J-Card$^+$: Zero-Knowledge Identity using Iden3

Capstone Project
Final Project Report
Fall 2022

Anshul Singhal, Liyin Li, Pratik Kayastha (Team ZKSnacks)
Capstone Mentor: Dr. Matthew Green

Abstract

Most existing authentication processes are third-party oriented. Maintaining duplicate information in a centralized manner is asserted to be a necessity. In this project, *J-Card*$^+$, an identity management wallet application, proves that verifying access rights requires neither data replication nor arbitration authority. Constructing upon a novel identity access control protocol proposed by Iden3, it shows that authentication can be self-sufficient between a prover and a verifier. It preserves better privacy such that identity information is individually maintained, minimally distributed, and confidentially verified using zero-knowledge proofs. Serving as a building block of *J-Card*$^+$, Wallet SDK and Issuer SDK are also built and delivered. To examine *J-Card*$^+$'s security and scalability, the tactics of threat modeling and performance testing are applied. Supported by the findings of *J-Card*$^+$ work, we argue that security need not be paid with the price of privacy. The proposing Iden3-based identity management solution is a prospective and rigorous alternative for third-party driven authentication systems.

1 Introduction

1.1 Significance

The proof of access rights contributes to an essential part of modern social functioning. Most online application accesses are third-party authorized, and the governance of verifying information is never user-centric. Evaluating such systems from a software engineering and a computer security perspective is neither maximizing system efficiency nor concerning best security practices. On the one hand, duplicate copies of information must be maintained (on the server side) for verification during authentication. On the other hand, the principle of least privileges is failed to en-place upon users' data concerning from the owner's angle. However, it is an essential asset for the overall system.

1.2 Objectives

Empowered by modern cryptography, this project aims to engineer privacy through designs. A zkWallet SDK based-off Iden3 [1], an innovative identity management protocol, was developed. It was also integrated into the *J-Card*$^+$, a proof-of-concept online application that targets the Johns Hopkins community. By its design nature, all personal user information is stored locally only on the user's device. And the process of identity verification of authorization is zero-knowledge, meaning that it is the access right being proved but the leverage of person-hood disclosure.

1.3 Definition

Under this project's scope, the authentication problem is characterized by three agents: 1) an identity Issuer, 2) an identity Holder, and 3) an identity Verifier. Corresponding relations and interactions can be visualized in figure 1. An Issuer can create claims for an identity while the Holder manages their identity claims. In requests by a Verifier, a Holder generates and sends associate identity proof based on the respective claim that has private data concealed.

The property that individual claims are restrictively distributed and controlled by the identity owner is said to be self-sovereign, namely, Self-sovereign Identity (SSI). To further experiment how SSI helps revolutionize social interactions in decentralized autonomous organizations (DAOs), this project delivers *J-Card*$^{+}$ application, which layouts the potential use cases in the Johns Hopkins University (JHU) community. A JHU student can govern their identity claims issued by the school administration office. And one's identity can be validated by generating and verifying 1) the proof of personhood in school events and/or 2) the proof of membership in a student organization, constructed upon the Zero-knowledge Proofs (ZKPs) cryptographic primitive.

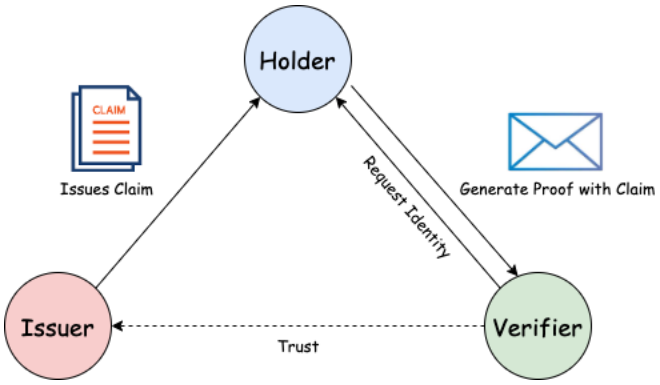

Figure 1: Trust triangle between the identity Issuer, Holder, and Verifier

2 Literature Review

This section provides the necessary background to understand zero-knowledge identity based on the iden3 framework. We first describe the role of zero-knowledge proof (§2.1) in developing software with privacy in mind. Then we describe different components such as zkSNARK (§2.2), Circom (§2.3), BabyJubJub Elliptic Curve (§2.4), Poseidon Hash (§2.5) and Sparse Merkle Tree (§2.6) used by the Iden3 to create a framework to develop privacy-focused blockchain software.

2.1 Zero-knowledge Proof

A Zero Knowledge Proof (ZKP) is a protocol that allows one party, known as the prover, to convince another, known as the verifier, that a statement about certain data is true without leaking or disclosing any additional information about the statement apart from the fact that statement is indeed true. The property of proving that someone possesses knowledge about specific data without revealing the data itself is desirable for privacy-focused blockchain software. For example, one can prove the following statements by creating zero-knowledge proofs for statements [2].

- "I know a private key associated with this public key": in this case, the prover does not reveal any information about the private key it is holding.
- "I know a pre-image of this hash value": in this case, the proof does not reveal any information about the pre-image value, but it still convinces the verifier that the prover knows the pre-image.
- "The transaction privately transfers a coin": in this case, the proof would not disclose any information about transaction properties such as origin, destination, or amount, but it guarantees that the coin has not been double spent.

Zero-knowledge proofs are classified into interactive zero-knowledge (ZK) and non-interactive zero-knowledge proofs. As the name implies, interactive ZK proofs require interaction between two parties, the prover (proving their knowledge of specific data) and the verifier (validating the proof). The interaction would be in the form of a challenge response. The verifier challenges the prover, and the prover responds without revealing any additional information other than the statement. The more correct responses the verifier receives from the prover, the less likely the prover is to lie. Non-interactive

Types of SNARKs				
	Size of Proof π	Size of S_p	Verification Time	Trusted Setup
Groth'16	O(1)	O($\|C\|$)	O(1)	yes (per circuit)
Plonk	O(1)	O($\|C\|$)	O(1)	yes (universal setup)
Bulletproofs	O($\log \|C\|$)	O(1)	O($\|C\|$)	no
STARK	O($\log \|C\|$)	O(1)	O($\log \|C\|$)	no
DARK	O($\log \|C\|$)	O(1)	O($\log \|C\|$)	no

Table 1: Comparison between different types of SNARKs

ZK proofs, on the other hand, are generated once on the prover's side and then sent to the verifier for verification. Non-interactive ZK-proof techniques such as zkSNARK and zkSTARK have been used in recent technologies. (Add references of zcash, zkEVM, etc.). We limit our scope in this project to the zkSNARK protocol because the Iden3 framework uses it [1]. We examine what zk-SNARK is and its desired characteristics in the following section.

In this project, we focus our scope on only the zkSNARK protocol because, under the hood, the Iden3 framework uses it.

2.2 zk-SNARK

ZK succinct non-interactive arguments of knowledge (zk-SNARKs) are one of the most efficient, popular, and general-purpose zero-knowledge protocols for generating proof [3, 4]. It has gained popularity due to its application in ZCash [5], a public blockchain based on bitcoin that uses this proof to ensure that parties involved in the transaction are verified without revealing any information about the parties to each other or the network. The zero-knowledge proof has the following properties, according to SNARK.

- **Succinct** - the size of the proof is minimal (around 200 bytes) regardless of the size of the statement or the witness
- **Non-interactive** - it does not require challenge-response communication between the prover and verifier
- **Argument** - we consider it secure only for provers with limited computational resources, which means that provers with sufficient computational power can persuade the verifier of an incorrect statement
- **Knowledge-sound** - The prover cannot construct proof without knowing a specific so-called witness for the statement

To create a zero-knowledge proof, we primarily focus on the short and non-interactive proof generation characteristics of zk-SNARKs. These characteristics make it very easy to create smart contracts for distributed ledgers that can execute code, such as Ethereum, to verify zero-knowledge proofs in a concise period of time. There are numerous tools available to use zk-SNARK, including circom, zokrates, zinc, snarky, and others. Most tools require developers to create arithmetic circuits for the use case to generate zkSNARK proofs. The main disadvantage of the zk-SNARK is that it necessitates an initial phase known as a trusted setup. The steps include generating random values to obtain a "proving key" and a "verifying key" that must be destroyed. If these random values are leaked, the security of the entire zk-SNARK protocol is jeopardized. However, new protocols, such as zk-STARK, are being proposed that do not require a trusted setup. We will concentrate on the zk-SNARK protocol for this project because the circom language uses zk-SNARKs to generate circuit proofs.

Table 1 shows different time complexity for different implementations of SNARKs. As we can observe from the table, Groth'16 and Plonk/Martin SNARK implementation can be efficient in terms of proof size and verification because they do not depend on the circuit, but proof generation takes linear time.

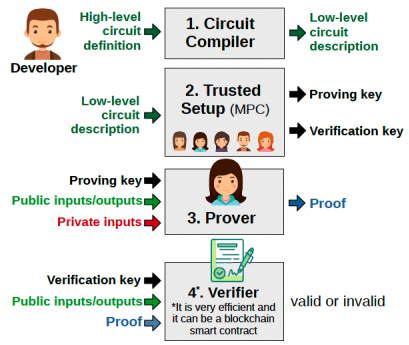

Figure 2: Generate and verify ZK proofs using Circom [2]

```
1  pragma circom 2.0.0;
2
3  /*This circuit template checks that c is the multiplication of a and b.*/
4
5  template Multiplier2 () {
6
7     // Declaration of signals.
8     signal input a;
9     signal input b;
10    signal output c;
11
12    // Constraints.
13    c <== a * b;
14 }
```

Listing 1: Multiplication Code in Circom - multiplier2.circom [6]

2.3 Circom 2.0

Circom is a newly developed domain-specific language (DSL) to define arithmetic circuits, and its associated compiler written in Rust [6]. The circom's primary goal is to give programmers a comprehensive framework for building arithmetic circuits using a simple user interface while abstracting the complexity of the proving procedures [7]. While zk-SNARKs, like most ZKPs, allow the proof of computational assertions, they cannot be used to solve the computational problem directly; instead, the statement must be transformed into the appropriate form. An arithmetic circuit must represent the computational statement for zk-SNARKs. A circom compiler can therefore be used to produce a zero-knowledge proof from an arithmetic circuit.

The figure (2) depicts the general steps for creating and validating proof using circom. Each step will be explained with the assumption that we want to prove the statement "I can factor number 33." To begin, we must create a program in arithmetic circuits using circom. To demonstrate the preceding statement, we will build the multiplication circuit shown in Listing (1). In line 5, we create a template or, as we commonly refer to it, a function in languages like Python, Java, C++, etc. In lines 8 & 9, we define private inputs or factors of some number. At 10, we define the output of the arithmetic circuit, which is a multiplication of two factors, as shown in line 13. First, we will take this higher-level code written in circom, compile it using circom compiler and convert it into lower-level language files with extension R1CS and Wasm format by running the following command.

circom multiplier2.circom –r1cs –wasm –sym –c

R1CS contains optimized constraints in quadratic, linear, or constant equations defined in higher-level circom code. The WASM file contains data to generate witnesses for zero-knowledge proofs. Second, once we get a low-level circuit description (r1cs & wasm files), we need to create a proving and verification key by performing an initial phase called trusted setup. For simplicity, we are not discussing different utilities developed by circom to create these keys, but https://docs.circom.io/getting-started/proving-circuits/ has a tutorial on creating them. Third, once we obtain

these keys, we create proof using the proving key, private inputs (in our example cases 11 & 3), and public input/outputs. In the Last step, the verifier can verify the proof by verification key, public input/output, and proof. The verifier can be a web2 application or a smart contract on the blockchain.

