A Stateless and Secure Delivery versus Payment across two Blockchains

Christian Fries [1,2,*] **Peter Kohl-Landgraf**,[1,*]
[1] DZ BANK AG, Deutsche Zentral-Genossenschaftsbank, Frankfurt a. M., Germany
[2] Department of Mathematics, Ludwig Maximilians University, Munich, Germany
[*] Correspondence: email@christian-fries.de, pekola@mailbox.org

November 9th, 2023

(version 2.0 [v5])

Disclaimer

The views expressed in this work are the personal views of the presenters and do not necessarily reflect the views or policies of current or previous employers.

ABSTRACT

We propose a lean and functional transaction scheme to establish a secure delivery-versus-payment across two blockchains, where a) no intermediary is required and b) the operator of the payment chain/payment system has a small overhead and does not need to store state. The main idea comes with two requirements: First, the payment chain operator hosts a stateless decryption service that allows decrypting messages with his secret key. Second, a "Payment Contract" is deployed on the payment chain that implements a function

```
transferAndDecrypt(uint id, address from, address to,
    string keyEncryptedSuccess, string keyEncryptedFail)
```

that processes the (trigger-based) payment and emits the decrypted key depending on the success or failure of the transaction. The respective key can then trigger an associated transaction, e.g. claiming delivery by the buyer or re-claiming the locked asset by the seller.

CONTENTS

Introduction **4**

Comparison of the Present Method to other DvP Methods . . . 4

Acknowledgment . . . 4

Setup **5**

Notation . . . 5

Contract Interfaces **6**

`ILockingContract` . . . 6

`IDecryptionContract` . . . 6

Decryption Oracle and Key Format **8**

Key Format . . . 8

Stateless API for the Decryption Oracle . . . 9

Key Generation: `requestEncryptedHashedKey` . . . 10

Key Verification: `verify` . . . 10

Key Decryption: `decrypt` . . . 10

Rationale for DvP . . . 10

Workflow of a Secure Delivery-vs-Payment without External State **12**

Key Generation . . . 12

Buyer to `ILockingContract` . . . 12

Buyer's Cancellation Option . . . 12

Seller to `ILockingContract` . . . 12

Seller to `IDecryptionContract` . . . 13

Verification against the Decryption Oracle . . . 13

Seller's Cancellation Option . . . 13

Buyer to `IDecryptionContract` . . . 13

Completion of Transfer on `ILockingContract` . . . 13

Upon Success: . . . 13

Upon Failure: . . . 14

Communication between `IDecryptionContract` and Decryption Oracle . . . 14

Advantages **16**

Remarks **17**

Encryption versus Hashing 17

Key Generation 17

Pre-Trade versus Post-Trade 17

Conclusion **19**

References **19**

INTRODUCTION

Within the domain of financial transactions and distributed ledger technology (DLT), the Hash-Linked Contract (HLC) concept has been recognized as valuable and has been thoroughly investigated, see [1, 3]. The concept may help to solve the challenge of delivery-versus-payment (DvP), especially in cases where the asset chain and payment system (which may be a chain, too) are separated. The proposed solutions are based on an API-based interaction mechanism which bridges the communication between a so-called Asset Chain and a corresponding Payment System or require complex and problematic time-locks ([3]). We believe that an even more lightweight interaction across both systems is possible, especially when the payment system is also based on a DLT infrastructure.

The smart contracts proposed here are available as ERC 7573, [2].

Comparison of the Present Method to other DvP Methods

The following table gives a brief comparison of the present method and the most common DvP methods.

Criterion	Proposed Method	HTLCs	Centralized DvP	API-Based DvP
Intermediary Required?	No	No	Yes	Yes
Storage Requirements	No external storage	Requires storage for time-lock	Centralized storage required	External API requires storage
Timeout Required?	No	Yes	No	No
Coupling of Payment & Delivery	Decoupled, via encryption	Directly coupled via hash preimage	Coupled via intermediary	Coupled via API function
Flexibility in Execution	High (stateless, no fixed time)	Medium (timeout can be problematic)	Low (dependent on central system)	Medium (dependent on API availability)
On-Chain Costs	Low (no time-locks, stateless)	Medium (gas fees apply to time-locks)	No direct on-chain costs, but high service fees	Medium (depends on API fees)
Security Risks	Low (no central entity)	Medium (reorgs, time-lock attacks possible)	High (centralization introduces risks)	Medium (depends on API integrity)

Table 1: Comparison of DvP Methods

Acknowledgment

We like to thank Giuseppe Galano, Julius Lauterbach and Stephan Mögelin for their valuable feedback.

