Citadel protocol specification

Dusk

November 22, 2023

Contents

1 Protocol overview

Citadel is a self-sovereign identity (SSI) protocol built on tope of Dusk that allows users of a given service to manage their digital identities in a fully transparent manner. More specifically, every user can know which information about them is shared with other parties, and accept or deny any request for personal information.

1.1 Properties

With Citadel, users of a service can request licenses that represent their *right* to use such a service. Citadel satisfies the following properties:

- *Proof of ownership*: users can prove that they own a valid license that allows them to use a certain service.
- *Proof of validity*: users with a valid license can prove that their license has not been revoked and is valid.
- *Unlinkability*: different services used by a same user cannot be linked from one another.
- *Decentralized session opening*: when users start using a service, the network learns that this happened and the license used to access to the service cannot be used again.
- *Attribute blinding*: users have the power to decide exactly what information they want to share and with whom.

1.2 The parties involved

Citadel involves three (potentially different) parties:

- The *user* is the person who interacts with the wallet and requests licenses in order to claim their right to make use of services.
- The *service provider* (SP) is the entity that offers a service to users. Upon verification that a service request from a user is correct, it provides such service.
- The *license provider* (LP) is the entity that receives requests for licenses from users, and upon acceptance, issues them. The LP can be the same SP entity or a different one.

1.3 The elements involved

Below there is the list of the elements involved in the protocol. The details of their structure and their role are explained in the following sections.

- A *request* is a set of information that the user sends to the network in order to inform the LP that they are requesting a license. It includes an stealth address where the license will be sent to.
- A *license* is an asset that represents the right of a user to use a certain service. In particular, a license contains a set of attributes that are associated to the requirements needed to make use of that service.
- A *session* is a set of public values sent by the user to the network that are associated with the initiation of the use of a service.
- A *session cookie* is a set of values that allows the SP to verify that a session was opened correctly.

1.4 Protocol flow

Citadel is a system that allows users to acquire a license to use a service and at the time of use, the SP can verify that they have a license but without identifying who is using it. The Citadel protocol has blockchain as its centerpiece, taking advantage of its high degree of integrity and allowing the interaction between the user and the SP not to be direct but via transactions on the network. The flow of the protocol is depicted in Figure 1 and goes as follows.

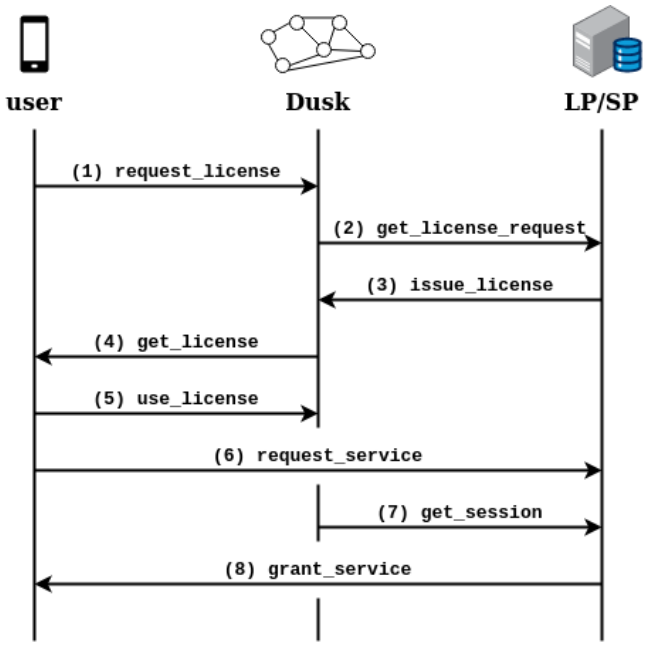

Figure 1: Overview of the protocol messages exchanged between the user, Dusk's network, and the LP/SP.

First, the user asks the LP for a license that will grant them access to a service. Typically, in order to grant this license to a user, the LP will have performed some prior checks on the user (probably off-chain). When the LP sends the license to the network, it gets included in a list of granted licenses. When the user wants to use the SP's service, he has to calculate his unique session identifier and also show that his license is included in the most recent version of the list of granted licenses. To do so, he has to generate a zero-knowledge proof that proves that the session identifier was computed correctly, and also that his license is part of the list of granted licenses that correspond to the latest state of the list. In particular, the user needs to show the SP three things: his session identifier, the latest state of the list, and the zero-knowledge proof. Finally, in order to grant the service, the SP does the following three checks:

1. The session identifier has not been used before.
2. It checks the state of the list is indeed the latest one.
3. It verifies the zero-knowledge proof provided by the user.

If all checks are correct, the user is allowed to use the service. Otherwise, access to the service is denied.

2 Cryptographic primitives

In this section, we detail the cryptographic primitives used in Citadel. We briefly introduce Merkle trees, the commitment scheme, encryption scheme, proof system, elliptic curves and hash functions used, specifying at each step the concrete parameters with which each of the primitives is instantiated.

