Galxe Identity Protocol v1.0.0

Abstract

In this paper, we present the Galxe Identity Protocol—an innovative, verifiable credential protocol exemplifying a self-sovereign identity (SSI) service—designed with a vision for future adaptability and grounded in state-of-the-art zero-knowledge proof (ZKP) techniques. This permission-less protocol constitutes chain-agnostic components catering to all roles within the ecosystem, namely credential issuers, holders, and verifiers, featuring unique developer experiences, e.g. no-code ZKP. It empowers issuers to generate not only scalable expiration-based credentials, but also revocable credentials and manage their status on-chain via a sparse merkle tree (SMT). Holders can selectively disclose requisite information for specific verification instances under a deterministic pseudonymous identity. Verifiers have the ability to specify a programmable trust schema, and by using the built-in identity nullifier feature, they can use the protocol for various applications requiring access control, both on-chain and off-chain. We further enhance verification efficiency by facilitating minting of attestations based on aggregation of revealed information from proofs.

Introduction

The concept of self-sovereign identity (SSI), in which individuals have full control over their personal information used for verification purposes, has gained significant attention over the past decade, fueled by the advancements in blockchain technology and the pursuit of decentralization. In an era marked by ever-increasing digital identity inter-connectivity, the limitations of traditional models of identity management have become a critical concern. Methods that rely on centralized authorities and third-party intermediaries have faced significant challenges in terms of privacy, security, and sovereignty.

Over the past two years, galxe.com has emerged as the leading user growth community within the Web3 ecosystem, relying extensively on credential-based systems for hosting marketing campaigns. During this period, our platform has facilitated the issuing of over one hundred million credentials to more than ten million users. At its peak, Galxe conducted over one million verification in a single day. Such extensive engagement has endowed us with firsthand experience and profound insights into the challenges associated with the existing solutions of verifiable credential and SSI, specifically regarding scalability, privacy, or the realization of self-sovereignty for users. While the W3C Verifiable Credentials Data Model has defined the structure of verifiable credentials and pioneers like Iden3 and Ethereum Attestation Service (EAS) have launched services in production, widespread adoption is still in its early stages and has not been fully realized.

In response to the challenges we faced on galxe.com, we have designed Galxe Identity Protocol, a SSI service that utilizes verifiable credentials, powered by zero-knowledge proof mechanisms. The architecture of this protocol is meticulously crafted to be **privacy-first, user-friendly, scale-able,** and **future-proofing**. Galxe identity protocol allows credential holders to selectively reveal requisite information in form of zero-knowledge proofs, while maintaining their pseudonyms. Proofs can be generated within seconds by Galxe identity vault in browsers or mobile devices, and verifications can happen on-chain at affordable costs. The protocol uses a flexible identity commitment schema to address the issue of digital identity multiplicity, so that nullifiers of proofs are deterministic and can be utilized by verifiers without introducing an extra layer of trust. This design ensures the privacy and non-traceability of the underlying identity, even in scenarios where collusion between issuers and verifiers occurs and verification activities are publicly disclosed, e.g. on-chain verification. In addition to holders, issuers and verifiers can leverage our ease-of-use Software Development Kits (SDKs) to apply ZKP without the necessity of understanding ZKP. Moreover, the typed credential design allows most issuers to create new credentials in a no-code fashion, and the unique aggregation mechanism further enhances verification efficiency by facilitating minting of a soul-bound token (SBT) or on-chain attestations (like EAS) for reduced cost verification in the future. Holders are not required to register for a global ID or reveal any commitment on-chain, and lazy registration is supported for issuers. The protocol promotes the use of non-revocable credentials with expiration for scalability, while also supporting revocable credentials on-chain. In recognition of the rapid-evolving nature of of ZKP technology, the protocol adopts a modular verification stack design, making it future-proofing.

The first verification stack is using groth16, a Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) system, for efficient proof generation and lower cost on-chain verification. It is implemented using circom and snarkjs. New verification stack, based on different proof systems like o1.js, can be added with full backward compatibility. Our open design also empower us to swiftly integrate other services in the ZKP ecosystem, e.g., cheap on-chain proof verification through proof aggregation, provided by Nebra's Universal Proof Aggregation.

Overview

One of the most important functions of digital credentials is their verifiability through cryptographic methods. Although the verifier still needs to trust the credential issuer, this functionality obviates the necessity for verifiers to interact with the data sources. Moreover, it empowers the holder of credentials with the complete ownership of the data, thereby endowing them with the discretion to decide where, when, who and what to share.

The Galxe identity protocol defines the 'how'. The protocol provides a set of ease-of-use SDKs, smart contracts and identity vault applications, hoping to bring the state-of-art SSI service to millions of users. Before jumping into the technical details of the protocol, we will give an overview of preliminary concepts and explain the capabilities and responsibilities of each role of the ecosystem. At the end, we will give an example of leveraging the protocol to run an on-chain NFT drop for angel users, where the security of application and user privacies are both protected.

Concepts

In this section, we provide a comprehensive overview of preliminary concepts, with a particular emphasis on notations used by this paper. We assume that readers are familiar with cryptographic concepts, including hash functions, asymmetric cryptography, digital signatures, sparse merkle trees, and basic understanding of zero-knowledge proofs.

Digital identity multiplicity problem

The term *digital identity multiplicity* emphasizes the multidimensional nature of modern digital identities, that a person can have multiple different identities across different platforms. For example, it is not uncommon that a user uses different nicknames, handles, and cryptographic IDs (namely addresses) for different platforms. A user might be called `MightyKitty` on twitter, while being `BraveBarbie` on discord, or uses address `0x7f7f7…abc` on EVM-compatible blockchains.

The phenomenon introduces a subtle challenge for designing a credential protocol: under what identity, should the issuer credential to? The design must allow a user to connect all his owned identities in order to reveal or generate zero-knowledge proofs against a comprehensive profile, while not leaking any privacy, e.g., the connections of accounts, to any other party.

One straightforward solution is to issue credentials under the ID of the domain that the issuer controls. That is, a credential showing that the user `MightyKitty` has more than 1000 Twitter followers, should be issued to the `MightyKitty`. However, this implementation can hinder the interoperability of credentials. When using credentials outside of the issuer's domain, the user has to prove that he is the ID shown on the credential. This process can be done through a 3rd party, which introduces another layer of trust, or the user has to reveal his ID to the verifier, sacrificing privacy. Another choice is using a global identifier schema, where each user is assigned to a globally unique identifier and issuers shall create credentials to that identifier. However, when issuers collude, a unique identifier can be used to track down footprints of the user behind, by linking together different identities from different issuers.

We proposes an innovative schema that, by embedding identity commitments into credentials, allows a user to harness his credentials from different platforms, while preserving his privacy.

Nullifier

In the context of zero-knowledge proof cryptography, a nullifier is a value that prevents double-spending, without revealing which resource is being spent. Essentially, it is a value that 'nullifies' (voids) the underlying resource in a privacy-preserving way. It plays a critical role in the applicability of zero-knowledge proof methods to practical applications. The lack of the ability to generate nullifiers with proofs compromises the verifier's ability to ascertain if a proof has been previously utilized. To draw an analogy, one may envision a scenario where a huge group of people is permitted entry into a concert event, but using a single ticket meant for only one person.

Identity commitment

An identity commitment is a public value, hiding two secret values: a secret identity and an internal nullifier. Both secret values are known only by the credential holder, while identity commitment can be public. In Galxe identity protocol, conceptually, identity secrets will be used by holders to prove the ownership of credentials, and internal nullifiers will be used, together with external nullifiers, to generate deterministic nullifiers per each verification.

Similar to Semaphore protocol, our protocol constructs identity commitment in the same simple way. It is a Poseidon hash of two secrets. Note that the order of two secrets is fixed when used as the inputs to the hash function. The first one is the identity and the second one is the internal nullifier, so that nullifiers and secrets can be distinguished in a zero-knowledge proof.

Compared with digital signatures, the power of identity commitment is actually smaller. You cannot use it for authentication without using a zero-knowledge proof, and secret values will have to be the inputs to the prover, unlike signatures where only public keys and signatures are required. However, this disadvantage can help to limit impersonation and fake profiles. For a credential, identity secrets cannot be shared without permanently sharing the ownership, while signatures can be shared with limited costs: one can temporarily transfer or lease the ownership of a credential to someone else, until the signature expires.