SETUP

Consider the setup of having two chains managing two different types of tokens. An example is a chain managing tokenized assets (the *Asset Chain*) and a chain that allows to trigger and verify payments (*Payment Chain*).

To facilitate a secure stateless delivery-versus-payment we introduce two interfaces:

- `ILockingContract`: a smart contract implementing this interface is able to lock the transfer of a token. The transfer can be completed by presentation of a success key (B), or reverted by presentation of a failure key (S). Presentation of B transfers the token to the buyer, presentation of S re-transfers the token to the seller.
- `IDecryptionContract`: a smart contract implementing this interface offers a transfer method that performs a conditional decryption of one of two encrypted keys ($E(B)$, $E(S)$), conditional on success or failure.

For the decryption, we propose a decryption oracle that offers a stateless service for generation and decryption of encrypted keys.

While the `IDecryptionContract`'s conditional transfer is prepared with encryptions $E(B)$, $E(S)$ of the keys B, S, the `ILockingContract`'s conditional transfer is prepared wth hashes utilizing a hashing $H(B)$, $H(S)$ of the keys B, S. This may be useful as hashing is a comparably cheap operation, while on-chain encryption may be costly.

To verify the consistency of the conditional transfers of the two contracts, the decryption-oracle offers a method that allows one to obtain $H(K)$ from a given $E(K)$ without exposing K.

If on-chain encryption is cheap, the hashes may be replaced with the encrypted keys, $H(K) = E(K)$, which slightly simplifies the protocol. We will describe the general case.

Notation

The method proposed here relies on a service that will decrypt a document observed in a message emitted by the smart contract implementing `IDecryptionContract` on the payment chain. In the following, we call these documents *key*, because they are used to unlock transactions (on the contact implementing `ILockingContract`).

We call the receiver of the tokens handled by the `ILockingContract` the *buyer* and the payer of tokens handled by the `ILockingContract` the *seller*. For the tokens handled by the `IDecryptionContract` the flow is reversed, but we still call the payer of the `IDecryptionContract`'s tokens the *buyer* and the receiver of the `IDecryptionContract`'s tokens the *seller*. This fits to the interpretation that the token on the `IDecryptionContract` are a payment for the transfer of the tokens on the `ILockingContract`.

There is a key for the buyer and a key for the seller. In Figure 1 and throughout, these keys are denoted by B (buyer, successful payment) and S (seller, failed payment), respectively. The encrypted keys are denoted by $E(B)$ and $E(S)$ and hashed of the keys are denoted by $H(B)$ and $H(S)$.

CONTRACT INTERFACES

ILockingContract

The interface `ILockingContract` is given by the following methods:

```
inceptTransfer(bytes32 id, int amount, address from, string memory keyHashedSeller, string memory keyEncryptedSeller);

confirmTransfer(bytes32 id, int amount, address to, string memory keyHashedBuyer, string memory keyEncryptedBuyer);

cancelTransfer(bytes32 id, int amount, address from, string memory keyHashedSeller, string memory keyEncryptedSeller);

transferWithKey(bytes32 id, string key);
```

A contract implementing this interface provides a transfer of tokens (usually resenting the delivery of an asset), where the tokens are temporarily locked. The completion or reversal of the transfer is then conditional on the presentation of one of the two keys.