Notation. Throughout the document, we use the following conventions. Given a set S, we denote sampling an element x uniformly at random from S by $x \leftarrow S$. Any group $\mathbb{G}$ used is of a large prime order, and we assume that the discrete logarithm problem is hard in $\mathbb{G}$. If two elements are denoted by the same letter in upper case and lower case, e.g. a, A, this often signifies the fact that A is a public key corresponding to the secret key a.

2.1 Elliptic curves

BLS12-381 [2] and Jubjub [3] are the elliptic curves used. More precisely, let

$$q = 400240955522166739341778982573590415655688281993900788533205813612403165049083786444268762912901566403789427255978 7,$$

$$p = 524358751751261904794477405081859658376905525005276378226036586 99938581184513.$$

Note that both are prime numbers, with bit-lengths 381 and 255, respectively. The curve BLS381-12 is the curve over $\mathbb{F}_q$ defined by the equation

$$E : Y^2 = X^3 + 4.$$

We have that $E(\mathbb{F}_q)$ has different subgroups $\mathbb{G}_1, \mathbb{G}_2$ such that $\#\mathbb{G}_1 = \#\mathbb{G}_2 = p$. This curve is pairing-friendly (with embedding degree $k = 12$), so pairings are efficiently computable. More precisely, Citadel makes use of the bilinear group

$$\mathbb{B} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T)\,.$$

By instantiating the zk-SNARK with the bilinear group $\mathbb{B}$, we are be able to prove statements about satisfiability of arithmetic circuits over $\mathbb{F}_p$, the so-called *scalar field* of E.

Furthermore, we are interested in proving certain operations with the zk-SNARK, like the correct verification of a Schnorr signature σ. Note that σ is an element of a certain elliptic curve J, but is represented as two coordinates in the base field $\mathbb{F}_s$ of J. Therefore, the verification can be best represented as arithmetic constraints modulo s. While it is possible to represent any NP statement using arithmetic modulo p to plug it into the zk-SNARK, this incurs into a significant efficiency loss if not done carefully. The natural thing is to set $s = p$. Therefore, the signature scheme must be instantiated with an elliptic curve over $\mathbb{F}_p$. For this, let

$$d = -\frac{10240}{10241} \bmod p.$$

Citadel uses the Jubjub curve, defined by the equation

$$J : -X^2 + Y^2 = 1 + dX^2Y^2,$$

over $\mathbb{F}_p$. In particular, it uses a subgroup $\mathbb{J}$ of order

$$t = 6554484396890773809930967563523245729705921265872317281365359162392183254199,$$

which is a 252-bit prime.

The primes and groups defined here will be used through the rest of the document.

2.2 Digital signatures

The Schnorr Sigma protocol [16] is used, compiled with the Fiat–Shamir transformation [5, 14], as a signature scheme. In particular, Citadel makes use of the standard scheme as well as a double-key version to be able to delegate computations later in the protocol. Let $G, G' \leftarrow \mathbb{J}$.

The single-key signature scheme is as follows.

- *Setup.* Sample a secret key $\mathsf{sk} \leftarrow \mathbb{F}_t$ and set the corresponding public key $\mathsf{pk} = \mathsf{sk}G$. Output $(\mathsf{sk}, \mathsf{pk})$.
- *Sign.* On input a message m and a secret key sk, sample $r \leftarrow \mathbb{F}_t$ and compute $R = rG$. Compute the challenge $c = H(m, R)$, and set

  $$u = r - c\mathsf{sk}.$$

  Output the signature $\sigma = (R, u)$.

- *Verify.* On input a public key pk, message m and signature $\sigma = (R, u)$, compute $c = H(m, R)$ and check whether the following equality holds:
$$R = uG + c\mathsf{pk},$$
If so, accept the signature, otherwise reject.

The double-key signature scheme is as follows.

- *Setup.* Sample a secret key $\mathsf{sk} \leftarrow \mathbb{F}_t$ and set the corresponding public key $(\mathsf{pk}, \mathsf{pk}') = (\mathsf{sk}G, \mathsf{sk}G')$. Output $(\mathsf{sk}, (\mathsf{pk}, \mathsf{pk}'))$.
- *Sign.* On input a message m and a secret key sk, sample $r \leftarrow \mathbb{F}_t$ and compute $(R, R') = (rG, rG')$. Compute the challenge $c = H(m, R, R')$, and set
$$u = r - c\mathsf{sk}.$$
Output the signature $\sigma = (R, R', u)$.
- *Verify.* On input a public key pk, message m and signature $\sigma = (R, R', u)$, compute $c = H(m, R, R')$ and check whether the following equalities hold:
$$R = uG + c\mathsf{pk},$$
$$R' = uG' + c\mathsf{pk}'.$$
If so, accept the signature, otherwise reject.

The signature scheme is existentially unforgeable under chosen-message attacks under the discrete logarithm assumption, in the random oracle model [10, Section 12.5.1]. While the Schnorr signature scheme is widely known, the double-key version has not been used before, to the best of our knowledge. In Citadel, as it happens in the Phoenix transaction model [4], this is leveraged to allow for delegation of proof computations without the need to share one's secret key with the helper.

