Emulation of Hash-Time-Locked Contracts of the Lightning network by a trusted, but publically auditable escrow service

C. J. Plooy (cjp@ultimatestunts.nl)

April 19, 2015

Contents

1 Introduction 2

2 Bi-directional microtransaction channels 3

3 Hash-Time-Locked Contract emulation 3

3.1 The output script . . . 4

3.1.1 Let E sign a token instead of the transaction . . . 4

3.1.2 Delay the signing by E . . . 5

3.2 The Transaction Conditions Document . . . 5

4 Data structures summary 6

5 Transaction sequences 6

5.1 Deposit into the channel . . . 6

5.2 Lock funds for a microtransaction . . . 7

5.3 Commit by settling . . . 7

5.4 Rollback by settling . . . 7

5.5 Commit by escrow . . . 7

5.6 Rollback by escrow . . . 8

5.7 Withdraw from the channel (with cooperative neighbor) . . . 9

5.8 Withdraw from the channel (with uncooperative neighbor) . . . 9

6 Remaining vulnerabilities **9**
6.1 Positive escrow failure 9
6.2 Negative escrow failure 11
6.3 Malleability of the deposit transaction 11

7 Conclusions and recommendations **12**
7.1 Comparison with the Lightning design 12
7.2 Bitcoin improvements 12

1 Introduction

The Lightning network[1] is a design for a decentralized, scalable network that allows for fast, cheap Bitcoin transactions without requiring trusted third parties. However, it requires the presence of new functionality in Bitcoin: without this new functionality, the Lightning network can not exist. So, in order to make the Lightning network a reality, the Bitcoin community must be convinced to accept this new functionality. While none of this new functionality is known to have any controversial characteristics, some of it has, so far, no use case outside the Lightning network [1].

Since any new functionality makes Bitcoin more complex, and more complex systems have more places where vulnerabilities can exist, the Bitcoin community might be reluctant to accept new functionality, unless convincing evidence is provided about the value of the new functionality. However, so far, the Lightning network only exists on paper, and has not been demonstrated to be useful in real-life conditions. This creates a *catch-22* situation: in order to convince the Bitcoin community, we would like to make a working implementation of the Lightning network, but in order to do that, we need to convince the Bitcoin community to include the required functionality.

There are a couple of ways to escape this catch-22 situation:

1. The Bitcoin community might accept the required functionality, even without a working Lightning network.

2. It is possible to demonstrate the Lightning network on an alt-coin (possibly one specifically designed for this purpose), or on a side-chain, once side-chains are realized.

3. It might be possible to emulate the missing functionality, with some loss of desirable properties, using only the already existing functionality of Bitcoin. This would allow Bitcoin users to become familiar with Lightning-like technology, while simultaneously increasing the pressure to "fix" the loss of desirable properties by including the missing functionality in Bitcoin.

[1] More specifically, I am thinking of the new SIGHASH types.

This paper describes a design that follows the third approach. The "no trusted third parties" requirement is relaxed, by introducing a partially trusted escrow party, which can be audited by arbitrary third parties. The design is such that, after the missing functionality is introduced to Bitcoin, the migration to a full-featured Lightning network can happen gradually: pairs of neighboring nodes are free to choose when they upgrade their link to a real Lightning link, and during the migration period, transactions can be routed through both old-style and new-style channels (in any order).

2 Bi-directional microtransaction channels

As a starting point, Alex Akselrod's bi-directional microtransaction channel[2] is used. Assuming Alice (A) and Bob (B) share a channel, and initial funds for the channel are provided by Alice:

- A deposit transaction (D) sends the initial funds from Alice to a 2-of-2 multisignature output that requires sinatures from both Alice and Bob.
- A withdraw transaction (W) spends the output of the deposit transaction, and sends a part of it to Alice and a part to Bob. The initial version of W sends all funds back to Alice and none to Bob.

While the channel is in use, W is *not* published on the block chain, so it can be updated after every micro-transaction. The initial version of W sends all funds back to Alice; subsequent microtransactions can change the balance. Every version of W has a lock time: as soon as the lock time expires, one of the versions of W can be published on the block chain.

To ensure that it is always the *last* version that gets published, each time a microtransaction is performed, the payer side gives his signature of the updated W to the receiver side. Each time the direction of the channel is reversed, the lock time is reduced. The effect is that the side who last received funds is the first to be able to spend the output of D, and it is in the interest of that side to spend the output of D with the most recent version of W.

3 Hash-Time-Locked Contract emulation

Since it is impossible to implement Hash-Time-Locked Contracts (HTLCs) with the current functionality of Bitcoin, this functionality is instead enforced by an escrow service (E). While a transaction is locked, but not yet settled, W contains a third[2] output, besides the outputs to Alice and Bob; this third output corresponds to the to-be-transferred funds. Depending on judgement by the escrow service, this output should become either spendable by Alice or by Bob. The escrow service should decide this based on whether it has received the

[2]There can be more than one transaction locked at the same time: in that case, there will be a fourth, fifth etc. output, all having the same structure.

transaction token (T) before a certain time-out. The transaction token can be recognized because it hashes to a value H that is known when locking: $H = h(T)$, where h is a secure hash function.