2.4 BabyJubJub Elliptic Curve

Baby JubJub is an Elliptic curve optimized for use with zk-SNARKs [8]. Traditionally, public and private key pairs are used to authenticate users. The public and private keys can be generated by performing addition or multiplication operations on the Elliptic curve. The public and private key pairs are used in the Iden3 Protocol to manage identity and authenticate in the name of identity. As a result, we must demonstrate ownership of the public key without revealing the private key. To put it another way, we must create an arithmetic circuit that verifies that a given secret key corresponds to a given public key. The BabyJubJub elliptic curve makes it possible to build this circuit quickly.

Definition of parameters of curve
Baby JubJub is an elliptic curve defined over a prime field F_p with

$$\text{p} = 18882428718392752222464057452572750885483644004160343436982041865758084956 17$$

and described with the equation,

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{1}$$

where a=168700 & d=68696

For simplicity, we will not discuss how circom implements different elliptic curve operations, such as multiplication and addition of points on the curve. We will use those circuits as black box circuits, but the definition of those circuits can be found in circomLib [9].

2.5 Poseidon Hash

A deterministic one-way function known as a cryptographic hash function converts data of any size to data of a specific size. They are mainly employed to recognize and validate the integrity of files, papers, and other kinds of data. Because they rely so heavily on bit operations, traditional hash algorithms like SHA256 are particularly ineffective when used in ZK protocols to create arithmetic circuits. For instance, the SHA256 implementation in circom includes almost 59,000 limitations. On the other hand, Poseidon hash construction offers a performance advantage in addition to being compactly stated as a circuit that can be customized for different proof systems using specifically created polynomials [10]. The Poseidon hash function is frequently used in the iden3 framework with Baby JubJub Elliptic curve, Merkle tree, and any other use cases needing a secure hash function.

2.6 Sparse Merkle Tree

A Merkle Tree or a hash tree is a tree data structure in which each leaf node contains the cryptographic hash of some data, and each non-leaf node contains the cryptographic hash of its child nodes. Merkle trees enable the linking of a set of data to a unique value, which is very efficient and useful in blockchain technology because it provides a secure and efficient way to verify large sets of data by storing only a small amount of information (the Merkle tree's root) on the blockchain. If most of the leaf nodes in a Merkle tree are empty, it is called a "Sparse Merkle Tree." The Merkle trees used in the Iden3 framework are Sparse Merkle trees [11]. Moreover, in sparse Merkle trees, each data block has an associated index that indicates its position as a leaf inside a tree. The sparse Merkle trees used in the Iden3 protocols have the following properties:

- **Binary** - Each node can only have two children
- **Sparse and Deterministic** - The contained data is indexed, and each data block is placed at the leaf corresponding to that data block's index, so insert order does not influence the final Merkle tree Root. This also means that some nodes are empty.
- **ZK friendly** - The hash function used for the Merkle tree, Poseidon, plays well with the zero-knowledge proofs (ZKP) used in different parts of the protocol.

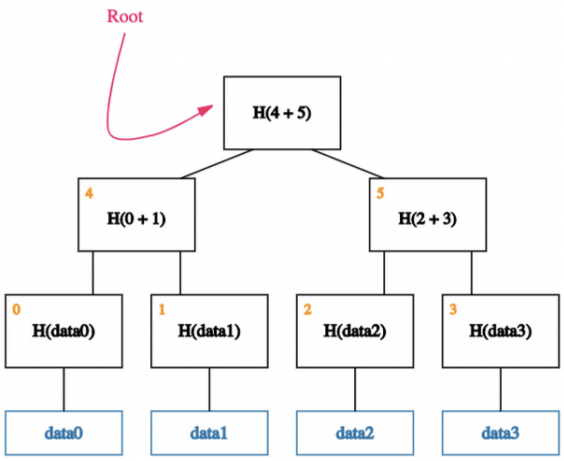

Figure 3: Merkle Tree
Source: https://docs.iden3.io/basics/key-concepts

Figure (3) shows what a Merkle tree looks like. Apart from inheriting temper-resistance, proof of membership, and scalability features from the Merkle tree, the sparse Merkle tree also provides the feature of proof of non-membership. It is easy to prove "proof non-membership" in sparse Merkle trees because most of the leaf node contains null data (empty). It is equivalent to proving the membership of a null data block.

At Iden3, sparse Merkle trees allow scalability. Any identity or user should be able to generate as many claims as he/she wants. Achieving this goal requires minimizing the amount of data stored on-chain. Sparse Merkle Tree provides a way to achieve that goal by storing only the root node on the chain. In other words, Merkle trees allow prolific claim (a statement about something) generators to add/modify millions of claims in a single transaction. This makes it easy to scale the claims. We discuss more claims in the Iden3 section of the report.

3 Technical Solution, Design, and Analysis

3.1 Iden3 Overview

To understand Iden3, we first have to understand a few terminologies.

- **Identity** - In the blockchain world, when we talk about identity, we mean accounts, which will be smart contracts. So you can think of identities as smart contracts, where the address of the contract is the identifier of that identity [12].
- **Claims** - Claims are statements said by an identity about itself or another identity. Claims can be public or private. For example, "I" as an identity on the blockchain can claim that I am a JHU student.

Now that we understand the terminologies, it brings us to the question, what is Iden3 providing us? Iden3 is a protocol to create and maintain identities and claims with the following properties:

- Scalability
- Temper resistance
- Zero-Knowledge verification.

The Iden3 organization has implemented libraries for their new protocol, mainly in Go.

3.2 Iden3 Protocol Specification

As the introduction states, Iden3 is a protocol to create and maintain identities and claims. Let's see how we store these components, what operation we can perform, and how the Iden3 protocol guarantees the above properties, i.e., Scalability, temper resistance, and ZK-verification.

3.2.1 Claims

3.2.1.1 Claim Data Structure

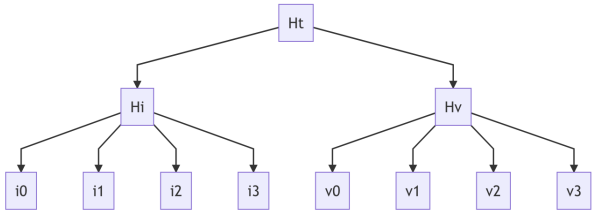

Figure 4: Claim Data Structure
Source: https://docs.iden3.io

Each claim is composed of two parts: the index part and the value part. The Indexes are i0, i1, i2, and i3, and the Values are v0, v1, v2, and v3. The protocol reserves i0, i1, and v0, v1 for its internal use; therefore, they cannot be utilized to store other information. The following defines these reserve slots in the claim data structure.

```
i_0 [253 bits]:[128 bits] claim schema hash(see below)
               [32 bits] header flags
                 [3] Subject:
                   000: A.1 Self
                   001: invalid
                   010: A.2.i OtherIden Index
                   011: A.2.v OtherIden Value
                   100: B.i Object Index
                   101: B.v Object Value
                 [1] Expiration: bool
                 [1] Updatable: bool
                 [27] 0
               [ 32 bits ] version (optional?)
               [ 61 bits ] 0 - reserved for future use


i_1 [253 bits]:[248 bits] identity (case b) (optional)
               [5 bits] 0

v_0 [253 bits]:[ 64 bits ]  revocation nonce (see below)
               [ 64 bits ]  expiration date (optional)
               [ 125 bits] 0 - reserved
v_1 [253 bits]:[ 248 bits] identity (case c) (optional)
               [  5 bits ] 0
```

Figure 5: Bit Structure

Note that the hash of the index slots (hash(i_0,i_1,i_2,i_3)) is unique for the whole tree. That means you cannot have two claims with the same index inside the tree. This property is critical; this is how we decide if the data corresponding to the claim will be stored in index slots(i2, i3) or value slots(v2, v3).

3.2.1.2 Claim Schema

Claim Schema is a JSON-LD document that defines which data should be stored inside which data slots(i2, i3, v2, v3). The Hash of these schemas is stored in the i0 slot when we create a claim.

3.2.2 Keys

In Iden3 or any cryptographic protocol, keys are used to authorize and authenticate new transactions. In Iden3, a transaction would be creating/updating/revoking claims; thus, keys are required to authorize these transactions. In Iden3, we use Baby Jubjub(BJJ) Elliptic Curve algorithm to generate public and private keys. This type of key is designed to be efficient while working with zkSNARK.

3.2.2.1 Auth Claim or Key Authorization Claim

Auth claim is a claim which claims that you own a private key or a public key. In other words, as an identity, you claim that you hold the BJJ private key of this BJJ public key stored in the claim. This claim is very important because whenever you issue a new claim, you need to prove that you are the owner of this identity. You do this using a digital signature. Cryptographically, you will issue a digital signature to a transaction (new/update/revoke a claim) using your private key. Now, any verifier can verify the validity of this transaction by confirming the digital signature (without revealing the private key). This is important because this will be the first claim you will issue when creating an identity.

3.2.3 Identity

Now that we understand what the claims look like, it is finally time to understand how identity is defined. An identity constitutes all the claims the identity has issued or received from other identities. For example, "I" as an identity can put forward many claims, i.e., I am a JHU student, an MSSI Student, and an International student. All these claims say different statements about me, but all these claims are made for one identity, which constitutes an identity. In other words, identity is built by what the identity and others have said about the identity.

3.2.3.1 Identity Data Structure

Three Sparse Merkel Trees define an Identity [12].

- Clt: Clt is the claims tree. This tree is where we store all of our claims.
- Ret: Ret is called a revocation tree. Remember, in the claims data structure, we stored revocation nonce in the v0 slot. Those nonces can be used to revoke a claim by simply adding the nonce to the Revocation Tree(Ret).
- RoT: This tree stores all the historical versions of Clt roots. In Simple words, when we add a new claim to the claims tree, the root of the Clt changes. But before changing the Clt, we add the previous root tree value to the Root tree (RoT). In future sections, we will explain why this is required.

That's it; these three above Merkel Tree are all that how we represent an Identity. With this construction, the identities can issue, update, and revoke claims. An identity must issue at least one Auth Claim to operate correctly. This is the first claim issued by identity, which must be added to the Clt.

3.2.3.2 Identity State

An Identity State ***IdS*** is represented by the hash of the roots of these three Merkle trees.

$$IdS = H(rootofClt||rootofRet||rootofRoT) \tag{2}$$

3.2.3.3 Genesis State

The very first identity state of identity is defined as Genesis State. To create a genesis state, we need at least one claim in the claims tree(Clt). As mentioned before, the first claim is always an AuthClaim, and the Rot and Ret will be empty. The IdS that we get at this state is called Genesis State.

3.2.3.4 Genesis ID or Identifier

We calculate the Genesis ID or Identifier (ID) from the Genesis State. This ID is unique and permanent for the entire lifecycle of an identity. This ID is stored as a mapping in the smart contract; more specifically, we store the mapping of ID ⇒ IdS in a smart contract. Therefore, as we add more claims to the claims tree, the IdS changes, and we update the value in the map. This way, we get timestamped data of all the history of identity.

3.2.3.5 Identity State Diagram

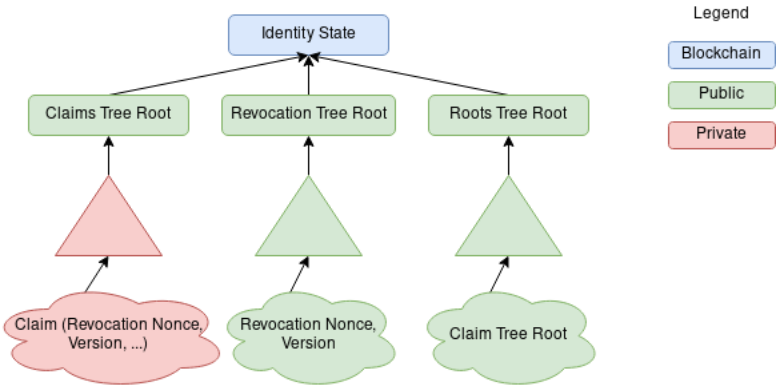

Figure 6: Identity State Diagram
Source: https://docs.iden3.io

This diagram denotes where we store the data we created we talked about. All the claims that we created would be off-chain and will be private. We will store IdS on the blockchain; everything else will be public information. This can be directly shared with the verifier or hosted on an HTTPS website.