2.3 Hash functions

Citadel uses hash functions H mostly in the case where given y, we want to prove knowledge of x such that $H(x) = y$. We will do so with PlonK, which requires statements to be written as arithmetic constraints modulo a large prime number p. Most hash function evaluations do not naturally translate to this language, incurring in a big efficiency loss. To avoid this, the Poseidon hash function [8] $H : \mathfrak{F}_p \to \mathbb{F}_p$, where $\mathfrak{F}_p$ is the set of tuples of $\mathbb{F}_p$-elements of any length, will be used whenever we compute a hash of which we need to produce a proof. This is because Poseidon is purposefully designed to work with modular arithmetic.

2.4 Encryption schemes

A symmetric encryption scheme [11] based on Poseidon is also used, as described below.

Poseidon is built by applying the sponge construction [1] to a permutation $\pi : \mathbb{F}_p^t \to \mathbb{F}_p^t$, for $t = r + c$, where

- r is the *rate*, i.e. the amount of $\mathbb{F}_p$-elements of the input that can be processed in a call to π.
- c is the *capacity*, which is a part of the permutation that is never output by the hash, and is required for security.

The permutation π is composed of linear (matrix multiplication over $\mathbb{F}_p$) and non-linear (S-boxes) operations. Some rounds are *full rounds*, and apply S-boxes to the whole input, and others are *partial rounds*, in which an S-box is applied to a single $\mathbb{F}_p$-element.

In this case, Citadel uses Poseidon-128 to target 128-bit security. Following the recommendations of [8], parameters are set as $r = 4$ and $c = 1$, so that a hash in the Merkle tree can be computed with a single call to the permutation. Internally, a permutation performs $R_F = 8$ full rounds and $R_P = 59$ partial rounds, and uses $S(x) = x^5$ as the S-box.

When no proofs involving hashes are required, the BLAKE2 family of hash functions [15] is used, which yields better efficiency. In particular, the BLAKE2b hash function is used, since it is optimized for 64-bit platforms.

The encryption scheme [11] is a variation of the one-time pad encryption in the field. It uses Poseidon as a pseudorandom function to extend an agreed-upon symmetric key and encrypt the message. Therefore, the encryption scheme is perfectly secure under the random oracle model.

Concretely, the encryption works as follows. Given a message in $\mathbb{F}_q^\ell$, each $\mathbb{F}_q$-component is added to the corresponding component of the key. The key is obtained using Poseidon by extending the symmetric-encryption key, which is an elliptic curve point, to obtain a key with the same size of the message. The initialization vector contains the two coordinates of the key and a nonce, it is passed to the Poseidon iteration which extends the key and outputs the ciphertext. The sender sends the encryption along with the nonce and the information needed to compute the key, so the receiver can use Poseidon with the same key and nonce to decrypt the message.

2.5 Commitments

As commitment scheme, Citadel uses the Pedersen commitment [13], which we now describe.

- *Setup.* Sample and output the commitment key $\mathsf{ck} = (G, G') \leftarrow \mathbb{J}^2$.
- *Commit.* On input a value v, sample randomness $r \leftarrow \mathbb{F}_t$ and output
$$c = \mathsf{Com}_{\mathsf{ck}}(v; r) = vG + rG'.$$
- *Open.* Reveal v, r. With these, anyone can recompute the commitment and check if it matches c.

This scheme is perfectly hiding, and computationally binding under the discrete logarithm assumption.

2.6 Proof systems

Citadel uses the zk-SNARK PlonK [7] as its proof system. PlonK allows anyone to prove satisfiability of any arithmetic circuit modulo a prime. Since arithmetic circuit satisfiability is an NP-complete problem, this proof system will allow us to prove any statement in NP. PlonK makes use of the KZG polynomial commitment scheme [9], as described in [7]. This requires instantiating PlonK over a pairing-friendly group, which is described in Section 2.1.

Below is a summary the efficiency of PlonK, for a circuit with n multiplication gates and ℓ public inputs.

- *Proving time:* $O(n)$ group and field operations.
- *Verification time:* $O(1 + \ell)$ group and field operations.
- *Proof size:* $O(1)$ group and field elements.

PlonK is sound in the algebraic group model [6], and statistically zero-knowledge. A complete and explicit description of the scheme can be found in [7, Section 8].

2.7 Merkle trees

A *Merkle tree* [12] is a tree that contains at every vertex the hash of its children vertices. More precisely, we consider a perfect k-ary tree of height h. The single vertex at level 0 is called the *root* of the tree, and the k^h vertices at level h are called the *leaves*. Given a vertex in level i, the k vertices in level $i+1$ that are adjacent to it are called its *children*. Two vertices are each other's *sibling* if they are children of the same vertex.

To each vertex in the tree, we will recursively associate a value, starting from the leaves.[1] Let H be a hash function.

