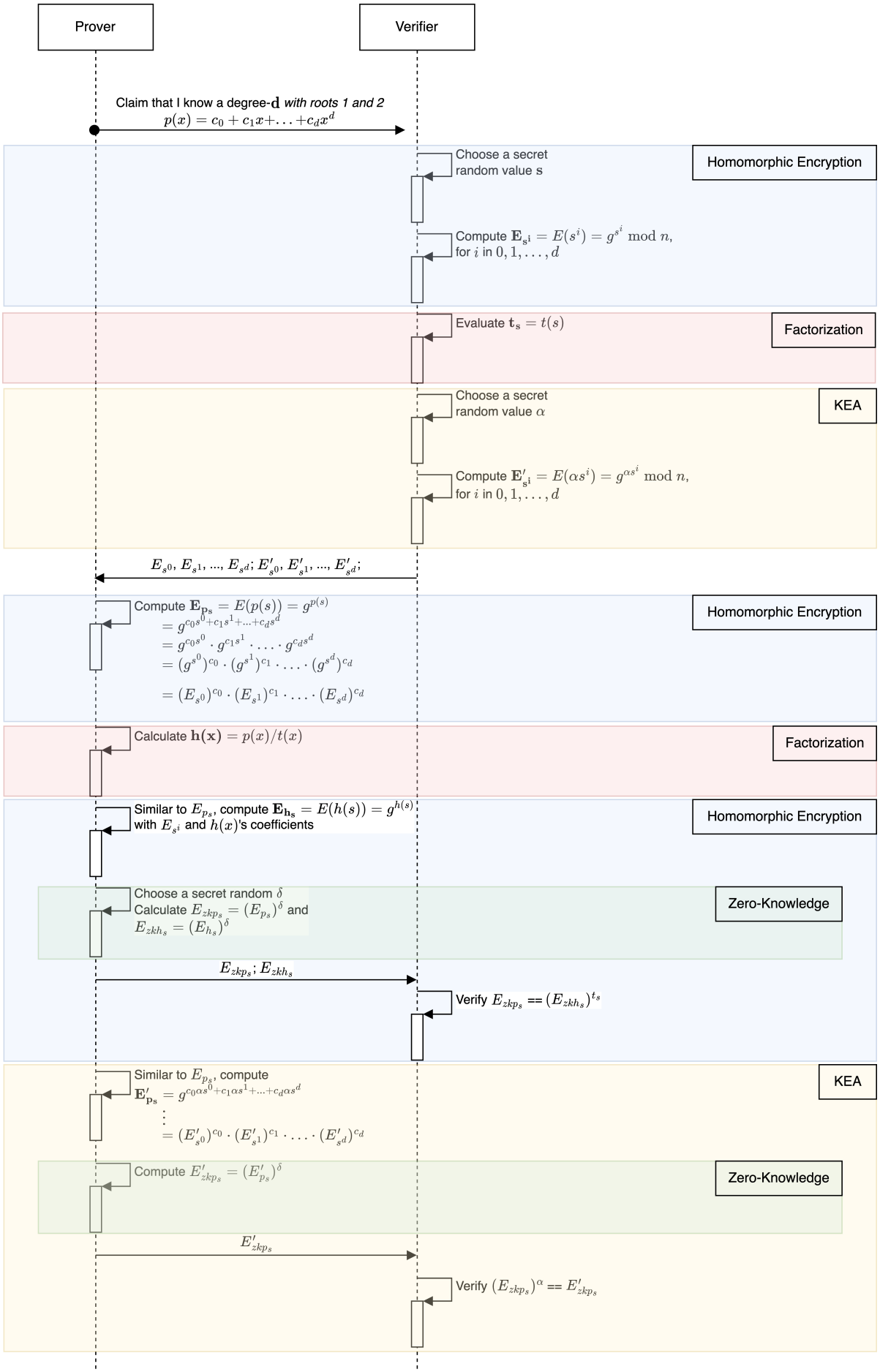


Figure 6: zk-SNARKs ZKP Sequence Diagram

The advantage of PLONK is that it supports a universal trusted setup, which means that the trusted setup can be performed once and used for any circuit. This is a major improvement over Groth16.

However, PLONK is not as efficient as Groth16 as it has a much slower performance and larger key and proof sizes. Taking our circuit as an example, with Groth16, generating the proving and verification keys took us about 2.5 minutes, and generating the proof took only around 15-20 seconds. The proving key size is 59MB. In contrast, with PLONK, it took us about 80 minutes to write 407 out 1163 Lagrange polynomials for the setup, and then the incomplete proving key exhausted all 33GB of free disk space on our machine...

