How to Iden3

Distributed Lab

2023

What is Iden3? 2
- Identity and claims 2
- Polygon ID 4
- DID and Verifiable Credentials 4
- Verifiable Credential proof types 5
- Zero-Knowledge Proof (ZKP) 5
- Iden3 ZKP Circuits 6

What can you achieve using Iden3 and PolygonID? 7

Base architecture 8
- Core services 8
- Issuer 9
  - Resources 9
  - Requirements 9
  - Polygon ID issuer node 10
  - Custom issuer 11
- Wallet (Holder) 12
  - Requirements 12
  - Polygon ID Wallet 12
  - Custom Wallet 13
- Verifier 15
  - Iden3 contracts system 15
    - State 15
    - ValidatorContracts 16
    - ZKP Queries hash 16
    - ZKPVerifier 17
    - Conclusions on the basic implementation 17

How to dive into digital identity? 18

Multichain with Rarimo 20
- Prerequisites 20
- Solution 20
- How to achieve cross-chain? 21
  - Issuer service particularities 21
  - Wallet changes 21
  - Smart contracts particularities 22
    - Main motivation 22

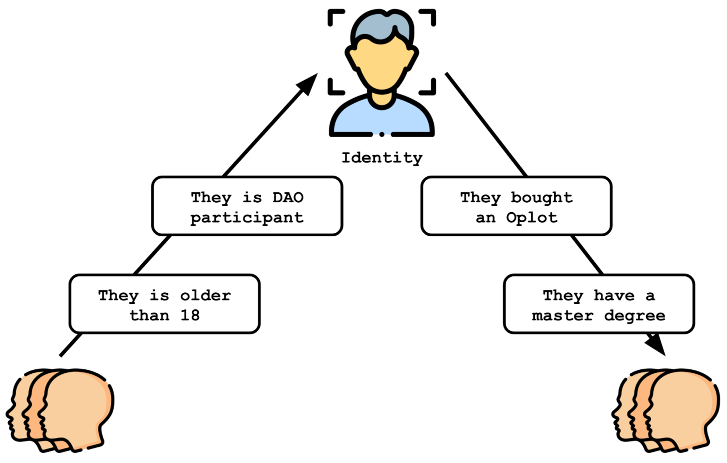
State and LightweightState contracts.. 22
QueryValidator.. 22
ZKPQueriesStorage.. 23
Verifier contracts... 23
Conclusions... 23
Appendix A. Sparse Merkle Tree operations and its usage... 24
Appendix B. Explanation of schemas in the Iden3 protocol.. 28

What is Iden3?

Identity and claims

Iden3 is the protocol that allows the implementation of self-sovereign identity that can be used in different systems. Each identity can be almost anything and is characterized by:

- claims issued by other identities about it
- claims issued by this identity about others



An identity can belong to a person, a company, an organization, a DAO, or a government. Identity can even belong to a thing: a chair, a room, a bot, and so on. When discussing identities, we refer to "identities as accounts".

Iden3 protocol categorizes identities into two types:

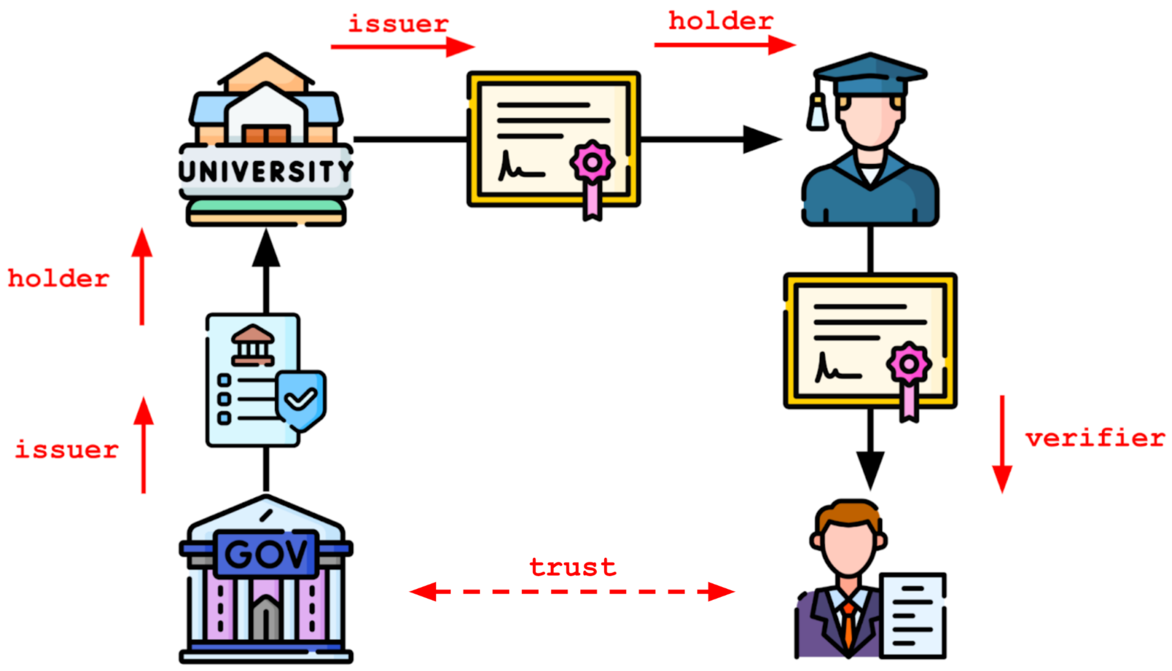
- Issuer
- Holder

However, an identity can simultaneously be an issuer and a holder. Let's consider a simple and transparent example.

Let's say we have two identities: a university and a university student. Upon completing their studies, the student will receive a diploma. In this case, we can consider the diploma as a claim, where the university is the issuer, and the student is the holder.

During employment, the student can present their claim, certifying their education at a specific university. The employer verifies the stamp on the diploma and is confident that the corresponding trusted issuer produced it. In this case, the employer acts as a verifier, verifying the data provided by the new employee.

But why does the employer trust a particular university? Because the university has the necessary license, which is also a claim issued by a government authority serving as another issuer. Therefore, the university acts as both an issuer and a holder simultaneously.

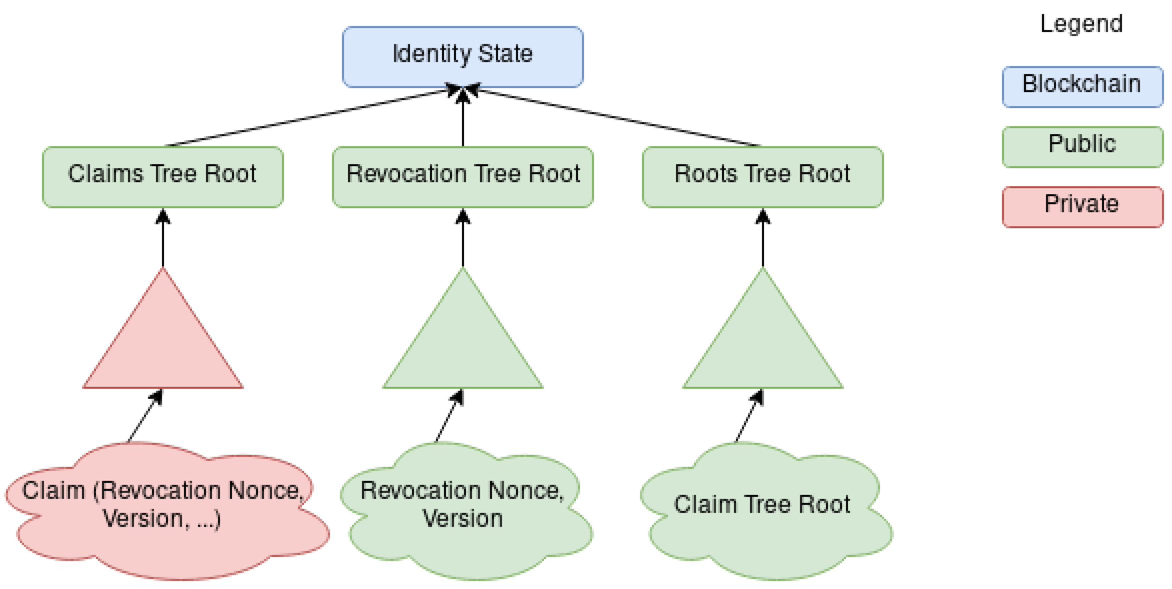


Examples of Claims:

- A certificate (e.g., birth certificate)
- A debt recognition
- An invoice
- An Instagram "Like"
- An endorsement (reputation)
- An email address
- A driving license
- A role in a company
- ... Almost anything!

Every identity has a state. The **identity states** are published on the blockchain under the identity's identifier, anchoring the state of the identity with the timestamp when it is published. In this way, the identity claims can be proved against the anchored identity state at a certain timestamp. Identities follow the transition functions to transition from one state to the other. The identity state consists of:

- Claims Merklee Tree
  Contains claims that are issued by this identity
- Revocations Merkle Tree
  Contains revocation nonces of the revoked claims
- Roots Merkle Tree
  Contains Claims Merkle tree's roots



Polygon ID

You can think of the Polygon ID as one of the Iden3 protocol implementations. The protocol defines, on a low level, how the parties listed above communicate and interact with each other. Polygon ID is an abstraction layer to enable developers to build applications leveraging the Iden3 protocol.

DID and Verifiable Credentials

Every identity is recognized by a unique identifier called DID (Decentralized Identifier). Every identity-based information is represented via Verifiable Credentials (VCs). In the simplest terms, a VC represents any information related to an individual/enterprise/object. The VC could be as simple as the entity's age or its highest degree, e.g., it could be a membership certificate issued by a DAO, for instance.

You can think that claims and verifiable credentials are the same things.

The toolset made available by Polygon ID and Iden3 fully complies with the W3C standards. PolygonID has a definition spec. for the Polygon ID DID method.

VCs contain a "proof" (i.e., some cryptographic data that can be verified to make sure of some statement) field that stores one or both of the issuance proofs. Also, VCs contain Non-revocation proof (see below).

Verifiable Credential proof types

- **Signature issuance proof (SIG)**
  It proves that the specific issuer has issued the claim (VC) by a **BJJSignature**, where "BJJ" is a BabyJubJub elliptic curve. That means the issuer isn't obligated to add the claim to the issuer's Claims tree.

- **Merkle tree issuance proof (MTP)**
  It proves that the specific issuer has issued this claim (VC) by a Merkle tree proof that this claim is included in the issuer's Claims Merkle tree, so in the issuer's state.

- **Non-revocation proof (NRP)**
  It proves that the claim hasn't been revoked by an issuer. Namely, the claim's revocation nonce (revocation id) isn't included in the issuer's Revocation tree.

The claim issuance proof can be received only once by a holder from the issuer. Because whenever a claim was signed, or it was included in the issuer's state, it means that you can use the same signature whenever and the claim can't be removed from the issuer's Claims Merkle tree even by the issuer. In contrast to the Non-revocation proof type, that must be updated every time during a Zero-Knowledge Proof generation.

Usually, an issuer first sends a VC to a holder only with **Sig proof**, but after some time, when a state that includes the holder's claim was published, the issuer sends a notification to the user that he can update a VC with an **MTP proof**.

Zero-Knowledge Proof (ZKP)

One of the most important features of the Iden3 protocol, enabling most of the magic, is Zero-Knowledge Proof technology. It is a method for one party to prove a statement's truth to another party without revealing any other information apart from the statement's validity. For illustrative purposes, let's consider the following example:

Say, you want to enter a nightclub, and you need to prove to the bouncer that you are over 18. But you don't want to reveal your name, address, or anything else irrelevant to him. With Zero-Knowledge Proof, you can prove that you hold the key that belongs to an identity that the state says is over 18 without revealing anything else about that identity.

Iden3 ZKP Circuits

Here are the main circuits written and used by the Iden3 Identity protocol.

- **Auth**
  The circuit is mainly used by holders for identity authentication. Prove that the user is an identity owner.

- **StateTransition**
  The circuit is used within state transition by an issuer. Prove that the state was changed correctly and changes correspond to the protocol. The changes can be either claim issuance or revocation that require the issuer's state changes.

- **CredentialAtomicQueryMTPV2**
  The circuit is mainly used by holders for proving some statements about verifiable credentials, like, that an "age" field is greater than 18.

  It uses the verifiable credentials' **Merkle tree issuance** and **Non-revocation** proofs.

- **CredentialAtomicQuerySigV2**
  The circuit is mainly used by holders for proving some statements about verifiable credentials, like, that an "age" field is greater than 18.

  It uses the verifiable credentials' **Sig issuance** and **Non-revocation** proofs.