Verification stack

A verification stack is a set of instantiations of cryptographic algorithms that will be applied during credential issuance, proof generation and verification. Because the domain of zero-knowledge proof technology is witnessing a meteoric rise, with innovative techniques being invented rapidly, alongside the consistent refinement of implementations, the protocol is designed with a vision for future adaptability. The design de-couples the signature and verification schema from the core protocol, allowing versatile use cases and smooth upgrades.

The initial version of Galxe identity protocol will come with one simple yet powerful verification stack called 'BabyZK'. It is a production-ready full-feature zero-knowledge tech stack that supports ultimate privacy protection, that can be used even publicly on blockchains, like Ethereum.

Credential schema

In Galxe identity protocol, credential schemas are defined by the two fields on credentials: **type** and **context**. Like types in programming languages, credential types are structs, specifying the list of typed claims. A type will be associated with a set of zero-knowledge circuits and verification parameters published on-chain. On the other hand, a credential context is used to construct a concrete instantiation from a credential type. Together, they form a credential schema, consisting of typed claims under a specific context. For example, assuming there is (there actually will be) a simple credential type with one unsigned integer field and we call this type 'scalar', different schemas can leverage this same type by linking different contexts, e.g., the number of loyalty points, the highest rank achieved in a game, or date of birthday expressed in UNIX timestamp. These credential schemas can all use the zero-knowledge circuits that have been created for the scalar type. This advantage allows issuers, holders, and verifiers to apply zero-knowledge proofs without writing any code.

Parametric on-chain attestation by aggregating public inputs

For on-chain verifications, users can mint an attestation that caches the revealed facts with auto aggregation. For each pair of `(type, context)` and an `issuer`, there will be an attestation collection. Every attestation has a unique token id, which is the nullifier generated by the owner, and its properties represent some facts of the owner. The facts are automatically aggregated based on all past information revealed, during on-chain verification. Because fields are typed, aggregations are also type-aware. Taking an unsigned integer field as an example, its lower bound and upper bound that have been checked will be stored. For example, if there were two proofs that one says `height > 100 && height < 150`, and the other says `height < 123`, the attestation of height will be `lower_bound: 101` and `upper_bound: 122`. Note that the attestation will not leak any extra information about the owner, because when the verification happened, those facts have already been revealed publicly, at the owner's will. For revocable credentials, the root hash of sparse merkle tree of revoked credentials that has been used during proof will be cached.

This attestation offers the potential for future verification at a reduced computational cost. On Ethereum, a ZK-SNARK verification of Groth16 on BabyJubjub curve generally costs around 300k gas. In contrast, querying an attestation property necessitates only a few thousand units. Because activities under the same pseudonym of the attestation will be public, protocol leaves the freedom to choose between the highest level of privacy and gas efficiency to verifiers and holders.

Roles

The protocol incorporates four main roles, each contributing uniquely to the protocol's functioning. This section will give a brief introduction to their capabilities and responsibilities.

Holder

The Holder role is the centerpiece of the whole ecosystem: the individual possessing verifiable credentials. Holders do not need to register anything on-chain and the ownership of a credential is controlled by identity commitments constructed from secrets that are only known by their holders. In the Galxe identity protocol, holders maintain total control over their digital identity by storing credentials in Galxe identity vault, an open-source application that runs on holders' local devices, e.g. browser, and native applications. Holders use the vault to manage different identity commitments for different issuers, and generate zero-knowledge proofs for verification based on owned credentials. By using ZKP, holders have the ability to reveal only requisite data under a deterministic and pseudonymous identity.

Issuer

Issuers create verifiable credentials to holders by using digital signatures to endorse claims about the holder. In our design, issuers do not need to know any extra information of a holder, except for the identity commitment, which can be considered as a random number. Any identity can issue non-revocable credentials to others without on-chain registration. However, for revocable credentials, issuers must be registered on-chain to manage credential validity by publishing the root hash of sparse merkle tree of revoked credentials. Moreover, public keys used for signing need to be registered on-chain to be linked to their issuers.

Issuers can use auxiliary contracts, such as Domain Name System Security Extensions (DNSSEC) verification, email verification and GAL staking contracts to enhance their perceived trustworthiness and credibility. Verifiers can leverage these records for setting up a dynamic trust schema. For example, instead of trusting credentials signed by a specific key, verifiers can program a policy that trusts issuers with DNSSEC verification of a specific domain, with some specific corp email ownership, or issuers with enough GAL staking.

Issuers must maintain a 1-to-1 mapping from users of the domain to identity commitments. The identity secret part can be changed with a zero-knowledge proof that ensures the internal nullifier remains the same.

Verifier

The verifier is the role that has the least dependence with other roles of the ecosystem. They are usually applications of gated services, allowing access only if the identities meet some specific requirements, like bouncers at bars. To host a verification, verifiers need to pick issuers that they can trust via a programmable trust schema, and verification requirements of credentials to be checked. Verifiers can use the protocol SDK to generate verification requirements that holders can use for generating proofs. To verify a proof, verifiers need to fetch verification key based on the type of the credential from blockchain, and check revocation status if there is any proof of revocable credentials. For each verification, verifiers should use a unique external nullifier and keep tracking of used nullifiers to prevent double-spending.

Credential type designer

Credential designers propose new types of credentials to the community, specified in a domain-specific language (DSL) of the protocol. These individuals, typically community contributors, actively participate in shaping the ecosystem of evolving needs and requirements, by introducing innovative and diverse credential types. With Galxe identity protocol SDK, designers turn the type specification written in DSL into automatically generated zero-knowledge proof circuits, smart contracts proposed. Depending on the verification stack, type designer may need to hold a decentralized trust set up ceremony and publish the proof key to permanent storages.

Designers must also provide a detailed specification of in-depth description of the credential type, outlining its purpose, intended use cases, and the specific attributes and fields it contains. These specifications clarify the semantic meaning of each field, define any restrictions or requirements associated with the credential, and provide guidelines for issuing and verification. The specifications ensure a shared understanding among ecosystem participants regarding the purpose and usage of the proposed credential type.

Although it is a permission-less process to introduce new types to the ecosystem, the protocol employs a governance framework to facilitate decision-making regarding inclusion of new credential types into official core types with long term support. The community, comprising users, developers, and other stakeholders, actively participates in the evaluation and acceptance process. Through transparent discussions and GAL voting mechanisms, the community collectively determines which proposed credential types are incorporated into the official type.

Case study: on-chain NFT drop for early users using Galxe identity protocol

The Galxe identity protocol offers a solution for services that require identity-based access control, just like a bouncer at a club. To demonstrate its application, let's consider an on-chain NFT drop scenario.

Scenario:

A decentralized exchange (DEX) project on EVM chains, named *Alpha*, intends to reward its early supporters (angel users) with a unique on-chain NFT drop. The criteria for qualifying as an angel user are:

1. Following their official Twitter account (X) within the first 30 days post-launch. Given that Alpha launched on 01/01/2021, the cut-off date is 01/31/2021.
2. The user's nationality isn't on the disallowed list.
3. For every $1000 trading volume, the user qualifies for one NFT drop.

Instead of creating an application that mandates users to link their EVM address, Twitter, and undergo KYC (potentially compromising privacy), Alpha can utilize the the protocol for a privacy-centric approach:

1. **Issue trading volume credentials**: Alpha becomes a credential issuer, providing scalar-typed credentials under the context "Project Alpha trading volume" to its users. Users must submit an identity commitment to Alpha, either on-chain via a registry contract based on `msg.sender` or off-chain through ECDSA signatures, to receive the credential. Alpha ensures one EVM address corresponds to a single identity commitment, as required by the protocol.
2. **Select 3rd-party credentials issuers**: Alpha collaborates with two reputable credential issuers to verify user's Twitter follows and user nationalities. We assume that Twitter follow credentials have two claims in its body: followed account ID and verification date, from issuer *deSocial*. As for nationality, we assume that the issuer, *DeKYC*, provides passport-type credentials which includes a claim about the holder's nationality.
3. **Deploy NFT drop smart contract**: Alpha deploys a smart contract that rewards NFTs based on trading volume to any address providing valid zero-knowledge proofs with unused nullifiers. The external nullifier is set to `0x18d...`, derived from the hash of "Alpha angel user NFT drop".