- Level h: leaves are initialized to a null value. Through the lifetime of the tree, they will progressively be filled from left to right with values.
- Level $0 \leq i < h$: each vertex has k children $c_1, \ldots, c_k$ at level $i+1$. We set the value of the vertex to $H(c_1, \ldots, c_k)$.

[1] We will often abuse notation and write the vertex to refer to the value associated with the vertex.

The tree is updated every time a new value is written into a leaf, by updating the $h+1$ elements in the path from the new value to the root. In particular, this means that the root changes after every update. A nice feature of Merkle trees is that, given a root r, it is easy to prove that a value x is in a leaf of a tree with root r. The proof works as follows:

- *Prove.* For $i = h, \ldots, 1$, let x_i be the vertex that is in level i and is in the unique path from x to the root. Let $y_{i,1}, \ldots, y_{i,k-1}$ be the $k-1$ siblings of x_i. Output

$$(x, (y_{1,1}, \ldots, y_{1,k-1}), \ldots, (y_{h,1}, \ldots, y_{h,k-1})).$$

- *Verify.* Parse input as $(x_h, (y_{1,1}, \ldots, y_{1,k-1}), \ldots, (y_{h,1}, \ldots, y_{h,k-1}))$, where x_h is the purported value and $y_{i,1}, \ldots, y_{i,k-1}$ are the purported siblings at level i. For $i = h-1, \ldots, 0$, compute[2]

$$x_i = H\left(x_{i+1}, y_{i+1,1}, \ldots, y_{i+1,k-1}\right).$$

This allows for proving membership in a set of size k^h by sending $O(kh)$ values. This proof is sound provided that the hash function is collision resistant [10, Section 5.6.2]. For our application, we will set $k = 4$ and $h = 17$.

3 Protocol

3.1 Keys

Let $G, G' \leftarrow \mathbb{J}$ be two generators for the subgroup $\mathbb{J}$ of order t of the Jubjub elliptic curve. In Citadel, each party involved in the protocol holds a pair of static keys with the following structure:

- *Secret key:* $\mathsf{sk} = (a, b)$, where $a, b \leftarrow \mathbb{F}_t$.
- *Public key:* $\mathsf{pk} = (A, B)$, where $A = aG$ and $B = bG$.

We use the subindices $\mathsf{user}, \mathsf{SP}, \mathsf{LP}$ to indicate the owner of the keys, e.g. $\mathsf{pk}_{\mathsf{user}}$ denotes the public key of the user.

3.2 Protocol flow

In this section, we describe the workflow of Citadel in detail.

3.2.1 (user) request_license()

1. Compute a license stealth address $(\mathsf{lpk}, R_{\mathsf{lic}})$ belonging to the user, using the user's own public key, as follows.
   1.1. Sample r uniformly at random from $\mathbb{F}_t$.
   1.2. Compute a symmetric Diffie–Hellman key $\mathsf{k} = rA_{\mathsf{user}}$.
   1.3. Compute a one-time public key $\mathsf{lpk} = H^{\mathsf{BLAKE2b}}(\mathsf{k})G + B_{\mathsf{user}}$.
   1.4. Compute $R_{\mathsf{lic}} = rG$.
2. Compute the license secret key $\mathsf{lsk} = H^{\mathsf{BLAKE2b}}(\mathsf{k}) + b_{\mathsf{user}}$ and an additional key $\mathsf{k}_{\mathsf{lic}} = H^{\mathsf{Poseidon}}(\mathsf{lsk})G$.
3. Compute the request stealth address $(\mathsf{rpk}, R_{\mathsf{req}})$ using the LP's public key, as follows.
   3.1. Sample r uniformly at random from $\mathbb{F}_t$.
   3.2. Compute a symmetric Diffie–Hellman key $\mathsf{k}_{\mathsf{req}} = rA_{\mathsf{LP}}$.

[2] To be precise, the prover also has to send $\lceil \log_2 k \rceil$ bits for each level, specifying the position of x_i with respect to its siblings, so that the verifier knows in which order to arrange the inputs of the hash.

3.3. Compute a one-time public key $\mathsf{rpk} = H^{\mathsf{BLAKE2b}}(\mathsf{k_{req}})G + B_{\mathsf{LP}}$.

3.4. Compute $R_{\mathsf{req}} = rG$.

4. Encrypt data using the key $\mathsf{k_{req}}$: $\mathsf{enc} = \mathsf{Enc}_{\mathsf{k_{req}}}((\mathsf{lpk}, R_{\mathsf{lic}})||\mathsf{k_{lic}}; \mathsf{nonce})$.
5. Send the following request to the network: $\mathsf{req} = ((\mathsf{rpk}, R_{\mathsf{req}}), \mathsf{enc}, \mathsf{nonce})$.

3.2.2 (LP) get_license_request()

The LP checks continuously the network to detect any incoming license requests addressed to them:

1. Compute $\tilde{\mathsf{k}}_{\mathsf{req}} = a_{\mathsf{LP}} R_{\mathsf{req}}$.
2. Check if $\mathsf{rpk} \stackrel{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{req}})G + B_{\mathsf{LP}}$.
3. Decrypt enc using nonce and $\tilde{\mathsf{k}}_{\mathsf{req}}$: $\mathsf{Dec}_{\tilde{\mathsf{k}}_{\mathsf{req}}}((\mathsf{lpk}, R_{\mathsf{lic}})||\mathsf{k_{lic}}; \mathsf{nonce})$.