The **CredentialAtomicQueryMTPV2** and **CredentialAtomicQuerySigV2** have two implementations: for on-chain and off-chain verification. The on-chain has some of the public data hashed, which reduces the size of input data for verification and the verification price.

What can you achieve using Iden3 and PolygonID?

1. **Privacy using Zero-Knowledge Proofs**: An Identity Holder can keep an identity's personal data private using zero-knowledge proofs. During the process of VCs verification, it just needs to show proof that he owns a VC that matches certain criteria without letting the Verifier know of the actual VCs. Another aspect of privacy comes from the fact that the Issuer would not be able to track the usage of VCs by an individual once it has been issued.

2. **Off-Chain and On-Chain Verification**: Verification of proofs can be done either off-chain or on-chain via Smart Contracts. For example, developers can set up a contract that airdrops tokens only to users that meet certain criteria based on their credentials' data.

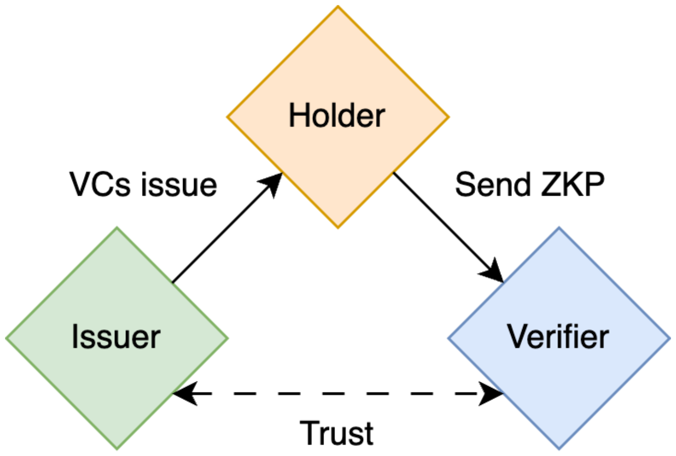
3. **Self-Sovereignty**: Polygon ID renders self-sovereignty in the hands of the user. The user is the only custodian of his/her private keys; user-controlled data can be shared with third parties without taking any permission from the Issuer that has issued the VCs to the user.

4. **Transitive Trust**: A transitive trust between the actors of the triangle means that the trust between two entities in one domain or context can be easily extended to other domains or contexts. For instance, the information generated by an Issuer can be conveniently used by more than one Verifier without asking for permission. Along similar lines, an Identity Holder can build up his/her trust by collecting multiple credentials from different Issuers in one digital wallet.

Base architecture

It must be noticed that Iden3 and PolygonID mainly use *Golang* language for the Back-End part, *Solidity* language for the EVM-compatible Smart contracts, *JavaScript* language for the web, and *Flutter* framework for mobile applications.

You can find libraries for almost everything that you need for the protocol realization in Iden3 and PolygonID GitHub repositories.



Core services

We can single out some main figures that participate in the different flows:

- *An **Issuer** service* - a centralized service that is controlled by some organization and responsible for VCs that contain some specific data, issuance for identities that fulfill certain conditions.
- *A **Verifier*** - an instance that can verify a ZKP about some data stored in VSc provided by an Identity and based on the result, makes a decision. *It can be either a Smart contract or a Back-End service.*
- *A **Wallet (Holder)*** - an application that can create and manage a user's Identity. Under the management, Identity creation, private data storing, VCs storing, proof generation, etc., is hidden.

The identity and claims management are different for the issuer and wallet.

Issuer

The back-end service, or set of services, includes both public and private APIs. These APIs are responsible for implementing all the logic related to identity and claim management, such as state transitions and revocation.

The public API methods are designed for interactions with holders and verifiers, while the private ones are meant for handling claims and identity management.

It is advisable not to overly expand the Issuer logic beyond the protocol to avoid unnecessary complexity in the service. Instead, the Issuer should be treated as the method responsible for issuing the claim. In contrast, an external service should handle the logic of determining who will receive the claim and conducting different user verifications.

Resources

The issuer service is resource-demanding. Usually, it needs to publish the issuer's state on-chain once per some period of time. During the publishing process, it generates a State Transition Zero-Knowledge proof that requires large enough resources for efficient processing.

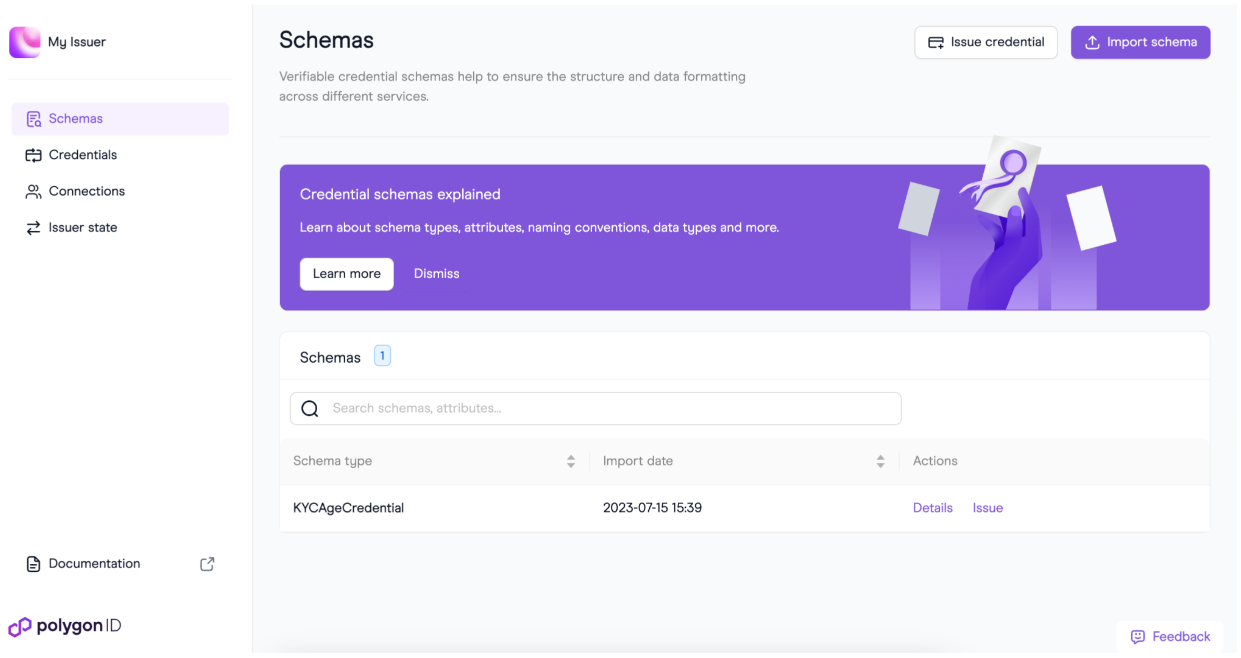
Requirements

- The service MUST be able to create and manage identity. Identities management consists of storing identities' info in a database. Retrieving the previous identity on service start, correctly managing its Merkle Trees.
- The service MUST be able to read the claims schemas, create and manage claims, and return verifiable credentials by request.
- Claims management implies adding claims into either a Claims Merkle tree or Revocation Merkle Tree and its versioning.
- The service MUST be able to publish the issuer's state on-chain by either request or in some configured period.
- The service MUST implement the Iden3 protocol and API for interactions with holders. It will provide the ability to retrieve an issued claim for the holder and update its non-revocation proof.

Polygon ID issuer node

Polygon ID has a well-maintained open-source implementation of the issuer. It has every feature presented in the Iden3 protocol, contains a set of services responsible for different parts, and has an admin panel.

With the admin panel, you can easily add new claims schemas, directly issue VCs, or create convenient QRCodes for claim issuance that mobile applications can use. It is also possible to manage already issued VCs.



Schemas

Verifiable credential schemas help to ensure the structure and data formatting across different services.

Credential schemas explained

Learn about schema types, attributes, naming conventions, data types and more.

Schemas 1

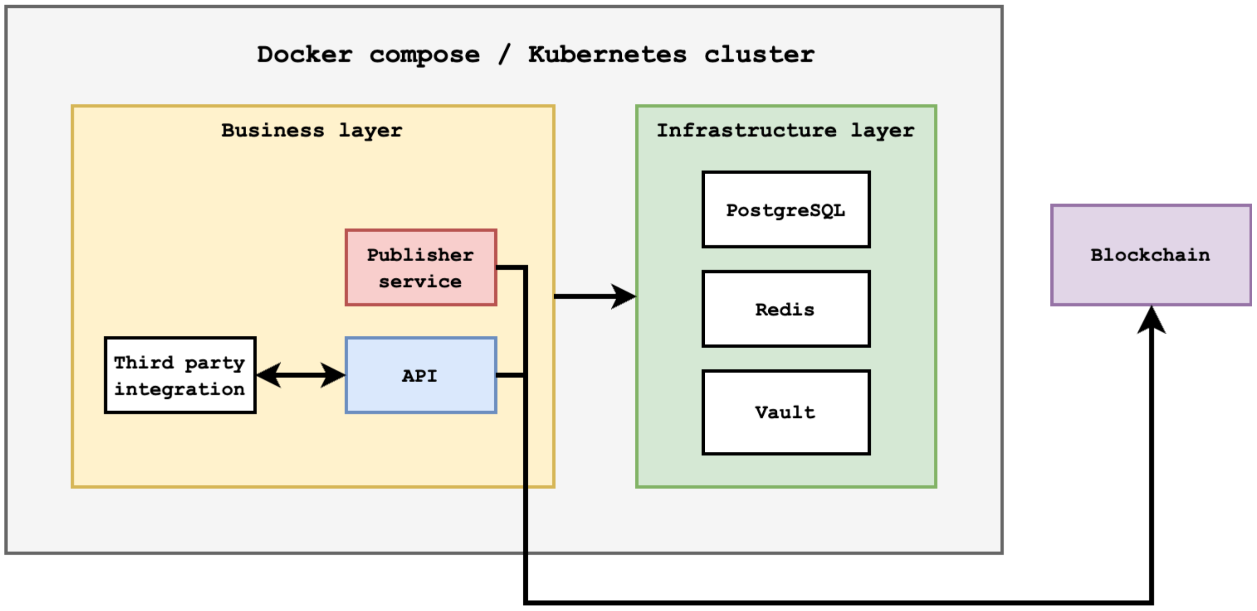
Search schemas, attributes...

Schema type	Import date	Actions
KYCAgeCredential	2023-07-15 15:39	Details Issue

The back-end part provides an easy-for-use API that is fully documented. It provides the ability to create several identities and conveniently manage them, issue verifiable credentials and communicate with users' wallets.

Services use Vault with an extension that improves security by managing the keys so that anyone can't receive direct access to keys but only can sign the messages if they have an access key.

Redis is used for caching the schemas we use in the Issuer Node. The schemas are downloaded from IPFS (or any other source) and stored on Redis. This way, every time the Issuer Node issues a Credential, it doesn't need to fetch the schemas from an external source; it can fetch it directly from Redis. This boosts the performance of the application.



Custom issuer

Also, it is possible to develop an issuer service with any custom logic based on the Polygon ID issuer node or from scratch using Iden3 libraries that implement all the low-level cryptography and protocol primitives like identities and claims management.

It is highly recommended that a custom issuer has backward compatibility with the Iden3 protocol.

Wallet (Holder)

A digital wallet is software that can hold and manage users' Credentials. Based on the principles of **Self-Sovereign Identity (SSI)** and cryptography, a wallet helps its Holder share data with others without exposing any other sensitive private information stored on it. Only the wallet holder can decide which information to share with other entities and what needs to remain private.

The wallet is a front-end or mobile UI application representing the user's identity. It includes necessary logic for managing identities, such as creating, removing, importing, and exporting them. Additionally, it handles the logic for managing claims, such as receiving claims from an issuer, removing claims, displaying them in a human-readable format, and utilizing them for user purposes. Moreover, it can generate Zero-Knowledge proofs for certain statements about the information it contains.

Optionally it may contain implementation for the profile nonces, which provides additional user anonymity.

It is important to know that Iden3 uses another from Ethereum elliptic curve called BabyJubJub. So, the cryptography is different from the default.

Requirements

- The wallet MUST be able to create and manage identity. Identities management consists of creating, removing, importing, and exporting it. Additionally, it involves utilizing the Profiles feature and supporting different networks and DIDs.
- The wallet MUST implement the Iden3 protocol for interactions with issuer and verifiers.
- The wallet MUST be able to generate *Auth*, *CredentialAtomicQueryMTPV2*, and *CredentialAtomicQuerySigV2* proofs and use them for off-chain or on-chain verification.
- The wallet MUST integrate with Metamask or any other wallet that can send transactions on-chain for proof verification.