Both Groth16 and PLONK are supported by *SnarkJS*, the tool we will use to generate the witness and proof. So, either of the two proving systems may be used in the project. However, due to the prohibitive cost of PLONK, we will just choose Groth16 in our implementation.

Implementing zk-SNARKs in the Project

To implement zk-SNARKs, we can define zk-SNARK constraints in the form of a circuit using the *circom language*, and then generate the witness and proof using *SnarkJS*.

Here are the general steps of the process:

A. Circuit definition and preparation for proving

1. Define the circuit in the circom language.
2. Compile the circom file to into a Rank-1 Constraint System (the .r1cs file) using the circom compiler. This step also generates the script for generating the witness.
3. Trusted Setup - Powers of Tau. With Groth16, the first phase, Powers of Tau, can be performed in advance. So, we just download the Powers of Tau file of the right size from a trusted source[15].
4. Trusted Setup - Phase 2. The phase 2 setup, however, is specific to the circuit. Ideally, it requires contributions from multiple trusted parties via MPC. In our demonstration, we just contribute our hardcoded entropy in the phase 2 setup. This step generates the proving key (.zkey file) and the verification key (verification_key.json).

B. Generating the witness and proof

1. Prepare the input data for the circuit.
2. Generate the witness by executing the witness generation script with the correct input data.
3. Prove the witness using the proving key, and obtain the proof with public inputs and outputs.

C. Verifying the proof

1. Verify the proof, public inputs and outputs using the verification key.

ZKP Workflow for zkKYC

In this section, we will illustrate the zero knowledge proof workflow using the following scenario:

- Issuer has a public DID `did_i` and has a key pair (`pub_i`, `priv_i`).
- Holder generates a peer DID `did_hi` to interact with Issuer.
- Verifier has a public DID `did_v`.
- Holder generates a peer DID `did_hv` to interact with Verifier.
- Government has a key pair (`pub_g`, `priv_g`).

Before the zkKYC process, the Holder has already registered with the Issuer. That is to say, the Issuer has the binding `{ did: "did_hi", real_name: "Evil Holder", passport_no: "A12345678" }` in its database.

1. Issuer signs the tuple `(did_i, did_hi)`.

When the Holder requests a KYC credential, the Issuer signs the tuple `(did_i, did_hi)` with its private key `priv_i`, and sends the signature `sig_i_hi` to the Holder in the credential. This conveys that the Holder is a recognized customer of the Issuer.

2. Holder generates the zkKYC token and proof for the Verifier.

When the Holder registers a business with the Verifier, the Holder generates a zkKYC token and proof for the specific Verifier.

To generate the zkKYC token, the Holder inputs `did_i`, `did_hi`, `did_hv`, `did_v`, `sig_i_hi`, `pub_i`, `pub_g` into the zkKYC token generation circuit. The circuit will perform the following steps:

- Verify `(did_i, did_hi)` with `pub_i` and `sig_i_hi`.
- Encrypt `(did_i, did_hi, did_hv, did_v)` with `pub_g` to obtain the *circuit output*, `encryptedPayload`.
- Include `did_hv`, `did_v`, `pub_i` and `pub_g` as *public inputs*.

Proof of the execution process is generated with the proving key from the trusted setup.

Then, the Holder sends public inputs, output and proof to the Verifier. These can be regarded as the zkKYC token and proof.

3. Verifier verifies the zkKYC token and proof.

When the Verifier receives the information from the Holder, it checks the following:

- Verify the public inputs and output with the proof using the verification key of the circuit, to ensure the enclosed information is valid.
- Check the public inputs `did_hv` and `did_v` to ensure that the zkKYC token is intended for the Verifier.
- Check the public input `pub_i` to ensure that the Holder is registered with a *recognized* Issuer.
- Check the public input `pub_g` to ensure that the `encryptedPayload` can be decrypted by the intended Government.

With this information, the Verifier cannot deduce any information about the Holder's identity beyond the Holder's verifier-dedicated DID `did_hv`, but can be assured that the intended party, Government, can decrypt the information for retrieving the Holder's real identity.

4. Government decrypts the zkKYC token.

When the Verifier spots suspicious activities of the Holder identified by `did_hv`, the Verifier reports the `encryptedPayload` to the Government for further investigation.