3.2.3 (LP) issue_license()

1. Upon receiving a request from a user, define a set of attributes associated to the license, collect them (e.g. by concatenation) in a variable attr_data, and compute a digital signature as follows:

$$\mathsf{sig_{lic}} = \mathsf{sign\_single\_key}_{\mathsf{sk_{LP}}}(\mathsf{lpk}, \mathsf{attr\_data}).$$

2. Encrypt the signature and the attributes using the license key:

$$\mathsf{enc} = \mathsf{Enc}_{\mathsf{k_{lic}}}(\mathsf{sig_{lic}}||\mathsf{attr\_data}; \mathsf{nonce}).$$

3. Send the following license to the network:

$$\mathsf{lic} = ((\mathsf{lpk}, R_{\mathsf{lic}}), \mathsf{enc}, \mathsf{nonce}, \mathsf{pos}).$$

3.2.4 (user) get_license()

In order to receive the license, the user must scan all incoming transactions the following way:

1. Compute $\tilde{\mathsf{k}}_{\mathsf{lic}} = H^{\mathsf{Poseidon}}(\mathsf{lsk})G$.
2. Check if $\mathsf{lpk} \stackrel{?}{=} H^{\mathsf{Poseidon}}(\tilde{\mathsf{k}}_{\mathsf{lic}})G + B_{\mathsf{user}}$.
3. Decrypt enc using nonce and $\tilde{\mathsf{k}}_{\mathsf{lic}}$: $\mathsf{Dec}_{\tilde{\mathsf{k}}_{\mathsf{lic}}}(\mathsf{sig_{lic}}||\mathsf{attr\_data}; \mathsf{nonce})$.

3.2.5 (user) use_license()

When willing to use a license, the user must open a session with an specific SP by executing a call to the license contract. The user performs the following steps:

1. Create a zero-knowledge proof π using the gadget depicted in Figure 2.
2. Issue a transaction that calls the license contract, which includesπ. Notice that here, the user signs session_hash using lsk. Likewise, the user here will need to compute $\mathsf{lpk}' = \mathsf{lsk}G'$.
3. The network validators execute the license smart contract, which verifies π. Upon success, the following session will be added to a shared list of sessions:

$$\mathsf{session} = \{\mathsf{session\_hash}, \mathsf{session\_id}, \mathsf{com}_0^{hash}, \mathsf{com}_1, \mathsf{com}_2\},$$

where $\mathsf{session\_hash} = H^{\mathsf{Poseidon}}(\mathsf{pk_{SP}}||r_{\mathsf{session}})$, and r_{session} is sampled uniformly at random from $\mathbb{F}_t$.

3.2.6 (user) request_service()

To request a service to the SP, the user establishes communication with it using a secure channel, and provides the session cookie that follows:

$$\mathsf{sc} = \{\mathsf{pk_{SP}}, r_{\mathsf{session}}, \mathsf{session\_id}, \mathsf{pk_{LP}}, \mathsf{attr\_data}, c, \mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2\}$$

3.2.7 (SP) get_session()

Receive a session from the list of sessions, where session.session_id = sc.session_id.

3.2.8 (SP) grant_service()

Grant or deny the service upon verification of the following steps:

1. Check whether the values $(\mathsf{attr\_data}, \mathsf{pk_{LP}}, c)$ included in the sc are correct.
2. Check whether the opening $(\mathsf{pk_{SP}}, r_{\mathsf{session}})$ included in the sc matches the session_hash found in the session.
3. Check whether the openings $((\mathsf{pk_{LP}}, \mathsf{s}_0), (\mathsf{attr\_data}, \mathsf{s}_1), (c, \mathsf{s}_2))$ included in the sc match the commitments $(\mathsf{com}_0^{hash}, \mathsf{com}_1, \mathsf{com}_2)$ found in the session.

Furthermore, the SP might want to prevent the user from using the license more than once (e.g. this is a single-use license, like entering a concert). This is done through the computation of session_id. The deployment of this part of the circuit has two different possibilities:

- By setting $c = 0$ (or directly remove this input from the circuit), the license can be used only once.
- If the SP requests the user to set a custom value for c (e.g. the date of an event), the license can be reused only under certain conditions.