- `inceptTransfer`: Called by the buyer of the token who's address is implicit (`to = msg.sender;`). Sets the hash of a key that will trigger re-transfer of the token to the seller (`from`).
- `confirmTransfer`: Called by the seller of the token who's address is implicit (`from = msg.sender ;`). Verifies that the seller's (`from`) and buyer's (`to`) addresses match the corresponding call (with the same (`id`)) to `inceptTransfer`. Sets the hash of a key that will trigger transfer of the token to the buyer (`to`).
- `transferWithKey`: Called by the buyer or seller of the token with a `key` whose hash matches the `keyHashedBuyer` or `keyHashedSeller` respectively (which ever key is released by the `IDecryptionCont`

IDecryptionContract

The interface `ILockingContract` is given by the following methods:

```
inceptTransfer(bytes32 id, int amount, address from, string memory keyEncryptedSuccess, string keyEncryptedFailure);

transferAndDecrypt(bytes32 id, int amount, address to, string memory keyEncryptedSuccess, string keyEncryptedFailure);

cancelAndDecrypt(bytes32 id, address from, string memory keyEncryptedSuccess, string memory keyEncryptedFailure);

releaseKey(bytes32 id, string memory key) external;
```

A contract implementing this interface provides a transfer of tokens (usually representing a payment), where a successful or failed transfer releases one of two keys, respectively.

- `inceptTransfer`: Called by the receiver of the (payment) token who's address is implicit (`to = msg.sender;`). Sets the encrypted keys that will be decrypted upon the success or failure of the transfer from the payer (`from`).

- `transferAndDecrypt`: Called by the payer of the (payment) token who's address is implicit (`from = msg.sender;`). Verifies that the payer's (`from`) and receiver's (`to`) addresses match the corresponding call (with the same (`id`)) to `inceptTransfer`. Tries to perform a transfer of the token. A successful transfer will emit the decryption of `keyEncryptedSuccess`, a failed transfer will emit the decryption of `keyEncryptedFailure`. Cannot be called if `cancelAndDecrypt` was called before.

  The decryption of the keys will (usually) handled by an external oracle, see below for a proposal of the corresponding functionality.

- `cancelAndDecrypt`: Called by the receiver of the (payment) token who's address is implicit (`to = msg.sender;`). Verifies that the payer's (`from`) and receiver's (`to`) addresses match the corresponding call (with the same (`id`)) to `inceptTransfer`. Cancels the inceptTransfer call and releases the decryption of `keyEncryptedFailure`. Cannot be called if `transferAndDecrypt` was called before.

- `releaseKey`: Called by the decryption oracle to release the corresponding key.

DECRYPTION ORACLE AND KEY FORMAT

The contracts rely on two keys, denoted by B or S. Let K denote any such key. The decryption of K is done by an decryption oracle, which listens to messages that request decryption, and injects decrypted keys into the `releaseKey` method of the `IDecryptionContract`.

We propose a key format that allows to ensured that the description oracle decrypts the key K only for the eligible contract. It seems as if this would require us to introduce a concept of eligibility to the description oracle, which would imply a kind of state. However, a fully stateless decryption can be realized by introducing a document format for the key K and a corresponding eligibility verification protocol.

We propose the following elements:

- A key documents K contain the callback `contract` address of the contract implementing `IDecryptionContract`.
- The decryption oracle offers a stateless function `verify` that receives an encrypted key $E(K)$ and returns the callback address (that will be used for `releaseKey` call) that is stored inside K without returning K.
- When an encrypted key $E(K)$ (i.e., $E(B)$ or $E(S)$) is presented to the decryption oracle, the oracle decrypts the document and passes K to `releaseKey` of the callback `contract` address found within the document K.

Key Format

We propose the following XML schema for the document of the decrypted key:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://finnmath.net/dvp/IDecryptionContract" attributeFormDefault="unqualified"
    elementFormDefault="qualified" targetNamespace="http://finnmath.net/dvp/IDecryptionContract"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="releaseKey">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="contract" type="xs:string" use="required" />
          <xs:attribute name="transaction" type="xs:string" use="optional" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Here, the `contract` attribute denotes a unique identification of a contract on a chain. This can be, for example, a CAIP-10 address, [4]. This attribute defines the contract for which the decryption should be performed. Depending on the implementation, this may be used to ensure that only eligible contracts trigger a decryption.