Government decrypts the `encryptedPayload` with its private key `priv_g`, and obtains the tuple `(did_i, did_hi, did_hv, did_v)`. Then, Government can - check that this is the zkKYC token of the suspicious Holder `did_hv` doing business with the reporting Verifier `did_v`, and - contact the Issuer `did_i` to obtain the Holder's real identity with `did_hi`.

Finally, Issuer can retrieve the Holder's real identity, `{ real_name: "Evil Holder", passport_no: "A12345678" }`, for the Government to take further actions.

Implementation Details

Now, we are clear about the workflow of the zkKYC process. In this section, we will discuss the implementation details of the workflow. ### DID Encoding By default, Circom signals are bounded by the field size of the curve, which is a 254-bit prime number. Therefore, for both convenience and efficiency, we decide to encode the DIDs in the form of an array of 248-bit (31-byte) signals. So, DIDs length limits will be multiples of 31 bytes, and the *multiple* is specified by the circuit construction parameter `n248Bits`.

Signing and Verification with EdDSA

The signing and verification of the tuple `(did_i, did_hi)` is done using EdDSA over the Baby Jubjub curve.

In practice, the signature is created on the *Poseidon Hash* of the encoded `did_i` and `did_hi` concatenated together. We have chosen the Poseidon hash function in favor of the well-known SHA-256 hash function, because the Poseidon hash function is a set of permutations over a prime field, which makes it more efficient for zk-SNARKs[16].

In contrast, the SHA-256 hash function is inefficient in our use case, and it also resulted in a large number of constraints in the circuits according to our experiments.

EdDSA Signing is performed in NodeJS using the *@iden3/js-crypto* library, and verification is performed in the circuit using the *circomlib* library.

The EdDSA signature consists of two elements, `R` and `S`. `R` is a point on the curve, and `S` is a scalar. We encode `R` as two 254-bit integers in an array, and `S` as a 254-bit integer.

In summary, the zkKYC token generation circuit takes the following inputs for signature verification:

```
signal input didI[248];  // encoded did_i
signal input didHI[248]; // encoded did_hi
signal input pubI[2];    // public key of Issuer
signal input sigS;       // S element of the EdDSA signature
signal input sigR[2];    // R element of the EdDSA signature
```

The circuit execution would fail if the signature is invalid.

Message Representation in ElGamal Encryption

Before we discuss the encryption and decryption process, we need to clarify the message representation in the ElGamal encryption.

Because we implement ElGamal encryption over the Baby Jubjub curve, what it encrypts is essentially a *point* on the curve. Therefore, we need a way to represent an arbitrary message with a point on the curve.

An existing solution is to pick a random point on the curve, and subtract the message from its x-coordinate[17] to get a `xIncrement` value. Then, the message can be represented by the point and `xIncrement`.

We improved this solution by performing XOR, instead of subtraction, between the lowest 253 bits of the x-coordinate and the message up to 253 bits, to get a `xmXor` value. Note that we do not utilize all 254 bits of the x-coordinate, because otherwise `xmXor` may overflow the BabyJub curve field size.

In the existing solution, we need to assert that the x-coordinate of the point is greater than the message, which can possibly fail, especially when the message integer is large. In our solution, the encoding can always succeed.

Also, our solution has the advantage that the representing point, which acts as the plaintext in the succeeding ElGamal encryption, can be *uniformly* chosen at random, while the existing solution cannot as its x-coordinate is lower-bounded by the message. This advantage increases the security level of the entire encryption process.

Token Encryption and Decryption with ElGamal and AES

This subsection describes how we achieve the public key encryption and decryption of the zkKYC token.

Encryption is performed in the circom circuit so that we can prove the proper encryption of the zkKYC token in the zk-SNARK proof, while decryption is done by the Government in any environment.

Briefly speaking, we first encrypt the payload with AES using a random symmetric key, and then encrypt the symmetric key with ElGamal using the Government's public key.

AES Encryption

As mentioned in the previous subsection, plaintext messages in ElGamal encryption are represented by a point and an `aesKeyXmXor` value. As the AES key will be later encrypted with ElGamal, it is input to the circuit as a point `aesKeyPoint`, and a *public* input value `aesKeyXmXor`. Therefore, the first step is to obtain the AES key by XORing the `aesKeyXmXor` value with the lowest 253 bits of the x-coordinate of the point.