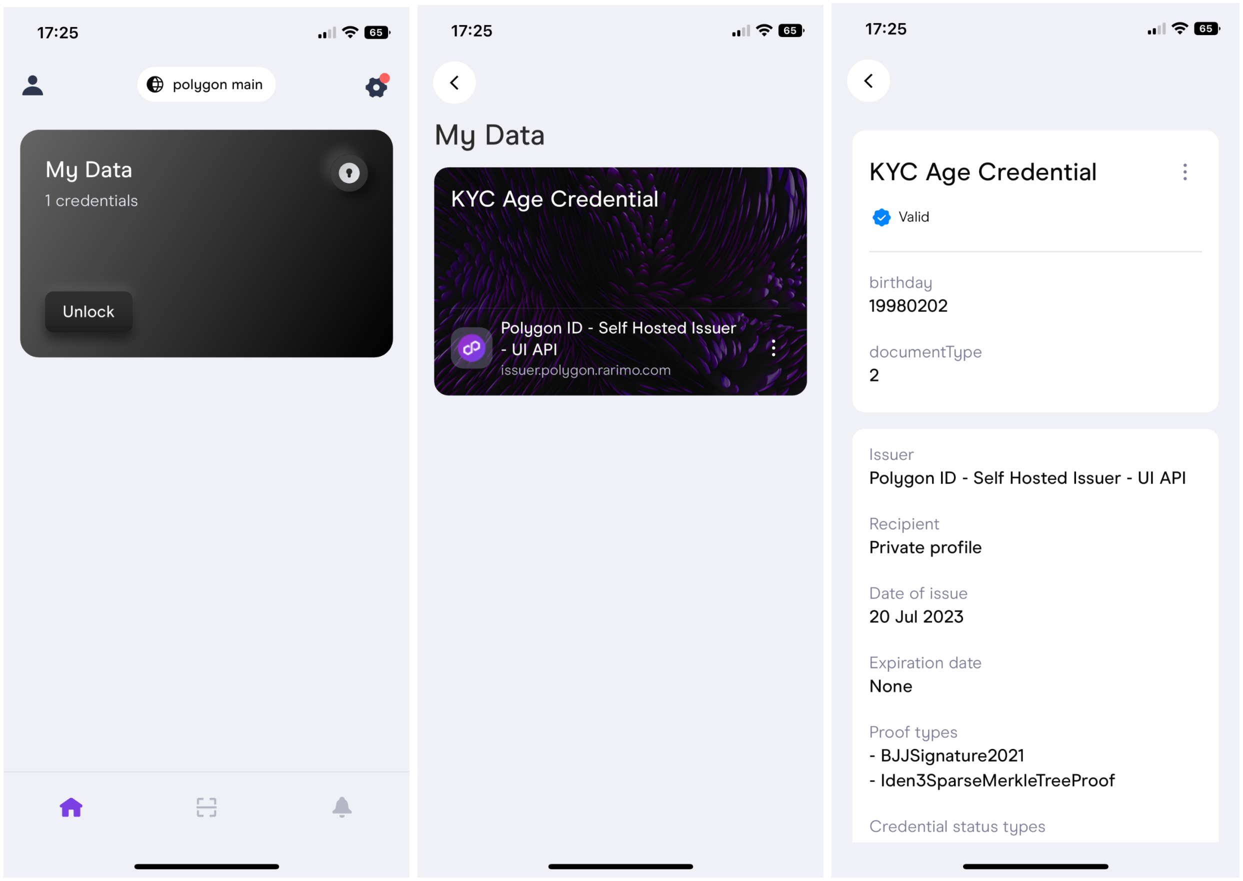
Polygon ID Wallet

Polygon ID offers a mobile wallet that fully implements all the features of the Iden3 protocol. You can easily find and download it from the Play Market for Android devices or the App Store for iOS devices.

The Polygon ID Wallet supports the following features:

- Privacy by design and Self-sovereignty: The user fully controls his/her identity data and exchanges credentials with other identities without needing an intermediary or centralized authority.
- Open and Permissionless.
- Fetching, storing, and managing credentials.
- Generating cost-optimized zero-knowledge proofs for credentials verification.
- Communication with Issuer and Verifier.
- Identity recovery using seed phrase.

It is worth noticing that, to comply with the principles of the Self-Sovereign Identity (SSI), all the credentials are stored only locally on the user's wallet and are not stored on-chain; this ensures strong privacy for the sensitive data related to the user's credentials.



birthday
19980202

documentType
2

Issuer
Polygon ID - Self Hosted Issuer - UI API

Recipient
Private profile

Date of issue
20 Jul 2023

Expiration date
None

Proof types
- BJJSignature2021
- Iden3SparseMerkleTreeProof

Credential status types

Custom Wallet

The wallet can be built in several ways. For example,

- **Mobile application (Android, iOS)**

For mobile applications, there is Flutter-SDK which allows to integrate Polygon ID identity system into Flutter apps.

- **Metamask extension (Snap)**
  Snaps is an open-source system that allows anyone to safely extend the functionality of MetaMask, creating new web3 end-user experiences. So it is possible to build digital Identity right into the Metamask just as the default browser extension.

- **Web wallet (Not recommended**)
  For the Web wallet, you'll find an abundance of JS libraries that provide every essential component of digital identity in alignment with Iden3. These libraries offer the necessary tools and functionalities to manage digital identities within the Web environment efficiently.

As you can see, both Iden3 and Polygon ID offer distinct SDKs and libraries that make it convenient to build various types of wallets from the list mentioned earlier. These SDKs and libraries provide developers with the necessary tools and resources to create wallets tailored to their specific requirements, ensuring flexibility and ease of implementation.

Verifier

The verifier is any platform that wants to authenticate users based on their credentials. Verifiers can be on-chain (smart contract) or off-chain (back-end service).

A verifier can provide Parameters that need to be verified using Query Language that can be transferred into a QRCode or deep-linking. The query can use several basic operators:

- Must be a verified human to vote for a DAO-specific proposal - **equals** (operator 1).
- Must have been born before 2000-01-01 to access an adult content website - **less-than** (operator 2).
- Must have a monthly salary greater than $1000 to get a loan - **greater-than** (operator 3).
- Must be an admin or a hacker of a Dao to enter a platform - **ìn** (operator 4).
- Must not be a resident of a country in the list of blacklisted countries to operate on an exchange - **not-in** (operator 5).
- Must not be a resident of a specific country - **not-equal** (operator 6).

Once the wallet application receives a query, it can use one of the requested circuits to prove the statement. After generating the proof, the wallet can transmit it either as an on-chain transaction or through a callback URL specified in the query to the back-end verifier service.

The verifier contract is generated from the circuits and trusted setup, but the business logic is based on the contract wrapper.

Iden3 contracts system

State

At the heart of the Iden3 contract-side system is the State contract. It is used for storing information about the user's states, which makes state transitions. The GIST (Global Identity State Tree) is also stored and created on this contract. Any identity can transit its state on-chain, but in most cases, states are published only by issuers. This action is necessary to be able to validate the **CredentialAtomicQueryMTPV2OnChain** and **CredentialAtomicQuerySigV2OnChain** circuits. To transit the state, you must call the **transitState** function, which takes the following parameters:

- **id** - identity’s ID that transits state
- **oldState** - previous state of this identity
- **newState** - identity’s new state
- **isOldStateGenesis** - whether the previous state is a genesis state or not

- **ZKP data** - parameters for verification of ZK proof of **StateTransition** scheme

ValidatorContracts

Validator contracts are the next layer of the protocol. The State contract is universal and can be one for the whole network, but there can be many validators.

The Iden3 protocol has an abstract *CredentialAtomicQueryValidator* contract that implements the *ICircuitValidator* interface. This contract has functions to get the circuit identifier index or the challenge index in the list of public ZK proof's outputs, but the main function of this contract is **verify**.

The **verify** function accepts a ZK proof (public signals and **a**, **b**, **c** points) and the hash of the query to verify a circuit.

During verification is checked:

1. The ZK proof itself.
2. Presence of GIST root from the proof in the State contract.
3. Presence of identity's id and identity's state from the proof in the State contract.
4. Checking that **issuerClaimNonRevocationState** is the current issuer's state. If it is not, the contract checks if the state is in the period when the proof is still considered valid.

To verify the ZK proof, a verifier contract is used, which can be generated using **SnarkJS** from the desired circom schema.

ZKP Queries hash

ZKP Query hash is required for the validator to check that the right claim's fields are checked in the ZK proof.

The logic for storing and retrieving a ZKP Query is placed in the *ZKPVerifier* contract from Iden3. The ZKP Query structure is placed in *ICircuitValidator*, and contains the following fields:

- **schema** - identifier of the schema of the claim
- **claimPathKey** - path to the field in the merklized claim that will be checked
- **operator** - operation to check the value (EQ, NON, IN and etc.)
- **value** - an array of values to which the value from the claim will be compared.
- **queryHash** - ZK Query hash
- **circuitId** - circuit name

The ZKP Query hash is calculated on the contract as follows:

queryHash = **Poseidon(**schema, slotIndex, operator, claimPathKey, claimPathNonExists, values sponge hash**)**

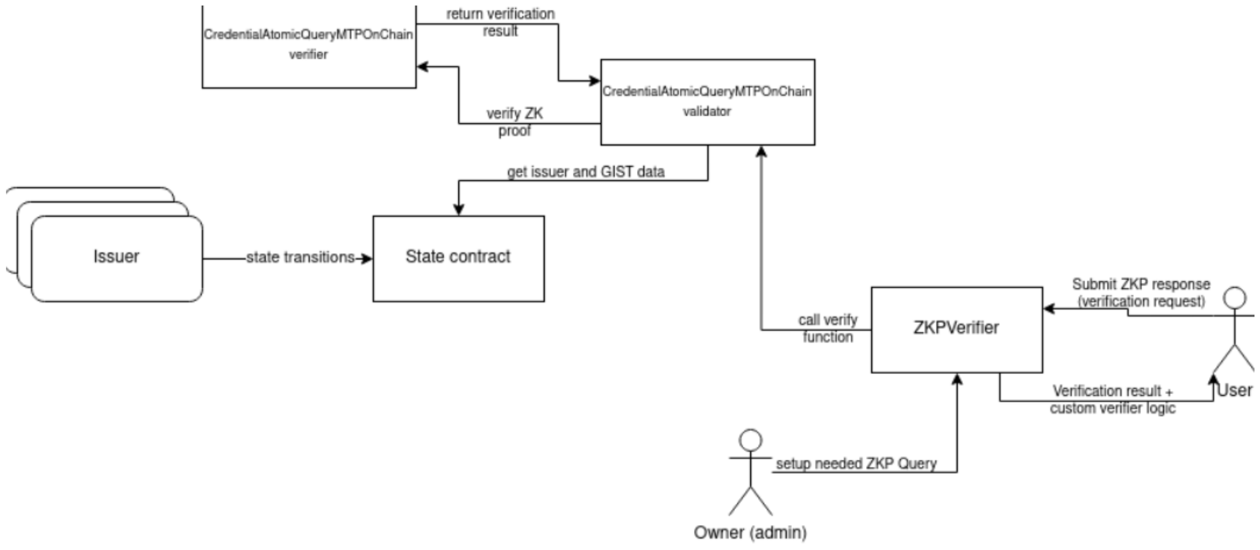
Where *slotIndex* and *claimPathNotExists* are always zero since the iden3 *ZKPVerifier* contract only works with merklized claims.

ZKPVerifier

The Verifier contract is the top layer of the Iden3 protocol, with which the user or other contracts interact directly.
Iden3 provides a *ZKPVerifier* contract that has the functionality to fully validate the user's claim using the *CredentialAtomicQueryValidator* contract described above. This is just an example implementation, so a different verifier can be written for each specific case.

Conclusions on the basic implementation

This implementation is great when your protocol supports only one network, as it is quite difficult to scale to other networks. All issuers will need to duplicate their states to all supported networks, which is quite complex and expensive.


CredentialAtomicQueryMTPOnChain verifier
return verification result
CredentialAtomicQueryMTPOnChain validator
verify ZK proof
get issuer and GIST data
Issuer
state transitions
State contract
Submit ZKP response (verification request)
call verify function
ZKPVerifier
Verification result + custom verifier logic
User
setup needed ZKP Query
Owner (admin)

How to dive into digital identity?

In this section, we consider that you already know for what you need a digital identity.

There are 3 obvious use cases:

- Develop an issuer or use an existing one for verifiable credential issuing.

  The issuer service's endpoints for verifiable credentials issuance is considered to be either private or use some authentication. So, there are two ways the issuance can be organized:

  - There can be administrators that, via an admin panel will directly issue claims to the end users. Sounds not good, doesn't it?
  - There might be a service that already exists within the system where the logic of issuance is placed. This service would act as a KYC service, determining who can receive claims based on the user's information. It will then utilize a method from the issuer service to issue the claim.

