# Off-Chain Insured Networks

Abhishek Sharma
Advised by Professor Maurice Herlihy

Spring, 2018

## 1 Introduction

Bitcoin's *Lightning Network*[1] introduced a means of solving the bitcoin scaling problem through off-chain channels which reduce the overall load on the network by reducing the total number of transactions that need to be broadcast on it. Users who know they will have many recurring payments between each other can set up *channels* which allow them to engage in secure, *atomic* payments between each other. However, concerns have been raised about the vaibility of the *lightning* network in practice, due to the need for a single user to have many channels funded in order to route their transactions to the rest of the network, thereby tying up much of users' funds. One overarching concern among bitcoin users is that the only users who will be able to fund enough channels to route transactions will be the ones who already have most of the capital on the network, introducing a centralizing effect whereby central 'banks' will form who offer the service of routing many lightning transactions for fees. For the goal of bitcoin and other cryptocurrencies in enabling scalable, cheap, and near-instant worldwide transactions on a decentralized network, the high capital requirements of the lightning network limit its potential to add value for cryptocurrency users.

In this project, we developed a framework for conducting similar off-chain transactions without the need for channels. Instead, funds are put into a 'pool' which can be used to trade with any party with funds placed in escrow on multiple blockchains. The atomicity of these cross-currency trades is ensured by hash-timelocked contracts of the type used in Bitcoin's *Lightning*[1] network as well as in Ethereum's *Raiden*[2] network.

We found that we gain an ability to trade with multiple parties in-exchange for the added risk that our counterparties may be double-spending their inputs when they trade with us. This risk can be mitigated through the involvement of a trusted insurance agent, who, through a provably-accurate mechanism of assigning blame on the smart-contract, developed by us, can identify double-spenders and compensate wronged-parties.

## 2 Overview

The two main components of off-chain networks are the *protocol* that participants use to engage in trades and the *smart-contract* they use to manage funds on-chain.

In the *protocol*, participants trade their escrowed funds by exchanging signed *IOU*'s which can be posted to the contract to obtain the agreed-upon funds. Our protocol enables parties to engage in reversible and atomic IOU swaps using *hash-timelocked contracts* (HTLC's) of the sort found in *lightning*[1], and enables them to use outputs of incoming IOU's as inputs to outgoing IOU's, i.e., to trade funds that they received via other trades, off-chain. This means that participants can engage in constant re-writing of their balances with other parties and act as 'middlemen' to allow funds to go from one escrow account to another, enabling trading at a near-unlimited rate and with the full expected functionality of a traditional cryptocurrency exchange.

The *smart-contract* contains logic for handling the postings of IOU's *to the network*. In practice, the smart-contract may be encoded in slightly different forms on all of the blockchains the participants wish to trade on, but it will have the same functionality regardless of which programmable blockchain it is coded on. We will assume that all blockchains have Turing-complete scripting languages for smart-contracts, and so the pseudo-code given later in this paper could be translated to any such blockchain. We also assume that all properties of blockchains, such as full visibility of called functions and their arguments, apply.

Because IOU's can be voided and can be dependent on payment received from other IOU's, the contract contains a system of timeouts to ensure that the funds cannot be withdrawn from a user's account until that user has had time to post all relevant IOU's and contest the posted IOU with proof if necessary. Once the timeout on an IOU expires, it can be redeemed directly from the escrow balance of the sending account by its intended recipient. Since IOU's can be dependent on other IOU's, the smart contract contains logic that makes the timeouts of IOU's which are dependent on other IOU's dependent on those IOU's timeouts.

Finally, because the protocol does not protect against double-spends, the contract contains logic to publicly label the perpetrators of double-spends as *at-fault* for those transactions, and ensure that IOU's sent by honest parties that depend on invalid IOU's are not executed, and that the party to blame for an IOU not being executed is publicly noted.

The contract has the property that if an IOU posted by an honest party can't be executed, at-least one perpetrator of the failure will always always be found and labelled. This provides a sort of 'API' for 3rd parties who wish to act as 'insurance agents' for the participants of an off-chain swap agreement. If an IOU cannot be completed as-promised, the intended recipient of funds can present the insurer with evidence that they are not to blame for the failure of the payment from the smart-contract itself, along with a proof that the receiving party was unaware that the IOU was fraudulent (discussed in the 'passing proofs'

subsection of the 'protocol' section), to request remuneration from the insurance-agent for the funds promised. The insurance agent role can be taken by a 3rd party with a reputation, or it can be codified in the form of another smart-contract. The incentives for the insurance agents to play the role could come in the form of payments or deposits from all members of the agreement, giving insurance agents the ability to keep deposits and penalize malicious parties, as well as take compensation for the risk they take on.

To conclude, we have greatly improved the speed and scalability of off-chain payments while sacrificing some of the security from double-spends, but we incorporate functionality that enables incentives for third parties to protect honest parties in off-chain agreements.

# 3 Shared Constructions

The following data-structures are used in both the off-chain protocol and the on-chain contract. Though they may differ in the addition or removal of one or two attributes, the equivalent version of the given off-chain/on-chain version of the struct can be produced by its recipient without any additional information.

## 3.1 The IOU

```
struct IOU {
    Currency uint;
    Amount uint;
    From address;
    To address;
    Hashlock uint;
    Timelock uint;
    Lasthash uint;
    Signature uint;
}
```

An IOU is a data-structure that is passed by a participant of the protocol, along with a signature, to counterparties to whom they wish to transfer funds *off-chain*. IOU's are the medium of cryptocurrency exchange in the network. An IOU is a data structure that contains all relevant information about a payment that a party in the network has agreed to. These IOU's can be published to the blockchain in order to retrieve promised funds. They do not, however, guarantee that the funds promised have not been double-spent.

It is important to note that the IOU stored in the contract has additional attributes, but they are initialized to null values and are only used by the internal contract logic.

A description of the struct attributes is given below:

- *Currency*: uint that maps via a previously agreed-upon map to a digital asset (cryptocurrency)

- *Amount*: The total units of digital asset to be sent
- *From*: the public key of the sender
- *To*: the public key of the intended recipient of the funds
- *Hashlock*: (optional) denotes the hash-value that must be matched by a hashlock
- *Timelock*: (optional) determines when the IOU expires
- *Lasthash*: (optional) the hash of the previous incoming IOU that is being voided-out (explained later)
- *Signature* A signature of the hash of the entire IOU contents (without the signature field) signed with the public key corresponding to the *From* field of the IOU

Also, it is important to note that the smart-contract's internal representation of an IOU contains more fields which are only relevant in the context of the contract, so those will be introduced and explained in a later section.

