Monetha private data exchange protocol
Design concept
Version 0.4

Sebastian Faust, Sasan Safai, Marius van der Wijden, Sebastian Stammler

Contributors: Dmitrij Koniajev, Viaceslavas Ruckis, Andrej Ruckij

1 Introduction

Monetha introduces a reputation engine which allows participants of a network to judge trustworthiness of another party by providing public and private information in a secure manner. The network enables participants to exchange it via Passports. The data stored in the passport should also be editable by the owner or by a trusted third party – the so-called Facts Provider. By editable we mean that the ”append-only” structure of the blockchain will be used. The passport owner can retrieve the confidential data to a requestor, who can be convinced by interaction with the passport contract stored on the blockchain whether this data is authentic, i.e., it was authenticated by the data provider and belongs to the corresponding passport owner.

2 Design Requirements

- R1: The proposed solution shall store information fully decentralized.
- R2: The cost for storage shall be minimal.
- R3: The time to retrieve data shall be within a reasonable timespan and the computational overheads on both the data provider and the requestor shall be minimal.
- R4: The proposed solution shall satisfy the highest possible security standards, ideally backed up by formal security proofs. Moreover, it shall use state-of-the art cryptographic algorithms to guarantee security against attacks that are relevant according to the proposed trust model.
- R5: The proposed technical solution allows for a user friendly integration i.e. into the Monetha-App. Also it should make sure that the protocoll is not disturbing the user by asking for too many confirmations.
- R6: In case the solution relies on frameworks or other software, its maturity should allow a production feasible implementation.

Actors/Roles/Entities of the system For phase 1 we consider the following main actors that are involved in the system:

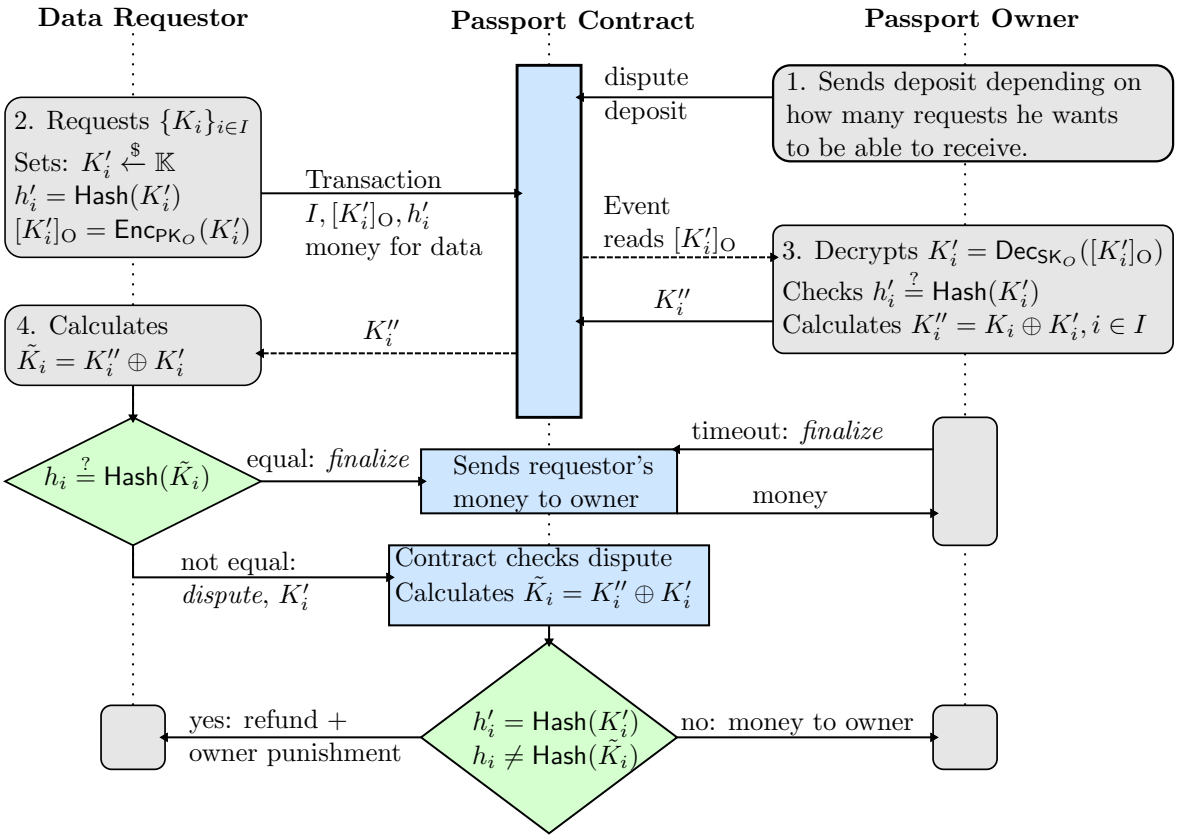
- Passport Owner (Primary Actor): he shall maintain a passport on the decentralized storage. He can request adding new data to his passport or can reveal private data to a requestor. He also may receive money for revealing certain protected data of his passport. Deleting Data on a blockchain is technically not possible and also for the moment not needed by an actor.
- Facts Provider (Primary Actor): he shall add new data to the passport of a user. Adding new data to the passport might be done without involvement of the user. This is needed to also allow for non-beneficial data to be added to the passport of a user (e.g., data that reduces one's reputation).
- Requestor (Primary Actor): he can request information from the passport of users. This is achieved by running an interactive protocol between the passport owner and the requestor.
- Passport Application (Secondary Actor): It is the logic to manage the passport on the blockchain. Managing permissions to access data and displaying history of transactions is part of its functionality. Passport owner is interacting with his passport via this application.
- Data storage (Secondary Actor): it is used to store and maintain the passport data (IPFS or blockchain). The data storage is involved in the protocol for adding new data to the passport and in a second protocol running between the requestor and the passport owner to reveal confidential data from the passport to the requestor.