3.3 ZKP Generation and Verification

In this section, we will describe how the data structures designed by Iden3 can be used to generate zero-knowledge proofs(ZKP) and their verification process. The Iden3 framework has written various circom circuits that can be utilized to generate ZKP. We want to redirect the reader to refer to the official Iden3 documentation for circuits [13] for further understanding.

3.4 *J-Card*$^+$ Design

The following components have been built in the implementation of *J-Card*$^+$. Source code can be found at [14], [15] and [16].

3.4.1 zkWallet SDK

Wallet SDK is a software developer kit for building a zero-knowledge identity wallet using the Iden3 framework. Wallet SDK provides all the functionality that is needed in an identity wallet, such as:

- Creating a new identity
- Adding a new claim in the identity wallet
- Generating zero-knowledge proof for a proof request
- Updating states on the Ethereum Blockchain
- Storing a copy of a claim issued by an issuer.

The purpose of developing this module is for any university/organization to create its identity wallet using JHU Wallet SDK.

3.4.2 zkIssuer SDK

Issuer SDK is also a software developer kit for building an issuer platform. It uses ZK wallet SDK and implements all the issuer-related functionality, such as:

- Implement all claims schemas related to a university.
- Gather students' data and generate Iden3 claims (using ZK wallet SDK §3.4.1) for the claims schemas.

This module is university specific and thus can be used by any university to build its issuer platform.

3.4.3 JHU Issuer

The issuer is the JHU issuer platform built using the Issuer SDK described in §3.4.2. This application reads JHU student data (fictitious), creates claims, and sends it to the student's wallet.

3.4.4 *J-Card*$^+$ Backend

The *J-Card*$^+$ backend is the JHU wallet application in Go built using Wallet SDK described in §3.4.1. Using this application, any JHU student can interact with the verifier to respond to the verifier's proof request query, i.e., they can accept/deny a proof request. Using this application, students can also request that the JHU issuer send their credentials to their identity wallet. For instance, a student can ask for AgeClaim, their status(domestic/international), their school of study(WSE, Carry Business School, etc.), and various other claims related to JHU and its affiliates. This application also handles interaction between a verifier, a wallet, an issuer, and a wallet.

3.4.5 *J-Card*$^+$ Frontend

It's a UI implementation of the wallet. Using UI, a student can interact with the *J-Card*$^+$ backend running in their systems. The UI provides all the functionalities:

- A student can request the JHU issuer to send their claims using a button.
- Students can scan a QR code to receive a proof request from a verifier.
- It displays proof requests using UI; students can accept or deny a proof request.
- Students can see all the claims they have received from the JHU issuer.
- Students can see their account details, such as private key, Identifier, and identity state.

3.4.6 Verify18

Verify18 is a sample application built to test our entire flow. To login into this web application, a student needs to be 18+. After clicking on the Signup button, it generates a QR code which is just an image version of the proof version. Any student with their JHU wallet installed can scan this QR code to verify that they are 18+. When student scans the QR code, it adds a proof request to their JHU wallet. Using wallet UI, a student can check what this verifier is trying to verify and thus can decide to accept or deny a request. If the student agrees with the request, the JHU wallet will generate the Zero-knowledge proof(ZKP) for the proof request and send the ZKP to the verifier.

Upon receiving the ZKP, this application verifies the ZKP using SnarkJS. It also confirms the states(public signals from the ZKP) on the Polygon blockchain. If the ZKP turns out to be valid, it notifies the user of "Authentication Successful" and lets the user in.

3.5 System Architecture

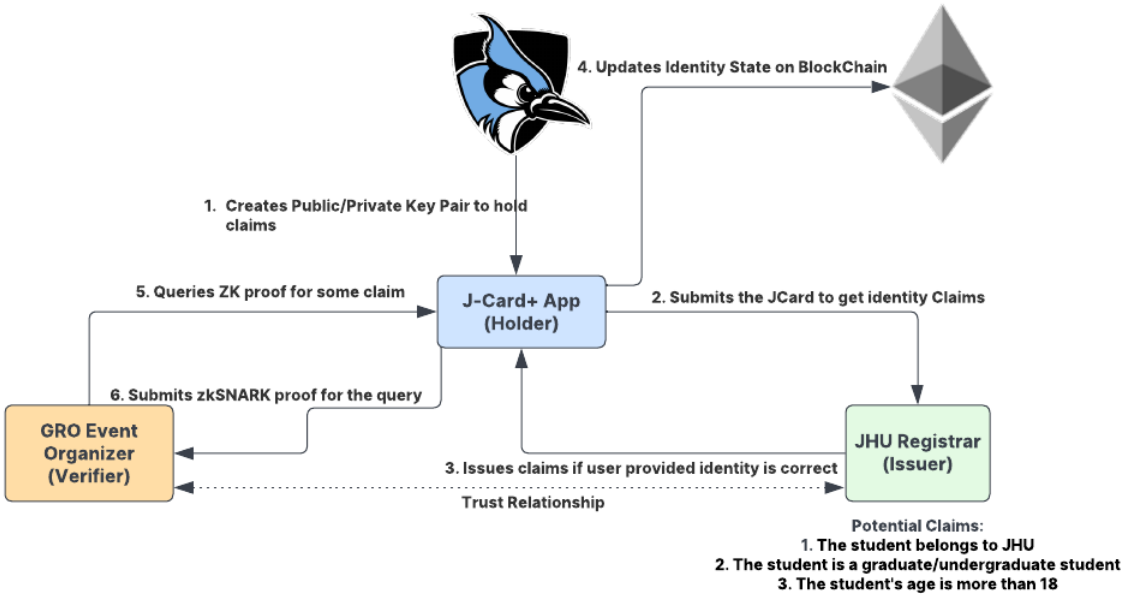

Figure 7: System Architecture

Figure (7) is the architecture diagram for the *J-Card*$^+$ ecosystem. *J-Card*$^+$ App is installed on the user's computer, and all the personal data, i.e., private keys, identity claims, and received claims, are only stored on the local system. JHU registrar is a web service responsible for issuing claims to JHU students. Using *J-Card*$^+$, students can send requests to the JHU registrar application to issue their respective claims. To authenticate the student, the JHU registrar expects a secret token from the *J-Card*$^+$, which will only be known to the student.

The verifier application in figure (7) is a sample application to authenticate a user using zero-knowledge proofs(ZKP). On requesting signup from a student, the Verifier application generates a QR code which a student can scan using *J-Card*$^+$. This QR code contains details about the query the verifier wants to perform. For example, verifiers may check if a student is above 18 years of age. Using *J-Card*$^+$, a student can accept or reject a proof request. If a student agrees with a proof request, *J-Card*$^+$ will submit the ZKP to the verifier using a callback endpoint exposed by the verifier.

We will explain the whole interaction in more detail further in section §3.9

3.6 Data Model

J-Card$^+$ stores all the data it needs in a JSON file (account.json). The JSON files contain private keys, created or received claims from other identities, and the current Identity State(IDS). This JSON file is all needed to recreate data structures such as Iden3 Sparse Merkle Tree and Auth Claim in the *J-Card*$^+$ memory.

J-Card$^+$ wallet is responsible for creating and updating this file upon starting and during the runtime of *J-Card*$^+$. *J-Card*$^+$ looks for the account.json file in the current directory; if not found, *J-Card*$^+$ creates a new account by randomly creating a new BJJ private key.

Whenever a user performs a write action to the Merkle Tree, such as Adding a new claim or upon receiving a new claim from another identity, *J-Card*$^+$ then dumps the current state of the application in the account.json file. This way, whenever you restart the application, *J-Card*$^+$ loads the last stored state in the memory, as shown in Figure 8. See Appendix (4) for a example account.json file

```
2022/12/03 11:35:15 Account loaded from saved file: account.json
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.
```

Figure 8: account.json loads at the application start

According to the Iden3 protocol specification, we must also store the IDS on the Blockchain. For *J-Card*$^+$, Iden3 State smart contract [17] is deployed at address [18] on the Polygon Chain [19] due to low latency and low gas fees.

3.7 Interface/API Definition

J-Card$^+$ and JHU Issuer are web applications that expose various REST endpoints so users can interact with these services. The following section takes a deeper look at the APIs.

3.7.1 *J-Card*$^+$ Backend

J-Card$^+$ Backend is the Go Application that runs on the user's local system. This application exposes the following endpoints which the front-end application interacts with:

3.7.1.1 GET /api/v1/getAccountInfo

This API returns all the account information, i.e., Private Key, Auth Claim, Current Identity State(IDS), Stored Claims, Identity, etc. The front-end application uses this endpoint to display this information on the web page, as shown in Figure 9.

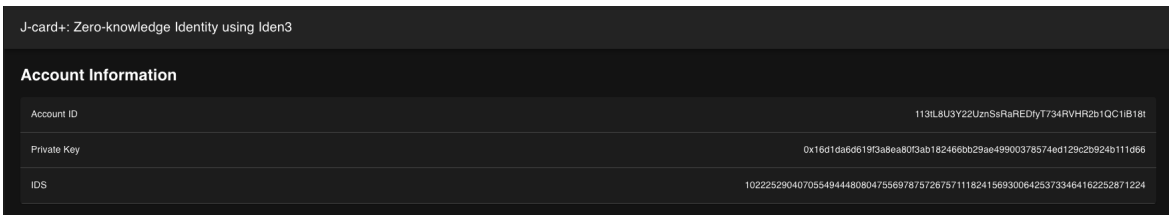

Figure 9: Account Info Screen

3.7.1.2 GET /api/v1/getClaims

This API returns all the claims the user has received from the JHU issuer. The front-end application uses this endpoint to create baseball cards for user display, as shown in Figure 10.

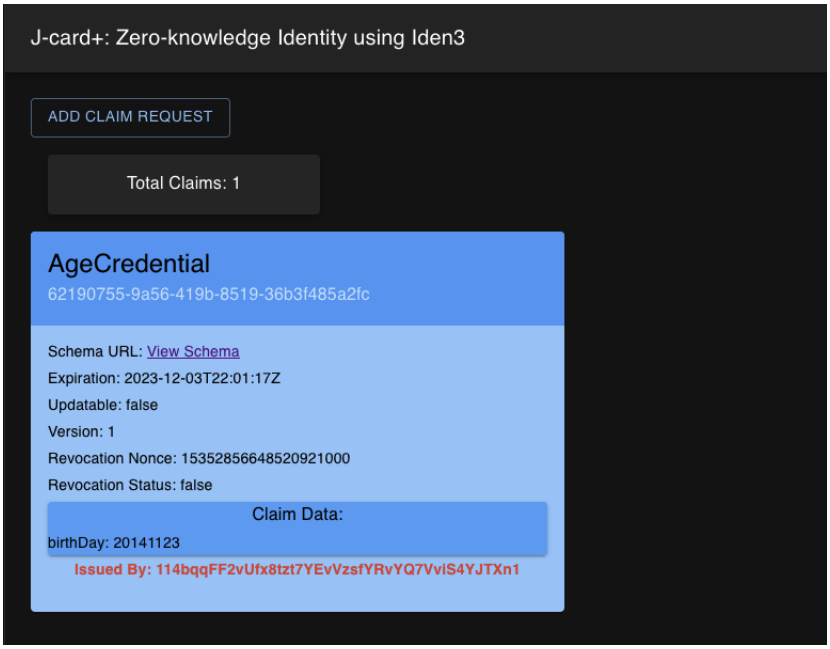

Figure 10: Claims Screen

3.7.1.3 POST /api/v1/fetchClaimsByIssuer

This API sends a request to the issuer to fetch claims. In the response, the issuer returns all the valid claims created for the user. This endpoint creates a local copy of received claims and stores them in the account.json. This endpoint expects the user to provide two parameters in the post body.