  If the Polygon ID issuer-node is used, any code changes can be done to achieve a desirable result. If was chosen a way to develop a custom issuer, there must be specific requirements, what should be done, and which features should be included.

  After preparing the issuer service, we should create schemas for the desired claims. This includes determining the number of schemas required, assigning suitable names to each schema, specifying the names and types of fields within them, and clarifying the purpose of the stored data.

  Once the issuer service is deployed, the schemas are prepared, and the claim issuance process is defined, we can consider the issuer environment as ready.

- Develop a wallet for verifiable credentials holding or embed into an existing one.

  First of all, the type of wallet should be defined. As well as features, ZKP circuits and networks will be supported. Then the target application can be developed.

- Embed a verifier into an existing system to verify Zero-Knowledge proofs based on any stored in verifiable credentials user data.

  The verifier can operate either on-chain or off-chain. In both cases, it is necessary to specify what will be verified and the circuits that will be used. In the on-chain scenario, the verifier contracts must be generated from circuits.

For on-chain verification, a query for verification requests should be defined and made accessible to the end user's wallet. This enables users to initiate verification processes. The query can be accessible, for example, via QRCode for mobile applications or via method call for Snap extension.

After that, a business logic that will be executed after a ZKP verification can be implemented and deployed.

Multichain with Rarimo

Prerequisites

In the basic Iden3 protocol, only one chain is supported. If you use the Polygon ID ecosystem, this chain is obviously either Polygon Mainnet or Polygon Mumbai for test purposes. You can run the entire ecosystem on another network, like Ethereum, but VCs issued on Polygon won't be accessible for proof generation on Ethereum.

Every proof about the data stored in VC verifies that the VC was issued and still wasn't revoked by a specific issuer. This verification is achieved by checking the inclusion or exclusion of certain data in the issuer's state, which is publicly available on-chain. Therefore, if the issuer's state has not been published to the Ethereum network, the proof cannot be verified using the Ethereum chain.

Of course, an issuer can publish the state on every chain, but it will be costly due to ZKP verification.

Solution

Rarimo cross-chain protocol allows users to transfer data across supported chains. For example, it can be a token transfer, identity transfer, or just a text message transfer. Such logic relies on several Rarimo components: decentralized oracles, decentralized ECDSA threshold signature producers, and Cosmos-based blockchain core.

After adding EVM compatibility into the Rarimo core, it receives an opportunity to deploy Solidity smart contracts directly into the blockchain. So, suppose identity providers use the Rarimo chain to publish their Iden3 states instead of others. In that case, they receive instant compatibility with cross-chain features in a co   dates price.

After publishing state updates on the state smart contract (Rarimo chain uses vanilla iden3 smart contract so that it will be compatible with all existing issuers) core system will immediately initiate the signing flow, and after several minutes, the signed by the threshold signature providers message with last state information will appear in Rarimo core state.

The caller side (wallet) can check and submit the signed information to any lightweight state smart contract on any supported EVM chain. The signed data contains compacted and easy-to-verify information about the issuers' state transitions. After that, user identity proofs become available on that chain, and anyone who wants can integrate their Verification smart contracts to use them.

And one more thing: all issuer states will be additionally aggregated into one Merkle tree (aka. **States Merkle tree**), so, by updating once a global identity state, all other states from all other issuers will also be updated. It significantly reduces the amount and cost of transactions to be published on target chains.

How to achieve cross-chain?

It is worth noting that the cross-chain feature doesn't break the backward compatibility with the basic Iden3 protocol. It is just an extension.

Issuer service particularities

The service must have an endpoint to return the claim's **Merkle tree issuance proof** by the VC's identifier.

The Merkle tree issuance proof, which is stored in the proofs field of the Verifiable credentials JSON structure, usually within an index 1, must have an additional field id that stores the link that can be used to receive an updated claim's Merkle tree issuance proof.

Wallet changes

The new endpoint, added to an issuer's service, is used the same as the one that returns a claim's Non-revocation proof. So, when the user wants to prove something with **CredentialAtomicQueryMTPV2** ZKP type or its on-chain version, they request a new claim's Merkle tree issuance proof that is tight to the last issuer's state by the corresponding link.

Also, a wallet has to implement the following flows to support the calling of the lightweight state smart contracts on target chains:

1. Updating the state with lightweight smart-contract on the direct chain.
2. Adding Merkle tree proof of the issuer's state inclusion into the States Merkle tree, among with a main ZKP.

State updating requires fetching the state data from Rarimo and target original chain StateV2 smart contracts, and if they are not equal - fetching witness (ECDSA threshold signature + Merkle tree proof) from Rarimo core RPC and submitting the corresponding transaction into the target chain light-weight State contract.

Merkle tree proof verifies that the requested state is included in the aggregated contract state (States Merkle tree).

Both witness and state inclusion Merkle proof can be fetched from Rarimo's backend systems.

Smart contracts particularities

Main motivation

As mentioned above, the standard iden3-based system works without problems within a single network but it has problems with scaling to other networks. The solution to this problem is the **Rarimo** network whose validators can issue different TSS signatures that can be easily verified on any EVM-like network.

State and LightweightState contracts

Standard Iden3 *State* contract is deployed on the Rarimo network. The interaction with the contract on the Rarimo chain and EVM is the same.

All other supported networks will use the **LightweightState** contract, which stores less data but enough for validator contracts.

LightweightState has a **signedTransitState** method that takes the following parameters:

- **identitiesStatesRoot** - identities' states MerkleTree root.
- **gistData** - GIST data from the main State contract.
- **proof** - TSS signature and all necessary associated data.

The main problem was that the core network can have a huge number of identities, so it would be very expensive to simply migrate all the stats of all identities. To solve this problem in Rarimo validators, there is a special oracle module that monitors events on the main State contract (that is deployed in the Rarimo network) and builds a MerkleTree whose leaves store the actual **issuerId**, **issuerState,** and **createdAtTimestamp** for each issuer. With this Merkle tree, we can confirm that there is a particular identity with the desired identity state and state publish timestamp.

Due to the added logic with MerkleTree, the LightweightState has a **verifyStatesMerkleData** method that can be used to verify that the identity data is correct.

QueryValidator

Similar to the main iden3 protocol, we have **QueryValidator** contracts that are needed to verify a ZK proof. Due to the usage of the **LightweightState** contract on the secondary networks, it was necessary to modify the validator contracts as well, namely the **verify** method. This method on the updated contract takes one additional parameter - the **StatesMerkleData** structure from the **ILightweightState** interface.

The StatesMerkleData structure has the following fields:

- **issuerId** - issuer id.
- **issuerState** - issuer's state.
- **createdAtTimestamp** - a timestamp when the state was transited.
- **merkleProof** - identities' states MerkleTree proof.

Inside the method, **GISTRoot** and **queryHash** are checked as in the original contracts. There is also a check that the ZK proof doesn't use a deprecated issuer state.

ZKPQueriesStorage

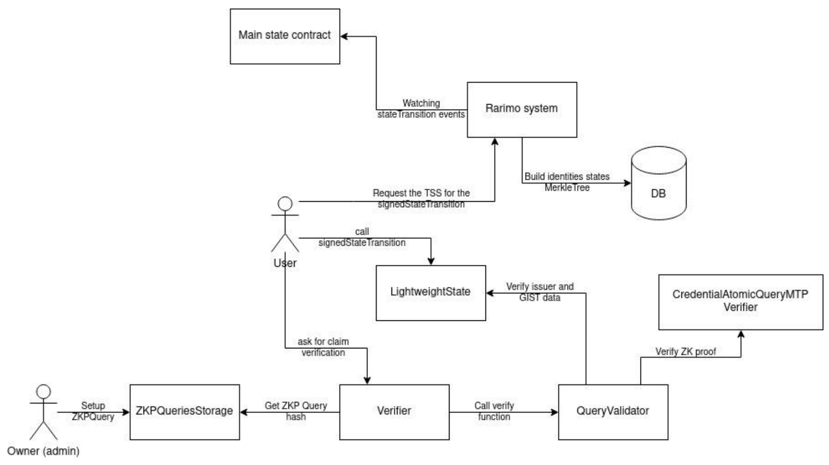
ZKPQueriesStorage was created for convenient storage and retrieval of **ZKPQueries**. It can be used to set different ZKPQueries by identifiers. Also, the contract has methods for retrieving the necessary data and separate functions for calculating **queryHash**.

Verifier contracts

The verifier contract has changed only slightly, with only one additional mandatory parameter added in the form of a **StatesMerkleData** structure. The verifier contract can get a queryHash with the required ID, so to verify the user's ZK proofs for a particular claim it just needs to call the **verify** method on the **QueryValidator** contract.

Conclusions

This solution is quite complex to implement technically, but with minor changes for the consumer, a fully working version of the cross-chain identity protocol can be achieved, which is a great result.


Main state contract
Watching stateTransition events
Rarimo system
Build identities states MerkleTree
DB
Request the TSS for the signedStateTransition
User
call signedStateTransition
LightweightState
Verify issuer and GIST data
CredentialAtomicQueryMTP Verifier
ask for claim verification
Verify ZK proof
Owner (admin)
Setup ZKPQuery
ZKPQueriesStorage
Get ZKP Query hash
Verifier
Call verify function
QueryValidator

Appendix A. Sparse Merkle Tree operations and its usage

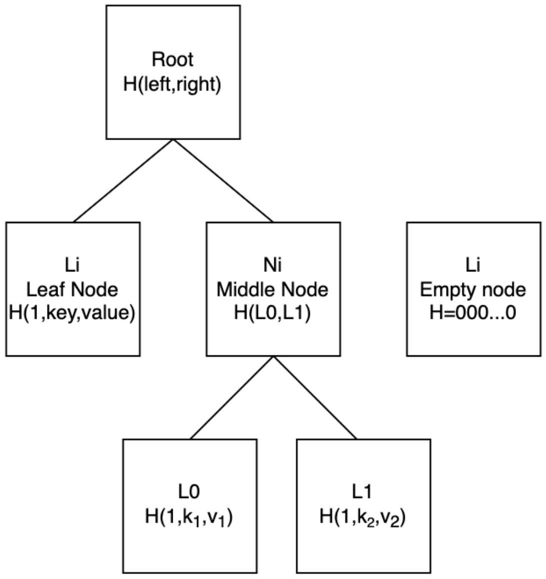
Sparse Merkle Tree (SMT) is a key-value binary Merkle tree widely used in the Iden3 protocol. It differs from the default Merkle tree in that, firstly, it has a key-value structure (two nodes with the same key can't exist in the tree at the same time, and the particular key always takes the same position in the tree), and secondly, if we reach an empty node (while inserting a new leaf), without going through the whole key, we can paste it in place and not go below.

What is a node?

The SMT has 3 types of nodes: empty, leaf, and middle. The node contains the key-value pair, a hash of the node (that depends on the node's type), and its two children hashes.

- Empty node - a precomputed node with a <0,0> key-value pair, hash of zeros, and no children.
- Leaf node - a node with key-value pair and Hash(1, key, value). Leaf nodes don't have children nodes.
- Middle node - a node with two children nodes (not empty). Stores the hash of children's hash concatenation (i.e., Hash(LeftNodeHash, RightNodeHash)). The middle node at height 0 is called the root node, which "describes" the whole tree.

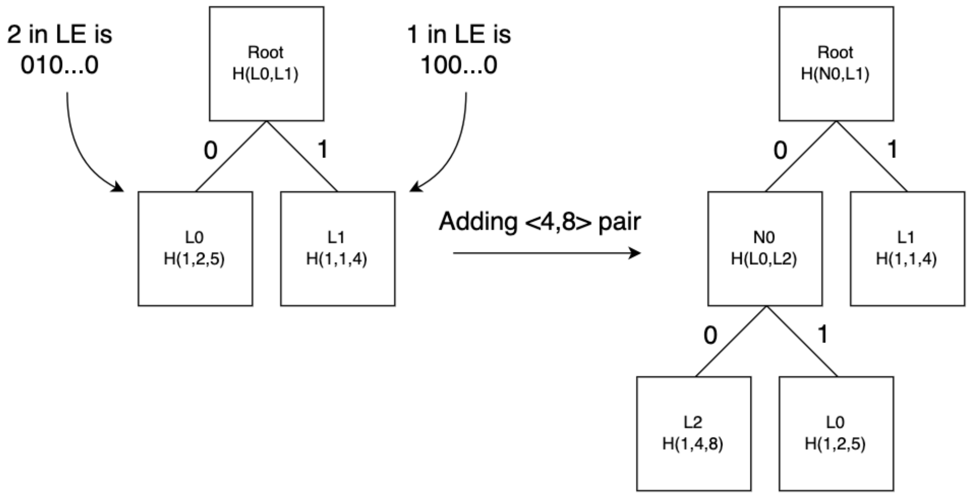
In visualization, we mark the root node as "Root", the middle node as "N_i" and the leaf node as "L_i". We mark an empty node as a leaf node with <0,0> key-value pair. The hash function is marked as H(values).