The circuit inputs also include the initial vector `iv` for AES, which is a randomly generated array of 128 bits.

After obtaining the AES key, we encrypt the payload (`did_i`, `did_hi`, `did_hv`, `did_v`) with the key and `iv`, into a ciphertext. The `iv` is then appended to the ciphertext, and the resulting bit array is the `encryptedPayload`.

To encrypt the payload, we use the AES-256-CTR cipher, and the implementation is adapted from *Electron-Labs/aes-circom*[18], where they implemented the AES-GCM-SIV.

We did not choose AES-GCM-SIV due to its computational complexity, and we do not require the integrity guarantee it provides. We just use AES-256-CTR circuit in their repository, and we modified it to match the bit endianness we are using in the rest of the project.

ElGamal Encryption

After AES encryption, we encrypt the AES key with ElGamal using the Government's public key. With the AES key represented by `aesKeyPoint` and `aesKeyXmXor`, we just apply the ElGamal encryption on the `aesKeyPoint`, and output the ElGamal ciphertext represented by two points `c1` and `c2`.

The random number required by ElGamal encryption is taken as an input signal `elGamalR`.

In summary, after the encryption process, the circuit takes the following inputs:

```
signal input didI[n248Bits];  // encoded did_i
signal input didHI[n248Bits]; // encoded did_hi
signal input didHV[n248Bits]; // encoded did_hv
signal input didV[n248Bits];  // encoded did_v
signal input aesKeyPoint[2];  // the point in AES key representation
signal input aesKeyXmXor;     // the xmXor value in AES key representation
signal input aesIV[128];      // the 128-bit AES initial vector
signal input govPubKey[2];    // public key of Government
signal input elGamalR;        // the random number for ElGamal encryption
```

and generates the following public information that Government can use to decrypt the zkKYC token:

```
[
  c1X, c1Y,             // circuit output
  c2X, c2Y,             // circuit output
  ...encryptedPayload,  // circuit output
  aesKeyXmXor           // circuit public input
]
```

We implemented the ElGamal encryption algorithm with the BabyJub curve in the circom circuit, in the following steps:

- Compute the **c1** point by multiplying the base point by `elGamalR`.
- Compute the shared secret point **s** by multiplying the public key point by `elGamalR`.
- Add the message point to the shared secret point to get the **c2** point.

Decryption Process

Using the above public information, the Government can decrypt the zkKYC token by the following steps:

- Decrypt `[c1X, c1Y, c2X, c2Y]` with the Government's private key to obtain the AES key point `aesKeyPoint`.
- XOR the lowest 253 bits of the x-coordinate of `aesKeyPoint` with `aesKeyXmXor` to obtain the AES key.
- Decrypt `encryptedPayload` with the AES key to obtain the payload (`did_i`, `did_hi`, `did_hv`, `did_v`).

The ElGamal decryption algorithm is implemented in the JavaScript with the BabyJub curve, in the following steps:

- Compute the shared secret point **s** by multiplying the `c1` point by the private key.
- Compute the inverse of **s** according to the specification of the BabyJub curve.
- Add the `c2` point to the inverse of **s** to get the message – `aesKeyPoint`.

Bit Lengths of Components — 248, 253, 254 or 256?

You may have noticed that we use various bit lengths in the project implementation. Those values are carefully chosen to maximize the security of the system, while avoiding overflow. In this subsection, we will summarize bit lengths for the components and explain the rationale behind those choices.

Generally, signals in a Circom circuit should be bounded by the finite field that the underlying elliptic curve is defined on. In our case, the *Baby Jubjub Elliptic Curve* is defined over the field F_p, where p is a *254-bit* prime number[19]. Given this fact, we define the bit lengths for the following components in the project:

DID Representation (248 bits): A DID is represented by an array of 248-bit unsigned integers. Each integer can be interpreted as a 31-byte string.

AES Key (253 bits): We use a 253-bit key for AES-256 encryption, instead of the standard 256-bit key. This is because this key will be encoded into a point on the Baby Jubjub curve and an XOR value as discussed in an earlier subsection. Therefore, as the point coordinate is bounded by a 254-bit number, we need to limit the key size to 253 bits to avoid overflowing the XOR value.

XOR Value (253 bits): The XOR value, named `xmXor`, is used to encode the AES key into a point on the Baby Jubjub curve. It is the result of XORing the 253-bit AES key with the lowest 253 bits of the x-coordinate of the encoding point `aesKeyPoint`.