- **Issuer URL:** Hostname for the issuer to fetch claims from.
- **Auth token:** Secret token for user authentication. The user only knows this token, and it is used to identify the user by the issuer uniquely.

3.7.1.4 POST /api/v1/addProofRequest

When users scan the QR code generated by a verifier, the front end uses this endpoint to add the corresponding proof request to the wallet database. This endpoint processes the data in the QR code and fetches the proof request details from the verifier.

3.7.1.5 GET /api/v1/getProofRequests

The endpoint is used to fetch all the proof requests from the backend. It then generates a table on the front end, using which a user can see details of a proof request such as Query, verifier ID, timestamp, etc., as shown in figure 11.

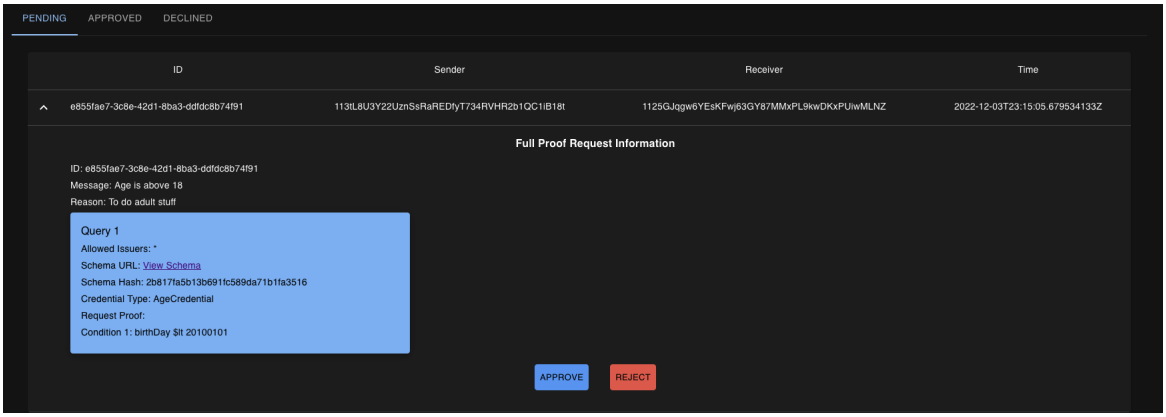

Figure 11: Proof Requests

3.7.1.6 POST /api/v1/acceptProofRequest

This endpoint is responsible for generating ZKP for a corresponding proof request. When a user accepts proof from the UI, the UI calls this endpoint, and in response, the backend service returns the status for the ZKP. If the ZKP generation were successful, the *J-Card*$^+$ would call the callback URL 3.7.3.3 provided by the verifier in the proof request with the payload as ZKP. If the ZKP generation fails because the user does not satisfy the criteria, the UI will display failed and remove the proof request.

3.7.2 JHU Issuer

JHU Issuer is the web service responsible for creating claims on behalf of all the students. This web application is a university-wide application responsible for creating/revoking claims for all the students on campus, like JHU SIS. *J-Card*$^+$ installed on the student's system will talk to the JHU issuer to fetch claims 3.7.1.3. It exposes only one endpoint, as explained below:

3.7.2.1 POST /api/v1/issueClaim

This endpoint expects the *J-Card*$^+$ to provide two parameters in the post body.

- **Identifier:** It corresponds to the student ID for which claims must be issued.
- **Auth token:** Secret token for user authentication. The user only knows this token, and the token is used to identify the user by the issuer uniquely.

When the issuer receives a request from the *J-Card*$^+$, it looks up the auth_token provided by the *J-Card*$^+$ to find the student in its database. If the lookup is successful, it creates various Iden3 claims for the student and responds with an array of Iden3Credential (2). Upon receiving the claims, *J-Card*$^+$ stores them in its account.json. This is important to generate ZKP for future proof requests.

Note: Due to time and access limitations, we have created fake data for students in a JSON file. In the future, JHU issuers can be connected with production databases like MySQL or PostgreSQL to fetch students' details.

3.7.3 Sample Verifier Application

To provide the end-to-end proof of concept of our wallet, we have created a sample application for the verifier. The following API specifications have been designed to create a compatible verifier. Every verifier application which uses *J-Card*$^+$ has to implement the following mandatory tasks:

3.7.3.1 The GET endpoint

J-Card$^+$ will use this endpoint to fetch the Iden3ProofRequest (3). While querying this endpoint, *J-Card*$^+$ will also pass its ID as sender_id. In response to this endpoint, the verifier application must return a valid Iden3ProofRequest.

3.7.3.2 The QR Code page

Since we have kept the first endpoint flexible, the verifier will provide details of the first endpoint in the QR code. A user can use their *J-Card*$^+$ to scan this QR code to get details of these endpoints, as shown in figure 12.

Figure 12: Example QR Code to the GET endpoint

3.7.3.3 The POST endpoint

J-Card$^+$ will use this POST endpoint to submit ZKP to the verifier. This endpoint will be provided in the Iden3ProofRequest as a parameter Callback URL. This is important because generating ZKP is time-consuming, and we can't do the entire workflow in one HTTP communication. Upon receiving the ZKP in the request payload, the verifier uses the Iden3 auth library([20]) to verify the validity of the received ZKP.

3.7.4 JHU claim schemas

The following claims schemas have been designed to create different claims in the *J-Card*$^+$ ecosystem:

- **AgeCredential:** As the name suggests, this schema will store a student's age information.
- **RoleCredential:** This claim schema is responsible for storing the role of an individual on the JHU campus. For example, an individual can be a student, a professor, or a staff member.
- **DepartmentCredential:** This claim schema is responsible for storing the department for an individual on the JHU campus. For example, an individual belongs to the department of computer science or physics, etc.

3.8 Iden3 Protocol Analysis

At the beginning of the discussion, we introduced three properties that the Iden3 protocol provides. This section will cover them in more detail.

3.8.1 Scalability

There are several reasons why the protocol is very scalable.

- We only store IdS on the blockchain, and the rest of the data is stored off-chain. This approach saves us a bunch of gas money and scales the protocol.
- Sparse Merkel Trees are designed to provide quick proof of membership and proof of non-membership. Thus, claim lookup is very efficient.
- We utilize BJJ and Posiden Hash functions designed to work efficiently with the zkSNARK.

3.8.2 Zero-knowledge Verifiable

ZK verification is the most crucial property of the protocol. The property states that any verifier can verify the claims without learning new information while executing the protocol. The zero-knowledge verification is done through zkSnarks.

3.8.3 Temper resistance

Temper resistance is the property that claims that data stored in the Identity Trees cannot be corrupted. Temper resistance is a property we inherit from the Merkel Trees. Also, changing the state requires proof of ownership, and this verification happens on-chain through zkSnark before changing the IdS. Therefore, we can be sure only the person who holds the private key has put forward claims, and these claims are not tampered with.

3.9 *J-Card*$^+$ Ecosystem Overview

In J-Card$^+$ ecosystem, there are two critical interactions 1) Issuer-*J-Card*$^+$, and 2) *J-Card*$^+$-Verifier. Corresponding sequence diagrams are displayed in sections §3.9.1 and §3.9.2.

3.9.1 Interaction between the JHU Issuer and *J-Card*$^+$

We developed an issuer SDK §3.4.2, which enables consumer applications to create new claims on behalf of holders. As previously stated, the Iden3 protocol specification heavily influences our issuer SDK. As the primary system function, the *J-Card*$^+$ & JHU issuer interaction results in the creation of identity claims and the registration of new claim data on the blockchain. The interaction is initiated by the holder, which means that the *J-Card*$^+$ initiates the interaction by requesting that the JHU issuer generate claims and return the claims data. The steps involved in the interaction between the *J-Card*$^+$ and the issuer are as follows.

- *J-Card*$^+$ sends a request to the JHU issuer with an authorization parameter to retrieve claims about itself
- The JHU issuer verifies the authorization parameters and creates claims for the *J-Card*$^+$ user with the appropriate expiration, revocation, and updatable flags.
- JHU issuer adds generated claims to its claims tree (Clt) and updates the roots tree (RoT).
- The JHU issuer publishes a new identity state (IdS) to the blockchain.
- The JHU issuer sends back claims data to the *J-Card*$^+$ to be stored in its wallet.

Figure (13) depicts a sequence diagram containing detailed events required to retrieve claims from the JHU issuer and how they will be presented in *J-Card*$^+$ front-end.

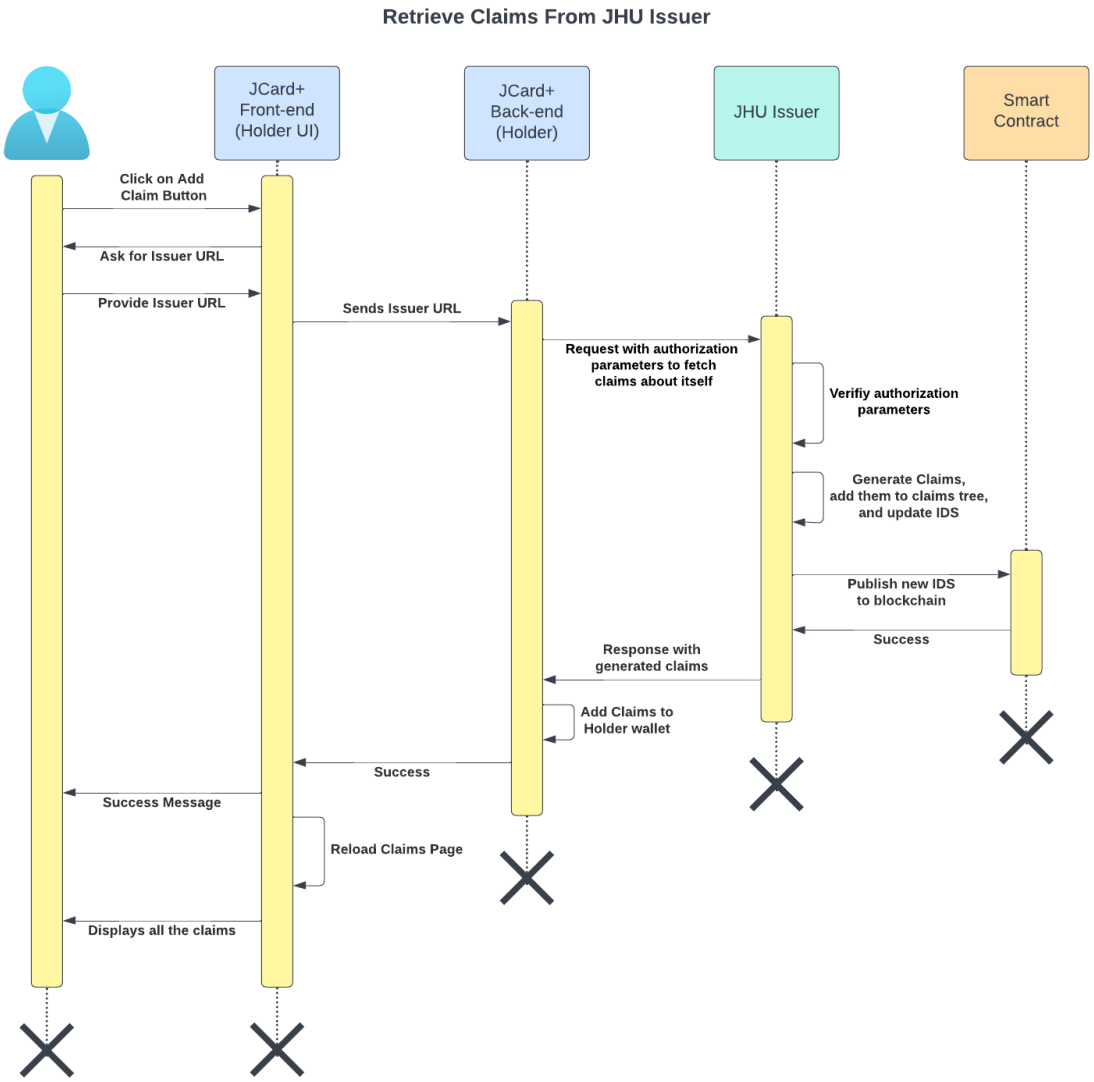

Figure 13: Sequence diagram of the *J-Card*$^+$ and JHU issuer interaction

3.9.2 Interaction between the *J-Card*$^+$ and Verifier

As another critical system function, the *J-Card*$^+$ & Verifier interaction is responsible for a zero-knowledge proof generation and verification upon a particular claim. Our implementation slightly deviates from the Iden3 protocol specification in this interaction. We structured the verification procedure into two parts in our implementation. 1) Adding ZK proof request to the *J-Card*$^+$ wallet 2) Generate and verifying the ZK proof. This choice was primarily motivated by the desire to introduce functionality allowing users to inspect ZK proof request data (in Step 1) before approving a request to create evidence and transmit it to the verifier. This strategy gives the user of *J-Card*$^+$ application the option to accept or reject ZK proof requests.