3.3 Circuits

3.3.1 License circuit

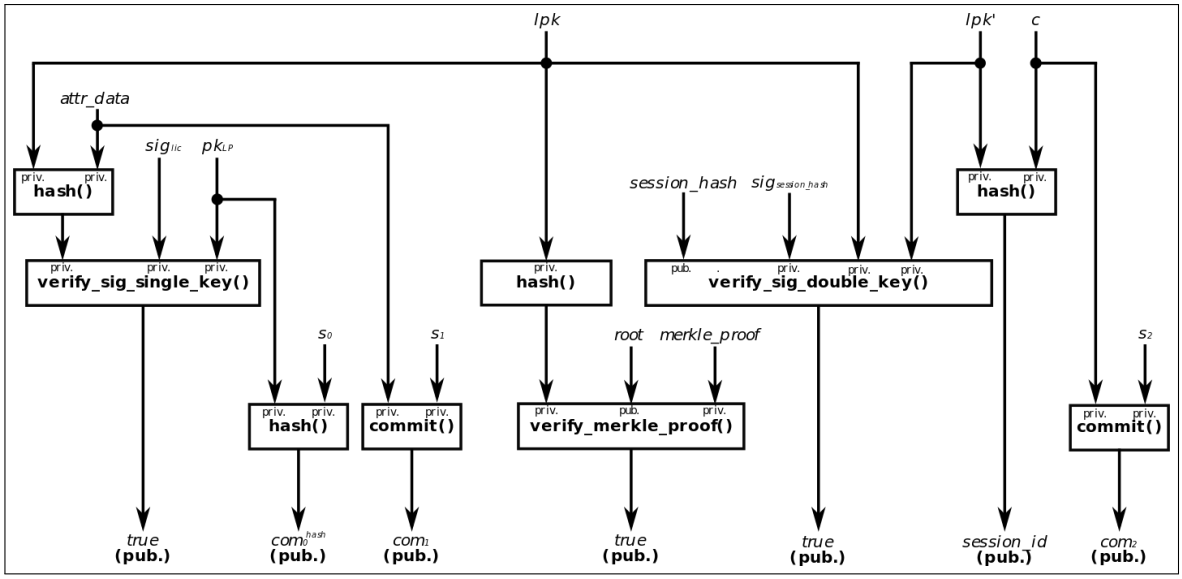

Figure 2: Arithmetic circuit for proving a license's ownership.

4 Implementation details

4.1 Elements structure

Here we describe the elements involved in Citadel. How they are used in the protocol is described in Section 3.

- *Request*: the structure of a request includes the encryption of a stealth address belonging to the user and where the license has to be sent to, and a symmetric key shared between the user and the LP.

Element	Type	Description
$(\mathsf{rpk}, R_{\mathsf{req}})$	StealthAddress	Stealth address for the LP.
enc	PoseidonCipher[6]	Encryption of a symmetric keys and of user's stealth address where the license has to be sent to.
nonce	BlsScalar	Randomness needed to compute enc.

- *License*: asset that represents the right of a user to use a given service. A license has the following structure:

Element	Type	Description
$(\mathsf{lpk}, R_{\mathsf{lic}})$	StealthAddress	License stealth address of the user.
enc	PoseidonCipher[4]	Encryption of the data of some user attributes and signature of these data.
nonce	BlsScalar	Randomness needed to compute enc.
pos	BlsScalar	Position of the license in the Merkle tree of licenses.

- *SessionCookie:* a session cookie is a secret value only known to the user and the SP. It contains a set of openings to a given set of commitments. The structure is as follows:

Element	Type	Description
$\mathsf{pk}_{\mathsf{SP}}$	JubJubAffine	Public key of the SP.
r_{session}	BlsScalar	Randomness for computing the session hash.
session_id	BlsScalar	ID of a session opened using a license.
$\mathsf{pk}_{\mathsf{LP}}$	JubJubAffine	Public key of the LP.
attr_data	JubJubScalar	Specific data concerning the attributes of the user.
c	JubJubScalar	Challenge value.
s_0	JubJubScalar	Randomness used to compute com_0^{hash}.
s_1	BlsScalar	Randomness used to compute com_1.
s_2	BlsScalar	Randomness used to compute com_2.

- *Session:* a session is a public struct known by all the validators. The structure is as follows:

Element	Type	Description
session_hash	BlsScalar	Hash of the SP's public key together with r_{session}.
session_id	BlsScalar	ID of a session opened using a given license.
com_0^{hash}	BlsScalar	Hash of the public key of the LP with s_0.
com_1	JubJubExtended	Pedersen commitment of the attributes data using s_1.
com_2	JubJubExtended	Pedersen commitment of the c value using s_2.

- *LicenseProverParameters:* a prover needs some auxiliary parameters to compute the proof that proves the ownership of a license. Some of the items of this table are related to the session and session cookie elements. The structure is as follows:

Element	Type	Description
lpk	JubJubAffine	License public key of the user.
lpk'	JubJubAffine	A variation of the license public key of the user computed with a different generator.
$\mathsf{sig}_{\mathsf{lic}}$	Signature	Signature of the license attributes data.
com_0^{hash}	BlsScalar	Hash of the public key of the LP with s_0.
com_1	JubJubExtended	Pedersen commitment of the attributes data using s_1.
com_2	JubJubExtended	Pedersen commitment of the c value using s_2.
session_hash	BlsScalar	Hash of the SP's public key together with r_{session}.
sig_session_hash	dusk_schnorr::Proof	Signature of the session hash signed by the user.
merkle_proof	PoseidonBranch	Membership proof of the license in the Merkle tree of licenses.