User Stories

- S1.1: A passport owner can create a passport on the data storage which contains public and private data.
- S1.2: It is guaranteed that the data shown to the requestor corresponds to the protected data stored on the data storage.
- S2 The data provider should be able to write protected data to the passport, whenever this is desired and allowed by the passport owner.
- S3.1: A requestor should be able to retrieve public data.
- S3.2: A requestor should be able to initiate retrieval of private data from a passport when granted by its owner.
- S4: The data of a passport owner should either be stored directly on the blockchain or on an underlying decentralized storage network like IPFS.

3 Protocol

The following figure shows the full protocol. It currently uses one-way hash functions as the commitment scheme, which are only computationally binding and hiding (unlike the commonly-used Pedersen commitments) but readily and cheaply available in Ethereum smart contracts.



Storing data into Passport

The following protocol describes the publishing procedure of a single data item data by the data provider. $\mathbb{K}$ denotes the symmetric encryption's key space, i.e., $\mathbb{K} = \mathbb{Z}_{2^l}$ for keys of bitlength l.

1. The passport owner publishes his public key PK_O on the passport. This may be implicit if the passport explicitly references its owner.
2. A data provider holds data data about the owner which she wishes to publish on the passport.
   (a) She randomly generates a symmetric encryption key $K \xleftarrow{\$} \mathbb{K}$.
   (b) She symmetrically encrypts data with K to get $[\mathsf{data}]_{\mathrm{K}} = \mathsf{Enc}^{\mathrm{sym}}_K(\mathsf{data})$.
   (c) She commits to the symmetric key by calculating its hash $h = \mathsf{Hash}(K)$
   (d) She encrypts the symmetric key with the passport owner's public key to get $[K]_{\mathrm{O}} = \mathsf{Enc}_{\mathsf{PK}_O}(K)$
3. The data provider publishes the just calculated values $[\mathsf{data}]_{\mathrm{K}}, h$ and $[K]_{\mathrm{O}}$ on the passport contract.

4. The owner can now reconstruct the symmetric key using his secret key SK_O as $K = \mathsf{Dec}_{\mathsf{SK}_O}([K]_\mathrm{O})$; then checks the data by decrypting it as $\mathsf{data} = \mathsf{Dec}_K([\mathsf{data}]_\mathrm{K})$. He also checks that the commitment h matches K.

In the later fair data exchange protocol, the encrypted data $[\mathsf{data}\,]_\mathrm{K}$ is not used for disputes and hence the data can also be stored outside of the contract, e.g., in IPFS or at a cloud provider (who would not be able to read the data since it is encrypted).

Requesting Passport data

The following protocol describes the data request and receive procedure between the owner and a data requestor.

To prevent a malicious owner from triggering a lot of requests in order to deplete the requestor's gas resources, as a precautionary measure the passport should only accept one request opening per requestor. Only after a request is either finished or times out, should a new request be possible to open. For that reason, we also propose to implement the possibility to request a list of data points, that is, a list of keys, at once, because this is much faster and cheaper than serial requests.

In the following we assume that the passport holds a list of keys K_i with hashes h_i from which a requestor wants to request a subset $\{K_i\}_{i\in I}$.

1. To make $\{\mathsf{data}_i\}_{i\in I}$ requestable, the data owner posts a deposit to the contract so that successful dispute requests can be reimbursed to requestors.

2. A data requestor generates $|I|$ random keys $K'_i \xleftarrow{\$} \mathbb{K}$, hashes them ($h'_i = \mathsf{Hash}(K'_i)$) and encrypts them with the owner's public key ($[K'_i]_\mathrm{O} = \mathsf{Enc}_{\mathsf{PK}_O}(K'_i)$). She posts $h'_i, [K'_i]_\mathrm{O}$, the list I, and the money for the data to the contract to open the request. If the owner doesn't reply within a timeout period, she may refund herself and prematurely close the data request.

3. The owner decrypts the request keys $K'_i = \mathsf{Dec}_{\mathsf{SK}_O}([K'_i]_\mathrm{O})$ and checks that they hash to h'_i. He now accepts the request by calculating keys $K''_i = K_i \oplus K'_i, i \in I$ and posting them to the passport. Otherwise, he just ignores the request.

4. The requestor calculates the keys $\tilde{K}_i = K'_i \oplus K''_i$ and checks for all i that they hash to $h_i \stackrel{?}{=} \mathsf{Hash}(\tilde{K}_i)$.

5. The protocol finishes depending on the outcome of this check:

   Check failed (*dispute*): She sends the K'_i to the passport contract, which can easily verify the failed checks by simple XOR and hashing operations. The requestor is refunded, including some of the owner's deposit. Note that she cannot send a false dispute, because only the keys K'_i hash to the public hashes h'_i to which the owner implicitly committed by replying with the keys K''_i.

   Check succeeds (*finalize*): She can decrypt the encrypted data $[\mathsf{data}]_\mathrm{K}$ and sends a finalization command to the contract, releasing all money in escrow to the owner. If she doesn't send the finalization request within a predefined timespan, the owner is allowed to finalize, preventing the escrow from being locked-up indefinitely.

Since in general a list of keys is requested, in a dispute situation it may happen that only some keys are disputable. Whether to punish the owner fully or only per wrong key is up to the implementation.