A Stateless and Secure Delivery versus Payment across two Blockchains

Christian Fries[1,2,*] Peter Kohl-Landgraf[1,*]

November 9th, 2023

(version 3.0 [v5])

[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

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, stateless and functional transaction scheme to establish secure delivery-versus-payment across two blockchains. Our approach eliminates the need for stateful intermediaries and ensures minimal overhead for the payment chain operator, who does not need to store state.

The main idea comes with two requirements: First, a stateless decryption service is attached to the payment chain that allows decrypting messages with the decryption service operators secret key. Second, a "Payment Contract" is deployed on the payment chain that implements a function

```
transferAndDecrypt(uint256 id, address from, address to,
    string keyEncryptedSuccess, string keyEncryptedFail)
```

that processes the (trigger-based) payment, requests decryption, 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 stateless decryption service could be performed using a threshold description scheme, in which case the requirement of a single trusted entity would be removed.

Contents

1. Introduction **3**
- 1.1. Contribution 3
- 1.2. Comparison of the Present Method to other DvP Methods 4
- 1.3. Acknowledgments 4

2. Problem Description **5**
- 2.1. Time Locks 6
- 2.2. Decryption Oracle 6
- 2.3. Decentralization and Disintermediation 7

3. Contract Interfaces **8**
- 3.1. Notation 8
- 3.2. `ILockingContract` 9
- 3.3. `IDecryptionContract` 9

4. Workflow of a Secure Delivery-vs-Payment without External State **11**
- 4.1. Key Generation 11
- 4.2. Buyer to `ILockingContract` 11
- 4.3. Seller to `IDecryptionContract` 11
- 4.4. Verification against the Decryption Oracle 12
- 4.5. Buyer's cancellation option on `ILockingContract` 12
- 4.6. Seller to `ILockingContract` 12
- 4.7. Seller's cancellation option on `IDecryptionContract` 12
- 4.8. Buyer to `IDecryptionContract` 12
- 4.9. Completion of Transfer on `ILockingContract` 13
- 4.10. Communication between `IDecryptionContract` and Decryption Oracle 13
- 4.11. Complete Sequence Diagram 13

5. Decryption Oracle and Key Format **15**
- 5.1. Key Format 16
- 5.2. Single Oracle Design: A Stateless API for the Decryption Oracle 16
- 5.3. Rationale for DvP 18
- 5.4. Distributed Oracle Design: Threshold Decryption 18

6. Remarks **19**
- 6.1. Encryption versus Hashing 19
- 6.2. Key Generation 19
- 6.3. Security Considerations 20
- 6.4. Pre-Trade versus Post-Trade 20

7. Conclusion **21**

A. Double-Locking on two Contracts **24**
- A.1. Steps to Create the Double Locking 24
- A.2. Race Conditions 26

1. Introduction

Delivery-versus-Payment (DvP) is a critical issue in financial markets, particularly for digital assets, tokenization, and DeFi. Traditional services, like Central Counterparty Clearinghouses (CCPs) and Central Securities Depositories (CSDs), play a crucial role in financial market infrastructure by mitigating counterparty risk and ensuring the efficient settlement of securities transactions. However, these systems rely on intermediaries, may require participants to post collateral, and introduce potential liquidity constraints.

Within the domain of financial transactions and distributed ledger technology (DLT) [1], the Hash-Linked Contract (HLC) concept has been recognized as valuable and has been thoroughly investigated, see [2, 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 a stateful, often API-based interaction mechanism with centralized trusted entities [3, 4], which bridges the communication between a so-called Asset Chain and a corresponding Payment System, or, require time-based constraints. These time-based constraints could be time-locks, which are required at least on one of the two chains, [5, 6], or verifiable timed signatures (VTS) [7].

Time-based constraints can be problematic, as they introduce short-term options, which will ultimately be part of the transaction cost, and still exhibit race conditions.

We present an alternative implementation that makes use of a stateless decryption oracle. This allows us to remove the dependency on time-based constraints (via time-locks or VTS) or stateful APIs.

The requirement for decryption does not imply the requirement of a single central authority. There could be many decryption oracles attached to the chain, and the two parties could choose a mutually trusted one on a transaction level. In addition, a threshold decryption scheme can be used, where multiple oracles perform partial decryption.

For settlement involving central bank digital currency or commercial bank money tokens [8], it may be reasonable to integrate the decryption oracle into the payment system. However, this is not a requirement.

The smart contracts proposed here are available as ERC 7573 [9] in the Ethereum Improvement Process, [10].

1.1. Contribution

With respect to HTLC, our approach removes the requirement of time locks. With respect to the use of a central trusted entity, our approach does not require the storage of state within the decryption oracle. With respect to on-chain verification, our approach does not require complex signing or encryption algorithms to run on-chain. As a by-product, we elaborate that the central problem in cross-blockchain DvP is that function arguments are observable before the function execution is final. We show that the use of decryption oracles solves this general problem; see Figure 1.