The smart contract checks the proofs against:

General requirements:

- The revealed identity specified in proof, backed by the fact that only the credential holder knows secrets of identity commitments, must match the `msg.sender`.
- The external nullifier is `0x18d...`.
- Nullifiers are either unused or associated with other addresses.

Credential-specific requirements:

1. **Twitter Follow**: The zero-knowledge proof must confirm the followed account is Alpha and the verification date is before 01/31/2021.
2. **Passport**: The proof should list countries that don't match the user's nationality, and the list is a super set to the Alpha's disallow list.
3. **Trading Volume**: The proof must show the lower bound of the trading volume, determining the number of NFTs a user can claim. To incentivize trading, unlike the above two credentials, this credential isn't voided but linked to an EVM address for future volume updates.

Credential	Type	Context	Public inputs	On-chain verification	Post-verification actions
Twitter follow	Custom, two claims: 1. Followed account ID. 2. Verification date.	Simple non-revocable twitter follow credentials with verification date, issued by DeSocial.	1. Equality of followed account ID v.s. "Alpha". 2. Upper bound of verification date. 3. External nullifier and nullifier `A` 4. Revealing identity is X.	1. Equality is true. 2. Upper bound is before 01/31/2021. 3. External nullifier is `0x18d...` 4. X is `msg.sender` 5. `A` has not been marked as used.	Mark nullifier `A` as used for twitter credentials.
Passport	Custom, containing a claim of nationality.	DeKYC's AI-power KYC solution.	1. Equalities of nationality v.s. a list of countries. 2. External nullifier and nullifier `B` 3. Revealing identity is X.	1. Equalities are all false, and the list of countries is a superset of the disallow list. 2. External nullifier is `0x18d...` 3. X is `msg.sender` 4. `B` has not been marked as used.	Mark nullifier `B` as used for passport credentials.
Trading volume	Scalar (a basic credential type shipped)	Project Alpha trading volume	1. Lower bound of trading volume. 2. External nullifier and nullifier `C`	1. External nullifier is `0x18d...` 2. X is `msg.sender` 3.	Bind `C` with `msg.sender`, as used for trading volume credentials, and set the

Credential	Type	Context	Public inputs	On-chain verification	Post-verification actions
			3. Revealing identity is X.	`C` has not been binded yet, or binded with `msg.sender`	number of available NFTs to mint to be `lower_bound_volume / 1000`

Once preparations for Project Alpha are complete, we can focus on the user experience. The user workflow is streamlined: collecting credentials and generating proofs for verification. All these steps can be effortlessly executed using the **Galxe Identity Vault**, a local application compatible with both browsers and mobile devices.

1. **Credential Collection**: Users are required to gather three distinct credentials from separate issuers. If a user has not previously requested credentials from these issuers, they would need to set up three unique identity commitments. The Galxe Identity Vault automates this process, ensuring that all confidential information remains safeguarded by the user's password.
   a. **Twitter follow**: Visit *DeSocial's* website or app, connect with Galxe Identity Vault*, and* potentially do an OAuth login with your twitter account to verify the follow status. Then *DeSocial* will request a identity commitment from the connected Galxe identity vault, and delivery the credential directly to the vault.
   b. **Passport**: By accessing *DeKYC's* platform, users may pay a service fee for KYC verification and credential issuance, then DeKYC will follow a process analogous to the one mentioned above.
   c. **Trading Volume:** Within Project Alpha's application, users can directly request a trading volume credential, again following a similar procedure.
2. **Proof Generation**: With the three credentials secured, users can effortlessly produce zero-knowledge proofs using the Galxe identity vault. The intricate technical details of generating these proofs remain hidden from users. Instead, Project Alpha forwards statements to be proved directly to the vault. The vault then displays the information set to be disclosed, such as "your nationality differs from countries A, B, C..." or "Your trading volume exceeds XXX", seeking user confirmation. Upon approval, the proofs are generated and sent to the smart contract for NFT minting.

It's worth noting that while this process is hosted on-chain, it can also be executed off-chain in a centralized manner.

Security Analysis:

1. **Identity Linkage Protection**: The protocol ensures that users' identities across various platforms remain concealed. These identities are interconnected using different identity commitments and the revealed identity. Importantly, even if multiple issuers were to collude, they will not be able to correlate a user's identity across different platforms, as long as users follow the best practice of using different identity commitments for each issuer.
2. **Pseudonymous Claims**: Users have the flexibility to claim NFTs under a pseudonym. This ensures that their trading address on the DEX remains separate from their NFT ownership, bolstering privacy. As a practical implication, a user can sell their NFTs on a KYC-mandated marketplace without the fear of their KYC data being associated with their trading activities on the DEX.
3. **Double-spending Prevention**: Nullifiers ensure credentials cannot be used more than once. This ensures that each angel user can only claim their rightful amount of NFTs.
4. **Minimal Data Exposure**: The protocol is designed to disclose only the bare minimum data, thereby reducing the risk of privacy breaches, such as birthday attacks. For instance:
   - For the Twitter follow date, users disclose the an upper bound of verification date, instead of a precise date. All angel users can standardize it to 01/31/2021.
   - Regarding nationality, the protocol doesn't directly reveal the country. Instead, it provides a list of countries that don't match the user's nationality, which is theoretically the least revealing method.
   - For trading volume, a lower bound is used in place of the precise volume, which can be easily used to identity an user. The lower bound can be adjusted to any legit N * 1000, based on the number of NFTs a user could and intend to mint.

Protocol specification

The protocol specification covers the domain-specific language (DSL) for definition of credential types, format of credentials, components and standard workflows throughout the lifecycle of credentials. Designed to be minimalism and future-proofing, the protocol isolates rapid-changing zero-knowledge proof technology from the core elements of a credential, using the modular verification stack design. For implementations that are specified by the chosen verification stack, we use BabyZK stack as an example for illustration.

Type DSL

Before we dive into the credential's structure, it is crucial to introduce the concept of a credential type. To facilitate the use of ZK credentials without requiring deep dive into the underlying technology, we introduced a straightforward Domain-Specific Language (DSL). Credential type designers use this DSL to define credential types, outlining the expected claims, their respective types, and whether the credential can be revoked. A credential type is consist of a sequence of declarations of ***claims*** (also referred as ***fields*** in this paper). Like definition of struct in other programming languages, a declaration of claim has two components: an identifier and a type.

Additionally, whether a credential can be revoked by its issuer is also specified by its type. By default, credential is non-revocable and its validity can only be managed using the intrinsic expiration field. The DSL allows for the specification of revocability through the `@revocable(n)` pragma, where `n` denotes a positive integer indicating the depth of the sparse merkle tree used for managing revoked signatures. An SMT of depth `n` can accommodate `2^n` signatures. Developers should determine an optimal value by estimating the maximum number of credentials an issuer might distribute for a given context or type. The range for the depth spans from a minimum of 2 to a maximum of 248.

A semi-formal definition in BFN is as following:

```
<credential-type> ::= <pragma> ";" <declaration-list> | <declaration-list>
<declaration-list> ::= <declaration> | <declaration> ";" <declaration-list>
<declaration> ::= <type> ":" <identifier>
<type> ::= <base-type> | <array-type>
<array-type> ::= <base-type> "[" <number> "]"
<base-type> ::= "bool" | "uint" "<" <number> ">"
                "prop" "<" <number> "," <prop-hash-algorithem> "," <number> ">"
<prop-hash-algorithem> ::= "p" | "k" | "c"
<pragma> ::= "@" <identifier> "(" <pragma-args> ")"
<pragma-args> ::= number | string
<identifier> ::= /[a-zA-Z_][a-zA-Z0-9_]/
```

```
@revocable(16);
user_id:prop<128,p,2>;
age:uint<8>;
email_verified:bool;
owned_nft_ids:prop<32,p,1>[10];
```

In the current version, we support 3 types of claims:

1. **Bool**: Representing a boolean value, during proof generation, credential owners can choose to reveal or hide its value. This is due to its inherent limitation of equality check operation, which, when executed, inevitably discloses the value. For each bool-typed field, there will be one corresponding public signal in the proof, where `0` means the value is hidden, and `1` means the value is false, and `2` means the value is true.
2. **Property (string)**: Referred as "property type" in this document, this type is primarily designed for representing innumerable properties, such as "name". However, issuers can apply this value to enumerations as well. Issuers can use this type to store a string value. Internally, the string value will be converted to a hash value. There are three type parameters:
    a. The `width` represents the width of the hash value. It must be byte-aligned and up to 248-bit;
    b. The `hash_algorithm` designates which hash is used to transform the string into a hash value. If the hash value exceeds the specified `width`, only the `width` least significant bits will be used. The accepted values are `p | k | c`, where `p` stands for poseidon hash, `k` for keccak256, and `c` for a user-defined hash algorithm.
    c. The optional parameter `num_equals_to_checks` , default to 1, specifies the number of values to be compared for one proof. There will be `num_equals_to_checks` output signals, where their least significant bit represents the equality, and the `[width+1…1]` bits convey the value that underwent comparison.
3. **Scalars**: Representing unsigned integers with bit-lengths spanning from 8 to 256, scalars are equipped to support range checks.
4. **Fixed-length array**: This type encapsulates arrays of the previously mentioned types. Operations specific to the underlying type can be executed on either an individual element or the entire array, facilitated by the `one_of` and `all_of` operators. Note that the `any_of` operator is redundant, as during the proofing phase, the holder is aware of which element is the desired value.

The following table summarizes a brief overview of the supported claim types, their syntax, and the operations applicable during proof generation.

Claim type	Description	Supported operations	Public inputs
bool	A boolean value	reveal, hide	`0` means the value is hidden, and `1` means the value is false, and `2` means the value is true.
prop<width, hash_algorithm, num_equals_to_checks>	A string representing some property of the holder, designed for innumerable properties. The big-length of property and the hash algorithm that maps the string to an unsigned integer	==, ≠	`N` output signals, where N equals to `num_equals_to_checks`. Their least significant bit represents the equality, and the `[width+1…1]` bits convey the value that underwent comparison.

Claim type	Description	Supported operations	Public inputs
	which fits the width must be provided.		
uint<width>	Scalar values, with width to be 8, 16, 32 ... 256.	≤, ≥	2 public inputs: `lower_bound` and `upper_bound` Note: for verification stack that does not support 256-bit numbers, e.g. babyzk, each number will be mapped to two signals, the lower 128 and the higher 128 bits.
T[N]	An fixed-length array of other values.	1. oneOf(A, I, OP rhs) 2. allOf(A, OP rhs)	One public signal indicating if check was oneOf or allOf: 1 for oneOf, 2 for allOf, and one or two other signals are depending on the element type T.

While the protocol does not impose a restriction on the number of fields in the body part, the verification stack might set a limit due to its inherent constraints of circuit size. For example babyzk stack limits the number of claims by setting the limit of public inputs to 256, including intrinsic signals.

Primitive types

There are some build-in credential types. Issuers can start to issue credentials of these primitives types without going through the type creation process. Based on Galxe's experience, these types should cover a large portion of common use cases.

Type ID	Name	Description	Definition	Example
1	Unit	The simplest credential type that does not contains any field. Owning a credential of this type means that the owner, indicated by the ID in header, is endorsed by the issuer for the context.	empty	1. Have you ever been followed by Elon Musk. 2. Have you ever made any donation to Wikipedia. 3. Have you ever been a LP for Uniswap V2 before 2021.
2	Boolean	A credential with a bool claim. Comparing with Unit type, theoretically the set has two values instead of one. A practical difference is that Boolean type gives the owner the ability to choose to hide or reveal the value in the proof. There are some use cases, but in most cases, the Unit type is preferred.	`val:bool;`	Do you have an early supporter role in Galxe's discord? Since Galxe only issues credentials of this context to its guild member, owner can choose to reveal this value, showing that he is an early supporter, or without revealing the value, proving that he is at least a member of the guild.
3	Scalar	A credential with an unsigned integer claim, 248-bit. Unless 256-bit range is necessary, this type is recommended for credentials storing an unsigned integer value.	`val:uint<248>;`	1. Holder's loyalty points of some brand. 2. UNIX Timestamp of the holder's first transaction on Ethereum. 3. ERC20 token balance on height 1000, when representing the token's `total_supply * decimals` requires less than 248 bits.
4	Scalar256	A credential with an 256-bit unsigned integer claim. Note that due to the scalar field size limit in some verification stack (including babyzk), the output signal of a 256-bit value will represented as two public inputs, making on-chain verification higher than the Scalar type.	`val:uint<256>;`	1. ERC20 token balance, where the ERC20 token balance need the full 256-bit encoding.
5	Property	A credential with a 248-bit property claim, using user-defined hash algorithm, with 1 equality check.	`val:prop<248,c,1>;`	1. Name of holder's first pet. 2. Holder's first name on passport.

Credential format

A credential is made from header, body, an array of signatures and attachments. This design allows a credential to be endorsed by multiple signatures from different issuers, or using different verification stacks. Credentials are primarily stored and presented to holders in JSON format, for readability, while other formats like YAML, XML can also be supported. Note the formats of storing and showing the credential are not related to how the digest (i.e., the hash value) is computed, and will not affect zero-knowledge proof generation. The digest algorithm is determined by the verification stack and it can evolve without upgrading the main protocol.

```
             +----------------------------------------+
    Header   |       Version, Type, Context, ID       |
             +----------------------------------------+
        Body  |                (Field)*                |
              +----------------------------------------+
              |    Metadata      |      Signature      |
Signature(s)  |----------------------------------------|
              |         ...more signatures...          |
              +----------------------------------------+
 Attachments  |       Signer ID, Public key....        |
              +----------------------------------------+
```

```
{
  "header": {
    "version": "1",
    "type": "778",
    "context": "666",
    "id": "9"
  },
  "body": {
    "token_balance": "100",
    "birthday": "200",
    "status": {
      "str": "enabled",
      "value": "2"
    },
    "followed": "true"
  },
  "signatures": [sig1, sig2...]
}
```

```
{
  "metadata": {
    "verification_stack": 1,
    "signature_id": "999",
    "expired_at": "100",
    "identity_commitment": "123",
    "issuer_id": "456",
    "chain_id": "0",
    "public_key": "public key bytes"
  },
  "signature": "signature bytes"
}
```

Header

Header contains the common fields that all every credential must have.

- **Version** (uint8) : The version of protocol. Currently, the only supported version is 1.
- **Type** (uint160) : The type ID this credential, specifying the order and type of the claims in the body. This value will be a public signal of a proof.
- **Context** (uint160) : The context ID of the credential. This context ID needs to be registered on-chain before using it. Issuers are allowed to re-use the context created by others. Context IDs are computed by taking the 160 least significant bits of the keccak256 hash of the context string. Optionally, the issue can attach the original context string in the attachments section for better readability. This value will be a public signal of a proof.
- **ID** (uint248) : An identifier of holder. For example, an EVM/Bitcoin/Solana/Cosmos/Mina address, a X (Twitter) handle, or a Discord handle. If the original identity is a string, issuer must convert it to an integer value, by taking the 248 least significant bits of the output of some collision-resistant hash function, or ways that can guarantee the 1-to-1 mapping from the original handle to the ID. For example, 160-bit EVM address can be directly used as IDs. This value does not need to be registered on-chain. It will not be a public signal of a proof. Instead, the proof will contains a equality check result that shows the ID equals or not equals to some ID.

For example, to create a credential of the loyalty points, that a Galxe.com user has earned in the Galxe space, assuming that we have registered the context "Galxe.com loyalty points" on-chain and the context ID is `0xdeadbeef....`, when the user's Galxe ID is `123123` , Here is an example header.

```
"header": {
  "version": "1",
  "type": "2",
  "context": "0xdeadbeef..",
```

Note that Its type ID is 2, meaning that it is using the primitive credential type containing just one scalar value.

```
  "id": "123123"
}
```

Body

The body part stores all claim values, corresponding to the claim types declared in the credential type. Below is an example of credential body and its type definition. Note that the `status` claim value stores not only the hash value, but also the original value. This is because that the claim declaration specify that the hash algorithm is user-defined (`c`), storing both values is mandatory by protocol's specification.

```
  "body": {
    "token_balance": "100",
    "birthday": "200",
    "status": {
      "str": "enabled",
      "value": "2"
    },
    "followed": "true"
  },
```

```
token_balance:uint<256>;
birthday:uint<64>;
status:prop<32,c,1>;
followed:bool;
```

Signature

The signature part of a credential is an array of pairs of metadata and signature, allowing one credential to have multiple signatures. Although the format of a signature is specified by the verification stack used by the issuer, all signatures shares a common metadata structure.