### 3.2 Hash-Timelocks

A *hash-timelocked* iou is one which will only be executed if the user calling its execution can present a value such that its hash matches the hashlock on the contract. Such an IOU can only be executed if it is posted within in a time-interval defined by the IOU.

Hash-timelocks are represented as fields in the IOU struct. When an IOU does have a hash-timelock, its Hashlock and Timelock attributes are set by the creator to non-null values (in the case where they are null, the contract views the IOU as possessing no hash- or time-lock). For a hash-timelocked IOU presented to the contract to be considered valid, its recipient must post the value $v$, such that under an agreed-upon hash function, $hash()$, gives $hash(v) = a$, where $a$ is the 'Hashlock' value of the signed struct. In addition, the current system time must be before the time given in the 'Timelock' field of the IOU.

Hash-Timelocks are used in a protocol similar to *lightning*'s[1] in order to ensure *atomic* swaps of IOU's between parties, thereby enabling trading.

## 4 The Protocol

### 4.1 Entering the Escrow Agreement

Before they can begin trading, participants in the pool must deposit their escrow funds into contracts on each of the blockchains they would like to trade on. The smart-contract pseudo-code is given in a later section.

## 4.2 Conducting Trades

### 4.2.1 IOUs:

In order for two participants to agree to a trade, they must engage in an atomic HTLC (hash-time-locked contract) protocol to swap IOU's. IOU's are exchanged in a protocol similar to *lightning*, in which users exchange signed, hash-time-locked revocable IOU's. When a user sends IOU's to a counterparty, they actually send several separate IOU's, one for each blockchain they wish to send currency from. In addition, each IOU has the ability to nullify one or more IOU's that were completed between the two parties before it. This provides the same functionality as *lightning*'s 'Breach-Remedy Transactions'[1], which are used to secure one-way payment channels.

### 4.2.2 HTLC Protocol:

This protocol enables users to trustlessly trade tokens, and ensures that they can adjust their channel-balances without settling-out on the main chain.

Here we assume that it takes $\Delta$ time for a party to read an event the blockchain and post to the next block. If parties $A$ and $B$ are members of the agreement, they are trading in tokens $t_1$ and $t_2$, and they would like to trade $n_A$ of token $t_1$ to party $A$ from $B$ and $n_B$ of token $t_2$ to party $B$ from $A$ from $A$, then the swap would proceed as-follows:

1. The two must agree on who will sign the first IOU. Assume that they both agree that party $A$ will sign and send the first IOU.
2. Party $A$ signs and sends sends IOU $I_A$, where:
   - $I_A.Currency$ is set to the positive integer that maps to $t_2$
   - $I_A.Amount = n_B$
   - $I_A.From$ is set to $A$'s address on the $t_2$ blockchain
   - $I_A.To$ is set to $B$'s address on the $t_2$ blockchain
   - $I_A.Hashlock = h_A$, some hashlock for which the hash-value is known by $A$
   - $I_A.Timelock = t + 2\Delta$, where $t$ is the current time.
   - $I_A.Signature$ is $A$'s signature of the hash of the rest of $I_A$
3. Party $B$ receives $I_A$ and creates, signs, and sends $I_B$ where:
   - $I_B.Currency$ is set to the positive integer that maps to $t_1$
   - $I_B.Amount = n_A$
   - $I_B.From$ is $B$'s address on the $t_1$ blockchain
   - $I_B.To$ is $A$'s address on the $t_1$ blockchain
   - $I_B.Hashlock = h_A$

- $I_B.Timelock = t + \Delta$
- $I_B.Signature$ is $B$'s signature of the hash of the rest of $I_B$

At this point, the HTLC atomic swap protocol has been completed. For convenience, the two parties can now exchange 'finalized' versions of these two IOU's without the hashlocks or timelocks (starting with A sending its finalized version). Note that it wasn't necessary to broadcast any messages to any of the main chains in order for both parties to trustlessly and securely agree to a trade.
**Example:**
If Alice wishes to trade Bob 1 NEO for 1 Ether, Alice will make an IOU with a payment of 1 NEO out to Bob with a timeout of 2*$\Delta$, hash-locked with $h_A$, and send it to Bob. Bob will then create an IOU with a payment of 1 ether to Alice with a timeout of $\Delta$, also hash-locked by $h_A$. At this point, the payment may be completed in any manner the counterparties choose. Once Alice sees that the payment has been ensured, she may send a non-hash-locked and non-time-locked version of the payment to Bob and Bob could then respond with a non-hash-locked and non-time-locked version of the payment.

## 4.3 Revoking Past IOU's

A key feature of IOU's is that they are *revocable*. That is, two parties may mutually agree to render old IOU's void at any point and create a new balance between them for any currencies included in the agreement. This is done using the *Lasthash* field of the IOU struct. The *Lasthash* field denotes the hash of the contract that has been voided by the current contract. An important feature of voiding is that it operates in a 'zipper' pattern, in which $A$'s outgoing payment to $B$ voids the previous payment from $B$ to $A$. (see figure 1) Using HTLC's ensures security when voiding old contracts, as a later contract cannot be used to void out an earlier one if the later contract's timelock has expired.

Contract hashes act as backpointers to the previous contract in IOU structs which enable compact 'proofs' that old IOU's have been voided-out by their recipients. Since the recipient of a payment only ever sends outgoing payments to its counterparty, incoming payments always act as invalidators for previous outgoing payments. The smart-contract enables parties to use the new IOU issued by a counterparty as a means to invalidate any attempt by the counterparty to execute said payment. This means that users must store old IOUs, even after they have been overwritten.

**Example:** We use the previous example in which Alice wishes to trade Bob 1.0 NEO for 1.0 Ether. We assume that this trade has been completed and finalized. Then, Alice and Bob decide to increase their trade to 1.5 NEO for 1.5 Ether. They can void their old trade by having Alice include the hash of Bob's 1.0 Ether payment in the *Lasthash* section of her payment, and Bob do the same, adding Alice's 1.0 NEO payment to his *Lasthash* field.