Addition of the node to the tree

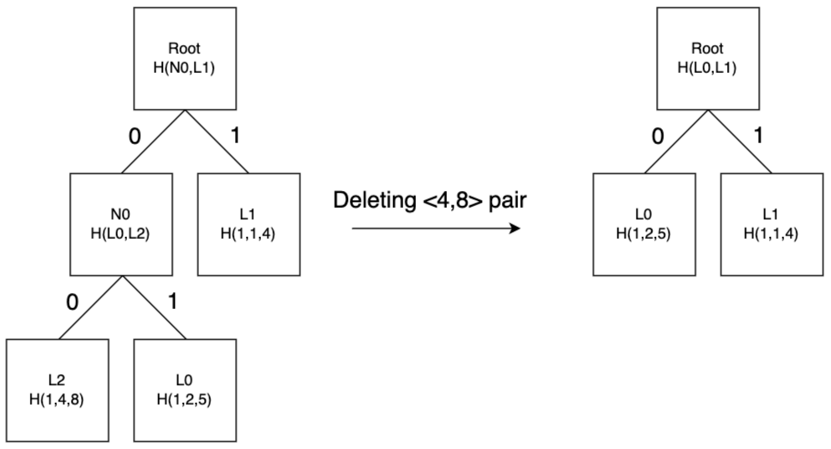
Basically, we go through the whole tree (from the root) using “path” (key in binary little-endian representation) until we find an empty node or a leaf node. If a leaf node is found, it will be split, and they will be inserted at the level where their bits differ.

For example, we have a tree below with two nodes (their keys are 2 and 1). We insert a <4,8> key-value pair (4 in LE is 0010...0), and the L0 node splits into two leaves (because their next bits differ).



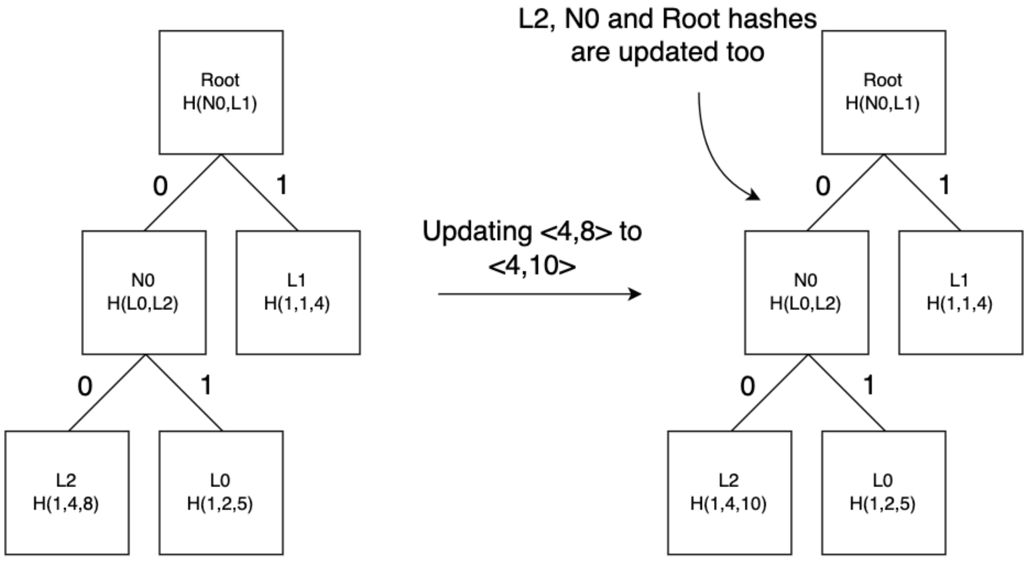
Deleting the node from the tree

Delete function works inversely to the addition function. It finds a node with a specified key, removes it, and recursively rebuilds the whole tree and the root. This functionality is available but not used in Iden3 trees.



Updating the node

You can update the node's value with a certain key. For example, we can update a node <4,8> to value 10 (so it will be <4,10>). It will change the hash of the node and all nodes higher (subtree and the root root).



Sparse Merkle Tree Proofs

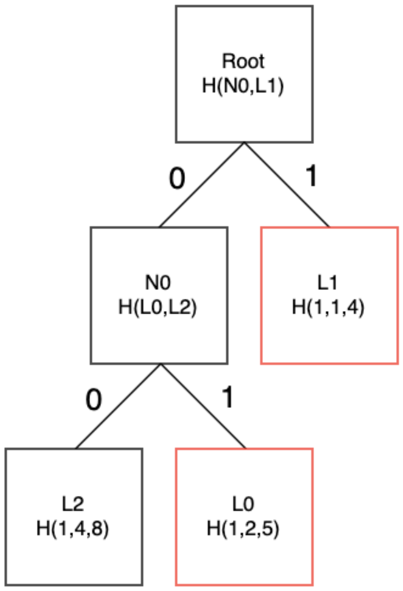
Sparse Merkle Tree can create two types of proofs - inclusion and non-inclusion (sometimes called exclusion).

To create such proofs, we go from the root (using the provided key) until we reach a leaf (or an empty node), and every time we choose a node, we store the sibling node's hash that we haven't chosen (our tree is binary, so every time we choose the left path, we store the hash of the right node and so on).

Assume, we want to prove that an entry with key 4 with value 8 exists in our tree (in the picture below). Because our key in the binary LE presentation is 0010…0 we choose the left path from the root twice (storing all right nodes' hashes as an array) and find the leaf node. Then we check whether provided and stored keys are the same, in the positive case, we return the value of this node, siblings array (in our case, it's all right nodes' hashes, i.e. hash of L1 and L0 at the picture) and this is called "**proof of inclusion**".

In the negative case (i.e. when keys are different, for example, if there was key 8 instead of 4) it is called "**proof of non-inclusion**". We provide node <8,8> as an "auxiliary node" and all siblings till the root, saying that this node is placed instead of the one we want.

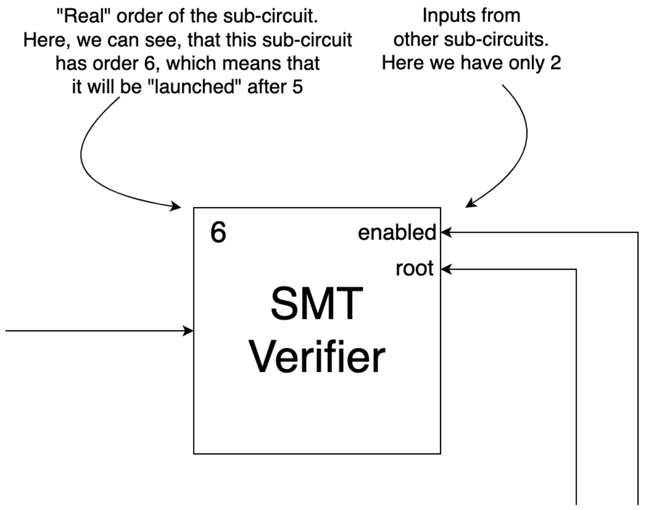
You can notice that we check only the “key” existence. This is because “value” is not often used in the Iden3 protocol, but this check can be done on the user side. The user, asking for the MTP of a certain node, knows its value, so receiving the siblings, he can calculate the hash of the node and check the MTP. If the root doesn’t match, then this node doesn’t exist.



Appendix B. Explanation of schemas in the Iden3 protocol

Most circuits have **input** and **output** parameters. Some inputs are public as shown next to their names in the tables, others are considered private (only a prover knows them). **All output** parameters are public, i.e. the prover must provide them to the verifier.

Here you can find a visualization and explanation of the main circuits in the Iden3 protocol. We can say that all circuits use smaller sub-circuits. In the visualization, their order was changed to make it visually cleaner, but every sub-circuit has a number in the top left corner, which indicates its "real" order. The upper right corner has additional inputs from other sub-circuits.


"Real" order of the sub-circuit.
Here, we can see, that this sub-circuit
has order 6, which means that
it will be "launched" after 5
Inputs from
other sub-circuits.
Here we have only 2
6
enabled
root
SMT
Verifier

Circuits are used to generate proofs that state something. We use 6 main circuits, which are explained and visualized in this document:

1. **AuthV2** - authentication of the user. Proof generated with this circuit will state that the prover knows his private key.
2. **StateTransition** - used for on-chain state transition (when identity state changes, it should be updated on-chain).

CredentialAtomicQuery circuits are used for most "activities" with claims. You can prove that you're at least 18 years old or that you own a certain NFT from the list without disclosing which one and many other things. It should be noted, that all of them receive requestID and issuerID as inputs. These inputs are not used in any subcircuit. RequestID is needed to identify requests from the verifier to the prover (and to store the query) and IssuerID can be used to block certain issuers (in the Iden3 protocol anyone can be an issuer) or vice versa, to "whitelist" certain issuers. There are 4 variations of these circuits, which differ in issuance method (how the claim was issued) and where the proof is checked (off-chain variations also interact with the blockchain, but they only read information from it and do not write, so no on-chain updates are performed):

3. **CredentialAtomicQueryMTPOnChain** - uses MTP issuance method and is verified on-chain in the smart contract.
4. **CredentialAtomicQueryMTPOffChain** - uses the MTP issuance method and is verified off-chain at the back-end.
5. **CredentialAtomicQuerySigOnChain** - uses the Signature issuance method and is verified on-chain in the smart contract.
6. **CredentialAtomicQuerySigOffChain** - uses the Signature issuance method and is verified off-chain at the back-end.

The MTP issuance method updates the state of the issuer and should be done on-chain (state transition should be performed), while Signature issuance can be done off-chain without updating the state.

AuthV2

At the high level, this circuit checks that the prover owns an identity. It can be used to give users access to the applications, and it is usually done in this way. Also, it can be used in other circuits as a sub-circuit, which is done in credentialAtomicQuery on-chain variations.

Scheme visualization

Inputs

Public / private	Input name	Description
Public	challenge	message that should be signed by the prover to prove identity ownership
Public	gistRoot	root of the GIST stored on-chain
Private	claimsTreeRoot	prover's claims tree root
Private	authClaimIncMtp [40]	auth claim inclusion MTP inside prover's claims tree
Private	authClaim[8]	prover's auth claim
Private	revTreeRoot	prover's revocation tree root
Private	authClaimNonRevMtp[40]	auth claim nonce exclusion MTP inside prover's revocation tree
Private	authClaimNonRevMtpNoAux	flag that indicates whether to check the auxiliary node or not (related to revTreeRoot)
Private	authClaimNonRevMtpAuxHi	auxiliary node index (key)

Private	authClaimNonRevMtpAuxHv	auxiliary node value
Private	rootsTreeRoot	prover’s roots tree root
Private	challengeSignatureR8x	signature of the challenge (Rx point)
Private	challengeSignatureR8y	signature of the challenge (Ry point)
Private	challengeSignatureS	signature of the challenge (S point)
Private	userState	prover’s identity state
Private	genesisID	prover’s genesis identifier
Private	gistMtp[64]	prover’s state inclusion MTP inside the global state
Private	gistMtpNoAux	flag that indicates whether to check the auxiliary node (related to gistRoot)
Private	gistMtpAuxHi	auxiliary node index (key)
Private	gistMtpAuxHv	auxiliary node value
Private	profileNonce	random number, stored by the user

Outputs

- userID - equals genesisID if provided profileNonce is 0 otherwise equals hash(ID, nonce).

Let’s explore this circuit part by part (it is desirable to read it in parallel with the visualization):

1. IdOwnership circuit receives a lot of parameters (challenge, claimsTreeRoot, authClaimIncMtp, authClaim, revTreeRoot, authClaimNonRevMtp, auxiliary information

for nonRevocation, rootsTreeRoot, challengeSignature data, and userState) as inputs, and it verifies:

   a. The validity of the prover's auth claim and its existence in the Claims tree.
   b. Non-revocation of the auth claim.
   c. Prover's signature on the challenge number.
   d. Finally, verification of the prover's identity state is performed (whether it is genesis or not and whether it exists on-chain).

The circuit uses VerifyAuthClaimAndSignature sub-circuit to check (1), (2), (3), and checkIdenStateMatchesRoots sub-circuit to check (4).

2. **CutId** receives the prover's genesisID as input. This circuit "cuts" unnecessary information from genesisID and outputs it in bit format. Concretely, it cuts type and checksum from the identifier.

3. **CutState** receives userState as input. It "cuts" unnecessary information from the IS and outputs it in bit format. Concretely, it cuts unnecessary zeros (due to how the Poseidon hash function is implemented, it outputs 32 bytes, but the actual hash is only 27, and the other 5 are zeros).