1.2. 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
External Storage Required?	No	Yes (storage for time-lock)	Yes	Yes
Timeout Required?	No	Yes	No	No
Coupling of Payment & Delivery	Decoupled, but order matters	Coupled via hash preimage and timeing.	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

1.3. Acknowledgments

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

2. Problem Description

The central issue of a secure delivery-versus-payment across two blockchains is a *race-condition* that arises from function arguments being visible before the function execution.

To illustrate this, consider two contracts, *AssetContract* and *PaymentContract*, operating on separate blockchains: *AssetChain* and *PaymentChain*. While these names suggest a specific use case, the two chains could manage tokens representing different types of assets. We aim to perform a transfer on the *AssetChain* if and only if a transfer on the *PaymentChain* was successful.

In addition, consider two participants, the *buyer* and the *seller*. The buyer is receiving the asset and delivering the payment. The seller is delivering the asset and receiving the payment.

Contracts can store values and calculate hashes, thus, compare hashes of given arguments with previously stored values. In addition, contracts can verify the caller of a function.

Assume that there is a sequence of function calls that brings the two contracts into a double-locking state that is as follows:[1]

- On the asset contract, the asset has been transferred from the seller to a "lock", and
  - if the buyer calls the function `AssetContract::transferWithKey(K)` with $K = B$, the asset will be transferred to the buyer, or
  - if the seller calls the function `AssetContract::transferWithKey(K)` with $K = S$, the asset will be transferred to the seller.
- On the payment contract, the payment has been transferred from the buyer to a "lock", and
  - if the seller calls the function `PaymentContract::transferWithKey(K)` with $K = B$, the payment will be transferred to the buyer, or
  - if the buyer calls the function `AssetContract::transferWithKey(K)` with $K = S$, the payment will be transferred to the seller.
- The seller knows B, but does not know S.
- The buyer knows B, but does not know S.

It appears as if this situation solves the secure delivery-versus-payment, because

- if the seller collects the payment, the buyer observes B an can collect the asset (completion of the transaction), or
- if the buyer re-collects the payment, the seller observes S and can re-collect the asset (cancelation of the transaction).

However, the above situation (and also its creation) suffers from a *race-condition*:

It must be ensured that the function `PaymentContract::transferWithKey(K)` can be called only once. So once it is called by one party, it is blocked by the other party. Now„ assume that both

[1]We show how to create this situation in Section A.

parties call this function at the same time. One call goes through, the other has to be blocked. If the function performs the blocking interally, the argument K can be observed on the blocked call. Hence, it may be possible that both keys S and B are observed, which clearly compromises the scheme.

For example, the seller likes to complete the transaction and calls `AssetContract::transferWithKey(B)`, but right before the buyer called `AssetContract::transferWithKey(S)`, which transferred the payment back to the buyer. The seller's call is unsuccessful, but the argument B has been observed, allowing the buyer to fetch the asset (without a payment), given that he is fast enough on the asset chain.

2.1. Time Locks

The above problem can be partially solved by *time-locks*, which, however, have similar vulnerabilities.

With a time-lock, the seller can collect the payment with a call to `PaymentContract::transferWithKey(B)` if and only if he performs this call within a specific time. After some pre-agreed time T_1 the payment will be transferred back to the buyer if not collected. The buyer has no option to collect/cancel the payment before (i.e., there is no cancellation key S).

Likewise, there is a time-lock on the asset chain that transfers the asset back to the seller after time T_2, with $T_2 > T_1$.

This setup still exhibits the same race condition: if the seller makes his call to `PaymentContract::transferWithKey(B)` slightly too late, he fails to collect the payment, but exposes B (allowing the buyer to collect the asset).

In addition, if the seller successfully collects the payment, the buyer may still fail to collect the asset if technical issues prevent him from completing his call before T_2. Hence, time-locks are susceptible to denial-of-service attacks, preventing one party from completing its function call within a given time-frame.

2.2. Decryption Oracle

A secure solution to the race condition problem is the introduction of *encrypted arguments* and a secure decryption oracle.

Assume that there is an offchain oracle that performs a decryption service exclusively for specific contract.

Encryption of a document K can be performed with a public key, but decryption of an encrypted document $E(K)$ can only be requested by the specific contract.

In that case, we may implement a `function` in a way that it consumes an encrypted argument `argumentEncrypted`, performs the necessary synchronization steps to prevent a race condition, i.e., checks if an encryption has already been performed, and then and only then, request decryption of the given argument. The processing of the decrypted argument is then continued in a callback. Alternatively, the synchronization could also take place in the decryption oracle, by preventing exposing a key that belongs to a transaction for which a key has already been exposed.