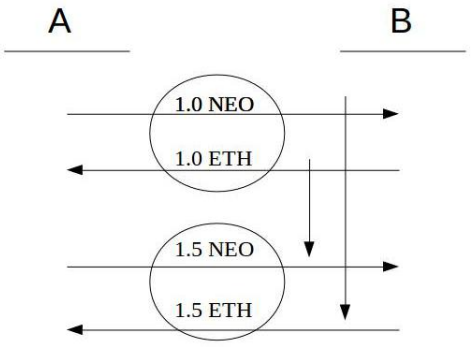


Figure 1: A visual illustration of revocable payments. Horizontal lines represent IOU's. Circles represent atomic groups of payments. Vertical lines represent revocation by the head of the arrow.

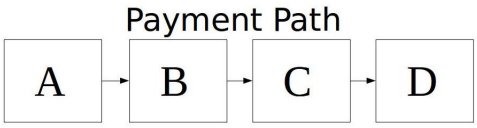


Figure 2: A string of payments representing a single asset being passed along the arrows

### 4.4 Passing Proofs

In addition to interacting with the smart-contract and engaging in the trading protocol, participants must, for each trade they make, pass on proofs of the full source of each payment, in the form of a list of IOU's from the source to the party they were paying. For example, if A were to pay B one token, who then went on to trade that token to C, then, when engaging in the trade with C, party B would need to send C A's IOU to B in addition to his own IOU to C. In concrete terms, participants must send the entire list of transactions that a given asset has moved through, from its original escrow account to the account of the sender. This proof would be provided to the insurance party in the case of assigning blame, to prevent a malicious party from bankrupting the insurance agent with malicious trades. The insurance-agent may also limit the total 'length' of a trade, in terms of how many accounts a particular asset passes through, in order to limit risk, and so the number of IOU's in a proof would be limited by this value.

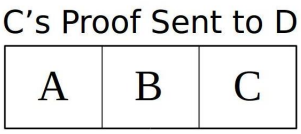


Figure 3: The array of IOU's which must be passed by C to D from figure 2

### 4.5 Payment 'Sourcing'

This protocol allows users to trade funds they received in other off-chain trades with any other member of the agreement - providing significant increases in off-chain payment freedom over the lightning network and other channel-based off-chain protocols. For example, if A received 10 bitcoin off-chain in a trade with B, A could then trade those bitcoin with C for any currency they agree upon. In the version being discussed currently, however, the protocol would limit A to send from that 10 bitcoin *only* to C if it wished to trade them.

### 4.6 Timeout Structure

Once IOU's are published to the smart-contract, they are instantly given a *time-out* by the contract, in terms of the number of *blocks* until those IOU's can be redeemed. Once an IOU's timeout has elapsed, its recipient can use it to pull funds directly from the sender's escrow account on the contract, and if this payment fails, the party to blame for its failure is instantly assigned.
Once an IOU is published to the contract, it is given an initial 3 block timeout period, during which the involved parties may post various messages to the contract in order to reach a conclusion on whether the IOU should be executed, and when. The intended activities for each block after an IOU is published are given below:

**Block 0**: Party $A$ publishes IOU $I$, in which party $B$ has promised to pay some funds to $A$, publicly on the smart-contract

**Block 1**: The following events may occur:

1. Party $B$ publishes the source (incoming) payment for $I$
2. Party $B$ proves that $A$ posted an old IOU, thereby nullifying it and publicly notifying the participants that party $A$ is guilty of foul-play
3. Party $B$ *sets* the source (by calling a function on the contract) for $I$ to an already-posted IOU, and $I$ is set to timeout after the payment $I$ is dependent on expires.

**Block 2**:

| Block | Party | Counterparty |
|---|---|---|
| 0 | | Publish IOU A where A.From = Party.Address |
| 1 | Publish IOU B where B.Amount >= A.Amount | |
| 2 | Set source of A to B | |
| 3 | | Attempt to withdraw if source not set or timer reaches 0 |

Figure 4: A visual depiction of the order of events when a user interacts with the contract for the first time

1. Party $B$ *sets* the source (by calling a function on the contract) for $I$ to an already-posted IOU, and $I$ is set to timeout after the payment $I$ is dependent on expires.

**Block 3**:

1. If the IOU wasn't nullified and the dependent payment for the IOU has been executed, or the *source* for the payment was the sender's own escrow funds (and therefore wasn't set), the timeout on the IOU expires and $A$ attempts to pull its funds. Either the payment succeeds, or it fails and the contract determines which party is at-fault.

Once the source on a contract has been set by its creator, if the given source is a valid one, the contract sets the contract timeout to dynamically timeout one block after its source does. The timeout is defined by the formula:

$$t(I_1) = t(I_0) + 1$$

where $t$ represents the *block-number* on which contract $I$ will expire, and $I_0$ is the listed source of $I_1$. This way, the payer has one block to collect the required

funds for the outgoing payment before the recipient will be able to siphon them from their escrow account.

### 4.7 Blame Assignment

Blame can be assigned via the *smart-contract* in the case where a given payment fails (a payee has insufficient funds to complete the payment) using the following logic:

```
if payment A fails:
    if A.source exists and A.source failed:
        A.AtFault = A.source.AtFault
    else:
        A.AtFault = A.From
```

There is one other situation in which blame needs to be assigned:

> If payment $A$ has been found to be an outdated and previously nullified payment, then $A.AtFault = A.From$

The proof of the correctness of this blame-assignment scheme is given in the appendix.

## 5 The Smart-Contract

The pseudocode for the smart-contract assumes a blockchain programming language with the following features:

- A *now*() function that returns the current system time, with all validators sharing a synchronized system time
- A *GetCurrentBlock*() function that returns the block-number of the current block it is mining on.
- User addresses encoded in a *address* datatype
- *map* datastructures which map keys (in the form of unsigned-integers) to values
- A *hash* function which can be applied to any byte array and which can be called on any struct or primitive datatype
- users may send assets to the contract along-with their method-calls, accessible to the language via the *msg.value* keyword
- events, which can be watched for on the blockchain and automatically initiate behavior on behalf of the watcher when an event is created with the *emit* keyword (similar to how they are used in solidity)

- The ability to verify a signature using public-key cryptography

The pseudocode itself is a modified version of Ethereum's *solidity* smart-contract programming language.

