A Proposal for a Lean and Functional 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 1.2 [v3])

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. The core idea is depicted in the following slightly simplified sequence diagram:

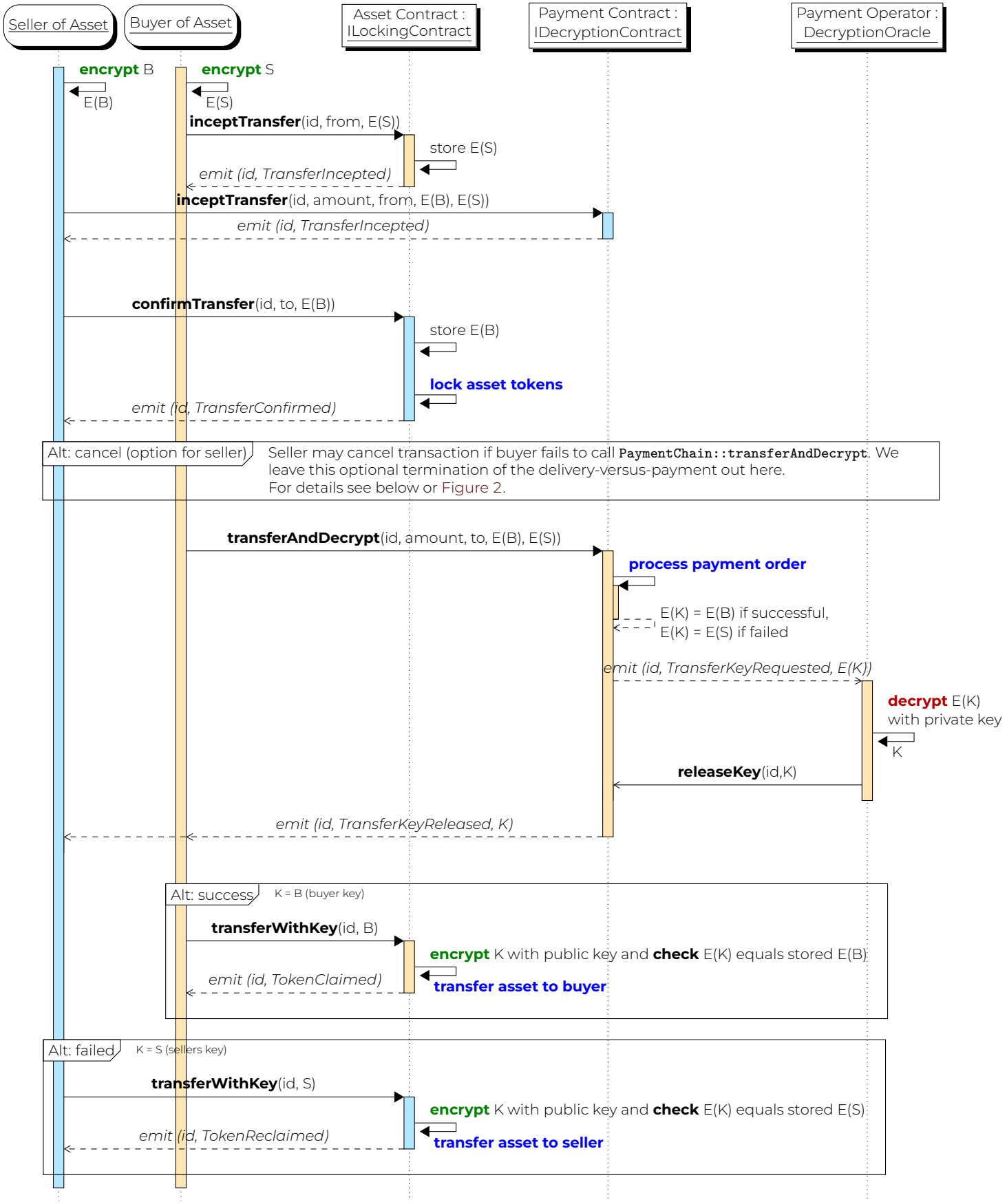

Figure 1: Sequence diagram of a delivery-versus-payment transaction between a buyer and a seller (of the asset), interacting on the asset and the payment chain. Note the low involvement of the payment chain and payment operator. For a complete version see Figure 2.