The optional `transaction` attribute can be used to link the key to a specific transaction, eliminating the risk of a replay of already observed keys.

A corresponding sample XML shown below.

```
<root xmlns:ilc="http://finnmath.net/dvp/IDecryptionContract">
  <ilc:releaseKey contract="eip155:1:0x1234567890abcdef1234567890abcdef12345678" transaction="3141">
        827364591027394857293847592374958273948572938475923749582739485729384
        43928... random data ensuring the uniqueness of this document... 29384
        749582739485729384759237495827394857293847592374958273948572938475923
    </ilc:releaseKey>
</root>
```

The decryption oracle should ensure that it performs decryption only for contracts matching the specification in the `contract`-attribute. The exact mechanism is an implementation detail of the decryption oracle.

Stateless API for the Decryption Oracle

We propose a stateless API for the decryption oracle. By adding a method that provides encrypted keys, there is no requirement that the encryption method is known to anyone else, except the decryption oracle. A simple hashing method is sufficient. In addition, participants can verify the encrypted key without exposing the key by a dedicated `verify` method.

The decryption oracle offers three stateless methods (endpoints):

- `requestEncryptedHashedKey(String contract, String transaction)`: internally generates K (with the provided attributes), creates encrypted the key $E(K)$ and the hashed key $H(K)$ and returns the pair $E(K), H(K)$, without exposing K.
- `verify(String encryptedKey)`: takes $E(K)$ and returns the corresponding `contract` address (stored inside K) and $H(K)$, without exposing K.
- `decrypt(String encryptedKey)`: takes $E(K)$, returns K, if the caller agrees with the `contractId` found in K.

The decryption oracle owns a public/secret key pair that for encryption/decryption of some key K.[1] The key K has the form

```
    class ReleaseKey {
        @XmlAttribute(name = "contract")
        String contract;      // Used to limit decryption request to a contract

        @XmlAttribute(name = "transaction")
        String transaction;   // User defined field

        @XmlValue
        String releaseKey;    // Secure random document
    }
```

Let $E(K)$ denote the encryption of K and $H(K)$ denote a hash of K. We describe the detailed stateless functionality of the decryption oracle.

[1] Since K will serve as a *key* to the unlocking of tokens, we call K key.

Key Generation: `requestEncryptedHashedKey`

```
EncryptedHashedKey requestEncryptedHashedKey(String contract, String transaction);
```

where

```
class EncryptedHashedKey {
    String encryptedKey;
    String hashedKey;
}
```

The method **`requestEncryptedHashedKey`** receives a contract id and (optionally) a transaction specification. It then internally generates a random key K incorporating the given **`contract`** and **`transaction`** attributed, performs encryption of K to $E(K)$ and hashing of K to $H(K)$, and returns $E(K)$ and $H(K)$ without exposing K.

Key Verification: `verify`

```
KeyVerification verify(String encryptedKey);
```

where

```
class KeyVerification {
    String contract;
    String transaction;
    String hashedKey;
}
```

The method **`verify`** internally decrypts the given $E(K)$ to K, extracts the **`contract`**-address from K, performs hashing of K to $H(K)$, and returns the **`contract`**-address and $H(K)$ without exposing K.

Key Decryption: `decrypt`

```
String decrypt(String encryptedKey);
```

The method **`decrypt`** takes E(K), internally decrypts it into K, extracts the `contract` from K, and verifies that the caller agrees with `K.contract`. If verified, it returns K. Here, **`encryptedKey`** is an $E(K)$, the encryption of some K, e.g., as generated by **`requestEncryptedHashedKey`**.

Rationale for DvP

For a secure DvP there will be two calls to **`requestEncryptedHashedKey`** to obtain the encrypted / hashed success key (buyer's key B) and the encrypted / hashed failure key (seller's key S).

The decryption contracts **`inceptTransfer`** is initialised with the encrypted keys for success and failure of the payment.

The locking contracts **`inceptTransfer`**/**`confirmTransfer`** is initialised with the hashed keys for success and failure of the payment.