4.2 Application layer

In the previous subsection, we explained how a user sends a ZKP onchain to use a license. In this process, the network validates that an unknown license has been used, and a session is opened. When the user communicates offchain with the SP, they provide a session cookie to verify that the session is opened onchain and the arguments are correct. One of these arguments, the attribute data attr_data, is what defines the license (e.g., a ticket token, a set of personal information...), and this data is leaked to the SP. However, some use cases could require attribute data to be verified according to some conditions, for instance, leaking the information only partially. We now introduce a scheme to perform several attribute verifications offchain.

In our scheme, each SP decides which requirements the users need to meet, and provides a circuit that performs such checks. Then, when the user wants to use a service, will provide the session cookie as explained in the generic protocol, with the difference that **shall not include** the opening to com_1. Instead, will provide a ZKP computed out of the circuit required by the SP. For this to work, an agreement between the different involved parties is needed, i.e. both LPs and SPs will need to agree on the language (or encoding) used to create the attributes of the license. In such regard, the value attr_data used in the license becomes the hash of some specific attributes, as follows:

$$\mathsf{attr\_data} = H^{\mathsf{Poseidon}}(\mathsf{attr}_0, \mathsf{attr}_1, ..., \mathsf{attr}_N, r_{\mathsf{attr}}),$$

where r_{attr} has to be a random value known by the user and the LP. For instance, the public key stored in their ID card.

The SP will accept the service if the user provides a valid session cookie and a valid proof out of the following sample circuit, where the value com_1 included in the public inputs must be equal to the value $\mathsf{session.com}_1$:

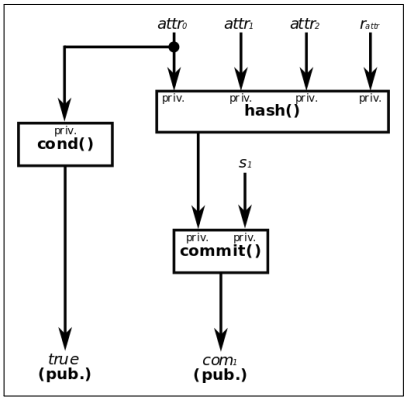

Figure 3: Arithmetic circuit for proving attributes off-chain.

The above circuit from Figure 3 can include as many conditions as desired for the attributes.

4.3 Participants interaction

Protocol implementation involves realization of the protocol building blocks as well as providing means of data communication between them. Building blocks are placed at the following locations, which correspond to protocol participants:

- User software.
- License Provider software.
- Service Provider software.
- License contract.

Data communication between protocol participants is realized in the following modes:

- Calling a contract state changing method.
- Calling a contract state querying method (not modifying the contract state).
- Storing data directly in blockchain.
- Retrieving data from blockchain.
- Delegating ZK proof calculation.
- Calling off-chain.

The following diagram in Figure 4 illustrates the interaction between protocol participants and indicates the communication means used.