CONTENTS

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].

Acknowledgment

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

SETUP

Consider the setup of having two chains. The first one is managing tokenized assets. We call this chain the *Asset Chain*. The second chain can run smart contracts that trigger and verify payments. We call this chain the *Payment Chain*.

The following setup is assumed:

- Trading parties interact on the asset chain over an *Asset Contract* and agree on trade terms in a functional way.
- Comparable to HLC, the asset contract can lock asset balances and will transfer them upon certain conditions are met.
- Trading parties interact with the payment chain either by their private keys or via an API interface.
- The payment chain is linked to the payment system and is operated by a payment operator - e.g. a central bank.
- Payment orders are initiated over a *payment contract* and may be processed, e.g. via the trigger concept [3] (not depicted in the diagram above).
- The payment chain operator (or, alternatively, a dedicated service provider) provides a public key that can be used to encrypt “documents”. This key is known to the trading parties interacting on the asset chain.
- The payment operator (or, alternatively, a dedicated service provider) offers a single additional functionality, namely to decrypt given documents (in the following called keys), when the smart contract on the payment chain emits a certain message.

Notation

The method proposed here relies on a service that will decrypt documents observed in messages emitted by the smart contract on the payment chain. In the following, we call these documents *key*, because they are used to unlock transactions (on the asset chain). There is a key for the buyer and the seller. In Figure 1, these keys are denoted by B and S, respectively. The encrypted keys are denoted by $E(B)$ and $E(S)$.

REALIZING DELIVERY-VS-PAYMENT WITHOUT EXTERNAL STATE

Based on the assumptions above, we propose a lean function pattern on both chains to implement Delivery-vs-Payment.

Functionality on Payment Chain: `inceptTransfer` and `transferAndDecrypt`

On the *payment chain* the following function is needed:

1. The payment chain operator offers functionality to decrypt any encrypted string (a key).
2. The payment is associated with two encrypted keys `keyEncryptedSuccess` and `keyEncryptedFail`. Both encrypted key strings have been encrypted using the payment chain operator's public key.[1]
3. If and only if the associated payment is successful, the `keyEncryptedSuccess` will be decrypted to the original key `keySuccess` and will be published by the payment contract as part of a message that the buyer can use to unlock the assets. In case of a failed payment transfer `keyEncryptedFail` is decrypted to `keyFail`, and the `keyFail` will be emitted by the payment contract, which the seller can use to retrieve back the locked assets.

This functionality is encapsulated in two functions: The first one is initiating the payment details. The second one is performing the payment transfer:

1. `PaymentChain::inceptTransfer(uint id, int amount, address from, string keyEncryptedSuccess, string encryptedKeyFail)` - the receiver of the payment (address: to) is implicitly the function caller: `to = msg.sender`.
2. `PaymentChain::transferAndDecrypt(uint id)` - requirements: `sender == from`.

The first one is sent from the seller of the asset; the second one is sent from the buyer of the asset. The ordering of the transactions as given in Figures 1 and 2 is relevant.

Important: The smart contract on the payment chain does not allow the inception of a payment transfer containing a key that has been used in a previous or current transaction. This is required to avoid a decrypted key being obtained from a different payment transaction. There are different ways of ensuring this.

Functionality on the Asset Chain: `inceptTransferWithKey` and `transferWithKey`

On the *asset chain*, the following three functions are needed:

1. `AssetChain::inceptTransfer(uint id, int amount, address from, string keyEncryptedSeller)` - called by the buyer to initiate the trade.
2. `AssetChain::confirmTransfer(uint id, int amount, address to, string keyEncryptedBuyer)` - called by the seller to confirm the trading terms. This call will result in a locking of the asset to be transferred.
3. `AssetChain::transferWithKey(uint id, string key)` - transfers the asset to the buyer if `encrypt(key)` = `keyEncryptedBuyer` or to the seller if `encrypt(key)` = `keyEncryptedSeller`.

[1] It is assumed that the keys are large and random, such that a collision with $E(B) = E(S)$ is excluded. Otherwise, the `inceptTransfer` / `confirmTransfer` with equal encrypted keys is rejected.