If necessary, the seller and the buyer can verify that the contract keys are valid and consistent, i.e., that

- $E(B)$ observed in **`IDecryptionContract`** has the hash $H(B)$ observed in **`ILockingContract`**,

- $E(S)$ observed in `IDecryptionContract` has the hash $H(S)$ observed in `ILockingContract`,
- `B.contractId` and `S.contractId` agrees with the contract id of the `IDecryptionContract`.

This can be achieved by the corresponding calls to the `verify` function of the decryption oracle.

WORKFLOW OF A SECURE DELIVERY-VS-PAYMENT WITHOUT EXTERNAL STATE

We describe the complete workflow of a secure Delivery-vs-Payment utilizing two smart contracts, implementing the `ILockingContract` and `IDecryptionContract`, respectively, and their interaction with a decryption oracle. See also Figure 1.

Key Generation

1. The buyer generates the `keyEncryptedSeller` ($E(S)$) and `keyHashedSeller` ($H(S)$).

Using the `contract`-address of the desired `IDecryptionContract` a call to the decryption oracle's `requestEncryptedHashedKey` generates the encrypted key $E(S)$ and the corresponding hashed key $H(S)$.

Alternatively, the buyer of the (asset) token that will be transferred can generate S (`keySeller`) and use the decryption oracle's public key to encrypt the key `keySeller` to `keyEncryptedSeller` ($E(S)$), and use the hashing of the `ILockingContract` to generate $H(S)$. He will keep S secret. In this case the seller needs to use the `verify` method later to verify the validity of $E(S)$, $H(S)$.

2. The seller generates the `keyEncryptedBuyer` ($E(B)$) and `keyHashedBuyer` ($H(S)$).

Likewise, a second call to `requestEncryptedHashedKey` with the same `contract`-address generates the encrypted key $E(B)$ and the corresponding hashed key $H(B)$.

Alternatively, the seller of the (asset) token that will be transferred can generate B (`keyBuyer`) and use the decryption oracle's public key to encrypt the key `keyBuyer` to `keyEncryptedBuyer` ($E(B)$), and use the hashing of the `ILockingContract` to generate $H(B)$. He will keep B secret. In this case the buyer needs to use the `verify` method later to verify the validity of $E(B)$, $H(B)$.

Buyer to `ILockingContract`

3. The buyer executes on `ILockingContract` (asset) `ILockingContract::inceptTransfer(bytes32 id, int amount, address from, string keyEncryptedSeller)`.

Buyer's Cancellation Option

3'. Buyer executes on `ILockingContract` (asset) `ILockingContract::cancelTransfer(bytes32 id, int amount, address from, string keyEncryptedSeller)`.

This call can occur only after `ILockingContract::inceptTransfer` (3) and before `ILockingContract::confirmTransfer` (4) and will terminate this transaction.

Seller to `ILockingContract`

4. Seller executes on `ILockingContract` (asset) `ILockingContract::confirmTransfer(bytes32 id, int amount, address to, string keyEncryptedBuyer)`.

After this call, the asset will be locked by the `ILockingContract` to be transferred to the buyer (upon successful payment) or to be transferred back to the seller (upon failed payment).

Seller to `IDecryptionContract`

5. Seller executes on `IDecryptionContract` (payment) `IDecryptionContract::inceptTransfer(bytes32 id, int amount, address from, string keyEncryptedBuyer, string encryptedKeySeller)`.

The buyer (receiver of the tokens on `ILockingContract`, payer of the tokens on `IDecryptionContract`) can now verify that the payment transfer has been incepted with the proper parameters. In particular he can verify that `keyEncryptedBuyer` is associated with `keyHashedBuyer`

Verification against the Decryption Oracle

If required, seller and buyer can verify the consistency of the encrypted/hashed keys.

- Buyer and/or seller call `verify` on the decryption oracle.

Seller's Cancellation Option

6. Seller executes on `IDecryptionContract` (payment) `IDecryptionContract::cancelAndDecrypt(uint id, address from, address to, string keyEncryptedBuyer, string encryptedKeySeller)`.