3.9.2.1 Add ZK Proof Request To *J-Card*$^+$

This interaction's primary duty is to ask for a ZK proof request from the Verifier18 and add it to the *J-Card*$^+$ so that capability to examine it later can be provided. This interaction also is initiated by the holder. The holder must be aware of the specifics of the endpoint to engage in interaction to retrieve ZK proof request data. The following are the main steps in the interaction between the *J-Card*$^+$ and the Verifier18.

- Before the interaction, the Verifier18 sends an endpoint or a QR code embedded with the endpoint to the *J-Card*$^+$ to obtain the proof query request it wants to verify.
- The *J-Card*$^+$ only knows the endpoint at the beginning of the interaction; it is unaware of the Verifier18's ZK proof request. The *J-Card*$^+$ user scans the QR code on the proof query request page on *J-Card*$^+$ front-end to obtain more details and then stores the proof query request for later examination in internal storage.
- The *J-Card*$^+$ user can now inspect the information contained in the proof query request and make informed decisions as to whether to accept or reject it.

Figure (14) shows a visual representation containing the sequence of events to add proof verification query to *J-Card*$^+$ internal storage and how they will be presented to *J-Card*$^+$ user.

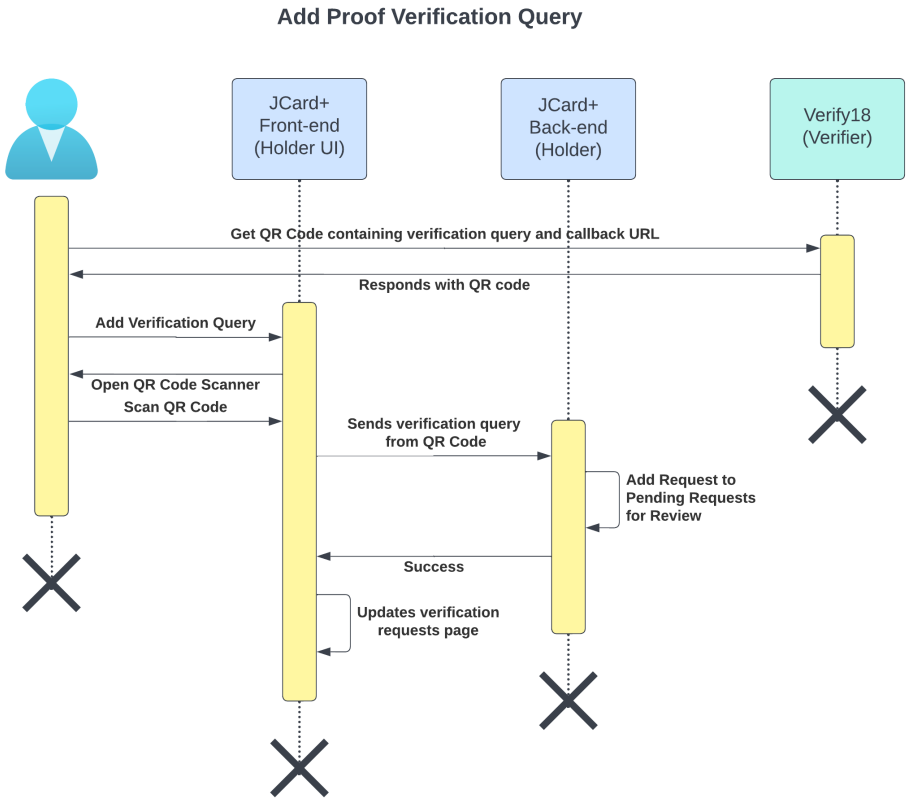

Figure 14: Sequence diagram of the interaction between the *J-Card*$^+$ and the Verifier18 to add proof query request

3.9.2.2 Generate and Verify ZK Proof Request

The interaction between the *J-Card*$^+$ and Verifier18 will begin to generate ZK proof associated with the proof query request once the holder has reviewed and approved the proof query request. The *J-Card*$^+$ back-end sends back ZK proof to the verifier to verify and prove its access right. The steps in the interaction between the *J-Card*$^+$ and the Verifier18 are as follows.

- Once the proof query request has been accepted, the *J-Card*$^+$ back-end creates the zero-knowledge proof and internally calls the callback URL associated with the Verifier18's request for a proof query.
- After receiving the request from *J-Card*$^+$, the Verifier18 performs the following actions:

- Payload associated with holder request is correctly associated with the proof query request
- The JHU issuer, with whom Verifier18 has already built a trusting relationship, has made a claim.
- The zero-knowledge proof is legitimate

- Verifier18 notifies the *J-Card*$^+$ that verification was successful and grants access to Verifier18 features.

Figure (15) shows a visual representation containing a sequence of events to generate and verify ZK proof for proof query requests sent by the Verifier18 and how they will be presented to *J-Card*$^+$ user.

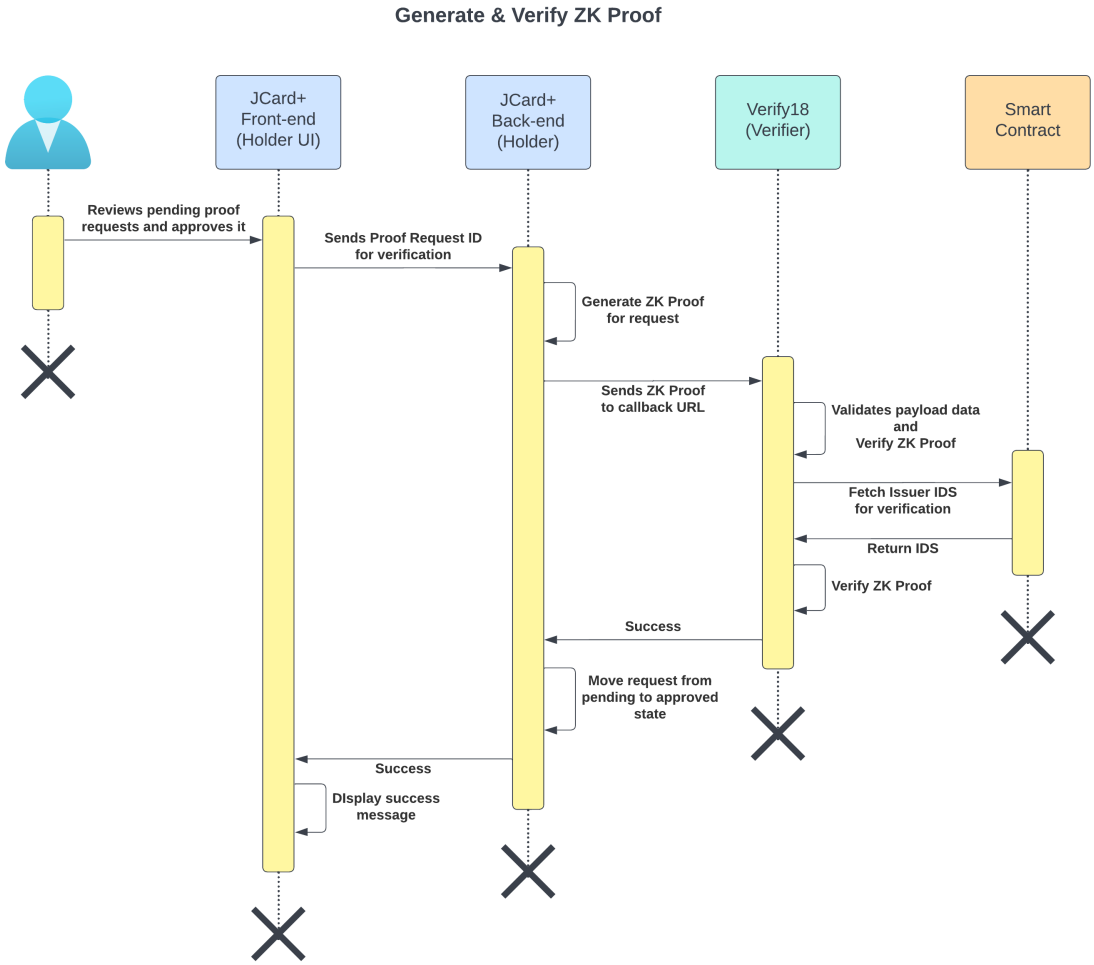

Figure 15: Sequence diagram of the interaction between the *J-Card*$^+$ and the Verifier18 to verify proof query request

4 Experimentation, Evaluation, and Result Analysis

4.1 Experimentation

Security needs to be practical for it to be useful. To envision the competence of *J-Card*$^+$, a stimulating *J-Card*$^+$ ecosystem consisting of an identity issuer (JHU Issuer) and an identity verifier (Verify18) is created. Integrated with the novel Iden3 identity access control protocol, *J-Card*$^+$ intends to prove

Critical Components	CPU Time	Memory Usage	Response Time
JHU Issuer: Claim issuing	0.14	16.4MB	5.85s
J-Card$^+$: Identity proof generation	0.11	16.7MB	9.88s
Verify18: Identity proof verification	0.13	16.3MB	0.3569s

Table 2: Software performance measured in terms of CPU time, memory usage, and response time

Subjects	Cost
Graduate student count	20,088 (10% event attendees)
GRO event count	35 / semester
Claim issuring	0.000687324004 MATIC / transaction (1 MATIC = 0.894 USD)
JHU Issuer Deployment	12 USD / month
Total	188 USD / year

Table 3: Economic analysis for *J-Card*$^+$ deployment in the JHU community

that it is an alternative identity management solution for the existing third-party-driven authentication system. It preserves better end-user privacy such that 1) identity information is self-sovereign, 2) access right verification is verifier-prover self-sufficient, and 3) identity claims can be formulated with autonomy.

4.2 Evaluation

The *J-Card*$^+$ ecosystem is assessed with metrics of software performance testing, economic analysis, and threat modeling. The former two executives are presented as follows, while the latter analysis is demonstrated in §5.

Software performance testing targets on three critical computations raised in the flow from JHU Issuer to *J-Card*$^+$ application, and then to Verify18. The corresponding code section executes claim issuing, identity proof generation, and identity proof verification accordingly. By deploying the three code modules on a macOS architecture with an Intel Core i5 processor, software performance is measured in terms of CPU time, memory usage, and response time. Specifically, the former two performance data are collected using an Apple-native process monitoring tool, Activity Monitor. At the same time, the latter is measured by wrapping the critical code section with a timer supported in the Go package, *time*. Table (2) enumerates performance results with respect to each critical function of the identity's issuer, holder, and verifier.