Workflow

Buyer to Asset Chain

1. Buyer generates a random key (string): `keySeller`.
2. Buyer uses the payment chain operator's public key to encrypt the key `keySeller` to `keyEncyptedSeller` (`E(S)`).
3. Buyer executes on asset chain `AssetChain::inceptTransfer(uint id, int amount, address from, string keyEncyptedSeller)`

Seller to Asset Chain

4. Seller generates a random key (string):`keyBuyer`.
5. Seller uses the payment chain operator's public key to encrypt the key `keyBuyer` to `keyEncyptedBuyer` (`E(B)`)
6. Seller executes on asset chain `AssetChain::confirmTransfer(uint id, int amount, address to, string keyEncyptedBuyer)`.

The asset will now be locked by the asset contract to be transferred to the buyer (upon successful payment) or to be transferred back to the seller (upon failed payment).

Seller and buyer can now observe the tuple `(keyEncyptedSuccess, keyEncyptedFail)` on the asset chain (we implicitly assume that it can be verified that both are well-formed encrypted keys, i.e., for each encrypted key $E(K)$ the owner of the secret key can generate K such that encrypting K with the public key results in $E(K)$.

Seller to Payment Chain

7. Seller executes on payment chain `PaymentChain::inceptTransfer(uint id, int amount, address from, string keyEncryptedBuyer, string encryptedKeySeller)`.

Calling the payment contract, the buyer can now verify that the payment transfer has been incepted with the proper parameters.

Buyer to Payment Chain

8. Buyer executes `PaymentChain::transferAndDecrypt(uint id, address from, address to, string keyEncryptedBuyer, string encryptedKeySeller)`.

Upon Success :

9. Payment contract emits message with `keyBuyer` (decrypted `keyEncryptedBuyer`).
10. On the asset contract, the buyer is now able to execute `AssetChain::transferWithKey(uint id, string key)` with `key = keyBuyer`.

Upon Failure :

11. Payment contract emits message with `keySeller` (decrypted `encryptedKeySeller`).
12. On the asset contract the seller is now able to execute `AssetChain::transferWithKey(uint id, string key)` with `key = keySeller`.

The asset contract will now apply the encryption algorithm with the payment operator's public key to convert the given `key` to `keyEncrypted`.

- The asset will be transferred to the buyer if `keyEncrypted == keyEncryptedBuyer`.
- The asset will be transferred (back) to the seller if `keyEncrypted == keyEncryptedSeller`.

Function Calls

To summarise, there are two interactions with the asset chain establishing the trade, two interactions with the payment chain preparing and executing the payment and one additional interaction with the asset chain to claim or re-claim the asset.

- Buyer to Asset Chain:
  `AssetChain::inceptTransfer(uint id, int amount, address from, string keyEncryptedSeller)`.
- Seller to Asset Chain:
  `AssetChain::confirmTransfer(uint id, int amount, address to, string keyEncryptedBuyer)`.
- Seller to Payment Chain:
  `PaymentChain::inceptTransfer(uint id, int amount, address from, string keyEncryptedBuyer, string keyEncryptedSeller)`
- Seller to Payment Chain:
  `PaymentChain::cancelTransfer(uint id, int amount, address from, string keyEncryptedBuyer, string keyEncryptedSeller)` (*optional*)
- Buyer to Payment Chain:
  `PaymentChain::transferAndDecrypt(uint id, address from, address to, keyEncryptedBuyer, string keyEncryptedSeller)`
- Buyer of Seller to Asset Chain:
  `AssetChain::transferWithKey(uint id, string key)`, where key is the observed `keyBuyer` or `keySeller`.

Cancellation

After `PaymentChain::inceptTransfer` and before `PaymentChain::transferAndDecrypt`, the seller can cancel the transaction by calling `PaymentChain::cancelAndDecrypt`.
This will decrypt the `keySellerEncrypted` to `keySeller`, and the seller can re-claim his asset.

ENSURING SECURE KEY DECRYPTION - KEY FORMAT

It has to be 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.

A fully stateless decryption can be realized by introducing a document format for the key K (S or B) and a corresponding eligibility verification protocol. We propose the following elements:

- The key documents (B and S) contain the address of the payment contract implementing `IDecryptionContract`.
- The decryption oracle offers a stateless function `verify` that 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.