This call can occur only after `IDecryptionContract::inceptTransfer` (5) and before `IDecryptionContract::transferAndDecrypt` (7). It will trigger the decryption of `encryptedKeySeller` (see `l1` below) and will terminate this transaction.

Buyer to `IDecryptionContract`

7. Buyer executes on `IDecryptionContract` (payment) `IDecryptionContract::transferAndDecrypt(uint id, address from, address to, string keyEncryptedBuyer, string encryptedKeySeller)`.

Completion of Transfer on `ILockingContract`

Upon Success:

If the call to `IDecryptionContract::transferAndDecrypt` resulted in a successful transfer of the (payment) tokens on the `IDecryptionContract`:

8. The `IDecryptionContract` emits an event `TransferKeyRequested` with `keyEncryptedBuyer` requesting decryption by the decryption oracle.
9. The decryption oracle reacts on this event and decrypts the `keyEncryptedBuyer` to `keyBuyer`, verifies that the event was issued by the corresponding `contract`, then calls `IDecryptionContract::releaseKey` with `keyBuyer`.
10. The buyer executes on `ILockingContract` `ILockingContract::transferWithKey(uint id, string key)` with `key = keyBuyer`.

Upon Failure:

11. The `IDecryptionContract` emits an event `TransferKeyRequested` with `keyEncryptedSeller` requesting decryption by the decryption oracle.
12. The decryption oracle reacts on this event and decrypts the `keyEncryptedSeller` to `keySeller`, verifies that the event was issued by the corresponding `contract`, then calls `IDecryptionContract::releaseKey` with `keySeller`.
13. The seller executes on `ILockingContract` `ILockingContract::transferWithKey(uint id, string key)` with `key = keySeller`.

Communication between `IDecryptionContract` and Decryption Oracle

The communication between the smart contract implementing `IDecryptionContract` and the decryption oracle is stateless.

The decryption oracle listens for the event `TransferKeyRequested`, which will show an `encryptedKey`, which is an encrypted document of the format suggested in the previous section.

The decryption oracle will decrypt the document `encryptedKey` without exposing the decrypted document `key`, verify that the event was issued from the contract specified in the decrypted `key`, and, in that case, submit the decrypted document `key` to the `releaseKey` function of *that* contract.

The following sequence diagram summarizes the proposed delivery-versus-payment process.