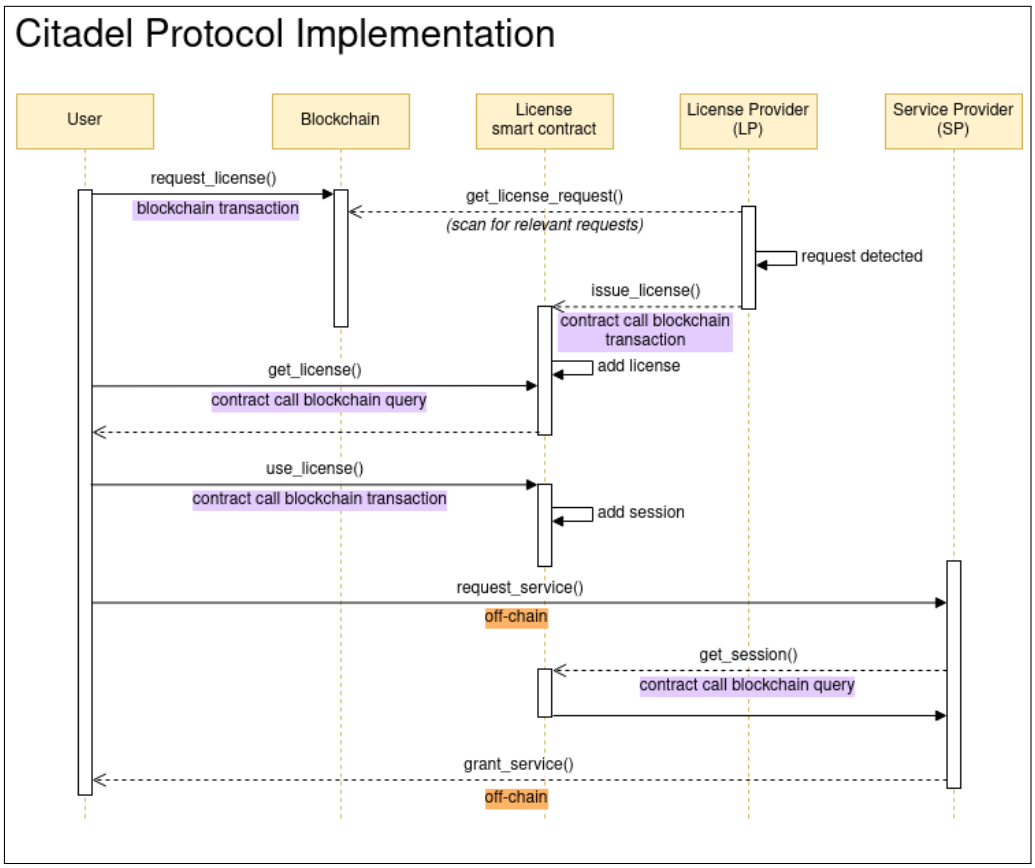

Figure 4: Interaction between protocol participants.

On the diagram, we can see various communication modes being used. Initially, the user submits to the blockchain a transaction which contains request as a payload. Subsequently, the License Provider, which continuously scans the blockchain, detects transactions containing requests and filters out requests which are addressed to it. The License Provider can obtain requests via other routes as well, for example via http or email, passing requests on the blockchain is only one of many possible ways of submitting a request, one that has the advantage of passing a payment along with the request. Once the License Provider gets a hold of a request, it can perform appropriate verification, and upon successful verification, it can issue a license. Issuing a license involves a smart contract transaction call. Smart contract transaction call is a blockchain transaction, yet to not confuse the reader, we show it on the diagram with details of blockchain involvement omitted. The user obtains licenses via a contract query. For privacy reasons, the user obtains a bulk of licenses not pertaining exclusively to her and filters it out by herself. To economize the volume of data transfers, block-height range is passed to allow for tranfer of a subset of available records. All modes of communication used so far were on-chain. Communication between the user and the Service Provider, on the other hand, is off-chain. When the Service Provider wants to establish a session, it calls a contract to obtain a session for the given session id.

The diagram illustrates the following flow of data and interactions between participants:

- User submits request to a License Provider by issuing a blockchain transaction.
- License Provider scans the blockchain and obtains the request.
- License Provider, upon necessary verification, issues a license.
- License Provider sends license to the License Contract via a smart contract call transaction.
- User obtains licenses for a given block-height range.
- User filters out licenses addressed to her/him.
- User calculates a proof (the proof calculation might be delegated).
- User calls *use-license* to redeem a license, via a smart contract call.
- License Contract attempts to verify the proof and, if verified, adds a new session to a list of sessions.
- User requests a service from a Service Provider (off-chain).
- Service Provider asks contract for a session.
- Service Provider grants service to the user (off-chain).

4.4 License contract

License contract maintains state consisting of the following data:

- List of sessions.
- Map of licenses and their positions in the Merkle tree.
- Merkle tree of license hashes.

Contract provides the following methods:

- *issue-license*: adds a license to a Merkle tree of licenses.
- *get-licenses*: provides a list of new licenses added in a given block-height range.
- *use-license*: attempts to verify the proof and, if verified, adds a new session to a list of sessions and nullifies the license in the Merkle tree.
- *get-session*: finds a session in a list of sessions and returns it to the caller.

4.5 License ZK proof calculation

License ZK proof calculation will either be performed by the user or delegated to a node. At the time of writing, the details of delegation are not yet known, they will be filled in here once the information becomes available.

References

[1] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponges (2011) 5

[2] Bowe, S., Grigg, J.: Implementation of the BLS12-381 pairing-friendly elliptic curve construction Available online: https://github.com/zkcrypto/bls12_381 (accessed on 1 June 2022) 4

[3] Bowe, S., Ogilvie-Wigley, E., , Grigg, J.: Implementation of the Jubjub elliptic curve group Available online: https://github.com/zkcrypto/jubjub (accessed on 1 June 2022) 4

[4] Dusk: The Phoenix transaction model (2023), https://github.com/dusk-network/phoenix-core/blob/master/docs/protocol-description.pdf 5

[5] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986) 4

[6] Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Annual International Cryptology Conference. pp. 33–62. Springer (2018) 6

[7] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PlonK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019) 6

[8] Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: USENIX Security Symposium. vol. 2021 (2021) 5

[9] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: International conference on the theory and application of cryptology and information security. pp. 177–194. Springer (2010) 6

[10] Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC press (2020) 5, 7

[11] Khovratovich, D.: Encryption with Poseidon Available online: https://dusk.network/uploads/Encryption-with-Poseidon.pdf (accessed on 1 June 2022) 5, 6

[12] Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987) 6

[13] Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991) 6

[14] Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: International conference on the theory and applications of cryptographic techniques. pp. 387–398. Springer (1996) 4

[15] Saarinen, M.J.O., Aumasson, J.P.: The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693 (Nov 2015), available online: https://www.rfc-editor.org/info/rfc7693 (accessed on 1 June 2022) 5

[16] Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Conference on the Theory and Application of Cryptology. pp. 239–252. Springer (1989) 4