ElGamal random value `r` (253 bits): The random value `r` is a signal in the ElGamal encryption circuit. Signals are bounded by a 254-bit number. When generating the `r`, however, we use a 253-bit random number to avoid overflowing the signal.

Baby Jubjub Point (254 bits): A point on the Baby Jubjub curve is represented by two 254-bit unsigned integers or buffers. Public keys, `R` in the EdDSA signature, the encoding point `aesKeyPoint`, "shared secret" `s` in ElGamal encryption, and the `c1` and `c2` of the ElGamal ciphertext are all Baby Jubjub points.

Private Key (256 bits): We use a 32-byte buffer as the private key for the EdDSA signature. Internally, EdDSA derives a 253-bit scalar from the private key, which is used for the signature generation process.

Private Key Scalar (253 bits): A 253-bit scalar derived according to the Baby Jubjub EdDSA signature scheme. It is also used as the private key for the ElGamal decryption.

AES Circuit Input (256 bits): As the AES-256 algorithm requires the size of the input to be the multiple of the unsigned integer size, we pad input units (integers representing DIDs) to 256 bits with trailing zeros. The 253-bit AES key is also padded to 256 bits with trailing zeros before inputting into the AES circuit.

Interfaces and Packaging

The ZKP part of this project is relatively independent of the SSI part. So, we group the ZKP actions by roles and define the interfaces as follows:

Issuer

```
generateKeyPair(privKey?: string) => { priv: string, pub: string[] }
```

Generate a key pair for the issuer. If `privKey` is provided, the key pair will be derived from the private key. Otherwise, a random private key will be generated.

```
signDidRecord(didI: string, didHI: string, privKey: string) => { s: string, r:
↪  string[] }
```

Sign the DID record with the private key.

Holder

```
generateZkKycProof(didI: string, didHI: string, didHV: string, didV: string, sigS:
↪  string, sigR: string[], issuerPubKey: string{[]{[]}, govPubKey: string{[]{[]}) => {
↪  proofJson: string, publicJson: string }
```

Generate the zkKYC proof and public information, including the encrypted contents and public keys.

Verifier

```
parsePublic(publicJson: string) => { aesKeyPointCipher: { c1: string[], c2: string[]
↪  }, encryptedPayload: string, didHV: string, didV: string, issuerPubKey: string[],
↪  govPubKey: string[], aesKeyXmXor: string }
```

Parse the public information generated by the Holder into an object. The parsed object can be easily read by the Verifier and passed to the government for decryption.

```
verifyZkKycProof(proofJson: string, publicJson: string) => boolean
```

Verify the zkKYC proof and public information to check if the information is valid.

Government

```
decryptToken(parsedPublic: string, privKey: string) => { didI: string, didHI: string,
↪  didHV: string, didV: string }
```

Decrypt the zkKYC token with the Government's private key to obtain the DIDs in the token.

```
generateKeyPair(privKey?: string) => { priv: string, pub: string[] }
```

Generate a key pair for the issuer. If `privKey` is provided, the key pair will be derived from the private key. Otherwise, a random private key will be generated.

These interfaces are packaged into a NodeJS module **zkkyc-js** and can be imported into the SSI controller written in NodeJS. For this method, we strongly suggest using separate *worker threads* to run the zkKYC actions to avoid blocking other tasks in the SSI controller thread, because the ZKP process is very time-consuming and NodeJS is single-threaded.

We also implemented GRPC services for these interfaces to facilitate the integration with SSI controllers written in other languages. You can start the GRPC server by running `npm run grpc` in the **zkkyc-js** directory.

Project Review

Business Requirements Review

This section provides a review of how the business requirements mentioned in the Solution Overview are met by the zkKYC solution.

Requirement ID	Approach to Meet the Requirement
BR01	We are just added the *zkKYC* token generation and verification process to the existing SSI solution. Therefore, all properties provided by SSI are preserved.
BR02	1. Personal identifiable information is stored in the Issuer's database. What is shared with the business (Verifier) is an encrypted token containing the information for the Issuer to retrieve the information of the given customer. Therefore, the user does not need to share personal identifiable information for KYC with the Verifier. There is also no way for the Verifier to retrieve the customer's information from the Issuer as the necessary information for doing so can only be decrypted by the Government.
BR03	This can be achieved by existing SSI solutions with verifiable credentials. Aries Framework can handle signing and verifying such credentials.
BR04	The business (Verifier) can report the zkKYC token to the Government. The Government can decrypt the token with its private key and take further actions.
BR05	Same as BR04.
BR06	The Government can decrypt the zkKYC token with its private key and get the Holder's DID registered in Issuer's database. Then, it can retrieve the Holder's real identity from the Issuer specified by the Issuer DID, which is also in the zkKYC token.
BR07	The zkKYC token is stored in the Verifier's secure storage and can be directly passed to the Government without notifying the Holder.
BR08	What the business holds is just an encrypted token dedicated to this business. Only the Government can decrypt the token.