## 5.1 Contract Structure

```
Contract EscrowAgreement {
    users []address;
    balances map[address]uint;
    iouMap map[uint]IOU;
    timeoutMap map[uint]Timeout
    joinTimeout time;
    tradingEnd time;
    minEscrow uint;
    maxPaymentLength uint;
    thisCurrency uint;
```

The contract attributes are described here:

- *users*: a list of addresses of users who are participating in the agreement
- *iouMap*: a mapping of IOU-hashes to the IOU structs they represent
- *timeoutMap*: a mapping of IOU-hashes to their timeouts
- *joinTimeout*: a time after-which new addresses will not be able to join the agreement
- *tradingEnd*: the time after-which parties can withdraw their funds from escrow
- *minEscrow*: an unsigned-integer representing the minimum amount that must be deposited in order to participate in the agreement
- *maxPaymentLength*: an unsigned-integer encoding the maximum number of intermediates an asset may go through in a trade
- *thisCurrency*: an integer set by a pre-defined mapping that corresponds to the blockchain and asset the contract is on

## 5.2 IOU Structure

```
    struct IOU {
        Currency uint;
        Amount uint;
```

```
        From address;
        To address;
        Hashlock uint;
        Timelock uint;
        LastHash uint;
        Signature uint;
        Source uint
        Invalid bool;
        Withdrawn bool;
        AtFault address;
    }
```

The contract-representation of an IOU contains the following additional attributes:

- *Source*: denotes the hash of the payment that the current payment depends on, or *null* if the source is the sender's escrow account
- *Invalid*: a boolean value that denotes whether the IOU has been invalidated due to counterparty action
- *Withdrawn*: a boolean value that denotes whether the funds promised in the IOU have been moved via the smart-contract
- *AtFault*: an address responsible for payment failure, if applicable

These attributes do not need to be provided as part of the protocol and are assumed to initially be filled in by the contract as null values.

### 5.3 Timeout Structure

```
    struct Timeout {
        Startblock: blocknumber;
        Timeout: uint;
        SourceSet: bool;
        Source: uint
    }
```

The timeout is a datastructure that represents the timeout structure for an IOU published to the contract.

- *Startblock* is the block number of the block the IOU was published on
- *Timeout* is the number of blocks after the source times-out that this payment will time-out
- *SourceSet* is a boolean value denoting whether the IOU this timeout represents has had its source set
- *Source* is the hash of the source-contract, if applicable

### 5.4 Contract Publish Event

We emit the following event in the case of an IOU successfully being published to the smart-contract:

```
event ContractPublish {
    From address;
    Amount uint;
}
```

This enables protocol-participants to watch the contract for IOU's involving them and respond in-time allowed by the protocol.

### 5.5 Failure Event

We emit the following event in the case of withdraw failure during an *execute* call.

```
event Failure {
    Hash uint;
    AtFault address;
}
```

The event tells the watcher what the contract hash of the failed contract was, and which party was at-fault. A 3rd-party insurer could use these events to watch the blockchain for failed withdrawals and determine who it needs to compensate by examining the state of the virtual-machine with regards to the smart-contract.

### 5.6 Invalid Contract Posted Event

```
event InvalidPosted {
    Hash uint;
    From address;
}
```

This event is emitted in the event of a correct call of *contestIOU*.

### 5.7 Joining the Agreement and Posting IOUs

```
func join() {
    require msg.sender not in users
    require msg.value > minEscrow
    require now() <= joinTimeout
    users.append(msg.sender)
    balances[msg.sender] = msg.value
}
```

```
func postIOU(iou IOU, hashval uint) {
    require that iou.Signature is valid and matches
       IOU
    require iou.Currency == thisCurrency
    require not iou timelock expired
    if iou.Hashlock is not null {
        require hash(hashval) == iou.Hashlock
    }
    require iou.To != iou.From
    iouMap[hash(iou)] = iou
    timeout = Timeout {
        Startblock: currentBlock(),
        Timeout: 3,
        SourceSet: false,
    }
    timeoutMap[hash(iou)] = timeout
    emit ContractPublish(iou.From, iou.Amount)
}
```

Joining the contract agreement involves a contract check that the user is depositing at least the minimum amount before the timeout has expired, after which the contract adds the user to its list of all users and adds an entry for it in its *balances* map.

*Posting* an IOU first requires checking that the signature passed with the IOU matches the IOU itself, and checking that the currency on the IOU matches the blockchain the contract is on. Then, if the contract has a hash-time-lock, the contract checks to see if the conditions for the hash-timelock have been met. If all checks pass, the contract is stored as-per its hash and it is given the standard 3-block timeout given to all newly-posted valid IOU's.

### 5.8 Setting the Source of an IOU

```
func setSource(iou_hash uint,
        source_hash uint) {
    require source_hash in iouMap
    require iou_hash in iouMap
    iou = iouMap[iou_hash]
    require iou.From == msg.Sender
    timeout = timeoutMap[iou_hash]
    require not timeoutExpired(timeout)
    source_iou = iouMap[source_hash]
    require not source_iou.Invalid
    require iou.Source is null
    require iou.Amount >= source_iou.Amount
```

```
        iou.Source = source_hash
        timeout.SourceSet = true
    }
```

The *setSource* function is intended to be called by the *sender* of an IOU once its *recipient* has posted it to the contract. In order for an IOU to have its source set, it must not have timed-out, the source iou must have been published on the contract, and only the paying party of an IOU can set its source. In addition, the source of an IOU must be able to pay for the entire promised amount of digital asset.

### 5.9 Contesting an Invalid IOU

```
    func contestIOU(contract_hash uint,
            iou IOU,) {
        require contrract_hash in iouMap
        contest = iouMap[contract_hash]
        require iou.Signature is valid and matches iou
        require iou.From == contest.To
        require hash(contest) == iou.LastHash
        contest.Invalid = true
        contest.AtFault = contest.To
        emit InvalidPosted(contract_hash, contest.To)
    }
```

The *contestIOU* function should be called by the *sender* of an already-revoked IOU once its *recipient* has posted it to the smart-contract. The contesting party should include the iou sent by the poster invalidating the old iou (which it should have saved) which includes the old iou's hash in its *LastHash* field.
Once an IOU is successfuly contested, it is marked as invalid and its poster is considered at-fault for any payments involving the old iou.

