Citadel protocol specification

Dusk Network

November 5, 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- *Proof of validity*: users with a valid license can prove that their license has not been revoked and is valid.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.
- The *LicenseProverParameters* are the set of parameters needed by the user to compute a proof that proves their license ownership.
- 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

[Missing explanation]

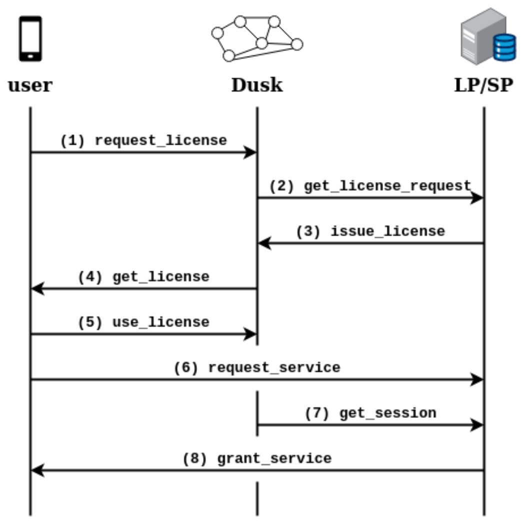

Figure 1: Overview of the protocol messages exchanged between the user, Dusk's network, and the LP/SP.

2 Cryptographic primitives

[TODO]

2.1 Elliptic curves

Marta: Include details of Jubjub group, generators, etc.

2.2 Digital signatures

2.3 Encryption

2.4 Commitments

2.5 Proof systems

2.6 Hash functions

Marta: Hashing - it is going to be Poseidon everywhere.

2.7 Merkle trees

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. $\mathsf{pk}_{\mathsf{user}}$.

3.2 Protocol flow

Marta: Change BLAKE2 to Poseidon and leave a footnote.

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

3.2.1 (user) $\mathsf{request\_license}()$

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 $\mathsf{k}_{\mathsf{req}} = rA_{\mathsf{LP}}$.
   3.3. Compute a one-time public key $\mathsf{rpk} = H^{\mathsf{BLAKE2b}}(\mathsf{k}_{\mathsf{req}})G + B_{\mathsf{LP}}$.
   3.4. Compute $R_{\mathsf{req}} = rG$.
4. Encrypt data using the key $\mathsf{k}_{\mathsf{req}}$: $\mathsf{enc} = \mathsf{Enc}_{\mathsf{k}_{\mathsf{req}}}((\mathsf{lpk}, R_{\mathsf{lic}})||\mathsf{k}_{\mathsf{lic}}; \mathsf{nonce})$.

   Marta: Include how the nonce is computed, if it is a random value as well.

5. Send the following request to the network: $\mathsf{req} = ((\mathsf{rpk}, R_{\mathsf{req}}), \mathsf{enc}, \mathsf{nonce})$.

3.2.2 (LP) $\mathsf{get\_license\_request}()$

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 $\tilde{\mathsf{k}}_{\mathsf{req}}$: $\mathsf{Dec}_{\tilde{\mathsf{k}}_{\mathsf{req}}}((\mathsf{lpk}, R_{\mathsf{lic}})||\mathsf{k}_{\mathsf{lic}}; \mathsf{nonce})$.

3.2.3 (LP) $\mathsf{issue\_license}()$

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 $\mathsf{attr\_data}$, and compute a digital signature as follows:

$$\mathsf{sig}_{\mathsf{lic}} = \mathsf{sign\_single\_key}_{\mathsf{sk}_{\mathsf{SP}}}(\mathsf{lpk}, \mathsf{attr\_data}).$$

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

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

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) $\mathsf{get\_license}()$

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

1. Compute $\tilde{\mathsf{k}}_{\mathsf{lic}} = H^{\mathsf{BLAKE2b}}(\mathsf{lsk})G$.
2. Check if $\mathsf{lpk} \overset{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{lic}})G + B_{\mathsf{user}}$.
3. Decrypt enc using nonce and $\tilde{\mathsf{k}}_{\mathsf{lic}}$: $\mathsf{Dec}_{\tilde{\mathsf{k}}_{\mathsf{lic}}}(\mathsf{sig}_{\mathsf{lic}}||\mathsf{attr\_data}; \mathsf{nonce})$.

3.2.5 (user) $\mathsf{use\_license}()$

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:

Marta: We should mention something about the *license contract* before this step. Rewrite Section 2 and add a small section about Dusk's blockchain (instead of leaving these details for last section)?