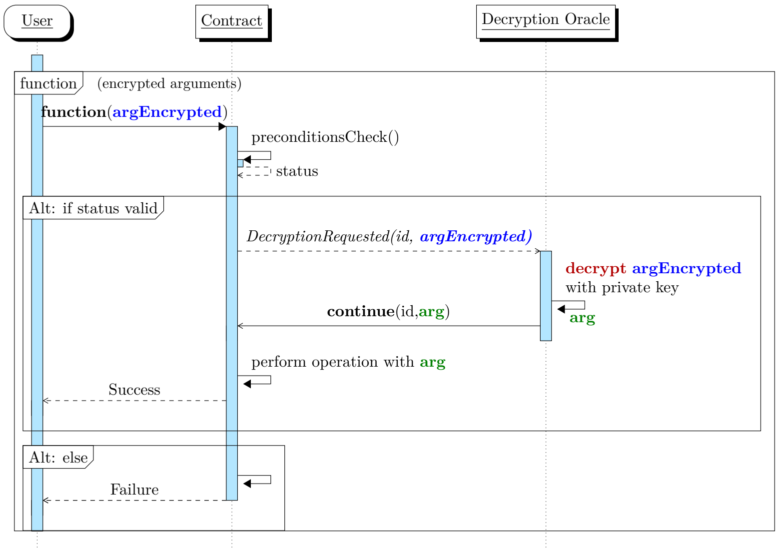

Figure 1: Example for a function with encrypted arguments interacting with a decryption oracle.

2.3. Decentralization and Disintermediation

It appears as if the requirement of a central trusted decryption oracle annihilates the advantages of a blockchain or distributed ledger with respect to decentralization and disintermediation. However, there could be many equivalent decryption oracle services attached to the network, and counterparties could agree on a decryption oracle per transaction basis. Therefore, the decryption oracle does not represent a monopolistic central role. It is an interchangeable service.

3. Contract Interfaces

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 back 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 decryption, we propose a decryption oracle that offers a stateless service for verification and decryption of encrypted keys and, for convenience, can also perform the generation of the 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 with 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.

3.1. Notation

The method proposed here relies on a service that will decrypt an encrypted document, where the encrypted document is 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 `ILockingContract` the *buyer*, and the payer of the tokens handled by the `ILockingContract` the *seller*. For tokens handled by the `IDecryptionContract`, the flow is reversed, the aforementioned *buyer* is the payer of the `IDecryptionContract`'s tokens, and the aforementioned *seller* is the receiver of the `IDecryptionContract`'s tokens. This fits to the interpretation that the tokens 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 2 and throughout, these keys are denoted by B (buyer key, successful payment) and S (seller key, 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 $H(S)$.

3.2. `ILockingContract`

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

```
inceptTransfer(uint256 id, int amount, address from, string memory keyHashedSeller, string memory keyEncryptedSeller);

confirmTransfer(uint256 id, int amount, address to, string memory keyHashedBuyer, string memory keyEncryptedBuyer);

cancelTransfer(uint256 id, int amount, address from, string memory keyHashedSeller, string memory keyEncryptedSeller);

transferWithKey(uint256 id, string key);
```

A contract implementing this interface provides a transfer of tokens (usually presenting 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 whose 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 whose 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 `IDecryptionContract`).