Security Review

In this section, we will discuss how our implementation of the zkKYC solution can protect users' privacy against the following threats:

Signature Correlation Signature correlation means that different Verifiers may collude to correlate the Holder's identity via identical signatures from the same Issuer for the Holder. This is prevented in our solution, because the signature verification is performed by the ZKP system in the circuit. The Verifiers will not get the signatures from the Issuer, but just a ZKP proof that the information is signed by the Issuer controlling a given public key.

Verifier Abusing KYC Data In traditional KYC, the Verifier may be able to abuse the information provided for KYC purposes. However, in our system, the Verifier only gets an encrypted token, without any personal information of the customer. The token is generated by the Holder specifically for this business. Therefore, it is also meaningless for other businesses.

Holder's Credentials Stolen If someone steals the Holder's credentials, they may be able to impersonate the Holder to generate zkKYC tokens for other businesses. Firstly, the credentials are stored in the

Holder's secure storage of the wallet, which should generally be safe. Secondly, in case the attacker gets access to the Holder's wallet, the Holder can contact the Issuer to revoke the credentials.

Data Leakage In traditional KYC, the Holder's personal information is stored in the every business they have conducted KYC with, and thus leads to a significant risk of data leakage. If one of the business is compromised, the user's data is threatened. In our solution, businesses do not store user's personal data, and the KYC information is only meaningful for dedicated businesses. Therefore, as long as the Issuer is secure, the Holder's data is safe. This significantly reduces the risk of data leakage threatening the user privacy.

Limitations

- For the sake of simplicity, we did not implement a full-fledged SSI system, because our focus is to demonstrate the possibility of integrating the zkKYC idea into an existing SSI system. In reality, we can find many more completed SSI solutions.
- The payload of the zkKYC token is limited to a small size. Limited by the efficiency of the proving system, we can only encrypt a small amount of data in the zkKYC token. In our project, the limit is four 31-byte DIDs, 124 bytes in total, and that will already take around 15 seconds to generate the zkKYC token.
- The zero-knowledge proof requires us to perform trusted setup by ourselves. Ideally, this should be performed by a group of reputable parties. Therefore, it may be difficult to apply the zkKYC solution with the current proving system in practice and get trusted by the public.

Future Work

Optimize Implementation For simplicity, we did not consider the performance and efficiency of the implementation in this project. We can identify several areas for optimization:

- Compared to ElGamal encryption, ECDH is more efficient and secure. We can use ECDH to derive the shared secret for symmetrically encrypting the DIDs payload.
- AES-256 encryption results in large circuits. For instance, our circuit uses the plain AES-256-CTR circuit to encrypt just 32 bytes of data, but has more than 120,000 constraints and generates a 21.7MB R1CS file! We should find some *zk-friendly* alternatives for token symmetric encryption.
- For clearer demonstrations, we used the uncompressed form of Baby Jubjub points in the project (two 254-bit integers). In practice, we can pack a point into one single 255-bit integer to improve efficiency.

Adapt to a Common Elliptic Curve We use the Baby Jubjub curve in this project, which is a special curve designed for zkSNARKs and the only reliable curve available at the moment. However, it is not widely used in practice. We may want to consider adapting the project to a more common elliptic curve, such as the Ed25519 curve. As a prerequisite, this will need an *efficient* circom implementation of the curve.

Explore Other Proving Systems We use the Groth16 proving system in this project. However, we may want to consider other proving systems in practice, because Groth16 requires a trusted setup for each circuit, and this may not always be practical. Further study and research are needed to find the most suitable proving system for zkKYC.