In addition, economic analysis is conducted for deploying a complete *J-Card*$^+$ ecosystem in the JHU community. According to table (3), each claim issuing which writes data to blockchain around 0.0006873 MATIC while 1 MATIC is equivalent 0.894 USD. Assuming there are 10% of grad student attending 35 events every year, the total cost of using *J-Card*$^+$ as an admission solution is estimated to be 188 UDS/year. It is including the deployment cost of JHU issuer.

4.3 Result Analysis

Apart from the extensibility driven by Wallet and Issuer SDKs, *J-Card*$^+$ is proven reliable and scalable with its efficient performance. As the CPU time revealed in Table (2), per critical computation only requires 0.13 clock ticks on average to complete. Although the process of proof generation is ten times more than its verification, *J-Card*$^+$ is still performing under a practical threshold. In a realistic setting, there are only limited claims maintained by the wallet application versus the number of claims needed to be processed by an identity verification agency. The payoff of computing proof at the claim-holder end yields a significant time decrease for a proof verifier as a return. In fact, the time contrast observed in proof generation and verification aligns with the theoretical time complexity presented in the Groth'16 entry of Table (1). Furthermore, the memory usage of proof generation is expected to be

high due to the involvement of the ceremony. As a trusted setup is required in constructing a Groth'16 zero-knowledge proof, extra JSON files are generated in the process of identity-proof generation.

5 Security Considerations

We created the SDK for wallet and issuer, as well as the *J-Card*$^+$ application, which illustrates their utility, as was covered in the prior parts. However, as is often stated in the security field, "No system is perfect, and every system carries some risk." In other words, every technology carries some level of risk, and by being aware of those risks, users and businesses may more successfully employ a better cybersecurity management strategy. This section will discuss some common risks to our produced application and the Iden3 protocol. We conducted the threat modeling exercise with our group members to identify threats and attacks. It is important to note that this part does not include all risks and primarily concentrates on threats and attacks pertaining to interactions between holders, issuers, and verifiers. It is built on the assumption that readers know some of the typical dangers of internet infrastructure.

5.1 Replay Attack

A replay attack [21] is a passive network attack in which an attacker replays the network traffic with valid network data. It is a type of man-in-the-middle attack. Replay attacks frequently have adverse effects, such as unauthorized access and denial of service.

In our project, the interaction between the holder and verifier to verify an access right is the most susceptible to a replay attack. If a malicious holder can persuade a trustworthy verifier that he produced the zero-knowledge proof correctly, the verifier will award the malicious holder access rights, creating a vulnerability for unauthorized access. In the threat scenario depicted in figure (16), an attacker monitors the network traffic between the prover and the verifier and then replays it to the verifier to persuade it to grant access rights.

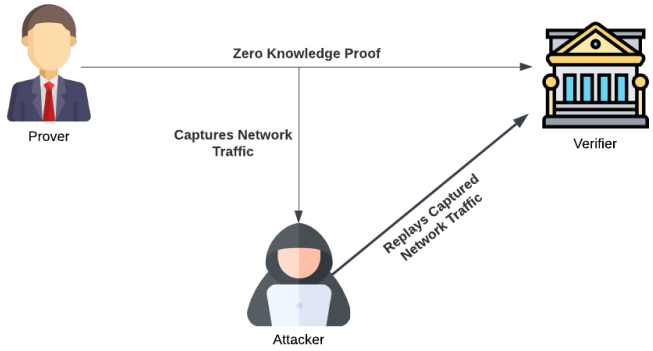

Figure 16: Replay Attack

The claim is being verified in our present implementation using the verifier **/api/v1/callback** endpoint. Instead of implementing a defense against the replay attack from our side, we are relying on implementing the Iden3 framework. According to the Iden3 protocol, verifiers must transmit a unique "challenge" along with each verification query while creating a proof verification request. From the holder's perspective, the holder must sign the challenge and return it with ZK proof. Verifiers can check the signatures on the challenge during the verification process to ensure that the responses are coming from the rightful owner.

5.2 Spoofing Attack

In a spoofing attack [22], an attacker tries to manipulate the data to get an unfair edge over the system. We looked into the threat scenario for this project, where "A dishonest holder will try to falsify the

claim and get access to the verifier system.". As discussed in the earlier sections, when the issuer issues a new claim, it updates its internal Merkle trees and deploys the hash of the root of those trees (IdS) to the blockchain.

Hence to successfully carry out this attack, an attacker needs access to the issuer system to generate or modify stored internal Merkle tree claims data claims. This would only be possible if the issuer system were compromised somehow. This would imply that the best security method against a spoofing attack is to store the issuer's private key in a secure location. Although the Iden3 standard implies that there would already be a trust connection between the issuer and verifier, the role duty of defense on the issuer is not optimal. Due to that assumption, verifiers only deal with blockchain-stored issuer data. In subsequent work, we can investigate solutions based on consensus algorithms to issue claims, removing central accountability from the issuer.

5.3 Information Leakage

An application flaw known as information leakage [23] enables attackers to access system data such as user-specific information, environment variables, or technical information about a web server. The implications of information leaking in the context of our implementation will be covered in the following two sections.

5.3.1 Private Key Data Leak/Issuer Account Data Leak

Our approach uses the environment variable to obtain details about the identity private key. Any security hole that enables an attacker to access environment variables could have disastrous effects on the issuer application. An attacker can issue new claims for anyone if they have access to the issuer's private key. Due to the trust connection between the verifier and issuer and the potential for bypassing the verification process in the event of issuer private key disclosure, this would also significantly influence the verifier's operation. It is important to note that since claim data is always saved in the issuer's local storage, an attacker who has access to the account's private key cannot access claims that have already been made. It's because the Iden3 protocol needs IdS to be published to the blockchain for verification and doesn't involve disclosing any claim data.

If the attacker gained access to all data pertaining to issuer identification, a terrible situation would develop. By doing this, the attacker would have access to the account.json file displayed in figure(4). If this occurs, an attacker would be given the ability to duplicate the original issuer and get access to all of its functionalities, including the ability to issue new claims, revoke existing claims, change existing claims, etc.

Our current approach does not provide any defense for the account.json file. As a first step for the security of account.json, the issuer can store account.json data in an encrypted format instead of storing it in plain text.

5.3.2 Claims Data Leak

In this section, we looked at the effects of a scenario in which an attacker was to obtain claim data about the victim's identity. Privacy would be lost, which would be the biggest effect. Data leaks involving claims effectively mean that all of the information kept there, including sensitive or private data, would be visible. In our scenario, if the identity makes a claim about its age credential, an attacker can see the identity's information about its date of birth.

At the time of writing this report, this kind of data leak is already present in PolygonID, another Iden3 protocol implementation. It is important to note that the claim data leak does not provide any personal information on the claim holders. In other words, it would be challenging to correlate claims with a specific individual in the actual world if the claim does not contain personal information about the holder. This situation is still bad because attackers frequently integrate information from many sources, such as blockchain transactions, to improve the accuracy of association with a person in the real world.

5.4 Failure of Blockchain Smart Contract Transaction

In the context of our application, we examine the implications of a blockchain smart contract transaction failing during the issuance of a new claim in this section. Blockchain transaction failures are not

currently handled gracefully by our implementation of the Iden3 protocol. Regardless of the results of the blockchain transaction updating the issuer IdS on-chain, we are currently adding claim data to the issuer's internal claims tree. In the event that a blockchain transaction fails, claims would already have been added to the issuer claims tree. As a side effect, the re-request to fetch the claim from the holder would be rejected because the claim data is already present in the issuer's internal storage; it would trigger a denial of service for the holder and the verifier. Additionally, the verifier won't be used because it won't have updated IdS during the verification process. We should try to leverage the context library of Go to reverse claim changes if a blockchain transaction fails to fix this problem in our current approach.

5.5 Impact of High Transaction Gas Fee

We encountered a problem with the transaction gas fee when testing our version of the Iden3 protocol on the polygon testnet [24]. This section will detail the problem and go through how it affects the Iden3 protocol. First, the blockchain will reject the issuer transaction to update IdS if it does not contain enough gas fees. The implication would be similar to what we described in section §5.4. Second, we discovered that the gas prices on the Polygon testnet would occasionally increase unexpectedly, causing blockchain transactions to fail. It is important to note that the issuer's operation would be impacted if an attacker were to increase transaction gas costs. Every transaction requires the issuer to pay an additional gas price, which could cause the issuer to run out of money.

We consider this scenario to be a very low-level threat because

- It would be extremely difficult for an attacker to take over the blockchain and raise gas prices.
- Relayer solutions can be added to improve scalability and lower gas costs per claim.

5.6 Resource Exhaustion Attack

Resource exhaustion [25] is an attack in which an attacker tries to crash, hang, or interfere with a targeted system. Denial-of-service to authorized users would be the primary effect of the attack. We employed an HTTP web server to deliver our *J-Card*$^+$ application capabilities in our solution. By submitting numerous unauthorized queries, an attacker can use a JHU issuer or Verifier18 web server as a target to take advantage of a resource exhaustion attack. Because the JHU issuer application only needs to issue a claim once and employs Web2 authentication to guard against unauthorized requests, we will primarily concentrate on *J-Card*$^+$ and Verifier18 interaction.

Without any prior authentication, the Verifier18 program sends verification to everyone. False claims can be generated by the malicious *J-Card*$^+$ application user, who will then try to flood Verifier18 with them. We rely on the implementation of the Iden3 zkSNARK to defend against this attack. No matter how ample the ZK proof is, zkSNARK always takes the same time to verify. This is also displayed in Table (2) in our identity proof verification component results. This suggests that traditional protections, such as API rate limitation and IP banning, would also be effective in our situation.

6 Related Work

The study of identity-based on zero knowledge is still extremely young. In this section, we go over more privacy-focused authentication methods based on zero-knowledge identities. PolygonID [26], created by the Web3 polygon organization, is the zero-knowledge identity solution that closely resembles our solution. It also makes use of the Circom ZK and the Iden3 toolkits. They also offer a function that allows you to install the issuer on their online platform. Comparing PolygonID to our solution, we discovered that all data is shared across all issuers and that issuer claims data is stored on open AWS S3 containers. Since anyone with internet access can read the claim data for any issuer, we view it as a privacy concern. In comparison to our strategy of building a website for *J-Card*$^+$ Wallet, we discovered that polygonID has a mobile application deployed as a wallet, which may help improve the adoption of their platform. We think that our work can be expanded in the future to produce smartphone applications.

zCloak Network [27] is another zero knowledge identity system that offers self-sovereign identity. Similar to the Iden3, the project is open source. However, it uses a different implementation for zero-knowledge identity than the Iden3 framework. Their architecture is divided into layers, including identity, storage, compute, and proof layers, each providing a different set of functionality. To manage and store claims, they offer a chrome extension [28]. The significant difference is that zCloak Network uses zkSTARK instead of the zkSNARK protocol to generate and verify proof. As shown in Table (1), zkSTARK does not require any trusted setup, unlike zkSNARK, but it takes logarithmic time with proof size to verify the ZK proof. As per our understanding, it might lead to resource exhaustion attacks discussed in the earlier section §5.6. But we didn't make a thorough comparison because their design is very different from the Iden3 protocol, but more research is needed to determine their platform's security.

7 Future Work

We think that our *J-Card*$^+$ project proved the value of the zero-knowledge identity offered by the Iden3 framework. However, it is not yet completely prepared for deployment to the JHU institution. We discovered a few areas where our project could be improved in the future.

First, how a tool is used typically significantly impacts its adoption. Due to the fact that our solution presently only functions on web browsers on laptops or PCs, we believe that is the biggest bottleneck. Adding functionality for mobile web browsers will help the *J-Card*$^+$ Wallet to improve adoption in the future. Additionally, we think users should have alternatives for how to use our tool across various gadgets. We recommend deploying *J-Card*$^+$ Wallet as a cross-platform smartphone application and browser extension to help with it. Second, as of right now, none of the functionality related to claims in the issuer, including updating, deleting, and revoking claims, is supported by our implementation. We advise implementing support for these functionalities before the *J-Card*$^+$ is widely deployed. This would permit JHU Issuer Admin to alter and delete claims in certain circumstances. For instance, the JHU issuer may have to retract some claims made about a student if they leave the university. Third, we performed thread modeling with the project team and code reviews on each other's code while putting the project into practice. Since the project will deal with students' private information, we advise having our code audited by a third party to ensure the highest level of security.