3.3. `IDecryptionContract`

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

```
inceptTransfer(uint256 id, int amount, address from, string memory keyEncryptedSuccess, string keyEncryptedFailure);

transferAndDecrypt(uint256 id, int amount, address to, string memory keyEncryptedSuccess, string keyEncryptedFailure);

cancelAndDecrypt(uint256 id, address from, string memory keyEncryptedSuccess, string memory keyEncryptedFailure);

releaseKey(uint256 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 whose 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 whose address is implicit (`from = msg.sender`). Verifies that the payer's (`from`) address, receiver's (`to`) address, and the encrypted keys match the corresponding call (with the same `id`) to `inceptTransfer`. Tries to perform a transfer of the token.

  A successful transfer will emit a request to decrypt `keyEncryptedSuccess`, a failed transfer will emit a request to decrypt `keyEncryptedFailure`, which then results in decryption (if the keys were valid). The decryption of the keys will (usually) handled by an external oracle; see below for a proposal of the corresponding functionality.

  Cannot be called if `cancelAndDecrypt` was called before.

- `cancelAndDecrypt`: Called by the receiver of the (payment) token whose address is implicit (`to = msg.sender;`). Verifies that the payer's (`from`) address, receiver's (`to`) address, and the encrypted keys match the corresponding call (with the same `id`) to `inceptTransfer`. Cancels the inceptTransfer call and emits a request to decrypt `keyEncryptedFailure`.

  Cannot be called if `transferAndDecrypt` was called before.

- `releaseKey`: Called by the decryption oracle with the decrypted key, to release (emit) the corresponding key.

4. 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 2. In this section we consider a single trusted decryption oracle. For the discussion of the decryption process and an the utilization of a distributed decryption process see Section 5.

4.1. Key Generation

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

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

4.2. Buyer to `ILockingContract`

3. The buyer executes on `ILockingContract` (asset) `ILockingContract::inceptTransfer (uint256 id, int amount, address from, string keyHashedSeller, string keyEncryptedSeller)`.

At this point $E(S)$ and $H(S)$ can be observed.

4.3. Seller to `IDecryptionContract`

4. Seller executes on `IDecryptionContract` (payment) `IDecryptionContract::inceptTransfer (uint256 id, int amount, address from, string keyEncryptedBuyer, string encryptedKeySeller)`.

At this point $E(B)$ and $E(S)$ have been observed.

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`

4.4. 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.

4.5. Buyer's cancellation option on `ILockingContract`

5*. Buyer executes on `ILockingContract` (asset) `ILockingContract::cancelTransfer(uint256 id, int amount, address from, string keyEncryptedSeller)`.

This call can occur only after `ILockingContract::inceptTransfer` (3) and before `ILockingContract::confirmTransfer` (5) and will terminate this transaction.

4.6. Seller to `ILockingContract`

5. Seller executes on `ILockingContract` (asset) `ILockingContract::confirmTransfer(uint256 id, int amount, address to, string keyHashedBuyer, string keyEncryptedBuyer)`.

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

At this point $E(B)$ and $H(B)$ can be observed.

4.7. Seller's cancellation option on `IDecryptionContract`