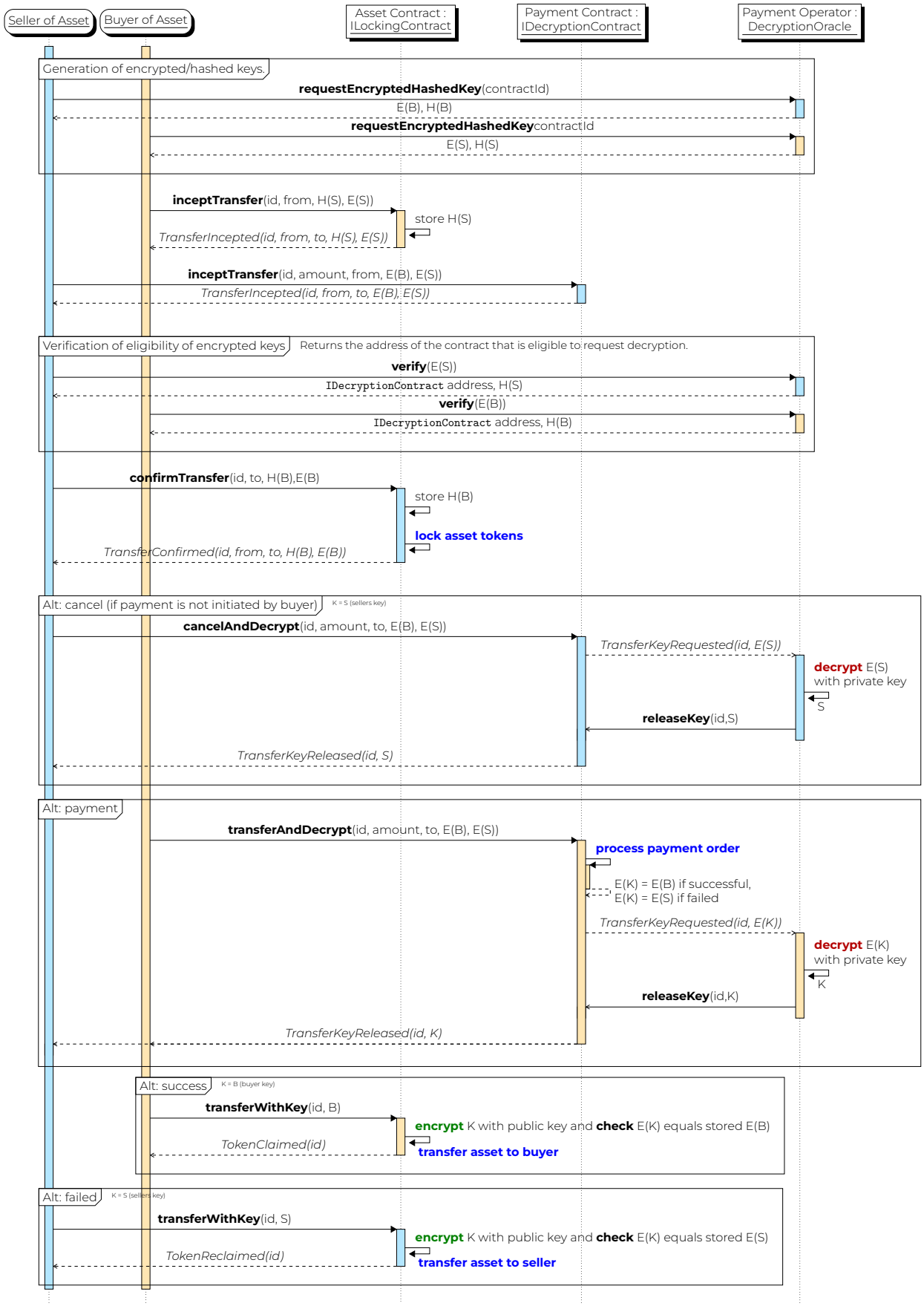

Figure 1: Complete sequence diagram of a delivery-versus-payment transaction.

ADVANTAGES

Regarding unlocking assets by providing an appropriate secret, the approach is similar to the Concept of a "Hash-linked Contract". Compared to this and other techniques, the main advantage of the present approach is:

- **No Intermediary Service Holding State**: Hashes or keys are not required to be stored by a third-party service. Hence, there is no additional point of failure. The payment operator's public key serves as the encryption key.

Besides this, other advantage, where some, but not all, are shared with variants of hash-linked contracts, are:

- **No Centralized Key Generation**: Keys are generated mutually by the trading parties at the trade inception phase and will not be needed afterwards.
- **No Timeout Scheme**: The transaction is not required to complete in a given time window, hence no timeout. The timing is up to the two counterparties. Either the buyer initiates the payment (via `IDecryptionContract::transferAndDecrypt`) or, in case of absence of payment initiation, the seller cancels the offer and re-claims the asset (via `IDecryptionContract::cancelAndDecrypt`).
- **No Coupling**: The payment chain and the payment chain operator do not need any knowledge of the associated asset. They only offer the possibility to perform a decryption with the payment operator's private key. The smart contract on the payment chain will trigger the decryption.
- **Lean Interaction**: The function workflow is structured and only consists of three main interactions:
    1. generate encrypted keys and lock assets,
    2. send payment order with encrypted keys,
    3. retrieve decrypted keys and unlock assets.

REMARKS

Encryption versus Hashing

The above scheme requires the use of an encrypted key $E(K)$ and a corresponding hashed key $H(K)$. This requires that the participants can check the consistency of the pair $E(K), H(K)$.

As encryption can be performed with a public key, in theory the hashing could be replaced by encryption. In that case the protocol simplifies slightly with $H(K) = E(K)$. The use of a separate hashing method is for practical reasons only, as on-chain encryption may be expensive.