### 5.10 Executing an IOU

```
    func execute(iou_hash uint, hashval uint) {
        require iou_hash in iouMap
        iou = iouMap[iou_hash]
        require not iou.Withdrawn
        require msg.sender == iou.To
        if iou.Hashlock is not null {
            require hashlockAccepted(iou, hashval)
        }
        require not iou.Invalid
```

```
        require not iou timelock expired
        timeout = timeoutMap[iou_hash]
        require timeoutExpired(timeout)
        if iou.Source is not null {
            source_iou = iouMap[iou.Source]
            // we only withdraw from an iou
            // if its source has been withdrawn
            if not source_iou.Withdrawn {
                if source_iou.Invalid {
                    iou.AtFault = iou.From
                } else {
                    if source_iou.AtFault is not null {
                        iou.AtFault = source_iou.AtFault
                    } else {
                        iou.AtFault = iou.From
                    }
                }
                emit Failure(iou_hash, iou.AtFault)
                return
            }
        }
        if balances[iou.From] < iou.Amount {
            iou.AtFault = iou.From
            emit Failure(iou_hash, iou.AtFault)
        } else {
            balances[iou.From] -= iou.Amount
            balances[iou.To] += iou.Amount
            iou.Withdrawn = true
        }
    }
```

The *execute* function should be called by the recipient of funds under a valid and timed-out IOU. Once all requirements are met, the caller receives all funds under the IOU. The requirements are:

- The IOU must be valid
- The IOU must have timed out
- If there is a hashlock, the caller must provide the correct hash value
- If there is a timelock, the timelock must have not elapsed
- The IOU must not have been withdrawn already
- The IOU source must have been withdrawn

If the funds cannot be paid, blame-assignment functions as-follows:

- If the iou has a valid source and that source-payment also failed, then the AtFault party is the same as the party who was at-fault for the source payment
- If the iou payment does not have a source, an invalid source, or its source-payment executed successfully, then the AtFault party is the payer (*From* attribute) of the IOU

The proof of the correctness of this blame-assignment mechanism is given in the Appendix

### 5.11 Entire Smart-Contract:

```
Contract EscrowAgreement {
    users []address;
    balances map[address]uint;
    iouMap map[uint]IOU;
    timeoutMap map[uint]Timeout
    joinTimeout time;
    tradingEnd time;
    minEscrow uint;
    maxPaymentLength uint;
    thisCurrency uint;

    struct IOU {
        Currency uint;
        Amount uint;
        From address;
        To address;
        Hashlock uint;
        Timelock uint;
        LastHash uint;
        Signature uint;
        Source uint
        Invalid bool;
        Withdrawn bool;
        AtFault address;
    }

    struct Timeout {
        Startblock: blocknumber;
        Timeout: uint;
        SourceSet: bool;
```

```
    Source: uint
}

event Failure {
    Hash uint;
    AtFault address;
}

event ContractPublish {
    From address;
    Amount uint;
}

event InvalidPosted {
    Hash uint;
    By address;
}

func join() {
    require msg.sender not in users
    require msg.value > minEscrow
    require now() <= joinTimeout
    users.append(msg.sender)
    balances[msg.sender] = msg.value
}

func postIOU(iou IOU, hashval uint) {
    require that iou.Signature is valid and matches
       IOU
    require iou.Currency == thisCurrency
    require not iou timelock expired
    if iou.Hashlock is not null {
        require hash(hashval) == iou.Hashlock
    }
    require iou.To != iou.From
    iouMap[hash(iou)] = iou
    timeout = Timeout {
        Startblock: currentBlock(),
        Timeout: 3,
        SourceSet: false,
    }
    timeoutMap[hash(iou)] = timeout
    emit ContractPublish(iou.From, iou.Amount)
}

func setSource(iou_hash uint,
```

```
        source_hash uint) {
    require source_hash in iouMap
    require iou_hash in iouMap
    iou = iouMap[iou_hash]
    require iou.From == msg.Sender
    timeout = timeoutMap[iou_hash]
    require not timeoutExpired(timeout)
    source_iou = iouMap[source_hash]
    require not source_iou.Invalid
    require iou.Source is null
    require iou.Amount >= source_iou.Amount
    iou.Source = source_hash
    timeout.SourceSet = true
}


func contestIOU(contract_hash uint,
        iou IOU,) {
    require contrract_hash in iouMap
    contest = iouMap[contract_hash]
    require iou.Signature is valid and matches iou
    require iou.From == contest.To
    require hash(contest) == iou.LastHash
    contest.Invalid = true
    contest.AtFault = contest.To
    emit InvalidPosted(contract_hash, contest.To)
}

func execute(iou_hash uint, hashval uint) {
    require iou_hash in iouMap
    iou = iouMap[iou_hash]
    require not iou.Withdrawn
    require msg.sender == iou.To
    if iou.Hashlock is not null {
        require hashlockAccepted(iou, hashval)
    }
    require not iou.Invalid
    require not iou timelock expired
    timeout = timeoutMap[iou_hash]
    require timeoutExpired(timeout)
    if iou.Source is not null {
        source_iou = iouMap[iou.Source]
        // we only withdraw from an iou
        // if its source has been withdrawn
        if not source_iou.Withdrawn {
            if source_iou.Invalid {
```

```
                    iou.AtFault = iou.From
                } else {
                    if source_iou.AtFault is not null {
                        iou.AtFault = source_iou.AtFault
                    } else {
                        iou.AtFault = iou.From
                    }
                }
                emit Failure(iou_hash, iou.AtFault)
                return
            }
        }
        if balances[iou.From] < iou.Amount {
            iou.AtFault = iou.From
            emit Failure(iou_hash, iou.AtFault)
        } else {
            balances[iou.From] -= iou.Amount
            balances[iou.To] += iou.Amount
            iou.Withdrawn = true
        }
    }

    func timeoutExpired(timeout Timeout) bool {
        totblocks = 0
        while timeout.SourceSet && timeout.Source != null
            {
            totblocks += timeout.Timeout
            timeout = timeoutMap[timeout.Source]
        }
        totblocks += timeout.Timeout
        end_block = totblocks + timeout.Startblock
        return end_block <= getCurrentBlock()
    }

    func withdraw() {
        require trading period over
        amount = balances[msg.sender]
        balances[msg.sender] = 0
        send amount to msg.sender
    }
}
```

## 5.12 Expanding the Construction to Allow for Batching Payments

In a situation where parties are conducting trades off-chain and in which parties act as intermediaries for payments to pass through, it seems natural to allow parties to batch incoming IOU's into one or more outgoing IOUs and to break down incoming IOU's into smaller outgoing IOU's. However when allowing for such transactions, the accounting process isn't as simple, and it isn't apparent if such a change would preserve the timeout structure and the proper detection of double-spends. In order to provide batching and splitting functionality while allowing the protocol to remain relatively simple and correct, we introduce additions to the IOU struct and contract, as well as a special kind of IOU contract transaction that is used to split up payments.