Finally, we have previously discussed the scope and adoption of our project with the Student DAO Community [29] and have received an encouraging response. However, to integrate our project with the Student DAO Community, we must develop a claim schema that gives students access to role-based functionality in addition to the fundamental claim of whether or not they are students. For instance, if a school allows some students to serve in administrative capacities, we must support the various positions. It's crucial to note that we currently provide the capability; all left to do is integrate our system with a student DAO validator and build a claim schema JSON-LD document.

8 Conclusion

With the emergence of the Web3 era, third-party driven authentications are still prevalent across most online applications. While excessive identity information is centrally maintained and justified as a necessity, access right verification is deemed to be renovated more than it ever needed. As an identity management alternative, *J-Card*$^+$ contrives an innovation of privacy-preserving authentication systems. It assures end-user privacy intrinsically such that: 1) identity information is self-sovereign, 2) access right verification is self-sufficient between identity owner and verifier, and 3) identity claims are autonomously managed among identity issuers and owners. To thoroughly evaluate its practical significance, *J-Card*$^+$ ecosystem composed of an identity issuer, holder and verifier is developed for simulating realistic use cases in the JHU community. Through assessment metrics such as software performance testing, threat modeling, and economic analysis, *J-Card*$^+$ reveals a great potential for wide-scale realistic deployment. In furthering the work of *J-Card*$^+$, a few suggestions such as profound functions inclusion, mobile migration, and third-party auditing are proposed concerning its extendibility, usability and reliability. Carrying on with our passions, we have taken the belief of "security need not be paid with the price of privacy" into action. Not only the *J-Card*$^+$ appplication is delivered as

a proof of concept, but also an issuer SDK and a wallet SDK are published as open source to lay the groundwork for next generation authentication systems.

9 References

[1] "iden3." [Online]. Available: https://iden3.io

[2] M. Bellés Muñoz, J. Baylina Melé, V. Daza Fernández, and J. L. Muñoz Tapia, "New privacy practices for blockchain software," *IEEE software*, 2021.

[3] "What are zk-snarks?" [Online]. Available: https://z.cash/technology/zksnarks/

[4] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, "A review of zk-snarks," *arXiv preprint arXiv:2202.06877*, 2022.

[5] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[6] "Circom 2 documentation." [Online]. Available: https://docs.circom.io/

[7] J. L. Muñoz-Tapia, M. Belles, M. Isabel, A. Rubio, and J. Baylina, "Circom: A robust and scalable language for building complex zero-knowledge circuits," 2022.

[8] B. WhiteHat, J. Baylina, and M. Bellés, "Baby jubjub elliptic curve," *Ethereum Improvement Proposal, EIP-2494*, vol. 29, 2020.

[9] "Circomlib." [Online]. Available: https://github.com/iden3/circomlib

[10] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for Zero-Knowledge proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 519–535.

[11] J. Baylina and M. Bellés, "Sparse merkle trees."

[12] "Iden3 documentation." [Online]. Available: https://docs.iden3.io/

[13] "Iden3 circuits documentation." [Online]. Available: https://docs.iden3.io/protocol/main-circuits/

[14] "Jcard+ ecosystem." [Online]. Available: https://github.com/zkSnack/jcard-plus-ecosystem/

[15] "Jcard+ frontend." [Online]. Available: https://github.com/zkSnack/jcard-plus-frontend/

[16] "Jcard+ schema." [Online]. Available: https://github.com/zkSnack/jcard-plus-schema-holder/

[17] "Iden3 state smart contract." [Online]. Available: https://github.com/iden3/contracts/blob/master/contracts/state/State.sol

[18] "Jcard state smart contract." [Online]. Available: https://mumbai.polygonscan.com/address/0x87B36cE5393D4ea6EEf3eb7b1ca6aAd7ae295D4F

[19] P. Community, "Ethereum's internet of blockchains." Polygon.technology.

[20] "Iden3 auth library." [Online]. Available: https://github.com/iden3/go-iden3-auth

[21] "Relay attack - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Replay_attack

[22] "Spoofing attack - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Spoofing_attack

[23] "Information leakage." [Online]. Available: http://projects.webappsec.org/w/page/13246936/Information%20Leakage

[24] "Polygon testnet." [Online]. Available: https://mumbai.polygonscan.com/

[25] "Resource exhaustion attack - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Resource_exhaustion_attack

[26] "Introducing polygon id, zero-knowledge identity for web3." [Online]. Available: https://polygon.technology/blog/introducing-polygon-id-zero-knowledge-own-your-identity-for-web3

[27] "zkid introduction." [Online]. Available: https://docs.zkid.app/guide/introduction

[28] "zkid wallet." [Online]. Available: https://chrome.google.com/webstore/detail/zkid-wallet/hkdbehojhcibpbcdpjphajfbgigldjkh

[29] "Student dao community." [Online]. Available: https://www.joinstudentdao.com/

[30] "Zksnacks docker hub repository." [Online]. Available: https://hub.docker.com/r/zksnacks/jcard3-wallet

[31] "Iden3comm." [Online]. Available: https://github.com/iden3/iden3comm

10 Appendices

10.1 User Manual

10.1.1 Installation

Installing *J-Card*$^+$ App on your device is straightforward. You can instantly use the published docker images to start using *J-Card*$^+$ App. Please note that you need to have docker installed on your system.

$ docker run -d -p 8080:8080 –name holder zksnacks/jcard3-wallet:latest

The above command will download the jcard3-wallet image from the zkSnacks docker hub repository [30] and start the container on your machine. After the container starts, you can go to **http://localhost:8080** to access the wallet UI.

10.1.2 Fetch claims from the JHU issuer

You can use wallet UI to fetch claims from the JHU issuer. Team ZkSnacks have deployed the JHU issuer on the digital ocean at *https://jhu-issuer-new-n6gay.ondigitalocean.app/*, so you can use that to fetch your claims. To fetch claims, click on Claims using the wallet UI left bar, and then you can use **Add Claim Request** button to open a dialog box as shown in Figure (17). If you don't have the issuer setup, see section 10.1.4 to launch your own issuer.

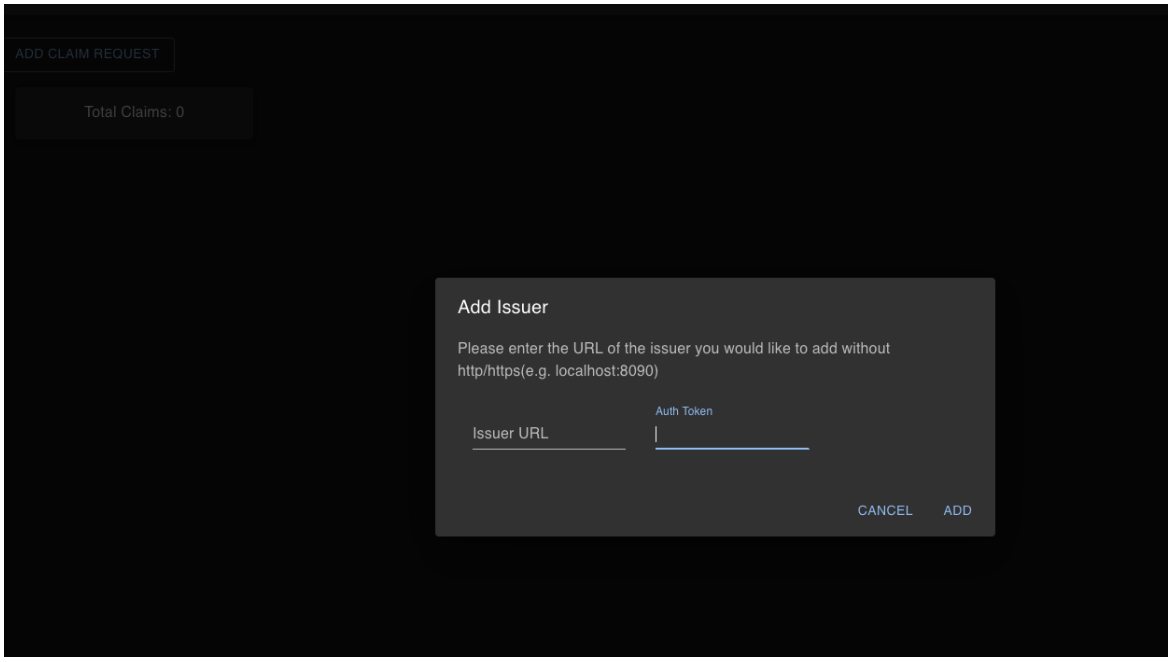

Figure 17: Fetch Claims using Wallet UI

You can insert the Issuer URL and your unique Auth Token in the dialog box to fetch your claims. After putting the values, the wallet will fetch your claims from the JHU Issuer, as shown in figure (18).

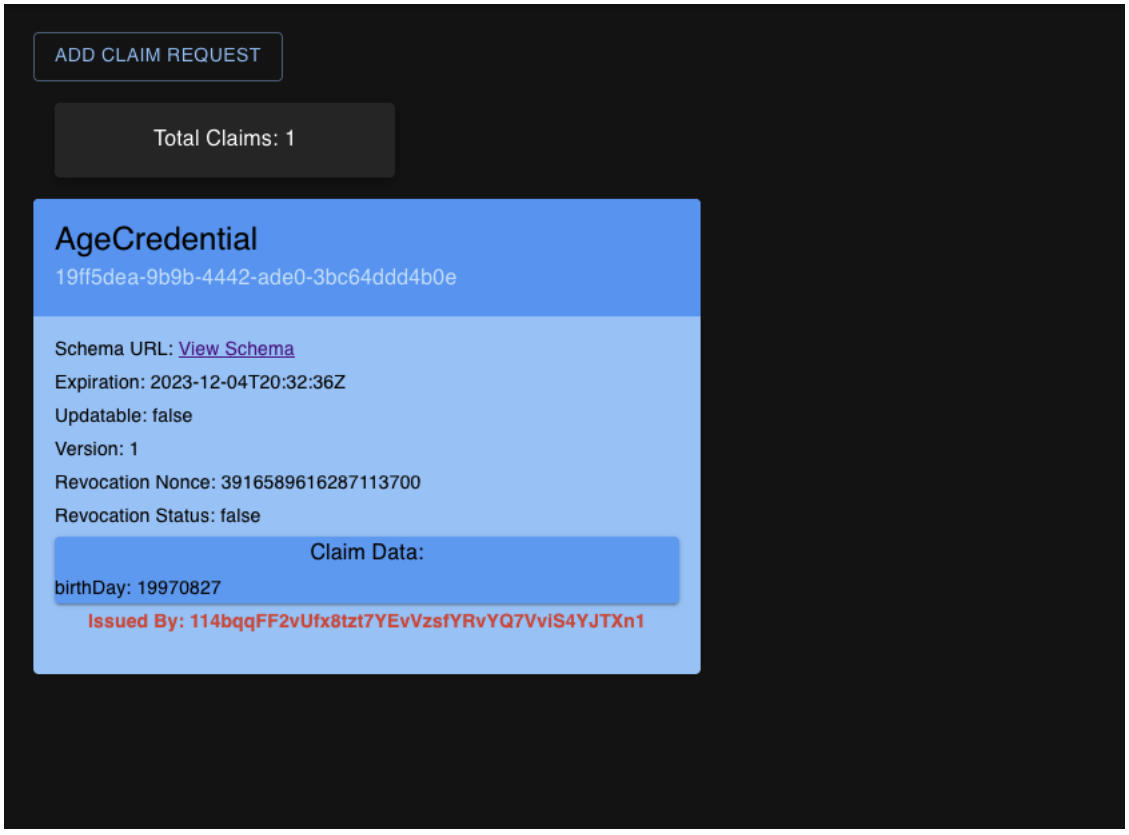

Figure 18: Claims from JHU issuer

10.1.3 Generating ZKP for a Proof Request

This section will show how we can use *J-Card*$^{+}$ to authenticate via ZKP. We have launched a sample verifier application on the digital ocean at *https://verifier-7iyjt.ondigitalocean.app/* to showcase the proof-of-concept. In this section, we will use this application to generate ZKP and authenticate on the application.