6*. Seller executes on `IDecryptionContract` (payment) `IDecryptionContract::cancelAndDecrypt(uint256 id, address from, address to, string keyEncryptedBuyer, string encryptedKeySeller)`.

This call can occur only after `IDecryptionContract::inceptTransfer` (4) and before `IDecryptionContract::transferAndDecrypt` (6). It will trigger the decryption of `encryptedKeySeller` (see 4.9.2 below) and will terminate this transaction.

The seller can cancel the payment and obtain the key to re-claim the asset in case the buyer does not complete the payment.

4.8. Buyer to `IDecryptionContract`

6. Buyer executes on `IDecryptionContract` (payment) `IDecryptionContract::transferAndDecrypt(uint256 id, address from, address to, string keyEncryptedBuyer, string encryptedKeySeller)`.

4.9. Completion of Transfer on `ILockingContract`

4.9.1. Upon Success:

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

7. The `IDecryptionContract` emits an event `TransferKeyRequested` with `keyEncryptedBuyer` requesting decryption by the decryption oracle.

8. The decryption oracle reacts to this event and decrypts `keyEncryptedBuyer` to `keyBuyer`, verifies that the event was issued by the corresponding `contract`, then calls `IDecryptionContract::releaseKey` with `keyBuyer`.

9. The buyer executes on `ILockingContract::transferWithKey(uint256 id, string key)` with `key = keyBuyer`.

4.9.2. Upon Failure:

7*. The `IDecryptionContract` emits an event `TransferKeyRequested` with `keyEncryptedSeller` requesting decryption by the decryption oracle.

8*. The decryption oracle reacts to this event and decrypts `keyEncryptedSeller` to `keySeller`, verifies that the event was issued by the corresponding `contract`, then calls `IDecryptionContract::releaseKey` with `keySeller`.

9*. The seller executes on `ILockingContract::transferWithKey(uint256 id, string key)` with `key = keySeller`.

4.10. 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.

4.11. Complete Sequence Diagram

The following sequence diagram summarizes the proposed delivery-versus-payment process; see Figure 2.

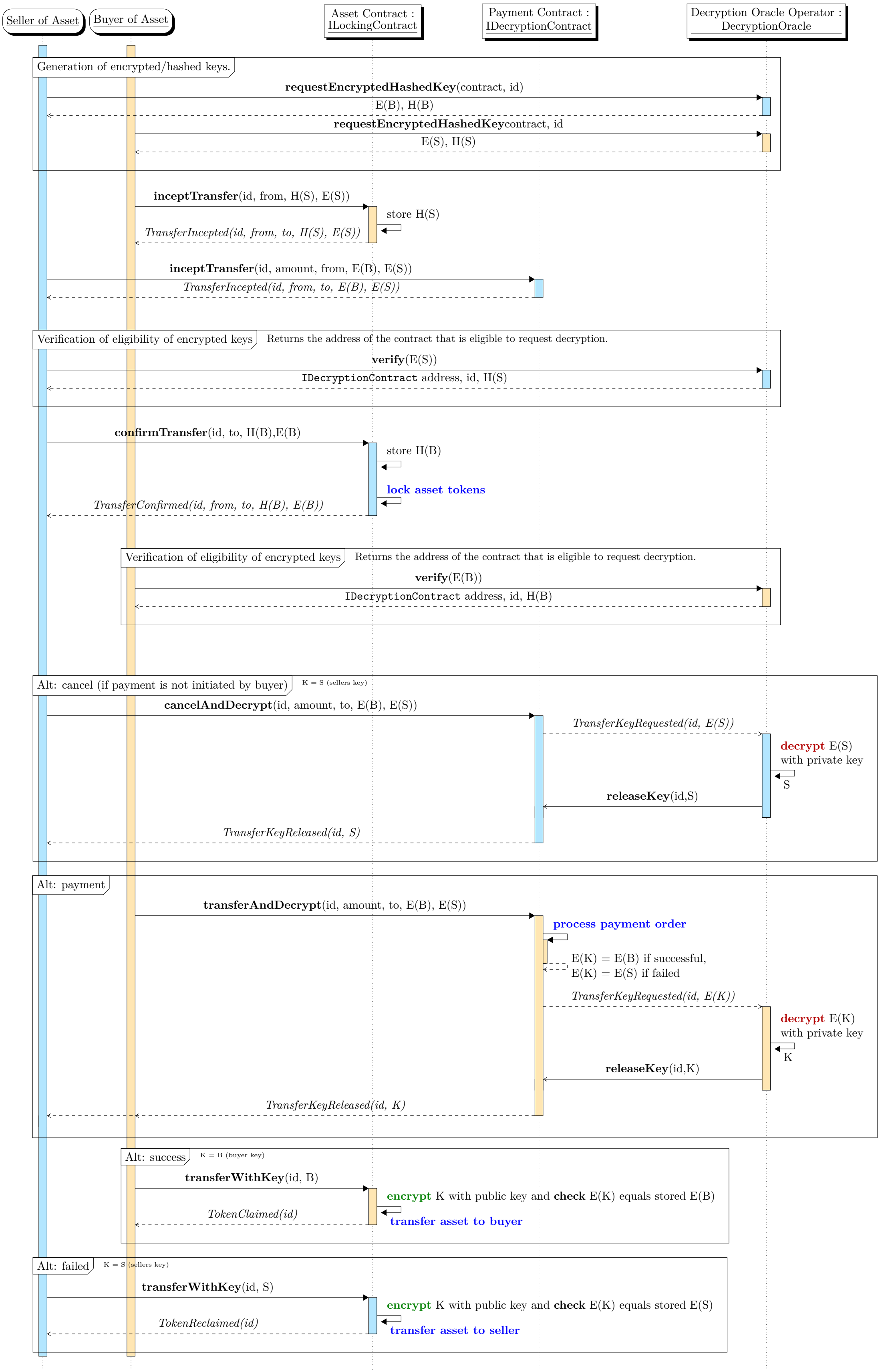

Figure 2: Complete sequence diagram of a delivery-versus-payment transaction.

5. 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`.

It is extremely important that the decryption oracle decrypts a key only if specific preconditions are met. The preconditions are

- the request is issued from the eligible contract,
- the request is issued for the eligible transaction id (and the contract ensures that there cannot be two open transactions with the same id).

We propose a key format that allows to ensure that the decryption oracle releases the key K only for the eligible contract / transaction.

It seems as if this would require us to introduce a concept of eligibility to the decryption 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 document K contains the `contract` callback address of the contract implementing `IDecryptionContract`.
- A key document K contains the `transaction` specification (the id and possibly other information) of the transaction created by `IDecryptionContract::inceptTransfer`.
- The decryption oracle offers a stateless function `verify` that receives an encrypted key $E(K)$ and returns the `contract` callback address (that will be used for `releaseKey` call), the `transaction` detail that is stored inside K, and the hash $H(K)$ 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, verifies the `contract` and `transaction` attributes, and, if verified, passes K to `releaseKey` of the callback `contract` address found within the document K.

5.1. Key Format

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