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 $\mathsf{session\_hash}$ using lsk. Likewise, the user here will need to compute $\mathsf{lpk}' = \mathsf{lsk}G'$.

   Marta: I think we should include the exact elements that are included in the transaction.

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

   Marta: It is not clear to me who samples r_{session}. Is it the smart contract? Can a smart contract do this?

3.2.6 (user) $\mathsf{request\_service}()$

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

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

Marta: Notation-wise: the acronym sc can be confused with the common abbreviation for `smart contract (sc)`, maybe use a different acronym?

3.2.7 (SP) $\mathsf{get\_session}()$

Receive a $\mathsf{session}$ from the list of sessions, where $\mathsf{session.session\_id} = \mathsf{sc.session\_id}$.

Marta: The action *receive* is not clear - Who sends this $\mathsf{session}$? Must the SP scan the network? Is it the user?

3.2.8 (SP) grant_service()

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

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

Marta: The following paragraph needs a bit more elaboration on the different use cases. Maybe discuss this in the first section of the document?

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

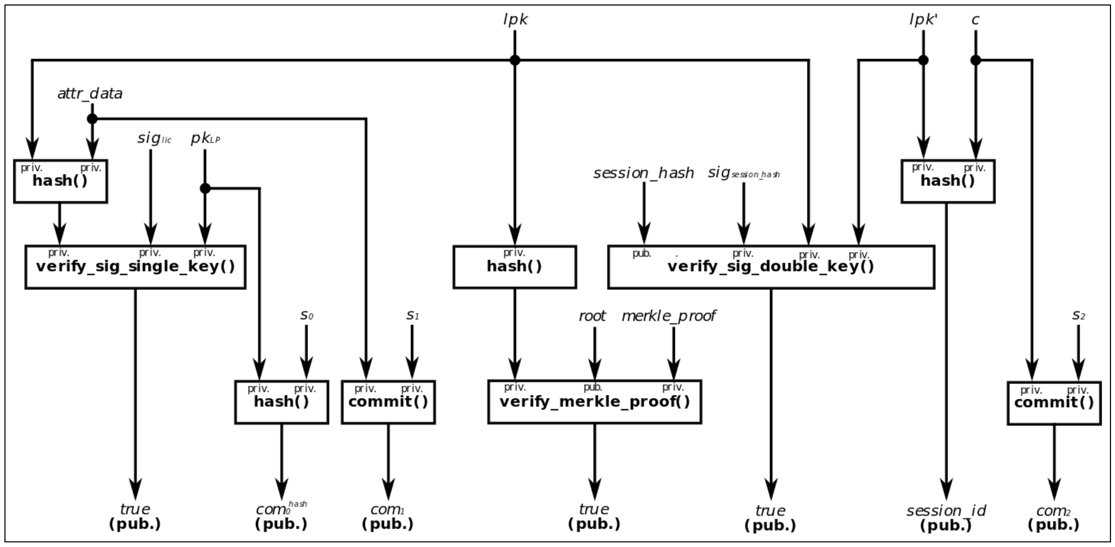

Figure 2: Arithmetic circuit for proving a license's ownership.

4 Implementation details

Marta: Add Milosz figure here or in Section 3. Protocol?

4.1 Elements structure

Marta: **Should this section be moved to 4. Implementation details?**

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:

Marta: The session cookie is not a secret value, it is an struct. Does it refer to `session_id`?

Marta: Clarify what the element `attr` is - is it a hash, an array?

Element	Type	Description
$\mathsf{pk}_{\mathsf{SP}}$	JubJubAffine	Public key of the SP.
r_{session}	BlsScalar	Randomness for computing the session hash.
$\mathsf{session\_id}$	BlsScalar	ID of a session opened using a license.
$\mathsf{pk}_{\mathsf{LP}}$	JubJubAffine	Public key of the LP.
$\mathsf{attr\_data}$	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:

Marta: What does *validators* mean?

Element	Type	Description
$\mathsf{session\_hash}$	BlsScalar	Hash of the SP's public key together with r_{session}.
$\mathsf{session\_id}$	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.

Marta: Add a section including the software that it is assumed each participant uses? For example, user makes use of wallet and does blockchain calls and queries. The blockchain stores a license contract that can be called blabla. The LP and SP software, etc. (see previous section 4.1 from Milosz).

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 $\mathsf{session.com}_1$:

The above circuit 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.