However, the participants still need to check that $E(K)$ represents a proper encryption of an eligible key, i.e., the `verify` step may still be necessary.

Key Generation

The protocol suggests that the generation of the buyer's key B is performed/requested by the seller, and that the generation of the seller's key S is performed/requested by the buyer.

This is someone intentional to avoid a *replay-attack* where it would be possible to re-use a previously observed key. It is in the interest of the seller that the hash of the buyer's key is not that of a previously observed key, and vice versa.

The `transaction` attribute of the key format can be used to eliminate the risk of a replay-attack.

Pre-Trade versus Post-Trade

If the buyer does not initiate `IDecryptionContract::transferAndDecrypt`, this might be considered a *failure to pay*. However, this depends on interpreting which actions are deemed part of the trade inception phase and which are regarded post-trade.

Assume that we interpret the first three transactions, i.e.,

1. `ILockingContract:inceptTransfer`,
2. `IDecryptionContract:inceptTransfer`,
3. `ILockingContract:confirmTransfer`,

as being pre-trade, manifesting a *quote* and the seller's intention to offer the asset for the agreed price.

We may interpret the fourth transaction, i.e.,

4. `IDecryptionContract::transferAnDecrypt`,

as manifesting the trade inception and initiating the post-trade phase. So we could interpret this transaction as a `IDecryptionContract:confirmTransfer` that also immediately triggers the completion of the transaction.

If we take the two interpretations above, then `IDecryptionContract::transferAnDecrypt` marks the boundary of trade event and post-trade transactions. This interpretation implies that a lack

of `IDecryptionContract::transferAnDecrypt` does not represent a *failure to pay* and a `IDecryptionContract::cancelAndDecrypt`, initiated by the seller, has the straightforward interpretation of invalidating a pre-trade quote (or offer).

It may be disputed if a failed payment resulting from a `IDecryptionContract::transferAnDecrypt` represents a failed inception of the trade (pre-trade) or a failure to pay (post-trade). In any case, the seller is not facing the risk of losing the asset without a payment, and the buyer is not facing the risk of losing the payment without a delivery of the asset.

Note that introducing a locking scheme for the payment is either unnecessary (because the transfer can be completed immediately) or raises a corresponding possibility of a failure-to-deliver (if the asset locking occurs after the payment locking).

CONCLUSION

We proposed a decentralized transaction scheme that allows to realize a secure delivery-versus-payment across two blockchain infrastructures without the requirement to hold the state outside the chains. The requirements for the payment system operator are comparably small and are detached from the corresponding asset transaction.

We invite the community to verify that the scheme does not permit attack scenarios or to jointly elaborate the conditions to prevent them. We mentioned one implementation requirement in this direction: the smart contract on the payment chain has to reject opening a payment transaction (`IDecryptionContract::inceptTransfer`) that contains an encrypted key that was part of a previous transaction or is currently part of an open payment transaction. This is required because otherwise, it would allow the decryption of a key in a payment transaction (e.g., one with a smaller amount) that can be used to unlock an asset associated with the other payment transaction.

REFERENCES

[1] European Central Bank and Bank of Japan, *Securities settlement systems: delivery-versus-payment in a distributed ledger environment*, STELLA - a joint research project of the European Central Bank and the Bank of Japan, (2018).

[2] C. P. Fries and P. Kohl-Landgraf, *Erc 7573*, Ethereum Improvement Proposals, (2023). See `https://eips.ethereum.org/EIPS/eip-7573`.

[3] R. La Rocca, R. Mancini, M. Benedetti, M. Caruso, S. Cossu, G. Galano, S. Mancini, G. Marcelli, P. Martella, M. Nardelli, and C. Oliviero, *Integrating dlts with market infrastructures: Analysis and proof-of-concept for secure dvp between tips and dlt platforms*, Bank of Italy Markets, Infrastructures, Payment Systems Working Paper No. 26, (2022).

[4] J. Thorstensson and M. Sena, *Ceramic improvement proposal 7: Caip-10 link.*, Ceramic Improvement Proposals, (2020). See `https://cips.ceramic.network/CIPs/cip-7`.