4. **IsEqual** receives cutId and cutState outputs as inputs (cropped state and cropped genesisID). It checks whether userState is genesis or not.

5. **Poseidon** is a realization of a SNARK-friendly hash function. In AuthV2 circuit it receives as input prover's genesisID to calculate the key in the tree (hash of the genesisID). It is not a theme of this document, but you can read more about the Poseidon hash function here and here.

6. **SMTVerifier** receives gistRoot, userState, gist MTP, auxiliary information for the gist MTP, isEqual output (determines whether inclusion or exclusion should be checked), and Poseidon output. Also, it receives the parameter "enabled" as 1 (hardcoded), which determines whether verification should be performed or not. This circuit has 11 components (so it's a bit complex), but at the high level, it verifies an inclusion or exclusion of some data (hash of the key-value pair) in the tree (using root, MTP, and auxiliary information if needed). In the AuthV2 circuit, it verifies that either the prover's identity is genesis and does not exist in the GIST or it's not genesis and should be in the GIST (and so, checks that it exists).

7. **SelectProfile** receives genesisID and profileNonce as inputs. If profileNonce is zero, it outputs genesisID, otherwise, it will compute profileID and output it from the circuit.

StateTransition

This circuit checks the correctness of the identity state transition execution by taking old and new identity states as inputs. This circuit is used every time the identity updates its state. Verification is performed in the smart contract.

Scheme visualization

Inputs

Public / private	**Input name**	**Description**
Public	isOldStateGenesis	flag that indicates whether oldUserState is genesis or not
Public	userID	prover's genesis identifier
Public	oldUserState	prover's identity state before the transition
Public	newUserState	prover's identity state after the transition
Private	claimsTreeRoot	prover's claims tree root
Private	authClaimMtp[40]	auth claim inclusion MTP inside prover's claims tree
Private	authClaim[8]	prover's auth claim
Private	revTreeRoot	prover's revocation tree root
Private	authClaimNonRevMtp[40]	auth claim exclusion MTP inside prover's revocation tree
Private	authClaimNonRevMtpNoAux	flag that indicates whether to check the auxiliary node or not

Private	authClaimNonRevMtpAuxHv	auxiliary node value
Private	authClaimNonRevMtpAuxHi	auxiliary node index (key)
Private	rootsTreeRoot	prover's roots tree root
Private	signatureR8x	signature of the challenge (Rx point)
Private	signatureR8y	signature of the challenge (Ry point)
Private	signatureS	signature of the challenge (S point)
Private	newClaimsTreeRoot	prover's claims tree root after the transition
Private	newAuthClaimMtp [40]	auth claim inclusion MTP inside prover's claims tree after the transition
Private	newRevTreeRoot	prover's revocation tree root after the transition
Private	newRootsTreeRoot	prover's roots tree root after the transition

This circuit has **no outputs**.

Let's explore this circuit part by part:

1. **CutId** receives userID as input and cuts unnecessary information from it (concretely, it cuts type and checksum from the identifier).
2. **CutState** receives oldUserState as input and also cuts unnecessary information from it. Concretely it cuts unnecessary zeros (due to how Poseidon implementation works).
3. **IsEqual** receives cut identifiers (CutId and CutState outputs) as inputs, calculates whether they are equal or not, and uses the result in the check (1 - IsEqual.output) * isOldStateGenesis === 0 (should be equal to 0). This check is performed to make sure that if user's old state is genesis, then userID should be derived from that state.

4. **IsZero** receives newUserState as input and checks that it is not zero.

5. **IsEqual** (another one) receives both user states (oldUserState and newUserState) as inputs and checks that they are not equal.

6. **Poseidon** also receives both user states and computes their hash (this hash is used as a "challenge", that should be signed by the prover).

7. **IdOwnership** receives a lot of parameters as inputs: Poseidon output as a challenge, oldUserState, claimsTreeRoot, authClaimMtp[40], authClaim[8], revTreeRoot, authClaimNonRevMtp[40], auxiliary information for auth claim non-revocation checks (authClaimNonRevMtpNoAux, authClaimNonRevMtpAuxHv, authClaimNonRevMtpAuxHi), rootsTreeRoot, and signature data (SignatureR8x, SignatureR8y, SignatureS).

   It verifies:
   a. The validity of prover's auth claim and its existence in the Claims tree.
   b. Non-revocation of the auth claim.
   c. Prover's signature on the challenge number.
   d. Finally, the prover's identity state is verified (whether it is genesis or not and whether it exists on-chain).

8. **CheckClaimExists** receives authClaim, newClaimsTreeRoot, and newAuthClaimMtp as inputs and checks that authClaim still exists in the new claims tree.

9. **CheckIdenStateMatchesRoots** receives the user's new tree roots (newClaimsTreeRoot, newRevTreeRoot, newRootsTreeRoot) and newUserState as inputs and checks that the roots match the user's new state.

credentialAtomicQueryMTPOnChain

This circuit checks that a claim issued to the prover (and added to the issuer's Claims Tree) satisfies a query set by the verifier, and the verification is performed in the smart contract in this case.

Scheme visualization

Inputs

Public / private	**Input name**	**Description**
Public	requestID	an identifier of the request
Public	issuerID	issuer's genesis ID
Public	gistRoot	root of the GIST, that is stored on-chain
Public	challenge	message that should be signed by the prover (to prove control of an Identity)
Public	issuerClaimIden State	issuer's Identity State (at the moment of the claim issuance)
Public	isRevocation Checked	flag that indicates whether claim revocation should be checked or not
Public	issuerClaimNonRev State	issuer's Identity State that is used in the non revocation checks (potentially should be equal to the on-chain one)
Public	timestamp	current time
Private	authClaim[8]	prover's auth claim

Private	authClaimIncMtp [40]	MTP of the auth claim inclusion inside prover's claims tree
Private	authClaimNonRevMtp[40]	MTP of the auth claim exclusion inside prover's revocation tree
Private	authClaimNonRevMtpNoAux	flag that indicates whether to check the auxiliary node (in the non-revocation check)
Private	authClaimNonRevMtpAuxHi	auxiliary node index (key)
Private	authClaimNonRevMtpAuxHv	auxiliary node value
Private	userGenesisID	prover's genesis identifier
Private	challengeSignature R8x	signature of the challenge (Rx point)
Private	challengeSignature R8y	signature of the challenge (Ry point)
Private	challengeSignature S	signature of the challenge (S point)
Private	profileNonce	random number, stored by user (0 if profile isn't used)
Private	gistMtp[64]	prover's state inclusion MTP inside the global state
Private	gistMtpAuxHi	GIST auxiliary node index (key)

Private	gistMtpAuxHv	GIST auxiliary node value
Private	gistMtpNoAux	flag that indicates whether to check the auxiliary node or not
Private	claimSubject ProfileNonce	nonce of user's profile that claim is issued to (can be 0)
Private	userState	prover's identity state
Private	issuerClaim[8]	claim data (4 indexes and 4 values, can be merklized)
Private	issuerClaimMtp[40]	MTP of the claim inclusion inside issuer's claims tree (related to the issuerClaimIdenState)
Private	issuerClaim ClaimsTreeRoot	issuer's claims tree root (related to the issuerClaimIdenState)
Private	issuerClaim RevTreeRoot	issuer's revocation tree root (related to the issuerClaimIdenState)
Private	issuerClaim RootsTreeRoot	issuer's roots tree root (related to the issuerClaimIdenState)
Private	userClaimsTreeRoot	prover's claims tree root
Private	userRevTreeRoot	prover's revocation tree root
Private	userRootsTreeRoot	prover's roots tree root
Private	issuerClaimNonRev Mtp[40]	MTP of the claim nonce exclusion inside prover's revocation tree

Private	issuerClaimNonRev MtpNoAux	flag that indicates whether to check the auxiliary node (in the revocation exclusion check)
Private	issuerClaimNonRev MtpAuxHi	auxiliary node index (key)
Private	issuerClaimNonRev MtpAuxHv	auxiliary node value
Private	issuerClaimNonRev ClaimsTreeRoot	issuer’s claims tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RevTreeRoot	issuer’s revocation tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RootsTreeRoot	issuer’s roots tree root (related to the issuerClaimNonRevState)
Private	claimSchema	claim schema that was used in the issuerClaim
Private	claimPathNotExists	flag, that indicates whether inclusion or exclusion should be checked in the merklized claim
Private	claimPathMtp[32]	MTP of the claimPathKey and claimPathValue pair inclusion in the merklized claim
Private	claimPathMtpNoAux	flag that indicates whether to check auxiliary node or not
Private	claimPathMtpAuxHi	auxiliary node index (key)
Private	claimPathMtpAuxHv	auxiliary node value
Private	claimPathKey	hash of path in merklized json-ld claim

Private	claimPathValue	value in merklized json-ld claim
Private	slotIndex	index of the slot, where value is stored, in case of not-merklized claim
Private	operator	query operator (<, >, =, !=, “in”, “not in” or “nothing”)
Private	value[64]	value that should be checked.

Outputs

- userID - equals genesisID if provided profileNonce is 0 otherwise equals hash(ID,nonce).
- merklized flag - shows, whether claim is merklized or not.
- circuitQueryHash - hash of the query (because query parameters are private, using a hash verifier can check that the prover didn’t cheat).

Let’s explore this complex circuit part by part:

1. **AuthV2** receives a lot of parameters, so they won’t be written here. This circuit checks that the prover is owner of the identity and sets userID as output from the circuit. It was explained better above, it is recommended to read it before you proceed.

2. **verifyClaimIssuanceAndNonRev** receives issuerClaimIdenState, isRevocationChecked, IssuerClaimNonRevState, issuerClaim, issuerClaimMtp, all issuer tree roots (issuerClaimClaimsTreeRoot, issuerClaimRevTreeRoot, issuerClaimRootsTreeRoot), issuerClaimNonRevMtp, auxiliary information for non-revocation check (issuerClaimNonRevMtpNoAux, issuerClaimNonRevMtpAuxHi, issuerClaimNonRevMtpAuxHv) and non-revocation trees (issuerClaimNonRevClaimsTreeRoot, issuerClaimNonRevRevTreeRoot, issuerClaimNonRevRootsTreeRoot) as inputs.

   It verifies:
   a. claim is included in the claims tree;
   b. issuer’s claims tree root is included in issuer’s identity state;
   c. non-revocation of the claim (through nonRevMtp and another state);
   d. issuer’s revocation tree root is included in another issuer idenState (NonRevIssuerState).

We use two identity states because once the claim is issued, it will be in the claims tree forever, but it may appear in the revocation tree later. It means that user should update only non-revocation proof, while issuance proof may remain old. Potentially, NonRevIssuerState should be equal to the on-chain one.

3. **verifyCredentialSubjectProfile** receives userGenesisID, claimSubjectProfileNonce and issuerClaim as inputs. It checks that the claim is issued to user's genesisID or one of its profiles.
4. **verifyCredentialSchema** receives issuerClaim and provided claimSchema as inputs and checks that schema inside the claim is equal to the provided schema.
5. **VerifyExpirationTime** receives issuerClaim and timestamp as inputs and checks that the timestamp is less than the expiration time in the claim.
6. **GetClaimMerklizeRoot** receives issuerClaim as input and outputs flag, which states whether the claim is merklized or not, and value (root that is stored in the claim).
7. **SMTVerifier** receives merklize.flag as enabled and merklize.output as root parameters, claimPathKey, ClaimPathNotExists (determines, whether inclusion or exclusion should be checked), claimPathMtp, auxiliary information (claimPathMtpNoAux, claimPathMtpAuxHi, claimPathMtpAuxHv) and claimPathValue. Performs verification of merklized claim (checks that provided MTP, key and value match the root, key-value pair is at the right place and exists)
8. **GetValueByIndex** receives slotIndex and issuerClaim as inputs and returns chosen (by slotIndex) value from the claim (i.e. 2 is passed, i_2 is returned, 6 is passed - v_2 is returned)
9. **Mux1** receives merklize.flag as selector, claimValue (from getValueByIndex) as first possible output and claimPathValue as second. Based on the selector (if it is 0 or 1) it returns first or second possible output. In simple words, if the claim is merklized - it returns provided value (claimPathValue), otherwise - it returns the value that was pulled out from the claim by slotIndex.
10. **SpongeHash** receives as input only value array (query specific array, can be date of birth, address, group of addresses, etc) and hashes the whole value array together, using 6 elements in one hash by default.

    SpongeHash is an implementation of the sponge hash function with Poseidon. It is used when data that should be hashed is huge.

For example, we want to hash an array of 23 elements (Poseidon can hash up to 6 elements at once, due to implementation). SpongeHash will hash the first 6 elements (let's call the result of this single procedure such as "finalHash", or simply f_0), add this finalHash as the first value to the next iteration, and add 5 more elements. It will be repeated until there are no more elements left (0…6, f_0+6…11, f_1+11..16, f_2+16…21, f_3+21…26. We have only 23 elements, so 24 and 25 elements will be zeros) and the output will be the final hash (hash(f_4+input[21,22,23] + two zeros).