Signature metadata

- **verification_stack_id** (uint8): an ID of verification stack used for this credential.
- **signature_id** (uint248): the id of this signature that will be used to manage revocation. This value will never be exported as a public value for user's privacy. It will only be used in the circuit when proving that the credential is not revoked, if required.
- **expiration** (uint64): the UNIX timestamp of when the credential will be expired. This field is mandatory and you can fill-in an `UINT64_MAX` if the credential never expires.
- **identity_commitment** (uint256): a field serves as the identity commitment for zero-knowledge proofs. The algorithm of computing the value is depending the chosen verification stack.

Identity commitment will be used as the key to the power of creating a link from an identity to another one, e.g., from a twitter handle to an EVM address. At verification, a holder must generate a proof that he is the actual owner of the credential, by showing that he knows the secrets behind the commitment, and output a number representing the pseudonymous identity the verifier will see. Thus, the verifier is assured that the verification originates from the holder and confirms the holder's identity as the entity communicating with the verifier in that session.

Note that `issuer_id` is not part of the signature. In our design, the `issuer_id` is intentionally omitted from the signature component. This approach prevents the embedding of trust relationships between the issuer and public keys within the credential, because the verification of a public key trustworthiness can only occur externally to the zero-knowledge proof circuit, as it involves stateful checks beyond cryptographic validation. This simplification eliminates the redundancy of including an `issuer_id` for zk proof generation. Furthermore, this flexibility enhances the system's scalability and the fluidity of trust relationships, thereby maintaining the simplicity of the credential's structure while accommodating a dynamic network of trust.

Attachments

The "attachments" section houses values that are external to the main body of the credential. It means that these values are not authenticated (or *signed*) by the issuer. Three fields are mandatory: `issuer_id`, `chain_id` and `public_key`. They do not require explicit signing, because the issuer's signature inherently implies those values. Issuers have the flexibility to introduce any arbitrary fields to this section. For instance, an issuer could provide a passport credential accompanied by a PNG photo. In such a case, issuer can place the hash of the photo in the credential body, while the actual binary data of the image would reside in the attachments.

- **issuer_id** (uint248): Represents the unique identifier of the issuer.
- **chain_id** (uint64): Represents the ID of the primary chain that the issuer was registered. Because the protocol is chain-agnostic and an issuer can register themselves in different chains using a consistent issuer_id, this primary chain ID is only a suggestion from issuer to verifier about where to check the public key status and revocable credential state. Note that although chain id may a 256-bit value, we do not support chain id larger than uint64_max.
- **public_key** (specific to the verification stack): Contains public keys essential for signature validation without querying it from the network.

- Additional fields as deemed necessary by the issuer.

Proof

Credential holders can generate zero-knowledge proofs to selectively disclose specific pieces of information on credentials. Regardless of the chosen verification stack, every proof consists of two main components: a proof and some public inputs. The structure and representation of a proof is varying by the verification stack. There are two types of public inputs: intrinsic public signals and user-defined signals. Intrinsic signals must be the first 8 or 9 signals (depending on if it is revocable) of the public input array. The protocol mandates the sequence of intrinsic signals as described below:

1. **out_type** (uint160): The type ID.
2. **out_context** (uint160): The context ID.
3. **out_nullifier** (uint256): A value derived from both `internal_nullifier` and `external_nullifier`. When using the babyzk verification stack, this value is computed as `poseidon(internal_nullifier, external_nullifier)`.
4. **out_external_nullifier** (uint160): The external nullifier used to generate the proof.
5. **out_reveal_identity** (uint256): The identity endorsed by the credential holder. Verifiers are obliged to ensure this value aligns with the session user's identity. For instance, for on-chain verification on EVMs, this value should be the direct caller, represented by `msg.sender`, or indirectly be authenticated through an EIP-712 signature.
6. **out_expiration_lb** (uint64): This signal delineates the lower boundary of the credential's expiration date.
7. **out_key_id** (uint256): Represents the hash of the public key that authenticated the credential.
8. **out_id_equals_to** (uint256): This field confirms the equality of the ID field. It adopts a compressed schema similar to `prop<248, c, 1>`. Here, the least significant bit designates equality, while its [249..1] bits convey the value that underwent comparison.
9. **(optional) out_sig_revocation_smt_root** (uint256): Only if the credential is revocable, there will be this intrinsic public signal, representing the hash of the root of the sparse merkle tree used provided during proof generation.
10. Public inputs introduced by claims in the body will start from here.

```
{
  "out_type": "778",
  "out_context": "666",
  "out_nullifier": "20587068051456727584720549518481563004145412893468610943364726469406950307723",
  "out_external_nullifier": "2056503065145424929058472778140048520960157098084297103754748891319000
  "out_reveal_identity": "0xdeadbeef",
  "out_expiration_lb": "99",
  "out_key_id": "17435824163656511673929665985298433476173638621066978183283107708096646071l7",
  "out_id_equals_to": "19",
  "out_sig_revocation_smt_root": "1243904711429961858774220647610724273798918457991486031567244100
  "out_token_balance_lb_msb": "0",
  "out_token_balance_lb_lsb": "50",
  "out_token_balance_ub_msb": "0",
  "out_token_balance_ub_lsb": "101",
  "out_birthday_lb": "199",
  "out_birthday_ub": "201",
  "out_status_eq0": "6",
  "out_status_eq1": "8",
  "out_followed": "0"
}
```

Technical components

This section describes all the components implemented by the protocol.

Verification stack: BabyZK

The initial zero-knowledge verification stack we have integrated is named "BabyZK". This stack employs the subsequent algorithms and parameters:

- **Curve**: BN254. All the following algorithms are using this curve.
- **Hash**: Poseidon.
- **Digest:** The digest is derived from the Poseidon hash of `poseidon(header, signature metadata)` and `poseidon(body)`. Initially, the header and signature metadata are cohesively hashed, according to the canonical sequence of header fields followed by signature metadata fields. Subsequently, the fields within the body are hashed in the type definition order. Finally we take the Poseidon hash of the two hash values.
- **Signature**: EdDSA Poseidon

- **Identity commitment schema:** a 2-input poseidon hash of BN254 curve of two fields: `identity secret` and `internal nullifier`, in this specified order.
- **Circuit language**: Circom
- **Proof system**: Groth16, powered by SnarkJS
- **Limits**: The total number of public inputs of claims in the body must be less than 256, including intrinsic public inputs. For body types encompassing more than 16 signals—such as 9 `uint256` fields or 20 `bool`—the fields are segmented into groups of 16 and individually hashed. The final step is hashing results from these group hashes. This approach is necessitated by Poseidon's constraint of 16 signals. Given the further restriction that there can't exceed 256 fields, this hashing methodology can be implemented through a two-layer hashing structure.

Smart contracts

Smart contracts in the protocol serve two primary functions:

- **On-chain data storage:**
  - Context registry
  - Issuer registry
    - Metadata
    - Public key management
    - Revocable credential state management
  - Type registry
    - Primitive type management
    - Metadata
      - Definition
      - Resource URI
    - Static proof verifier
    - Public input parser
  - Trust schema
- **On-chain Verification**:
  - Stateful credential zk verifier contract
  - Aggregated on-chain attestation

The Galxe identity protocol is crafted to be chain-agnostic, with plans to deploy these smart contracts across various EVM-compatible chains, including Ethereum, BSC, Polygon, and several Layer 2 solutions.

Context registry

A context is a string that, when paired with a credential type, forms a credential schema. This contract facilitates issuers in registering contexts, ensuring the meanings of credentials are publicly accessible.

- Issuers can introduce a new context by submitting a string to the contract. The Context ID is derived from the keccak256 hash of this string, truncated to the lower 160 bits.
- Contexts are designed to be universal, instead of binding to a credential type. While it might seem unconventional to associate a boolean-typed credential with the context "Trading volume," we entrust this decision to the issuer.