```xml

<?xml version="1.0" encoding="utf-8"?>
<xs:schema 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:attribute name="transaction" type="xs:unsignedShort" use="required" />
                </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, [11]. This attribute defines the contract for which the decryption should be performed. The `transaction`-attribute should contain the id of the transaction opened with `inceptTransfer` (where the `IDecryptionContract` ensures that two transaction ids cannot be open simultaneously).

A corresponding sample XML is shown below.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<releaseKey contract="eip155:1:0x1234567890abcdef1234567890abcdef12345678" transaction="3141" xmlns="http://finnmath.net/erc/ILockingContract">
    <!-- random data -->
    zZsnePj9ZLPkelpSKUUcg93VGNOPC2oBwX1oCcVwa+U=
</releaseKey>
```

The decryption oracle should ensure that it performs decryption only for contracts matching the specification in the `contract`-attribute and transactions matching the specification in the `transaction`-attribute. This prevents replay attacks and the misuse of `inceptTransfer` and `cancelAndDecrypt` functions. The exact mechanism is an implementation detail of the decryption oracle.

5.2. Single Oracle Design: A Stateless API for the Decryption Oracle

We consider the case of a single trusted decryption oracle and propose a stateless API for it.

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 attributes provided), creates the encrypted 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` and `transaction` fields (stored inside K) and $H(K)$, without exposing K.
- `decrypt(String encryptedKey)`: takes $E(K)$, returns K, if the caller agrees with `contract` found in K.

The decryption oracle owns a public/secret key pair for encryption/decryption of some key K.[2] The key K has the form

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

    @XmlAttribute(name = "transaction")
    String transaction;   // limit decryption request to eligible transaction

    @XmlValue
    String value;        // 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.

5.2.1. Key Generation: `requestEncryptedHashedKey`

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

where

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

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

5.2.2. Key Verification: `verify`

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

where

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

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

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

5.2.3. Key Decryption: `decrypt`

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

The method `decrypt` takes $E(K)$, internally decrypts it into K, verifies that the caller agrees with `K.contract` and that the calling transaction agrees with `K.transaction`. If verified, it returns K, otherwise it returns nothing / fails.

Here, `encryptedKey` is an $E(K)$, the encryption of some K, e.g., as generated by `requestEncryptedHashedKey`.

5.3. 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 contract's `inceptTransfer` is initialized with the encrypted keys for success and failure of the payment.

The locking contract's `inceptTransfer`/`confirmTransfer` is initialized 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.

5.4. Distributed Oracle Design: Threshold Decryption

The decryption oracle does not need to be a single trusted entity. Instead, a threshold decryption scheme [12] can be employed, where multiple oracles perform partial decryption, requiring a quorum of them to reconstruct the secret key. This enhances security by mitigating the risk associated with a single point of failure or trust.

In such cases, each participating decryption oracle will observe the decryption request from an emitted `TransferKeyRequested` event, and subsequently call the `releaseKey`(id,K_i) method with a partial decryption result K_i, see Figure 3.

Once sufficient partial decryptions K_i have been submitted to the decryption contract, the key K can be reconstructed. The reconstruction of K is an implementation detail. An option is that the decryption contract performs the reconstructions and emits a final `TransferKeyReleased` event with the reconstructed key. If on-chain reconstruction is considered computationally too intense, another option is that the partial decryptions are merely published and the reconstruction is left to the user.

To allow both implementation variants, the `TransferKeyReleased`-event has a `sender` argument that exhibits the caller of the `releaseKey`-function. This allows to distinguish partial decryptions from different external oracles, while a final reconstruction could show the decryptions contracts address as the sender in the `TransferKeyReleased`-event.

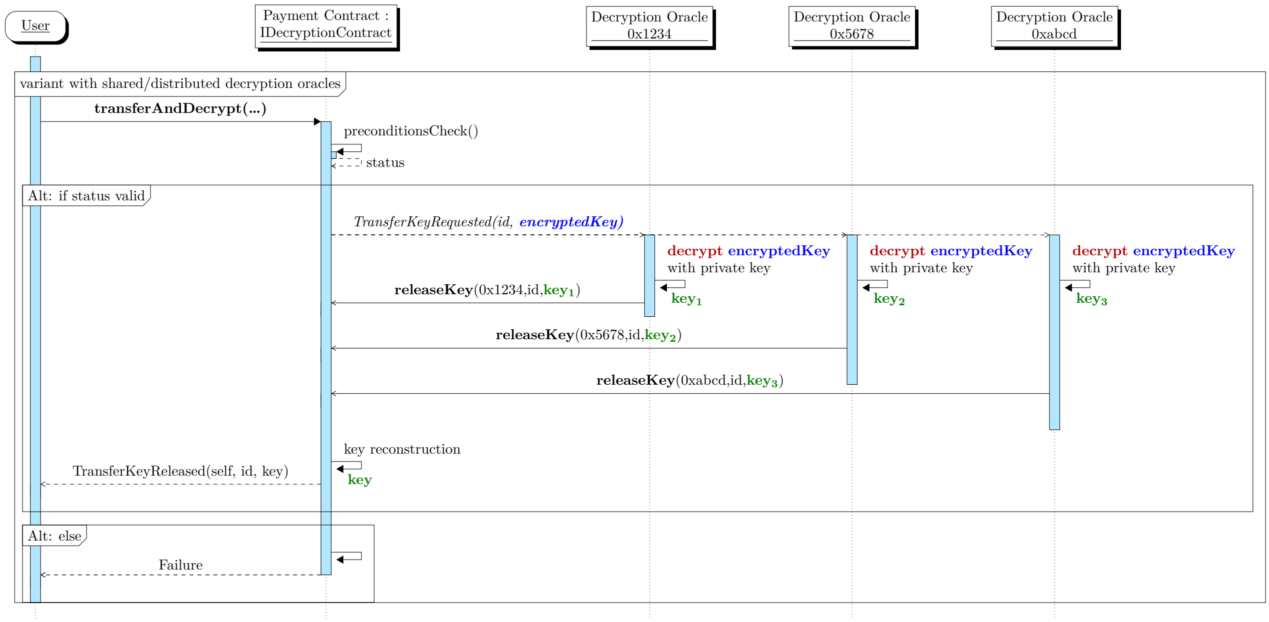

Figure 3: Shared/distributed decryption with a theshold decryption scheme.

6. Remarks

6.1. 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.

6.2. 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 somewhat intentional to avoid a *replay-attack* where it would be possible to reuse 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 has to be used to eliminate the risk of a replay-attack.

6.3. Security Considerations

6.3.1. Preventing Replay Attacks