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/erc/ILockingContract" attributeFormDefault="unqualified"
    elementFormDefault="qualified" targetNamespace="http://finnmath.net/erc/ILockingContract" 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:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A corresponding sample XML shown below.

```
<root xmlns:ilc="http://finnmath.net/erc/ILockingContract">
  <ilc:releaseKey contract="eea9e1da-d56a-4c0a-9c08-f2e76f616426">
        827364591027394857293847592374958273948572938475923749582739485729384
        43928... random data ensuring the uniqueness of this document... 29384
        749582739485729384759237495827394857293847592374958273948572938475923
    </ilc:releaseKey>
</root>
```

The description oracle now offers the two fully stateless functions:

- **`verify`**: decrypts an encrypted document $E(K)$ and exposes the value of the **`contract`** attribute without exposing the value K.
- **`decrypt`**: decrypts an encrypted document $E(K)$, verifies that the request of this decryption came from the **`contract`** address specified inside K, and, in that case only, exposes the document K.

Rationale

In the incept/confirm phase of a transaction, the two counterparties utilize the `verify` function to mutually verify the eligibility of the encrypted documents $E(S)$ and $E(B)$. They can verify that these documents are tied to the correct *IDecryptionContract*.

INTERFACES

Smart Contract on the Asset Chain

```
inceptTransfer(uint id, int amount, address from, string keyEncryptedSeller)

confirmTransfer(uint id, int amount, address to, string keyEncryptedBuyer)

transferWithKey(uint id, string key)
```

Smart Contract on the Payment Chain

```
inceptTransfer(uint id, int amount, address from, string keyEncryptedBuyer, string keyEncryptedSeller)

transferAndDecrypt(uint id, address from, address to, keyEncryptedBuyer, string keyEncryptedSeller)

cancelAndDecrypt(uint id, address from, address to, keyEncryptedBuyer, string keyEncryptedSeller)
```

ADVANTAGES

Regarding unlocking assets by providing an appropriate secrete, 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 `PaymentChain::transferAndDecrypt`) or, in case of absence of payment initiation, the seller cancels the offer and re-claims the asset (via `PaymentChain::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.

EXTENSIONS AND VARIANTS

Avoiding a Complex Encryption on the Asset Chain

The above scheme requires that the asset chain smart contract can *encrypt* the given key with the payment operator's public key and compare the result to the given `keyEncrypted`. This is somewhat involved since it requires the implementation of an encryption algorithm on the chain. If this is not an option, the scheme can be modified to rely only on the given key's simple hashing (e.g., sha256).

In the proposed setup above, the encrypted keys $E(S)$ and $E(B)$ serve as a *hash*. The asset will be released if S or B are presented, and the corresponding encrypted value matches the stored value. This functionality can be implemented with a simple hashing algorithm. In that case, the asset chain will store hashed values $H(S)$ and $H(B)$ and release the asset to the corresponding owner if the hash of the presented key matches the stored hash value.

Thus, the function calls to the asset chain are performed with hashes of the keys, i.e., (`keyEncrypted` is replaced with `keyHashed`).

The encrypted keys, $E(S)$ and $E(B)$, are still needed to retrieve the key upon successful payment.

However, in this modified scheme, it has to be ensured that the hashes correspond to the encrypted key. So $H(K)$ corresponds to $E(K)$, i.e., both are derived from the same K. To allow verification that a hashed key $H(K)$ (`keyHashed`) corresponds to the encrypted key $E(K)$ (`keyEncrypted`), the payment system operator (the owner of the secret key) offers the simple function that returns the hashed key for a given encrypted key without exposing the decrypted key. So it provides the function f with $f(E(K)) = H(K)$ without exposing K.

By that additional function, the buyer and seller can verify that the hashes stored on the asset chain are in accordance with the encrypted keys that initialized the payment transfer.

PRE-TRADE VERSUS POST-TRADE

If the buyer does not initiate `PaymentChain::tranferAndDecrypt`, 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. `AssetChain:inceptTransfer`,
2. `PaymentChain:inceptTransfer`,
3. `AssetChain: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. `PaymentChain::transferAnDecrypt`,

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

If we take the two interpretations above, then `PaymentChain::transferAnDecrypt` marks the boundary of trade event and post-trade transactions. This interpretation implies that a lack of `PaymentChain::transferAnDecrypt` does not represent a *failure to pay* and a `PaymentChain::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 `PaymentChain::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 (`PaymentChain::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.