Issuer registry

This contract empowers issuers to register and authenticate their identities on-chain and manage the status of public keys and state of revocable credentials.

- **Issuer Metadata**:
  - **Issuer ID**: Crafted to be chain-agnostic and deterministic, the issuer's EVM address is directly adopted as their Issuer ID. For chains not based on EVM, addresses can still be utilized, albeit truncated to 160 bits.
  - **Name:** Issuer can specify their name.
  - **Identity Verification** (built in separate contracts):
    - **DNSSEC**: Issuers can use DNSSEC records to validate their ownership of web2 domains.
    - **GAL Staking**: Issuers can manifest their commitment by staking GAL tokens.
    - **ENS/SpaceID & Other Bindings**: This facilitates the association of issuers with a web3 identity.
- **Issuer State**:
  - **Public Key Manager**: This contract caters to various verification stacks and is structured as a mapping from `(issuer_id, key_id)` to the key's status. The key ID is derived from the hash of the public key, with the hashing algorithm contingent

on the selected verification stack. Key statuses are bifurcated into:

        - **Active**: Denotes keys currently in use.
        - **Revoked**: Signifies that all credentials signed with these keys are invalidated.

    - **Signature State Manager**: For credentials that can be revoked, this contract provision allows issuers to regulate the revocation status for each combination of credential type and context. This is achieved by uploading the hash root of the sparse merkle tree containing signature IDs of the revoked credentials. Issuers need to publish the whole tree to a permanent storage for holders to access.
        - URI to the sparse merkle tree.
        - Root hash.

Credential Type Registry

The credential type registry contract serves as a comprehensive registry for managing credential types. Credential type designers can utilize this contract to introduce new types into the ecosystem, by storing all necessary artifacts on-chain. Issuers, holders, and verifiers will use this contract for creating new credentials, proof generation and verification.

- Type Registration
    - Allows for the registration of new credential types by users.
    - Type metadata storage
        - Stores type metadata such as revocability, admin address, name, definition in DSL, description, and a resource URI for proof generation artifacts.
- Type Management
    - Functionality to update the verifier and public signal getter for a type and a verification stack, adapting to evolving verification needs.
    - Functionality to transfer type ownership.
    - Check if a type is fully initialized for a given verification stack, ensuring readiness before deployment.
- Utility Functions
    - Check if a type is fully initialized for a given verification stack, ensuring readiness before deployment.
    - Calculate the typeID based on the creator's address and the name of the type

Stateful verifier

Besides using the verifier contract, that is associated with the credential type, to verify the validity of zero-knowledge proof, a full verification requires additional on-chain state validations, including:

1. Do type and context matches verifier’s expectation?
2. Can the revealed credential expiration lower bound prove that the credential is not expired? (greater or equal to the current blocktime)
3. Is the public key still active for the expected issuer?
4. If the credential is revocable, is the proof generated with the current SMT root?

For Babyzk, we built a contract that does all the checks in a simple function call `verifyProofFull`, so that verifiers can validate a proof with just one function call. It supports both:

- Static Verification: Checks the validity of proofs without considering the state of the issuer's public key or revocation lists.
- Full Verification: Extends static verification by also checking the issuer's public key activity and whether the revocation tree check is up-to-date.

Trust Schema Manager

This smart contract empowers verifiers to establish trust schemas tailored to specific verification scenarios. A straightforward example might involve enumerating a list of approved credential issuers. More intricate configurations could involve placing trust in issuers with active GAL staking, those operating under delegated trust, or issuers with a valid DNSSEC verification tied to some web2 domains. This contract transforms the trust establishment process into a communal activity, allowing trust schemas to be shared, voted upon, and dynamically adjusted.

Verifiers have the capability to introduce a new trust schema by supplying the address of a callback function. This function must be designated as a view function. The trust schema ID is then derived using a combination of the verifier's address and a nonce.

SDK

The protocol provides an open source implementation of SDKs that allows developers to build application on Galxe identity protocol. The SDK provides:

1. Credential maker function: generate a credential based on its type.
2. BabyZK verification stack

a. circuit generator: generate circom code based on type definition.

b. signer: Sign a credential with private key.

c. proof generator: Generate a proof based on the type and conditions.

d. proof verifier: Based on the credential type, query verification key from blockchain and return the verification result. Users should always use the SDK's proof verifiers or on-chain verifiers, when possible to enable type-based security checks.

Hosted service

Galxe BabyZK trust setup MPC node

This is a hosted service that can participate in phase 2 trust setups for BabyZK's Groth16 circuits. The service is not a necessity for protocol, but it can improve the user experience of credential type designer and its users. They do not need to worry about the trust setup when a new type is newly invented. The Galxe's MPC node will make sure to add secrets into the result, seal and publish the proving and verification keys to decentralized storage and corresponding verifier.

Application: Galxe identity vault

At the core of Galxe identity protocol's philosophy lies a deep commitment to the idea of self-sovereign identity. In pursuit of ultimate sovereignty, we developed the **Galxe identity vault**, a local storage solution for identity data, with embedded zero-knowledge provers, that gives users

1. **self-evident privacy safeguard**: given that data is stored locally, users can ensure that their personal information is kept private. Data does not need to pass through or be stored on any external servers where it could potentially be accessed by unauthorized parties.
2. **total access control**: Local storage gives users full control over their own data. They can decide what to do with it, when to share it, how to share it, and who to share it with. This is in contrast to storing data with centralized authorities or intermediaries, where users must trust those authorities will handle their data appropriately, and hope their services are live when needed. With local storage, users can ensure that their data is stored in their own jurisdiction, avoiding potential legal and regulatory complications.
3. **no-code ZKP**: It allows credential type designer, issuer, verifier, and holder to run all the workflow without any coding.

The Galxe identity vault will be developed as an open-source reference implementation, and any party can build their own vaults as long as it follows the specification. The reference implementation of Galxe identity vault will be available as browser extensions and native Apps. Once installed, the user can start to collect verifiable credentials to build his self-sovereign identity.

By using zero-knowledge proof technologies, users will be able to practice data minimization, down to the least required. In practice, all computation on the credentials will happen on user's local device via embedded zk-circuits, and only the public input and output will be shared with verifiers.

The wallet app must support:

- all the workflows defined below and
- generating a key pair on Baby Jubjub curve, which will be used for signing.
- sync and fetch credentials from issuers.
- checking on-chain state of revocable credentials.
- issuing credentials to any other identity

Storage add-on(s)

The protocol can work with decentralized permanent storage to improve its user experiences, by storing these information:

1. Proving keys, which is required for holders to generate zero-knowledge proofs. The size of these keys range from one megabytes to even hundreds of megabytes, depending on the number of constraints of the circuit.
2. Sparse merkle tree of revoked credentials. Without the SMT data, holders cannot prove the validity of revocable credentials.

Workflow

Most workflows are depending the chosen verification stack. For demonstration, we will explain them of BabyZK.

Type designer: create and publish a new credential type

Adding a new type to the protocol is a permission-less process. Type designer can use Galxe identity protocol SDK, or he can use Galxe identity vault for the no-code solution.

To introduce an official type, which will have long-term support of verification stacks, Galxe identity protocol employs a governance framework to facilitate decision-making regarding the acceptance and inclusion of new official credential types proposed by credential designers. The community, comprising users, developers, and other stakeholders, actively participates in the evaluation and acceptance process. Through transparent discussions and GAL voting mechanisms, the community collectively determines which proposed credential types are incorporated into the protocol.

The common workflow is:

1. Idea and Proposal: An individual or a group comes up with a new credential type. They draft a proposal document explaining the concept, its benefits, and technical specifications, including name and type of the fields, supported verification stack, zero-knowledge proof circuits, and details of the trust setup ceremony if required.
2. Community Feedback: The proposal is shared with the community for review and feedback. This can be done through forums, mailing lists, social media, or dedicated development channels like the official Galxe identity protocol github repository.
3. (Primitive) Type Number Assignment: If the proposal gains support and consensus within the community, an type number is assigned to it. The community uses these numbers to refer to specific proposals. Only applies to primitive types.
4. Audit and Security: Depending on the complexity and potential impact, an external security audit might be conducted to identify vulnerabilities or weaknesses in the proposal: (1) whether the wire format might be ambiguous, (2) if circuits were under-constrained or over-constrained, and (3) security level of the trust setup ceremony.
5. Voting and Adoption: Once the implementation passes security audits and testing and passes GAL voting process, it can be deployed to the Galxe identity protocol as a new official type.