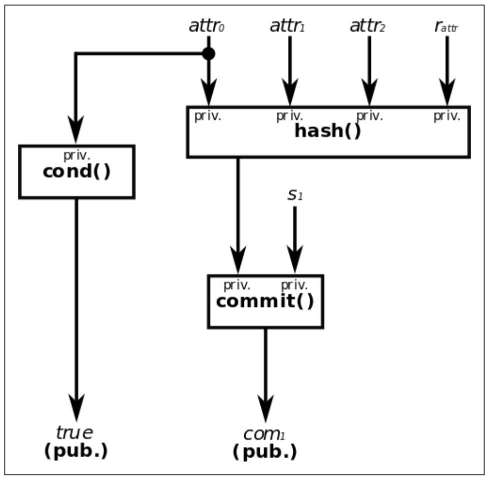

Figure 3: Arithmetic circuit for proving attributes off-chain.

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 illustrates an interaction between protocol participants and indicates the communication means used.

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.

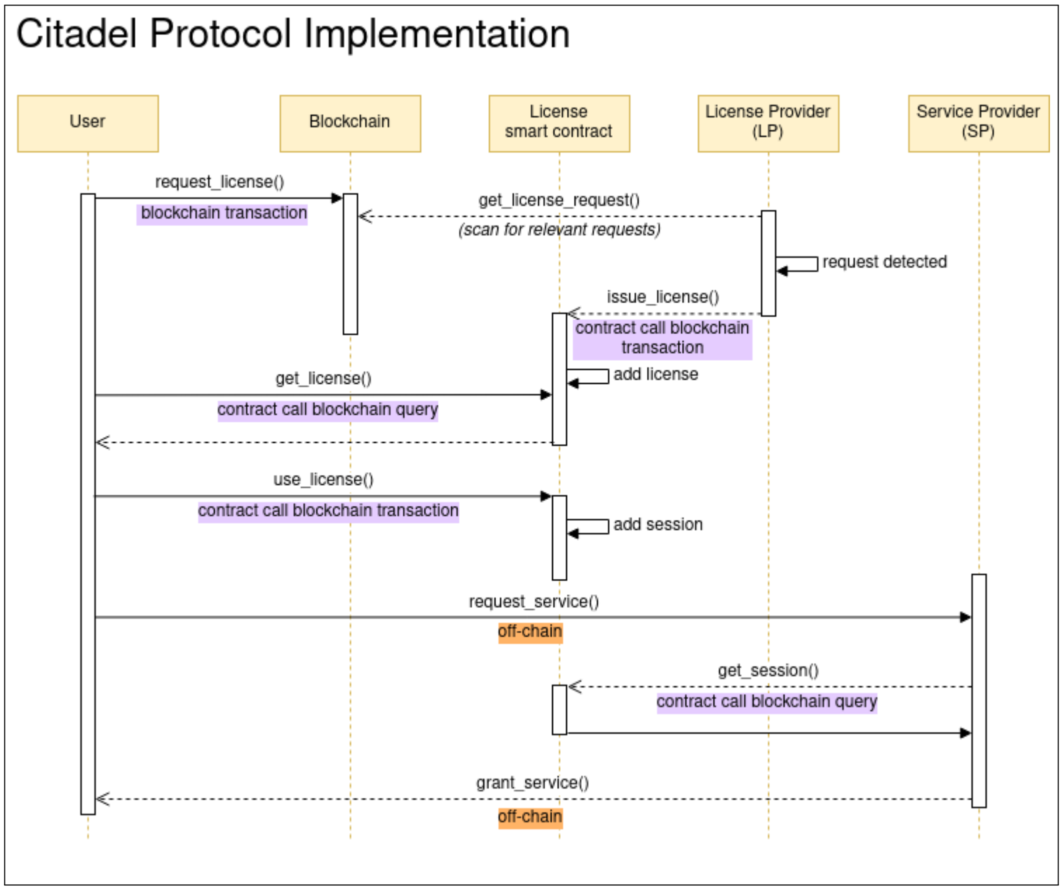

Figure 4: Interaction between protocol participants.

- 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] Benarroch, D., Campanelli, M., Fiore, D., Gurkan, K., Kolonelos, D.: Zero-knowledge proofs for set membership: efficient, succinct, modular. In: Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I. pp. 393–414. Springer (2021)

[2] Catalano, D., Fiore, D.: Vector commitments and their applications. In: Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16. pp. 55–72. Springer (2013)

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

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

[6] Palatinus, M., Rusnak, P., Voisine, A., Bowe, S.: BIP-39: Mnemonic code for generating deterministic keys (2013), https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[7] Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Advances in Cryptology—CRYPTO'89 Proceedings 9. pp. 239–252. Springer (1990)