To authenticate on the application, go to the verifier link and click on the Sign-Up button. This action will generate a QR code which you can scan using the wallet UI as shown in figure (19)

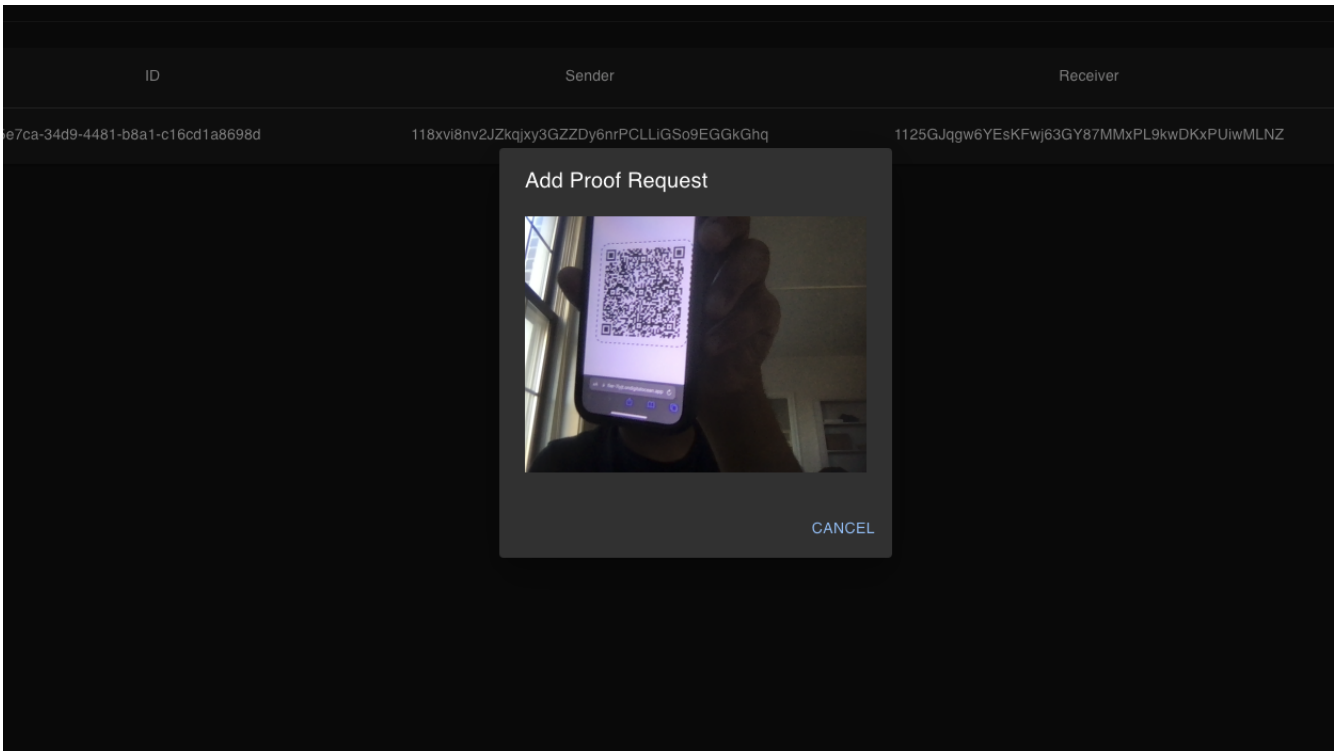

Figure 19: Scan QR code using Wallet UI

As soon as you scan the QR code, an entry will be created in the requests table as shown in figure (20)

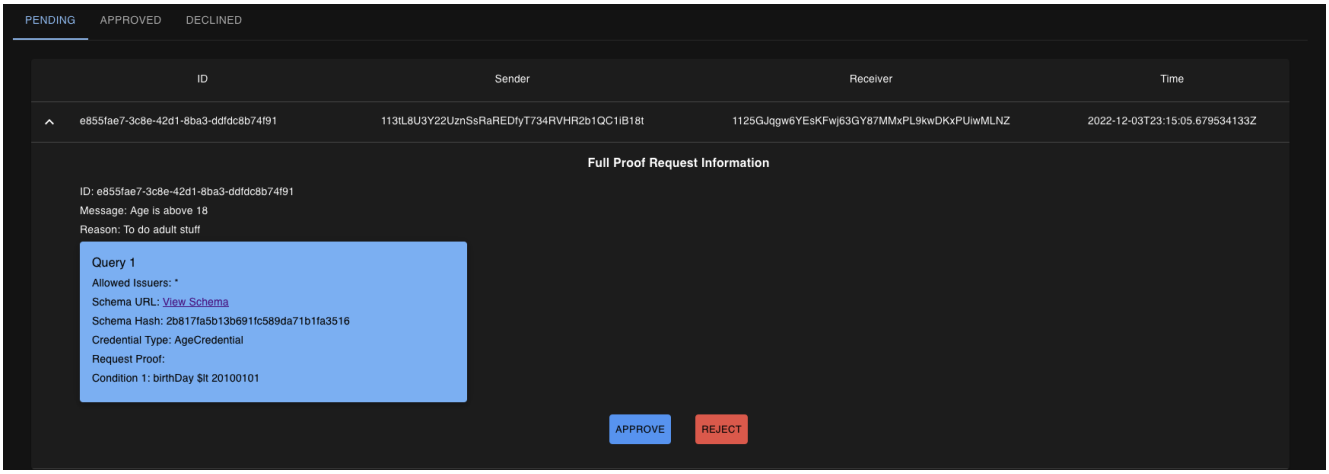

Figure 20: Entry in Proof Request Table

Now in the table entry, you check what the query presented by the verifier is. For example, in this

case, the verifier is querying that your birthday is less than 20100101. If you want to go ahead and generate the ZKP for this proof request, you can click on Approve Button. After clicking on approve button, the wallet will generate ZKP for you and send the same to the verifier, and a success message will be displayed on the UI as shown in figure (21).

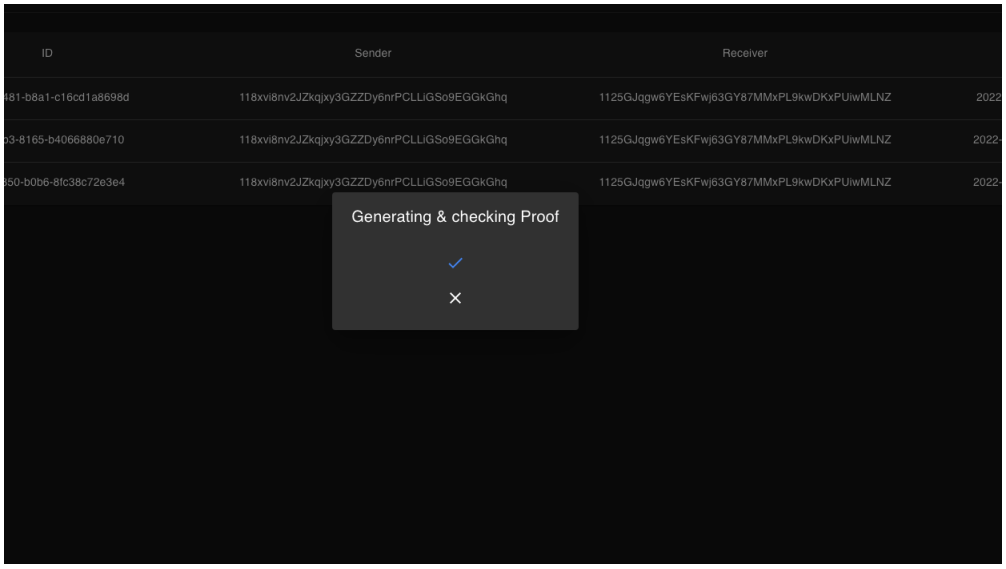

Figure 21: Success on ZKP generation

10.1.4 Installation of JHU Issuer

Launching the JHU issuer is very simple. You can instantly use the published docker image to launch your JHU issuer *J-Card*$^{+}$ App. Please note that you need to have docker installed on your system.

$ docker run -d -p 8090:8090 –name issuer zksnacks/jcard3-issuer:latest -e 'PRIVATE_KEY=Polygon account private key ' -e 'WEB3_ENDPOINT=https://rpc-mumbai.matic.today/'

The above command will download the jcard3-issuer image from the zkSnacks docker hub repository [30] and start the container on your machine. After the container starts, your issuer endpoint URL would be *http://localhost:8090*.

10.2 Code samples

```
type Iden3Credential struct {
        ID                string                `json:"id"`
        Context           []string              `json:"@context"`
        Type              []string              `json:"@type"`
        Expiration        *time.Time            `json:"expiration,omitempty"`
        Updatable         bool                  `json:"updatable"`
        Version           uint32                `json:"version"`
        RevNonce          uint64                `json:"rev_nonce"`
        CredentialSubject map[string]interface{} `json:"credentialSubject"`
        CredentialStatus  *CredentialStatus     `json:"credentialStatus,omitempty"`
        SubjectPosition   string                `json:"subject_position,omitempty"`
        CredentialSchema  struct {
                ID   string `json:"@id"`
                Type string `json:"type"`
        } `json:"credentialSchema"`
        Proof interface{} `json:"proof,omitempty"`
}
```

Listing 2: Iden3Credential Struct [31]

```
// AuthorizationRequestMessage is struct the represents iden3message authorization request
type AuthorizationRequestMessage struct {
        ID       string                          `json:"id"`
        Typ      iden3comm.MediaType             `json:"typ,omitempty"`
        Type     iden3comm.ProtocolMessage       `json:"type"`
        ThreadID string                          `json:"thid,omitempty"`
        Body     AuthorizationRequestMessageBody `json:"body,omitempty"`

        From string `json:"from,omitempty"`
        To   string `json:"to,omitempty"`
}

// AuthorizationRequestMessageBody is body for authorization request
type AuthorizationRequestMessageBody struct {
        CallbackURL string                       `json:"callbackUrl"`
        Reason      string                       `json:"reason,omitempty"`
        Message     string                       `json:"message,omitempty"`
        DIDDoc      json.RawMessage              `json:"did_doc,omitempty"`
        Scope       []ZeroKnowledgeProofRequest `json:"scope"`
}
```

Listing 3: Iden3ProofRequest Struct [20]

```
{
        "id": "115BJorcHigfT8KwCxJeAb8EDn4xjqhM1fHEPjUsnr",
        "identity_state": "106654784261141523734219400293901606065863432848027559042867575781924419486 32",
        "private_key": [
                117,
                162,
                28,
                156,
                68,
                185,
                129,
                217,
                211,
                243,
                89,
                186,
                17,
                156,
                144,
                163,
                122,
                108,
                186,
                141,
                16,
                61,
                192,
                204,
                174,
                4,
                25,
                34,
                123,
                224,
                19,
                28
        ],
        "authClaim": [
                "304427537360709784173770334266246861770",
                "0",
                "804973618350926051883590469794331510063623129697435400745169906920243703663 9",
                "161133039387229215688726192545281470851660818170399403675357761124257209218 86",
                "5577006791947779410",
                "0",
                "0",
                "0"
        ],
        "claims": [],
        "clt": {},
        "ret": {},
        "rot": {},
        "received_claims": {}
}
```


Listing 4: A example account.json file