	Role (who's doing it)	What	Input	Output	Modules
1. Create credential type	Credential designer	Design a new credential type expressed in our DSL.	Credential type specification in DSL	ZK circuit	SDK, circuit generator
2. Phase2 trusted setup for the type and circuit. Depending on the verification stack, this may be skipped	Credential designer + other parties (Galxe)	Phase2 setup for zk-related, specifically for babyzk stack's proof system: Groth16	ZK circuit	1. proving key 2. verification key	The type designer needs to host a trusted setup ceremony for the circuit. Galxe identity protocol will provide an optional MPC node for participating the ceremony.
3. Publish credential type	Credential designer	Type specification will be published on-chain, and a deterministic Type ID will be generated. For babyzk, verification key parameters must be updated to the verifier contract.	1. proving key 2. verification key 3. Circuit artifacts like witness calculation WASM. 4. onchain verifier contract.	1. Type ID. 2. add the verification key to the verifier. 3. proving key will be stored on decentralized storage.	contract: 1. Credential type registry. 2. Type-specific verifier contract. Storage: 1. decentralized permanent storage.

```
.--------..-----------------..-------------.       .----------------------------.
|Designer||   Protocol SDK   ||Trusted setup|       |(optional: Galxe-hosted MPC)|
'--------''-----------------''-------------'       '----------------------------'
    |              |                 |                            |
    |  Type DSL    |                 |                            |
    |------------->|                 |                            |
    |              |                 |                            |
    |circom circuit|                 |                            |
    |<-------------|                 |                            |
    |              |                 |                            |
    |         circom circuit         |                            |
    |------------------------------->|                            |
    |              |                 |                            |
    |              |                 |          *.zkey            |
    |              |                 |--------------------------->|
    |              |                 |                            |
    |              |                 |proving and verification key|
    |              |                 |<---------------------------|
    |              |                 |                            |
    |  proving and verification key  |                            |
    |<-------------------------------|                            |
```

```
.--------.        .-------..-----.      .-------------.
|Designer|        |Storage||Chain|      |Type Registry|
'--------'        '-------''-----'      '-------------'
    |                 |       |                |
    |Circuit artifacts|       |                |
    |---------------->|       |                |
    |                 |       |                |
    |  Resource URI   |       |                |
    |<----------------|       |                |
    |                 |       |                |
    |Deploy verifier contract |                |
    |------------------------>|                |
    |                 |       |                |
    |Type specification, resource URI, verifier|
    |----------------------------------------->|
```

Issuer: register and issue credentials

	Role (who's doing it)	What	Input	Output	Modules
4. Register as an issuer	Issuer	Register to become an issuer.	1. metadata 2. public keys 3. proof of identities.	issuer ID	Issuer registry + pubkey manager for different verification stack
5. Register a new credential context	Issuer	Add a new context on-chain.	1. CredType ID. 2. Context string	context ID	contract: context registry Context ID: hash of context string cut to 160 bits, from LSB.
6. Issue a new credential	Issuer ←→ owner	Sign a new credential to user.	1. credential header 2. claim values. 3. user's identity commitment 4. signature	a credential in JSON	SDK

To become an issuer, one can register on-chain. The issuer ID is computed deterministically based on the caller's address. Issuers can add proofs of identity to show his accountability, e.g., DNSSEC verification, and GAL staking. It is recommended to use a smart contract wallet, e.g. ERC-4337 wallet, to register the issuer, for permission management.

```
.------.            .---------------..-----------------.
|issuer|            |Issuer Registry||Identity contract|
'------'            '---------------''-----------------'
   |                        |                 |
   |        metadata        |                 |
   |----------------------->|                 |
   |                        |                 |
   |            proof of identity             |
   |----------------------------------------->|
   |                        |                 |
   |active/remove public key|                 |
   |----------------------->|                 |
```

Before issuing a credential, issuer must decide the credential schema, which is the type and context of credential. The context is a string that describes the content of the credential. For example, for a credential that proves a user's total USD value of financial assets, issuer can use the basic credential type 'scalar', and its context can be a string of "total USD value of financial assets". The separation of context and type in a the credential schema allows users to reuse the pre-compiled zero-knowledge proof circuits. Note that credential schema is universal, meaning that you can use a schema that is created by some other issuers. This can also be useful during verification: a verifier can program the logic of which schema to accept, from a simple list of schema IDs to dynamic logic programmed on-chain.

```
.------.              .----------------.
|Issuer|              |Context Registry|
'------'              '----------------'
   |                          |
   |Type ID and context string|
   |------------------------->|
   |                          |
   |        context ID        |
   |<-------------------------|
```

One important benefit of Galxe identity protocol is that, credential issuance can happen completely off-chain. The issuing process requires holder to provide an identity commitment to the issuer. An issuer must maintain a strict 1-to-1 mapping from the unique ID of the under the application to the identity commitment. In another word, a user must use the same commitment for an issuer. If the issuer failed to do so, the holder can create multiple proofs of different nullifiers, which allow him to 'double-spend' the credential.

Note that this interactive process can be improved by using PLUME, if the ID is a public key. However, due to the high cost (over 2M constraints) under BabyZK stack, it is currently not supported.

```
  .------.           .------.                    .---------------------.
  |Holder|           |Issuer|                    |Identity Protocol SDK|
  '------'           '------'                    '---------------------'
     |                  |                                   |
     |identity commitment|                                  |
     |----------------->|                                   |
     |                  |                                   |
     |                  |credential header, claims, signing key|
     |                  |----------------------------------->|
     |                  |                                   |
     |                  |        verifiable credential      |
     |                  |<-----------------------------------|
     |                  |                                   |
     |    credential    |                                   |
     |<-----------------|                                   |
```

After issuing credentials, issuers can still manage their validity on-chain, if they are revocable. the protocol provides different ways to do it

1. Signing key management. Signing keys can be marked as active or deprecated. Verifiers should never trust any credential that is signed by a deprecated key.
2. Expiration date embedded in the signature.
3. Revocable credentials: sparse merkle tree (SMT) of signature IDs.

	Role (who's doing it)	What	Input	Outcome	Modules
6.1 Credential revocation	issuer	Issuers can management the 'signatures' they signed on credentials.	1. expiration. 2. signature ID via SMT. 3. explicitly seal the credential.	Credential will be revoked, holder cannot generate proofs that satisfy the goal.	Issuer Registry

Owner: proof generation and identity management

	Role (who's doing it)	What	Input	Outcome	Modules
7. zk-proof generation	owner	Credential holder can generate a proof against a set of conditions.	1. credential. 2. conditions. 3. proving key. 4. external nullifier	1. proof 2. public data	Galxe identity vault or any apps that have integrated identity protocol SDk.

Owners can generate proofs that they hold a credential satisfying some constraints. This is done by using the circuit generated from compiled for the credential type. Using the credential and the conditions that will be checked as input, user can generate a succinct proof showing that the holder has a credential satisfying conditions.

```
 .-------.           .-----.                                              .------------.
 |Storage|           |Owner|                                              |Protocol SDK|
 '-------'           '-----'                                              '------------'
     |                  |                                                       |
     |Proof gen artifacts|                                                      |
     |----------------->|                                                       |
     |                  |                                                       |
     |                  |identity secrets, pseudonym, credential, and statements|
     |                  |------------------------------------------------------>|
     |                    |                                                          |
     |                    |                 proof and public input                   |
     |                    |<---------------------------------------------------------|
 .-------.             .-----.                                                      .------------.
 |Storage|             |Owner|                                                      |Protocol SDK|
 '-------'             '-----'                                                      '------------'
```

Verifier: verify proofs

Verifiers can validate a proof both on-chain and off-chain. Their respective workflows are shown as below:

	Role (who's doing it)	What	Input	Outcome	Modules
9. zk-based credential verification	verifier	Owners can submit a proof to verifier that some statements about them is true.	1. proof.json 2. public.json	accept/reject	Stateful verifier, type registry, issuer registry