First, we add a new field to IOU's to tell the contract whether or not the given IOU is a 'batched' payment. So, the modified (contract version) IOU would be defined as follows:

```
struct IOU {
        Currency uint;
        Amount uint;
        From address;
        To address;
        Hashlock uint;
        Timelock uint;
        LastHash uint;
        Source []uint
        Signature
        Invalid bool;
        Withdrawn bool;
        AtFault []address;
        Batched bool;
    }
```

- *Source* is now a list of IOU hashes which the given IOU has as source
- *AtFault* is now a list of at-fault addresses for the failure of a given payment
- *Batched* tells the contract whether the IOU has been formed by combining more than one IOU

In order to batch payments, we add a method to the contract which creates one or more unique type of IOU's that represent a transaction from a user to themselves. The sources for these IOUs are incoming IOUs which are being combined, and the *To* and *From* values are both the address of the user who wishes to combine or break-up funds.

We make the same modification to the Timeout struct:

```
    struct Timeout {
        Startblock: blocknumber;
        Timeout: uint;
        SourceSet: bool;
        Source: []uint;
        Batched: bool;
    }
```

### 5.12.1 Creating Batched IOU's

In order to create batched IOU's, users call the following contract method

```
func combineAndCreate(inputs []IOU, hashvals []uint,
    outputs []IOU) {
    tot_in = 0
    for each input in inputs {
        require that input.proof is  valid
        require input.To == msg.Sender
        require input.currency == this.currency
        require not input.Timelock expired
        require input.Hashlock matches hashval if not
            null
        tot_in += input.Amount
    }
    tot_out = 0
    for each output in outputs {
        require output.currency == this.currency
        require output.To == msg.Sender
        require output.From == msg.Sender
        tot_out += output.Amount
    }
    require tot_in >= tot_out
    for each input in inputs {
        if hash(input) not in iouMap {
            iouMap[hash(input)] = input
        } else {
            input_iou = iouMap[hash(input)]
            require not input_iou.Withdrawn
        }
    }
    for each output in outputs {
        output.Source = inputs
        timeoutMap[hash(output)] = Timeout {
            Startblock: currentBlock(),
            Timeout: 1,
```

```
            SourceSet: true,
            Source: inputs,
            Batched: true,
        }
        iouMap[hash(output)] = output
    }
}
```

The contract first checks that the input IOU's and output IOU's are valid and that the sum of the outputs does not exceed that of the inputs. If these requirements are met, the input IOU's are added to the contract. Then, special output IOU's are created that have these input IOU's as their source, and the user calling the function as both their sender and receiver. The calling user may then use these output IOU's as payment-sources for other IOU's.

In terms of the timeout structure described in section 4.6, if a user wishes to combine IOUs to use as a source for another payment, they may call the *combineAndCreate* function on the contract in the first block after an IOU has been posted rather than the *postIOU* function, and the result will be that they have an IOU on the contract that they can set as the source for an outgoing payment in the next block. So this functionality does not change the timeout structure of the protocol.

We also modify the logic of the *timeoutExpired* function as follows:

```
func timeoutExpired(timeout Timeout) bool {
    end_block = getEndBlock(timeout)
    return end_block <= getCurrentBlock()
}

func getEndBlock(timeout Timeout) uint {
    if timeout.Batched {
        var timeouts []uint
        for source in timeout.Source {
            batch_timeout = timeoutMap[source]
            timeouts.append(getEndBlock(batch_timeout))
        }
        last_timeout = max(timeouts)
        return last_timeout + timeout.Timeout
    }
    if timeout.SourceSet && timeout.Source != null {
        source = timeoutMap[timeout.Source[0]]
        return getEndBlock(source) + timeout.Timeout
    }
    return timeout.Startblock + timeout.Timeout
}
```

As we trace back through the source-payments, we must now account for the existence of intermediary 'batched' payments. When a batched payment is

reached, the timeout value used by the contract is the one that is last out of all of its source payments.

Batched payments also change the logic of blame assignment slightly, as if a batched payment is the source of a payment, and more than one of its incoming payments failed, then all of those payments that failed would be considered *at-fault* for the payment which the batched payment was the source of.

This also means that, after submitting a batched payment to themselves, a party must call the *execute* function on that payment when the timeouts on all of its sources expire, so that if the batched payment fails, blame can be assigned and used for payments that use it as a source.

We modify the logic of blame-assignment in the *execute* function in the following ways:

```
func execute(iou_hash uint, hashval uint) {
        iou = iouMap[iou_hash]
        require not iou.Withdrawn
        require msg.sender == iou.To
        require hashlockAccepted(iou, hashval)
        require not iou.Invalid
        require not iou timelock expired
        timeout = timeoutMap[iou_hash]
        require timeoutExpired(timeout)
        if iou.Batched {
            var at_fault []address
            for source in iou.Source {
                source_iou = iouMap[source]
                if not source_iou.Withdrawn {
                    if source_iou.Invalid {
                        at_fault.append(source_iou.From)
                    } else {
                        if source_iou.AtFault is not null
                            {
                            at_fault.append(source_iou.
                                AtFault)
                        } else {
                            at_fault.append(source_iou.To
                                )
                        }
                    }
                }
            }
            if len(at_fault) > 0 {
                iou.Withdrawn = false
                iou.AtFault = at_fault
                emit Failure(iou_hash, iou.AtFault)
                return
```

```
            }
        } else if iou.Source is not null {
            source_iou = iouMap[iou.Source]
            // we only withdraw from an iou
            // if its source has been withdrawn
            if not source_iou.Withdrawn {
                if source_iou.Invalid {
                    iou.AtFault = [iou.From]
                } else {
                    if source_iou.AtFault is not null {
                        iou.AtFault = [source_iou.AtFault
                            ]
                    } else {
                        iou.AtFault = [iou.From]
                    }
                }
                emit Failure(iou_hash, iou.AtFault)
                return
            }
        }
        if balances[iou.From] < iou.Amount {
            // this case signifies a double-spend of
            // promised inputs, in both a batched and
            // non-batched case.
            iou.AtFault = iou.From
            emit Failure(iou_hash, iou.AtFault)
        } else {
            if not iou.Batched {
                // We only move funds between accounts in
                    the case of non-batched payments
                balances[iou.From] -= iou.Amount
                balances[iou.To] += iou.Amount
            }
            iou.Withdrawn = true
        }
    }
```