To prevent replay attacks that allow to perform a decryption of some $E(K)$ through a different contract or a different transaction the contract address and the transaction id should be part of the key K *and* the description oracle should perform decryption only for eligible keys.

For example, without such a check, a simple attack is to initialize a payment transaction with `inceptTransfer(id, from, E(Y), E(X))` followed by a call to `cancelTransfer(id, from, E(Y), E(X))`. This will result in a decryption of the presumed failure-key X. If $E(X)$ is the success-key of some other transaction, this then allows the attacker to obtain the asset without payment. If $E(X)$ is the failure-key of some other transaction, this then allows to re-claim the asset after a buyer has paid.

This attack cannot occur if the locking of the asset with `ILockingContract::confirmTransfer` is performed after the call to `IDecryptionContract::inceptTransfer` and the `IDecryptionContract` does not allow to open a transaction with the same id.

6.3.2. Consensus Decryption Oracle via Threshold Encryption

The keys that are decrypted by the decryption oracle are useless to a third person, as smart contracts can still ensure that a buyer or seller, but no other party, can receive the asset(s). However, the decryption oracle has to be a trusted entity. It should be noted that it is not necessarily a central trusted entity. First, an appropriate decryption oracle could be chosen out of many on a per-contract basis. In addition, an option could be to implement a k-out-of-n threshold decryption. Such an algorithm allows to encrypt a message using n public keys, where the decryption requires the decryption using at least k secret keys.

6.4. 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).

7. 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 proposed a key format that prevents replay attacks.

The proposed decryption oracle can also be used to resolve race-conditions in other applications.

Compared to other delivery-versus-payment schemes, 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 decryption oracle 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 can be generated mutually by the trading parties at the trade inception phase and will not be needed afterwards. Centralized key generation is a convenient option.

- **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`). The absence of time-outs remove the possibility of unwanted race-conditions.

- **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 observe the transaction state, which then triggers 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.

The proposed smart contract interfaces are available as ERC 7573, [9]. An open source reference implementation of the decryption oracle is available at [13]

References

[1] Alexander Bechtel, Agata Ferreira, Jonas Gross, and Philipp Sandner. The future of payments in a dlt-based european economy: A roadmap. In Markus Heckel and Franz Waldenberger, editors, *The Future of Financial Systems in the Digital Age: Perspectives from Europe and Japan*, pages 89–116. Springer Singapore, Singapore, 2022.

[2] 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.

[3] Rosario La Rocca, Riccardo Mancini, Marco Benedetti, Matteo Caruso, Stefano Cossu, Giuseppe Galano, Simone Mancini, Gabriele Marcelli, Piero Martella, Matteo Nardelli, and Ciro 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] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *Proceedings of the VLDB Endowment*, 13(9):1319–1331, May 2020.

[5] Yunyoung Lee, Bumho Son, Huisu Jang, Junyoung Byun, Taeho Yoon, and Jaewook Lee. Atomic cross-chain settlement model for central banks digital currency. *Information Sciences*, 580:838–856, 2021.

[6] Philipp Hoenisch, Subhra Mazumdar, Pedro Moreno-Sanchez, and Sushmita Ruj. Lightswap: An atomic swap does not require timeouts at both blockchains. In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, and Nicola Dragoni, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 219–235, Cham, 2023. Springer International Publishing.

[7] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1299–1316. IEEE, 2022.

[8] Lee Braine, Shreepad Shukla, Piyush Agrawal, Shrirang Khedekar, and Aishwarya Nair. Payments use cases and design options for interoperability and funds locking across digital pounds and commercial bank money, 2024.

[9] Christian P. Fries and Peter Kohl-Landgraf. Erc-7573: Conditional-upon-transfer-decryption for delivery-versus-payment. *Ethereum Improvement Proposals*, 2023. See https://eips.ethereum.org/EIPS/eip-7573.

[10] Ethereum Community. Ethereum request for comments (ercs), 2024. See https://eips.ethereum.org/erc.

[11] Joel Thorstensson and Michael Sena. Ceramic improvement proposal 7: Caip-10 link. *Ceramic Improvement Proposals*, 2020. See https://cips.ceramic.network/CIPs/cip-7.

[12] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 307–315, New York, NY, 1990. Springer New York.

[13] finmath.net. finmath decryption oracle: Reference impementation of an erc 7573 decryption oracle, 2025. See https://gitlab.com/finmath/finmath-decryption-oracle.

A. Double-Locking on two Contracts

In the following S, B, C, X and Y are secret documents (e.g., long random sequences) and $H(S)$, $H(B)$, $H(C)$, $H(X)$, $H(Y)$ are *hashes* of the respective document. For a given K the calculation of $H(K)$ is easy, for a given $H(K)$ the determination of K is hard (presumed impossible)