11. **Query** receives "value" array, operator, and mux1.output (value, received from the claim, that should be checked) and performs operator on other inputs. Value array will contain more than 1 element if "in" (or "not in") operation is performed, otherwise, only the first (value[0]) element will be checked (e.g., it will contain the age as value[0] and it will be checked that this age satisfies certain query, for example, it is higher than 18).

12. **Poseidon** receives claimSchema, claimPathNotExists, claimPathKey, slotIndex, operator, and output of the spongeHash circuit as inputs, hash them together, and outputs the circuitQueryHash at the end, which will be checked by the verifier.

credentialAtomicQueryMTPOffChain

This circuit checks that a claim issued to the prover (and added to the issuer's Claims Tree) satisfies a query set by the verifier and the verification is performed by the backend service in this case.

Scheme visualization

Inputs

Public / private	Input name	Description
Public	requestID	an identifier of the request
Public	issuerID	issuer's genesis ID
Public	issuerClaimIden State	issuer's Identity State (at the moment of the claim issuance)
Public	isRevocation Checked	flag that indicates whether claim revocation should be checked or not
Public	issuerClaimNonRev State	issuer's Identity State that is used in the non-revocation checks (potentially should be equal to the on-chain one)
Public	claimPathKey	hash of path in merklized json-ld claim
Public	claimPathNot Exists	flag, that indicates whether inclusion or exclusion should be checked in the merklized claim
Public	claimSchema	claim schema that was used in the claim
Public	timestamp	current time

Public	slotIndex	index of the slot, where value is stored, in case of not-merklized claim
Public	operator	query operator (<, >, =, !=, “in”, “not in” or “nothing”)
Public	value[64]	value array that should be checked
Private	issuerClaimNonRev Mtp[40]	MTP of the claim nonce exclusion inside prover’s revocation tree
Private	issuerClaimNonRev MtpNoAux	flag that indicates whether to check the auxiliary node (in the non-revocation check)
Private	issuerClaimNonRev MtpAuxHi	auxiliary node index (key)
Private	issuerClaimNonRev MtpAuxHv	auxiliary node value
Private	issuerClaimNonRev ClaimsTreeRoot	issuer’s claims tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RevTreeRoot	issuer’s revocation tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RootsTreeRoot	issuer’s roots tree root (related to the issuerClaimNonRevState)
Private	issuerClaim Mtp[40]	MTP of claim inclusion inside issuer’s claims tree (related to the issuerClaimIdenState)
Private	issuerClaim ClaimsTreeRoot	issuer’s claims tree root (related to the issuerClaimIdenState)

Private	issuerClaim RevTreeRoot	issuer’s revocation tree root (related to the issuerClaimIdenState)
Private	issuerClaim RootsTreeRoot	issuer’s roots tree root (related to the issuerClaimIdenState)
Private	userGenesisID	prover’s genesis identifier
Private	claimSubject ProfileNonce	nonce of the profile that claim is issued to (can be 0)
Private	profileNonce	random number, stored by user (0 if profile isn’t used)
Private	issuerClaim[8]	claim data (4 indexes and 4 values, can be merklized)
Private	claimPathMtp[32]	MTP of claimPathKey and claimPathValue pair inclusion in the merklized claim
Private	claimPathMtpNoAux	flag that indicates whether auxiliary node should be used
Private	claimPathMtpAuxHi	auxiliary node index (key)
Private	claimPathMtpAuxHv	auxiliary node value
Private	claimPathValue	value in merklized json-ld claim

Outputs

- userID - equals user’s genesisID if profileNonce is zero otherwise equals hash(ID,nonce)
- merklized - flag that shows, whether claim is merklized or not

Let’s explore this circuit part by part:

1. **verifyClaimIssuanceAndNonRev** receives issuerClaimIdenState, isRevocationChecked, IssuerClaimNonRevState, issuerClaimNonRevMtp, auxiliary information for non-revocation check (issuerClaimNonRevMtpNoAux, issuerClaimNonRevMtpAuxHi, issuerClaimNonRevMtpAuxHv), issuerClaimNonRevClaimsTreeRoot, issuerClaimNonRevRevTreeRoot, issuerClaimNonRevRootsTreeRoot, issuerClaimMtp, issuerClaimClaimsTreeRoot, issuerClaimRevTreeRoot, issuerClaimRootsTreeRoot and issuerClaim as inputs and checks, that:

    a. claim is included in the claims tree
    b. issuer's claims tree root is included in the issuer's identity state
    c. non-revocation of the claim (through "nonRevMtp" and another state)
    d. issuer's revocation tree root is included in another issuer idenState (NonRevIssuerState), which potentially should be equal to the on-chain one

    We use two identity states because once the claim is issued, it will be in the claims tree forever, but it may appear in the revocation tree later. It means that the user should update only non-revocation proof, while the issuance claim may remain old.

2. **verifyCredentialSubjectProfile** receives userGenesisID, claimSubjectProfileNonce, and issuerClaim as inputs. It checks that the claim is issued to the user's genesisID or one of its profiles.

3. **verifyCredentialSchema** receives issuerClaim and provided claimSchema as inputs and checks that the schema inside the claim equals the provided schema.

4. **VerifyExpirationTime** receives issuerClaim and timestamp as inputs and checks that the timestamp is less than the "expiration time" in the claim.

5. **GetClaimMerklizeRoot** receives issuerClaim as input and outputs flag, which states whether the claim is merklized or not, and the root that is stored in the claim.

6. **SMTVerifier** receives merklize.flag as enabled and merklize.output as root parameters, claimPathKey, ClaimPathNotExists (determines, whether inclusion or exclusion should be checked), claimPathMtp, auxiliary information (claimPathMtpNoAux, claimPathMtpAuxHi, claimPathMtpAuxHv) and claimPathValue. Performs verification of merklized claim (checks that provided MTP, key, and value match the root, key-value pair is at the right place and exists)

7. **GetValueByIndex** receives slotIndex and issuerClaim as inputs and returns chosen (by slotIndex) value from the claim (i.e., 2 is passed, i_2 is returned, 6 is passed - v_2 is returned)

8. **Mux1** receives merklize.flag as the selector, claimValue (from getValueByIndex) as the first possible output, and claimPathValue as the second. Based on the selector (if it is 0 or 1), it returns the first or second possible output. In simple words, if the claim is merklized - it returns provided value (claimPathValue), otherwise - it returns the value that was pulled out from the claim by slotIndex.

9. **Query** receives "value" array, operator, and mux1.output (value, received from the claim, that should be checked) and performs operator on other inputs. Value array will contain more than 1 element if "in" (or "not in") operation is performed, otherwise, only the first (value[0]) element will be checked (e.g., it will contain the age as value[0], and it will be checked that this age satisfies certain query, for example, it is higher than 18).

10. **SelectProfile** receives userGenesisID and profileNonce as inputs. If profileNonce is zero, it outputs genesis ID, otherwise, it will compute profileID and output it from the circuit.

credentialAtomicQuerySigOnChain

It checks that a claim issued to the prover (through Issuer's signature, the claim is not added to Issuer's claims tree), satisfies a query set by the verifier, and the verification is performed in the smart contract in this case.

Scheme visualization

Inputs

Public / private	**Input name**	**Description**
Public	requestID	an identifier of the request
Public	issuerID	issuer's genesis ID
Public	gistRoot	root of the GIST, that is stored on-chain
Public	challenge	message that should be signed by the prover (to prove control of an Identity)
Public	isRevocation Checked	flag that indicates whether claim revocation should be checked or not
Public	isserClaimNonRev State	issuer's Identity State that is used in the non-revocation checks (potentially should be equal to the on-chain one)
Public	timestamp	current time
Private	authClaim[8]	prover's auth claim
Private	authClaimIncMtp [40]	MTP of the auth claim inclusion inside prover's claims tree

Private	authClaimNonRevMtp [40]	MTP of the auth claim nonce exclusion inside prover's revocation tree
Private	authClaimNonRevMtpNoAux	flag that indicates whether to check the auxiliary node or not
Private	authClaimNonRevMtpAuxHi	auxiliary node index (key)
Private	authClaimNonRevMtpAuxHv	auxiliary node value
Private	userGenesisID	prover's genesis identifier
Private	challengeSignature R8x	prover's signature of the challenge (Rx point)
Private	challengeSignature R8y	prover's signature of the challenge (Ry point)
Private	challengeSignature S	prover's signature of the challenge (S point)
Private	profileNonce	random number, stored by user (0 if profile isn't used)
Private	gistMtp[64]	MTP of user's state inclusion inside the global state
Private	gistMtpAuxHi	GIST auxiliary node index (key)
Private	gistMtpAuxHv	GIST auxiliary node value
Private	gistMtpNoAux	flag that indicates whether to check the auxiliary node in the GIST
Private	claimSubject ProfileNonce	nonce of the profile that claim is issued to

Private	userState	prover's identity state
Private	issuerClaim[8]	claim data (4 indexes and 4 values, can be merklized)
Private	issuerAuthClaim[8]	issuer's auth claim
Private	issuerAuthClaimMtp [40]	MTP of the auth claim inclusion inside issuer's claims tree
Private	issuerAuthClaims TreeRoot	issuer's claims tree root, that is used to check auth claim inclusion
Private	issuerAuthRev TreeRoot	issuer's revocation tree root, that is used to check auth claim inclusion
Private	issuerAuthRoots TreeRoot	issuer's roots tree root, that is used to check auth claim inclusion
Private	issuerAuthClaimNonRev Mtp[40]	MTP of the auth claim nonce exclusion inside prover's revocation tree
Private	issuerAuthClaimNonRev MtpNoAux	flag that indicates whether to check the auxiliary node (in the revocation exclusion check)
Private	issuerAuthClaimNonRev MtpAuxHi	auxiliary node index (key)
Private	issuerAuthClaimNonRev MtpAuxHv	auxiliary node value
Private	issuerClaim SignatureR8x	issuer's signature of the issued claim (Rx point)
Private	issuerClaim SignatureR8y	issuer's signature of the issued claim (Ry point)

Private	issuerClaim SignatureS	issuer's signature of the issued claim (S point)
Private	userClaimsTreeRoot	prover's claims tree root
Private	userRevTreeRoot	prover's revocation tree root
Private	userRootsTreeRoot	prover's roots tree root
Private	issuerClaimNonRev Mtp[40]	MTP of the claim nonce exclusion inside prover's revocation tree
Private	issuerClaimNonRev MtpNoAux	flag that indicates whether to check the auxiliary node (in the revocation exclusion check)
Private	issuerClaimNonRev MtpAuxHi	auxiliary node index (key)
Private	issuerClaimNonRev MtpAuxHv	auxiliary node value
Private	issuerClaimNonRev ClaimsTreeRoot	issuer's claims tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RevTreeRoot	issuer's revocation tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRev RootsTreeRoot	issuer's roots tree root (related to the issuerClaimNonRevState)
Private	claimSchema	claim schema that was used
Private	claimPathNotExists	flag, that indicates whether inclusion or exclusion should be checked in the merklized claim

Private	claimPathMtp[32]	MTP of claimPathKey and claimPathValue pair inclusion in the merklized claim
Private	claimPathMtpNoAux	flag that indicates whether auxiliary node should be checked
Private	claimPathMtpAuxHi	auxiliary node index (key)
Private	claimPathMtpAuxHv	auxiliary node value
Private	claimPathKey	hash of path in merklized json-ld claim
Private	claimPathValue	value in merklized json-ld claim
Private	slotIndex	index of the slot, where value is stored, in case of not-merklized claim
Private	operator	query operator (<, >, =, !=, "in", "not in" or "nothing")
Private	value[64]	value that should be checked.

Outputs

- userID - equals user's genesisID if profileNonce is zero otherwise, equals hash(ID, nonce).
- merklized flag - shows whether the claim is merklized or not.
- issuerAuthState - issuer state based on the provided trees (trees with "issuerAuth" prefix).
- circuitQueryHash - hash of the query (because query parameters are private, the verifier can check that the prover has not cheated using query hash).

Let's explore this circuit part by part:

1. **AuthV2** receives a lot of parameters so they won't be written here. This circuit checks that the prover owns the identity and sets userID as output from the circuit. It was explained better above, it was recommended to read it before you proceed.