In the case where a batched payment is being executed, the contract first checks to see that all of its source-payments succeeded. If any of them failed, the contract adds all of the failed payments to the batched payment's *AtFault* field and marks it as incomplete. In the case where the source iou wasn't withdrawn but it doesn't have an at-fault value, and so *execute* was never called, then the user batching the payments is at-fault for never calling execute.

If the batched payment has all successful source payments, then the contract checks that the funds in the users account match the amount being bundled. If they do, the payment is marked successful (no funds need actually be trans-

ferred, as they are already in that user's escrow balance). If they do not, then the user has double-spent at-least one of those source payments and thus is at-fault for the batched payment.

# 6 Appendix

## 6.1 Proof of Correctness of Blame-Assignment

The blame-assignment logic of the smart-contract is reproduced here:

```
if iou.Source is not null {
    source_iou = iouMap[iou.Source]
    // we only withdraw from an iou
    // if its source has been withdrawn
    if not source_iou.Withdrawn {
        if source_iou.Invalid {
            iou.AtFault = iou.From
        } else {
            if source_iou.AtFault is not null {
                iou.AtFault = source_iou.AtFault
            } else {
                iou.AtFault = iou.From
            }
        }
        return
    }
}
if balances[iou.From] < iou.Amount {
    iou.AtFault = iou.From
} else {
    balances[iou.From] -= iou.Amount
    balances[iou.To] += iou.Amount
    iou.Withdrawn = true
}
```

Once the timeout on an IOU has expired, the user may call the *withdraw* function and attempt to pull funds from the payee's escrow-balance.

For the rest of the section we assume that the currency is fixed, as IOUs on different blockchains do not affect one another, and so if we prove correctness for an arbitrary blockchain, we've proved correctness for the entire protocol.

### 6.1.1 Protocol Model

Assume that at a given instant $t$, a party $P$ has

- an escrow account on the smart-contract with $e_P$ units of the given asset

- $n_i$ incoming IOU's, labeled $I_1, ..., I_{n_i}$, where each $I_{n_k}$ has *not* been revoked by any subsequent IOU.
- $n_o$ outgoing IOU's, labeled $O_1, ..., O_{n_o}$, where each $O_{n_k}$ has *not* been revoked by any subsequent IOU.

We assume that these IOU's will eventually be published to the contract and executed in an order that depends on the contract logic and the user's behavior. That is, they will be executed in the order $H_{l_1}, H_{l_2}, ..., H_{l_{n_o+n_i}}$, where each $H_{l_k}$ refers to a unique incoming or outgoing IOU.

**Definition 6.1.1.** *: An IOU $H_{l_k}$ from party A fails if the contract executes A's incoming and outgoing payments in the order $H_{l_1}, ..., H_{l_k}$ and (here we will use $H_k$ as shorthand for $H_k$.Amount)*

$$e_A + \sum_{i=1}^{l_k-1} \mathbb{1}\{H_i \text{ is incoming and doesn't fail}\} * H_i - \sum_{i=1}^{l_k-1} \mathbb{1}\{H_i \text{ is outgoing and doesn't fail}\} * H_i - H_{l_k} < 0. \quad (1)$$

**Lemma 6.1.1.** *If an IOU fails according to this definition, it fails in the contract (a Failed event is generated).*

*Proof.* In the situation where equation (1) holds, then either the check in the *execute* function that determines if the escrow account has sufficient balance fails, and so a *Failed* event is generated and no funds are transferred, or a *Failed* event is generated prior to that. □

**Definition 6.1.2.** *An IOU $I_1$ is directly-responsible for the failure of payment $I_2$ if $I_1$ was attempted to be executed before $I_2$, $I_1$ failed, and if $I_1$ had not failed, then $I_2$ would not have failed.*

**Definition 6.1.3.** *A party P behaves honestly if*

1. *$e_P + \sum_{I_i \text{ is incoming}} I_i - \sum_{I_o \text{ is outgoing}} I_o > 0$ at any time t, where all are $I_k$ are un-executed, un-revoked payments*
2. *P follows the protocol outlined in section 4*

**Theorem 6.1.1.** *Honest parties will not be identified as at-fault by the blame-assignment logic of the smart-contract*

*Proof.* Assume an outgoing IOU from $A$, $I_A$, fails, and that $A$ is an honest party. Also assume incoming and outgoing IOU's for $A$ are attempted to be executed (from the start of trading) in the order $I_1, I_2, ..., I_A$

Assume $I_A$ fails. Let all of the outgoing IOU's successfully executed before $I_A$ be $I_{o_1}, ..., I_{o_k}$. We can divide these into two groups, $\{I_{o_j}\}$ and $\{I_{o_k}\}$, where the first consists of payments with a source set and the second consisting of

payments without a source set, therefore coming from the escrow account of $A$. Because $A$ follows the part of the protocol outlined in section 4.6, $\forall I_{o_j} ! \exists I_{i_j}$ where $I_{i_j}$ is incoming, $I_{i_j} >= I_{o_j}$, and $I_{i_j}$ is set by $A$ as the source for $I_{i_j}$. Let $l$ be the leftover value when taking $\sum I_{i_j} - I_{o_j}$. By equation (1) we have $l + e_A - \sum I_{o_k} - I_A < 0 \rightarrow I_A + \sum I_{o_k} > e_A + l$. If $I_A$ were an escrow payment, then $A$ would've spent more from escrow than it was able. However, A is honest, so it must be the case that $I_A$ had a source-payment. Also given that $A$ is honest, it didn't double-spend the source-payment to $I_A$. However, if the source of $I_A$ succeeded, $I_A$ would be cancelled by $l$ and we'd get $\sum I_{o_k} > e_A + \epsilon$ where $\epsilon > 0$ and $\sum I_{o_k} < e_A$, a contradiction.