We consider two contracts, *asset* and *payment*, each running on a separate chain. The names are purely illustrative. The aim is to perform a transfer on *asset* if and only if a transfer on *payment* was successful.

There shall be parties, the *buyer* and the *seller*. The buyer is receiving the asset and delivering the payment. The seller is delivering the asset and receiving the payment.

The contracts can store values, calculate hashes, and compare them with previously stored values. Execution of transfer is conditional to a successful comparison. The contracts can verify the caller of a function.

A.1. Steps to Create the Double Locking

Key Generation

0. **Key Generation**: Buyer knows S, Y. Seller knows B, C, X.

Partially Locking the Asset

1. **Buyer**: `asset.incept(H(S), H(Y))`

   Buyer knows S and Y. The asset contract internally stores $H(S)$ and $H(Y)$.

2. **Seller**: `asset.confirm(H(B), H(C), H(X))`

   Seller knows B and C and X. The asset contract internally stores $H(B)$ and $H(C)$ and $H(X)$.

   The asset contract moves the asset from the seller account to a lock.

From this point on and until 5), the seller can call the cancel function on asset at any time.

2* **Seller**: `asset.cancel(C)`

   Transfers the asset back to the seller. The buyer can observe C (i.e., the buyer knows C, once the seller cancels).

The asset is now locked with a cancel option:

- Seller can cancel by presenting C (asset is transferred back to seller).
- Buyer has no stake in the transaction yet.
- Buyer can retrieve asset with B (but does not know B yet).
- Buyer can remove Seller's cancellation option by presenting (X,Y) (but does not know X yet).

Locking the Payment

3. **Buyer**: `payment.incept(H(B), H(S), H(C), H(X))`

   The payment contract internally stores H(B) and H(S) and H(C) and H(X).

3* **Buyer**: `payment.cancel(H(B), H(S), H(C), H(X)`

   The buyer can cancel the previous incept if the seller does not confirm. In this case the seller will use C to cancel the asset locking.

4. **Seller**: `payment.confirm(H(B), H(S), H(C), X)`

   The payment contract verifies that $H(X)$ equals the previously stored $H(X)$.

   The payment is moved from the buyers account to the lock. The buyer can no longer cancel the payment locking without presenting C.

The payment is locked:

- Seller can retrieve the payment with (B,Y), but Y has not been observed yet.
- Buyer can cancel the payment with C (if Seller cancels asset with C).
- The value of X has been observed.

Buyer has stake in the transaction, Seller can cancel with C. Buyer cannot cancel (without Seller's cancellation), hence he likes to remove Seller's cancellation option. He is next. Now, that he has observed X he will remove the Seller's cancellation option asap. He could not do this before, because this requires X.

Locking the Asset

Continuation on asset contract requires that X has been observed in the previous step on the payment contract.

5. **Buyer**: `asset.lock(H(B),H(S), X, Y)`

The asset is locked.

- Buyer can retrieve asset with B (but does not know B yet).
- Seller can retrieve asset with S (but does not know S yet).
- Seller cannot cancel with C anymore.
- Y has been observed. Seller can retrieve payment.

Both parties have stake in the transaction. The symmetric double-locking is established.

Completing the Transaction: Payment

Continuation on Payment Contract requires that Y has been observed in the previous step on the asset contract

Either of the following two will finalize the transaction.

6. **Seller**: `payment.retrieve(B,Y)`

   Seller gets payment, exposes B, Buyer can take asset.

6* **Buyer**: `payment.retrieve(S)`

   Buyer gets payment, exposes S, Seller can take asset.

Completing the Transaction: Asset

Continuation on Asset Contract requires that either B or S has been observed in the previous step on the asset contract.

7. **Buyer**: `asset.retrieve(B)`

   Buyer can take asset, knows B.

7* **Seller**: `asset.retrieve(S)`

   Seller gets asset, knows S.

A.2. Race Conditions

The above protocol exhibits three race conditions:

1. The cancellation of the payment by the buyer in 3* could conflict with the confirmation of the payment by the seller in 4. If the first call goes through the buyer receives the payment back. If the second call is observed, the buyer observes X and can lock the asset with 5.

2. The cancellation of the asset by the seller in 2* could conflict with the locking of the asset by the buyer in 5. If the first call goes through the seller receives the asset back. If the second call is observed, the seller observes Y and can retrieve the payment with 6.

3. The retrieval of the payment by the seller in 6 could conflict with the retrieval of the payment by the buyer in 6*. If both calls are observed, we observe B and S simultaneously, leaving it open who gets the asset.

While hashes allow to synchronize the executions across two chains, we are left with the issue that simultaneous function calls on the same chain could allow the observation of their arguments, which may compromise the scheme.

Notes

Keywords

Delivery vs Payment, DvP, Settlement, Atomic Swap, Hashed Timelock Contract, HTLC, Smart Contract, Blockchain, ERC 7573

JEL Codes

E42, G20

Suggested Citation

Fries, Christian P. and Kohl-Landgraf, Peter: *A Stateless and Secure Delivery versus Payment across two Blockchains* (November 9, 2023). Available at SSRN: https://ssrn.com/abstract=4628811 or http://dx.doi.org/10.2139/ssrn.4628811

For updates see https://ssrn.com/abstract=4628811.