On-chain verification is extremely simple. Verifiers just need to submit the proof and the public input to the stateful verifier. For most of the cases, we recommend applications to use on-chain verification even for off-chain verification, by making an external call to the stateful verifier contract via RPC.

```
.--------.         .-----------------.
|Verifier|         |Stateful verifier|
'--------'         '-----------------'
    |                       |
    |proof and public inputs|
    |---------------------->|
    |                       |
    |  verification result  |
    |<----------------------|
```

Off-chain verification also rely on some on-chain states, for example, verification key and public key status. However, note that these values can be cached in some use cases, for example, when applications are sure that some public keys are always valid. In those cases, after initial setup, verifiers do not need to query anything on-chain.

```
.--------.             .--------------..-----------------..---------------..------------.
|Verifier|             |Typed registry||Verifier contract||Issuer registry||Protocol SDK|
'--------'             '--------------''-----------------''---------------''------------'
    |                         |                 |                 |              |
    |verifier contract address|                 |                 |              |
    |<------------------------|                 |                 |              |
    |                         |                 |                 |              |
    |             verification key              |                 |              |
    |<------------------------------------------|                 |              |
    |                         |                 |                 |              |
    |           public key status, revocation SMT root            |              |
    |<------------------------------------------------------------|              |
    |                         |                 |                 |              |
    |            vkey, key status, SMT root, proof and public inputs             |
    |--------------------------------------------------------------------------->|
    |                         |                 |                 |              |
    |                         |  verification result              |              |
    |<---------------------------------------------------------------------------|
```

Use Cases

Sybil Prevention

Sybil attack is one of the most prominent attack vectors in the digital space. Currently there is no foolproof solution to prevent sybil attacks. However, the greater amount of identity related data one can access, the easier it becomes to protect sybil attacks.

One solution to protect sybil attacks is through the implementation of KYC (know-your-customer) mechanisms. However, current KYC solutions raise privacy concerns as centralized entities collect and store users' personal identity information such as address, nationality, name, etc. Galxe Identity Protocol offers a solution to solve this issue by allowing users to have ownership and control over their sensitive identity information. With the Galxe Identity Protocol, users can securely share their data with third parties in a privacy-preserving way using zero-knowledge-proof verification stacks. By empowering users with control over their sensitive information, Galxe Identity Protocol enhances privacy while addressing the need for identity verification in preventing sybil attacks.

Other solutions such as Gitcoin Passport are also effective approaches to tackle sybil attacks. These solutions request and utilize multi-dimension behavior and social data from users to estimate the likelihood of them being real humans and provide quantitative results. However, solutions like Gitcoin Passport typically require each vendor to collect a vast amount of data from users through centralized authentication processes. In contrast, with Galxe Identity Protocol, users' credentials are aggregated and controlled by the users by default. All developers can thus build such scoring systems without the need to collect additional information from the users.

Reputation System

Traditional reputation systems often operate independently, resulting in data silos where reputation information is fragmented and not easily portable or shareable between different platforms or services. This lack of interoperability can limit the effectiveness and potential of reputation systems.

Galxe Identity Protocol provides the opportunity to build a decentralized reputation system that can aggregate an individual's reputation. By utilizing the Galxe Identity Protocol, it becomes possible to create a reputation system that is decentralized, transparent, and secure, enabling reputation to be collected and shared across various platforms or applications. This decentralized approach can enhance the trustworthiness and usability of reputation systems in a more interconnected manner.

Credit System

Current credit systems often rely on private data, which can raise concerns about data security and privacy. Centralized credit systems typically require access to personal and sensitive information, creating a risk of data breaches or unauthorized access. Additionally, these systems are often permissioned, meaning only certain entities have control over the data and how it is used.

Implementing a permission-less credential protocol can offer several advantages for building a credit system. By leveraging Galxe Identity Protocol, it becomes possible to access a fuller and more diverse dataset for verifying and evaluating creditworthiness. This protocol can enable the integration of data from various sources, allowing for better accuracy and a more comprehensive understanding of an individual's creditworthiness.

Achievement System

Existing achievement systems often operate independently within specific applications or platforms. This creates data silos where the achievements earned in one application cannot be easily utilized or recognized in other applications. As a result, the value and impact of achievements are limited within their respective ecosystems.

An ideal scenario would be the ability to use achievements across different applications to unlock benefits or gain recognition. If achievements were transferable and recognized across various platforms or applications, it would enhance their value and provide users with tangible benefits beyond individual ecosystems. This would incentivize users to pursue and showcase achievements more actively, fostering a sense of accomplishment and motivation.

Personal Data Market

In the present scenario, application owners typically have control over users' personal data. They own, share, and may even monetize this information without direct user consent or control. This lack of control raises critical privacy concerns, while depriving users of the opportunities for monetization, and this could even been deemed as a deprivation of users' assets.

The Galxe Identity Protocol offers a paradigm shift by allowing individuals to regain control over their identity data. With the Galxe Identity Protocol, users can have ownership and agency over their personal credentials. They can decide how their data is used and shared, including the option to exchange data for improved services or monetary incentives. The privacy-enabled verification stack of the Galxe Identity Protocol ensures that users' privacy is protected while engaging in these data transactions.

Identity Verification

Traditional methods of identity verification, such as presenting physical documents or IDs, are prone to fraud. Methods like photoshopping IDs or faking credentials like education degrees or drivers' licenses have become increasingly common. These fraudulent practices undermine the reliability and trustworthiness of the verification process.

Issuing digital verifiable credentials can help address these identity fraud concerns. Verifiable credentials are digitally signed pieces of information that are tamper-proof and can be easily verified by relying parties. By issuing these credentials, individuals can securely and conveniently present their verified identity information without the risk of forgery or manipulation.

Privacy-Enabled Access Control

Many existing access control systems require individuals to disclose private information in order to gain access to certain privileges or events. This can include revealing specific holdings like NFT collections and disclosing personal addresses. These requirements can potentially compromise privacy and expose individuals to risks associated with sharing sensitive information.

By leveraging privacy-enabled verification stacks, it becomes possible to ensure privacy for all kinds of access control systems.

Decentralized Review System

Credentials in Galxe Identity Protocol go beyond personal identity data and can also include review-like credentials for other individuals or entities such as restaurants, clothing brands, and more. By aggregating reviews for all the entities within the network, developers can build decentralized review systems such as a decentralized version of Yelp. Moreover, the reputation scores built using the protocol can be utilized to weight the review aggregations, resulting in more robust and trustworthy review results.

Future works

In this section, we list some future works that we can think of.

Expanding to Mina ecosystem

Because of the swift progression in zero-knowledge proof technology, there exists an opportunity to expand the Galxe identity protocol into more ecosystems. Within this realm, Mina emerges as a paramount option. It is equipped with a proof system that can also run in user's local devices, especially in web browsers, thereby enabling the privacy-preserving proof generation and on-chain proof verifications on the Mina protocol. Additionally, this proof system is distinguished by numerous benefits, such as the ability for make recursive proofs and verifications, and the potential to support more universally recognized signature algorithms, for instance, ECDSA.

Same as BabyZK, the verification can happen both off-chain (using SDK), and on-chain, by sending proofs to contracts on Mina protocol. Because that Mina protocol has native support for zero-knowledge verification, an gas-saving attestation design is considered to be redundant in this context. In another word, smart contracts on Mina protocol should be able to consume proofs directly with affordable gas.

For issuers, when issuing the same credential for the same identity using different stacks, they must request a proof of same identity commitment from the owner, because the output of poseidon hash under different curves are different.

Verification stack: o1js

- **Curve**: **Pasta Curves**. All the following algorithms are using this curve.
- **Hash**: Poseidon.
- **Digest:** The digest is derived from the Poseidon hash of `poseidon(header, signature metadata)` and `poseidon(body)`. Initially, the header and signature metadata are cohesively hashed, according to the canonical sequence of header fields followed by signature metadata fields. Subsequently, the fields within the body are hashed in the type definition order. Finally we take the Poseidon hash of the two hash values.
- **Signature**: EcDSA. Because EcDSA signature is not yet live on Mina mainnet (only available for zkApps), we may change the signature schema.
- **Identity commitment schema:** a 2-input poseidon hash of curve of two fields: `identity secret` and `internal nullifier` , in this specified order.
- **Circuit language**: o1js, a typescript-native SDK
- **Proof system**: Kimchi