Integrate into DeFi Protocols This could be the most exciting potential of the project, and has also been discussed in the succeeding zkKYC paper – *zkKYC in DeFi*[1]. This project can be easily extended to integrate with DeFi protocols, as Circom can generate Solidity smart contracts for proof verification. Then, a DeFi protocol can include the zkKYC verification process in its smart contract, and require users to submit a valid zkKYC proof before they can use the protocol. This will enable DeFi protocols to provide a more trustful and still privacy-preserving service to authenticated users.

Conclusion

In conclusion, we have developed a privacy-preserving KYC system, using the zkKYC solution concept, which leverages self-sovereign identity and zero-knowledge proof. To provide an implementation for the solution concept, we designed and built a ZKP system with the Groth16 proving system using Circom and SnarkJS, for generating and verifying the zero-knowledge proof of user identity. Finally, we integrated the ZKP system into the existing SSI framework to demonstrate the workflow of the entire privacy-preserving KYC system.

References

[1] P. Pauwels, J. Pirovich, P. Braunz, and J. Deeb, *Zkkyc in defi: An approach for implementing the zkkyc solution concept in decentralized finance*, Cryptology ePrint Archive, Paper 2022/321, https://eprint.iacr.org/2022/321, 2022. [Online]. Available: https://eprint.iacr.org/2022/321.

[2] Sovrin Foundation, *What is self-sovereign identity?* https://sovrin.org/faq/what-is-self-sovereign-identity/, accessed March 27, 2023.

[3] Linux Foundation. "Getting started with self-sovereign identity," edX. (2022), [Online]. Available: https://www.edx.org/course/getting-started-with-self-sovereign-identity.

[4] World Wide Web Consortium, "Decentralized identifiers (dids) v1.0," W3C, W3C Recommendation, Jul. 2022. [Online]. Available: https://www.w3.org/TR/did-core/.

[5] World Wide Web Consortium, "Verifiable credentials data model 1.0," W3C, W3C Recommendation, Mar. 2022. [Online]. Available: https://www.w3.org/TR/vc-data-model/.

[6] The Linux Foundation. "Becoming a hyperledger aries developer," edX. (2021), [Online]. Available: https://www.edx.org/course/becoming-a-hyperledger-aries-developer.

[7] Hyperledger Foundation, *Hyperledger aries cloud agent python*, https://github.com/hyperledger/aries-cloudagent-python, 2023.

[8] M. Petkus, "Why and how zk-snark works," *CoRR*, vol. abs/1906.07221, 2019. arXiv: 1906.07221. [Online]. Available: http://arxiv.org/abs/1906.07221.

[9] J. Groth, *On the size of pairing-based non-interactive arguments*, Cryptology ePrint Archive, Paper 2016/260, https://eprint.iacr.org/2016/260, 2016. [Online]. Available: https://eprint.iacr.org/2016/260.

[10] S. Li. "Zkp - intro to groth16 (chinese)." (2019), [Online]. Available: https://learnblockchain.cn/2019/05/27/groth16/.

[11] M. Binello. "Groth16." (2019), [Online]. Available: https://www.zeroknowledgeblog.com/index.php/groth16.

[12] A. Pruden and A. Matlala, *Setup ceremonies*, https://zkproof.org/2021/06/30/setup-ceremonies/, Accessed: March 27, 2023, Jun. 2021.

[13] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, *Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge*, Cryptology ePrint Archive, Paper 2019/953, https://eprint.iacr.org/2019/953, 2019. [Online]. Available: https://eprint.iacr.org/2019/953.

[14] Unknown. "Understanding plonk." (2019), [Online]. Available: https://vitalik.ca/general/2019/09/22/plonk.html.

[15] iden3, *Snarkjs*, https://github.com/iden3/snarkjs, 2023.

[16] A. Bakhta, E. B. Sasson, A. Levy, and D. L. Gurevich, *Eip-5988: Add poseidon hash function precompile*, https://eips.ethereum.org/EIPS/eip-5988, Accessed: March 27, 2023, 2022.

[17] K. W. Jie, *Elgamal encryption, decryption, and rerandomization, with circom support*, https://ethresear.ch/t/elgamal-encryption-decryption-and-rerandomization-with-circom-support/8074, Accessed: March 27, 2023, Oct. 2020.

[18] Electron Labs, *Aes-gcm implementation in circom*, https://github.com/Electron-Labs/aes-circom, 2022.

[19] J. B. Barry WhiteHat Marta Bellés, *Erc-2494: Baby jubjub elliptic curve*, https://eips.ethereum.org/EIPS/eip-2494, Accessed: March 27, 2023, 2020.