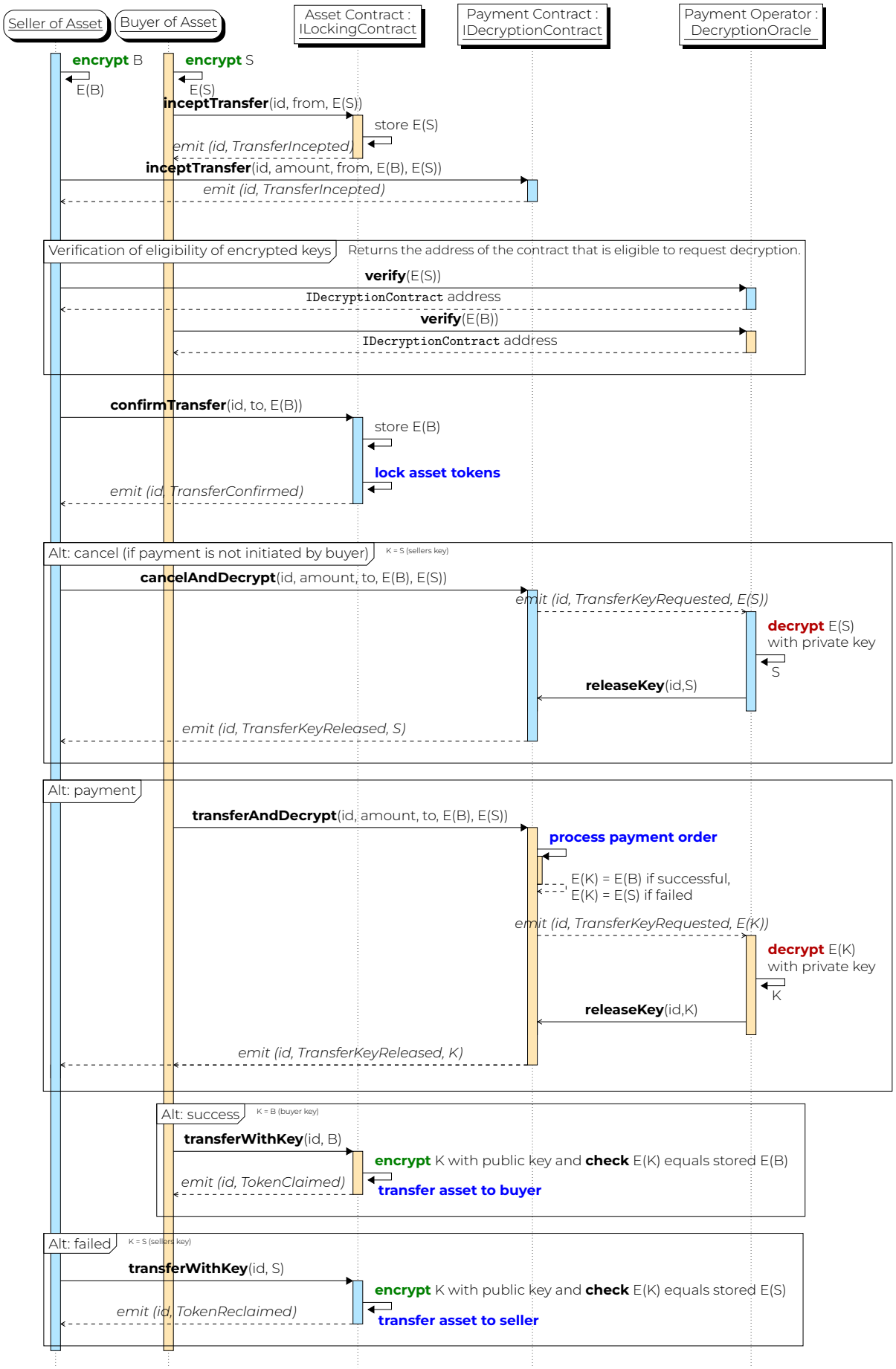

Figure 2: Complete Sequence diagram of a delivery-versus-payment transaction between a buyer and a seller (of the asset), interacting on the asset and the payment chain. Note the low involvement of the payment chain and payment operator.

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).