So, the source-payment for $I_A$ must have failed. If the source-payment for $I_A$ failed, then by the contract logic, some party other than $A$ would've been at-fault for its failure (as long as the source wasn't a payment from $A$ to itself, which is not allowed).

As-per the contract, when $I_{i_A}$ was executed and failed, its at-fault value must have been set to some address not belonging to A, so A would not be considered at-fault. An honest party couldn't be considered at-fault for its own outgoing payment. Therefore, it couldn't be considered at-fault for *any* payment. □

**Definition 6.1.4.** *A party $P$ is at-fault for payment $I$ if $\exists I_P$ such that $I_P.From = P.Address$ and $\exists I_1, ..., I_n$ published IOU's that all failed, $I_P$ is directly-responsible for $I_1$, $I_1$ is directly-responsible for $I_2$, and so on, $I_P$ is directly-responsible for $I$, and there is no published $I_0$ such that $I_0$ is directly-responsible for $I_P$.*

We see here that the definition of at-fault leaves some ambiguity. Multiple parties could be considered at-fault for the failure of a single payment.

**Lemma 6.1.2.** *If a party $P$ is at-fault for payment $I$ according to the definition, and all other members of the agreement were honest, then $P$ will be labeled at-fault for payment $I$.*

*Proof.* If $I_P$ is directly-responsible for $I_1$'s failure, then it must be the case that $I_P$ was set as the source of $I_1$. This is because of equation (1) along with the fact that the creator of $I_1$ followed the protocol. Assume the string of payments attempted to be executed involving $I_1.From$ leading up to $I_1$ was $H_1, ..., H_k$. We know that for each sourced outgoing payment in $\{H_i\}$ there is an equivalently-valued incoming payment, so we can reduce equation () to $l + e_A - \sum N_i - I_1 < 0$, where $N_i$ are all escrow payments. $I_1 + \sum N_i > e_A + l$ so $I_1$ couldn't have been an escrow-payment. Since the only source that could have failed was $I_P$ (every other member followed the protocol), it must be the case that $I_P$ was set as the source for $I_1$, and since no payment was directly responsible for $I_P$'s failure, it must be the case that $I_P$'s source was either successful or didn't exist, and so the contract would label $P$ as at-fault for $I_P$ and therefore for $I_1$.
If $I_P$ is *at-fault* for $I_k$, and $I_k$ is directly-responsible for $I_{k+1}$, then with similar logic we see that $I_k$ must have been the source for $I_{k+1}$. $l + e_{k+1} - \sum N_i - I_{k+1} > 0$, $I_{k+1}$ must have been sourced, and $I_k$ is the only candidate for $I_{k+1}$'s source, and so $P$ is labeled *at-fault* for $I_{k+1}$.

We've shown that $P$ will be labeled *at-fault* for payment $I$ under the given conditions. □

## 6.2 Correctness of At-Fault assignment under Bundled payments

With bundled payments, the notion of blame must change slightly to accommodate philosophical questions regarding fault assignment. If payments $P_1$ and $P_2$ are bundled payments that pay out $P_3$, $P_4$, and $P_5$, and $P_1$ and $P_2$ both fail, then which of $P_1$ and $P_2$ should be considered at-fault for the failure of payments $P_3$, $P_4$, and $P_5$? Our solution is to name $P_1$ and $P_2$ both responsible for the failure of all three, and this motivates the modified definition of *at-fault* when bundled payments are involved.

**Definition 6.2.1.** *An IOU $I_1$ is partially-responsible for the failure of payment $I_2$ if $I_1$ was executed before $I_2$, and $I_1$ failed, and there is a set of payments $I_1 \in \{I_k\}$ which had they not failed, then $I_2$ would not have failed.*

We modify the at-fault definition in the bundled-payments model:

**Definition 6.2.2.** *A party $P$ is at-fault for payment $I$ if $\exists I_P$ such that $I_P.From = P.Address$ and $\exists I_1, ..., I_n$ published IOU's that all failed, $I_P$ is partially-responsible for $I_1$, $I_1$ is partially-responsible for $I_2$, and so on, $I_P$ is partially-responsible for $I$, and there is no published $I_0$ such that $I_0$ is partially-responsible for $I_P$.*

The proof of the correctness of modified at-fault assignment is essentially the same as the proof of correctness of the original algorithm, however slightly more machinery is required to consolidate the difference between the two algorithms.

**Theorem 6.2.1.** *Honest parties will not be identified as at-fault by the blame-assignment logic of the smart-contract*

*Proof.* Assume an outgoing IOU from $A$, $I_A$, fails, and that $A$ is an honest party. Also assume incoming and outgoing IOU's for $A$ are executed (from the start of trading) in the order $I_1, I_2, ..., I_A$

Assume $I_A$ fails. Let all of the outgoing IOU's executed before $I_A$ be $I_{o_1}, ..., I_{o_k}$. In the case where the source of $I_A$ is not batched, the proof is the same. If $I_A$'s source is batched, then because $A$ follows the part of the protocol outlined in section 5.9, there are payments $I_1, ..., I_n$ set as the source of $I_A$ by $A$. The sum of the values of those sources is at least that of $I_A$, so the sum in the previous proof remains positive. That means that if $I_A$ fails, one of those payments must have been attempted to be executed and failed, and so the at-fault for $I_A$ is not $A$.

So, an honest party $A$ couldn't be considered at-fault for an outgoing payment. Therefore, it couldn't be considered at-fault for *any* outgoing payment. □

Due to the weaker notions of partial-responsibility, we are unable prove the equivalent of lemma 6.1.2 in this case.

### 6.3 Safety and Correctness of the HTLC Protocol

The HTLC protocol is nearly identical to *lightning*'s[1], and a proof of its correctness can be found in the lightning whitepaper.

## References


[1] Joseph Poon and Thaddeus Dryja, *The Bitcoin Lightning Network*, https://lightning.network/lightning-network-paper.pdf

[2] https://raiden.network/101.html