2. **VerifyCredentialSubjectProfile** receives userGenesisID, claimSubjectProfileNonce, and issuerClaim as inputs. It checks that the claim is issued to the user's genesisID or one of its profiles.
3. **VerifyCredentialSchema** receives issuerClaim and provided claimSchema as inputs and checks that the schema inside the claim equals the provided schema.
4. **VerifyExpirationTime** receives issuerClaim and timestamp as inputs and checks that the timestamp is less than the expiration time in the claim.
5. **VerifyCredentialSchema** receives issuerAuthClaim and "identifier" of the auth claim schema (hardcoded) as inputs and checks that the schema inside the claim is the same as the provided authBJJ schema.
6. **GetIdenState** receives the issuer's auth tree roots (issuerAuthClaimsTreeRoot, issuerAuthRevTreeRoot, issuerAuthRootsTreeRoot) and outputs issuerAuthState from the circuit.
7. **CheckClaimExists** receives issuerAuthClaim, issuerAuthClaimMtp, and issuerClaimsTreeRoot as inputs and checks that authClaim exists in the claims tree.
8. **CheckClaimNotRevoked** receives issuerAuthClaim, issuerClaimNonRevRevTreeRoot, issuerAuthClaimNonRevMtp, and auxiliary information (issuerAuthClaimNonRevMtpNoAux, issuerAuthClaimNonRevMtpAuxHi, issuerAuthClaimNonRevMtpAuxHv) and it is enabled by default (i.e., enabled parameter is hardcoded as 1). It checks that auth claim of the issuer is not revoked in his tree.
9. **GetPubKeyFromClaim** receives issuerAuthClaim as input and returns a public key (X and Y coordinate of a point on the BabyJubJub curve).
10. **VerifyClaimSignature** receives the issuer public key (from GetPubKeyFromClaim), issuerClaim, and issuerClaimSignature data (issuerClaimSignatureR8x, issuerClaimSignatureR8y, issuerClaimSignatureS). It checks that the issuer's signature on the issued claim is valid and made with his key pair.
11. **CheckIdenStateMatchesRoots** receives issuerClaimNonRevState and three tree roots (issuerClaimNonRevClaimsTreeRoot, issuerClaimNonRevRevTreeRoot, and issuerClaimNonRevRootsTreeRoot) as inputs and checks that provided state matches to the hash of three roots.

12. **CheckClaimNotRevoked** receives issuerClaim, issuerClaimNonRevMtp and auxiliary information (issuerClaimNonRevMtpNoAux, issuerClaimNonRevMtpAuxHi, issuerClaimNonRevMtpAuxHv), issuerClaimNonRevRevTreeRoot and isRevocationChecked parameter that determines whether non-revocation is checked or skipped. It checks that the issued claim nonce is not in the revocation tree.

13. **GetClaimMerklizeRoot** receives issuerClaim as an input and outputs flag, which states whether the claim is merklized or not, and the value of the root in case if merklization flag is one.

14. **SMTVerifier** receives merklize.flag as enabled and merklize.output as root (from the GetClaimMerklizeRoot circuit), ClaimPathNotExists (determines whether inclusion or exclusion should be checked), claimPathMtp, MTP auxiliary information (claimPathMtpNoAux, claimPathMtpAuxHi, claimPathMtpAuxHv), claimPathKey and claimPathValue. Performs verification of merklized claim (checks that provided MTP, key, and value match the root).

15. **GetValueByIndex** receives slotIndex and issuerClaim as inputs and returns chosen (by slotIndex) value from the claim (i.e., 2 is passed, i_2 is returned, 6 is passed - v_2 is returned).

16. **Mux1** receives merklize.flag as selector, claimValue (from getValueByIndex) as the first possible output, and claimPathValue as the second. Based on the selector (if it is 0 or 1), it returns the first or second possible output. In simple words, if the claim is merklized - it returns provided value (claimPathValue); otherwise - it returns the value that was pulled out from the claim by slotIndex.

17. **SpongeHash** receives a “value” array as input (query-specific array, can be age, address, group of addresses, etc.) and hash it with the Poseidon hash function.

18. **Query** receives “value” array, operator, and mux1.output (a value that should be checked) and performs operator on both value and mux1.output (i.e. if operator “=” is used, then mux1.output should be equal to the first element in the value array). Value array will contain more than 1 element if “in” (or “not in”) operation is performed. Otherwise, only the first (value[0]) element will be checked (e.g., it will contain the age as value[0], and it will be checked that this age satisfies the certain query, for example, it is higher than 18).

19. **Poseidon** receives claimSchema, slotIndex, operator, claimPathKey, claimPathNotExists and output of the spongeHash circuit as inputs, hash them together and outputs the circuitQueryHash at the end, which will be checked by the verifier.

credentialAtomicQuerySigOffChain

Checks that a claim issued to the prover (through Issuer's signature, the claim is not added to Issuer trees) satisfies a query set by the verifier, and the verification is performed in the backend service in this case.

Scheme visualization

Inputs

Public / private	Input name	Description
Public	requestID	an identifier of the request
Public	issuerID	issuer's genesis ID
Public	claimPathKey	hash of path in merklized json-ld claim
Public	claimPathNotExists	flag, that indicates whether inclusion or exclusion should be checked in the merklized claim
Public	isRevocation Checked	flag that indicates whether claim revocation should be checked or not
Public	issuerClaimNon RevState	issuer's Identity State that is used in the non-revocation checks (potentially should be equal to the on-chain one)
Public	claimSchema	claim schema that was used (in the issued claim)
Public	timestamp	current time

Public	slotIndex	index of the slot, where value is stored in the claim. This value is used when the claim is not merklized
Public	operator	query operator (<, >, =, !=, “in”, “not in” or “nothing”)
Public	value[64]	query specific value that should be checked.
Private	issuerClaimNonRevMtp[40]	MTP of the claim nonce exclusion inside prover’s revocation tree
Private	issuerClaimNonRevMtpNoAux	flag that indicates whether to check the auxiliary node in the revocation exclusion check
Private	issuerClaimNonRevMtpAuxHi	auxiliary node index (key)
Private	issuerClaimNonRevMtpAuxHv	auxiliary node value
Private	issuerClaimNonRevClaimsTreeRoot	issuer’s claims tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRevRootsTreeRoot	issuer’s roots tree root (related to the issuerClaimNonRevState)
Private	issuerClaimNonRevRevTreeRoot	issuer’s revocation tree root (related to the issuerClaimNonRevState)
Private	claimPathMtp[32]	MTP of claimPathKey and claimPathValue pair inclusion in the merklized claim
Private	claimPathMtpNoAux	flag that indicates whether auxiliary node should be checked or not

Private	claimPathMtpAuxHi	auxiliary node index (key)
Private	claimPathMtpAuxHv	auxiliary node value
Private	claimPathValue	value in merklized json-ld claim
Private	claimSubject ProfileNonce	nonce of the profile that claim is issued to
Private	userGenesisID	prover’s genesis identifier
Private	profileNonce	random number, stored by user (0 if profile isn’t used)
Private	issuerClaim SignatureR8x	issuer’s signature of the claim (Rx point)
Private	issuerClaim SignatureR8y	issuer’s signature of the claim (Ry point)
Private	issuerClaim SignatureS	issuer’s signature of the claim (S point)
Private	issuerClaim[8]	claim data (4 indexes and 4 values, can be merklized)
Private	issuerAuthClaims TreeRoot	issuer’s claims tree root, that is used to check auth claim inclusion
Private	issuerAuthRev TreeRoot	issuer’s revocation tree root, that is used to check auth claim inclusion
Private	issuerAuth RootsTreeRoot	issuer’s roots tree root, that is used to check auth claim inclusion
Private	issuerAuthClaim[8]	issuer’s auth claim

Private	issuerAuthClaim Mtp[40]	MTP of the auth claim inclusion inside issuer's claims tree
Private	issuerAuthClaim NonRevMtp[40]	MTP of the auth claim nonce exclusion inside prover's revocation tree
Private	issuerAuthClaim NonRevMtpNoAux	flag that indicates whether to check the auxiliary node in the auth claim revocation exclusion check
Private	issuerAuthClaim NonRevMtpAuxHi	auxiliary node index (key)
Private	issuerAuthClaim NonRevMtpAuxHv	auxiliary node value

Outputs

- userID - equals genesisID if profileNonce is 0 otherwise equals hash(ID,nonce).
- issuerAuthState - issuer's state based on the provided trees (trees with "issuerAuth" prefix).
- merklized - a flag that indicates whether a claim is merklized or not.

Let's explore this circuit part by part:

1. **VerifyCredentialSubjectProfile** receives userGenesisID, claimSubjectProfileNonce, and issuerClaim as inputs. It checks that the claim is issued to the user's genesisID or one of its profiles.

2. **VerifyCredentialSchema** receives issuerClaim and provided claimSchema as inputs and checks that the schema inside the claim equals the provided schema.

3. **VerifyExpirationTime** receives issuerClaim and timestamp as inputs and checks that the timestamp is less than the expiration time in the claim.

4. **VerifyCredentialSchema** receives issuerAuthClaim and identifier of the auth claim schema (hardcoded) as inputs and checks that the schema inside the auth claim equals the provided authBJJ schema.

5. **GetIdenState** receives the issuer's tree roots (issuerAuthClaimsTreeRoot, issuerAuthRevTreeRoot, issuerAuthRootsTreeRoot) and outputs issuerAuthState from the circuit.

6. **CheckClaimExists** receives issuerAuthClaim, issuerAuthClaimMtp, and issuerAuthClaimsTreeRoot as inputs and checks that authClaim exists in the claims tree.

7. **CheckClaimNotRevoked** receives issuerAuthClaim, issuerAuthClaimNonRevRevTreeRoot, issuerAuthClaimNonRevMtp and auxiliary information (issuerAuthClaimNonRevMtpNoAux, issuerAuthClaimNonRevMtpAuxHi, issuerAuthClaimNonRevMtpAuxHv) as inputs and it is enabled by default (i.e., "enabled" parameter is hardcoded as 1). It checks that auth claim of the issuer is not revoked in his tree.

8. **GetPubKeyFromClaim** receives issuerAuthClaim as input and returns the public key from it (X and Y coordinates of the point on the BabyJubJub curve).

9. **VerifyClaimSignature** receives the issuer's public key (X and Y coordinates from the GetPubKeyFromClaim), issuerClaim, and issuerClaimSignature data (issuerClaimSignatureR8x, issuerClaimSignatureR8y, issuerClaimSignatureS). It checks that the issuer's signature on the issued claim is valid and made with his key pair.

10. **CheckIdenStateMatchesRoots** receives issuerClaimNonRevState and three tree roots (issuerClaimNonRevClaimsTreeRoot, issuerClaimNonRevRevTreeRoot, and issuerClaimNonRevRootsTreeRoot) as inputs and checks that provided state match the hash of three tree roots.

11. **CheckClaimNotRevoked** receives issuerClaim, issuerClaimNonRevMtp, auxiliary information (issuerClaimNonRevMtpNoAux, issuerClaimNonRevMtpAuxHi, issuerClaimNonRevMtpAuxHv), issuerClaimNonRevRevTreeRoot and isRevocationChecked parameter that determines whether non-revocation is checked or skipped. It checks that the claim nonce is absent in the issuer revocation tree.

12. **GetClaimMerklizeRoot** receives issuerClaim as input and outputs the root (as "out"), which is stored either at index 2 or value 2 slots and the flag that states whether the claim is merklized or not.

13. **SMTVerifier** receives merklize.flag as enabled and merklize.out as root (from the GetClaimMerklizeRoot circuit), ClaimPathNotExists (determines whether inclusion or exclusion in the tree should be checked), claimPathMtp, MTP auxiliary information

(claimPathMtpNoAux, claimPathMtpAuxHi, claimPathMtpAuxHv), claimPathKey and claimPathValue. Performs verification (if the claim is merklized) of the claim (checks that provided MTP, key, and value match the root and that provided key-value pair exists in the tree).

14. **GetValueByIndex** receives issuerClaim and slotIndex as inputs and returns a specific value from the claim (i.e., 2 is passed, i_2 is returned, 6 is passed - v_2 is returned).

15. **Mux1** receives merklize.flag as selector, claim's value (from getValueByIndex) as the first possible output, and claimPathValue as the second. Based on the selector (if it is 0 or 1), it returns the first or second possible output. If the claim is merklized - it returns provided value (claimPathValue); otherwise - it returns the value that was pulled out from the claim by slotIndex parameter.

16. **Query** receives "value" array, operator, and mux1.output (value from the claim that should be checked) and performs the action based on the operator on both value array and mux1.output (i.e. if operator "=" is used, then mux1.output should be equal to the first element in the value array). Value array will contain more than 1 element if the "in" (or "not in") operation is performed; otherwise, only the first (value[0]) element will be checked (e.g., it will contain the birth date as value[0]. It will be checked that the age satisfies the certain query, for example, it is higher than 18).

17. **SelectProfile** receives userGenesisId and profileNonce as inputs. If profileNonce is zero, it outputs genesisID, otherwise it will compute profileID and output it from the circuit.