3.1 The output script

The output script of W that holds the funds corresponding to the locked transaction determines what needs to be signed by who. The standard way of implementing escrow in Bitcoin is with a 2-of-3 multisignature output, requiring two signatures corresponding with two of three given public keys; in this case, the public keys of Alice, Bob and the escrow service.

With the existing SIGHASH types, E can not generate a signature for spending W without knowing the transaction ID of W. Since the transaction ID of W depends on the signatures of both A and B in W's input ScriptSig, both these signatures need to be given to E before E can sign a transaction that spends the output of W. This creates an unsolveable transaction malleability problem (besides the other types of transaction malleability, which also apply here): when W is published on the block chain, the side who publishes it (either A or B) can change his own signature to another valid signature[3] of himself; this changes the transaction ID of W, and makes E's signature useless.

Two methods are considered for avoiding the unsolveable malleability problem:

3.1.1 Let E sign a token instead of the transaction

For this method, the output script would be something like this:

```
OP_IF
  OP_DUP OP_HASH160 <pubKeyHash_E> OP_EQUALVERIFY <token_A> OP_CHECKSIGDATAVERIFY
  OP_DUP OP_HASH160 <pubKeyHash_A> OP_EQUALVERIFY OP_CHECKSIG
OP_ELSE
  OP_DUP OP_HASH160 <pubKeyHash_E> OP_EQUALVERIFY <token_B> OP_CHECKSIGDATAVERIFY
  OP_DUP OP_HASH160 <pubKeyHash_B> OP_EQUALVERIFY OP_CHECKSIG
OP_ENDIF
```

token_A and token_B should be unique tokens, which are only used for this transaction. If a combination of token and address of E is re-used anywhere, a signature of E for one transaction might be abused in another transaction. Because of this, it is recommended that the tokens are calculated in some deterministic way (e.g. with a secure hash), based on all relevant information, so that neither party can re-use an already signed token in a different transaction to his own advantage.

The corresponding scriptSig would be one of the following:

[3]Bitcoin uses ECDSA for digital signatures. The creation of an ECDSA signature uses a random value; using a different random value results in a different signature, even when re-signing the same data with the same key.

```
<sig_A> <pubKey_A> <sig_E1> <pubKey_E> 1
<sig_B> <pubKey_B> <sig_E2> <pubKey_E> 0
```

The output script requires an op-code like OP_CHECKSIGDATAVERIFY, which verifies that a signature corresponds with a given public key, and *signs a given piece of data on the stack* (in this case: token_A or token_B). Unfortunately, such an op-code does not exist yet, so this method is discarded.

3.1.2 Delay the signing by E

In this method, a conventional 2-of-3 multisignature output script is used, but the malleability problem is avoided by delaying the signing by E until after the lock time of W has expired, and W has been included into the block chain. In order to provide a faster resolution, E can sign a *promise* that it will sign a spend of W in a certain way. The promise can be signed as soon as E has determined the outcome of the transaction: as soon as it receives T, or as soon as the time-out happens (whichever happens first).

Note that E does not have to remember all its promises: it only needs to remain capable of verifying its own signatures, when it receives back its own signed promise from either Alice or Bob.

Since this method works with current Bitcoin functionality, this method is selected.

3.2 The Transaction Conditions Document

How does E know the hash value H, the time-out value, and what to sign? Simple: Alice and Bob create a *Transaction Conditions Document* (TCD), containing all this information. The TCD should contain:

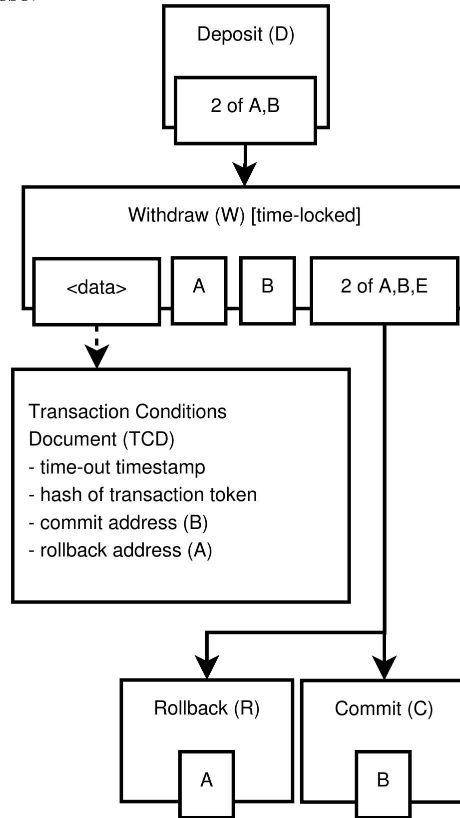
- The time-out timestamp
- H, the hash of the transaction token T
- The address where to send the funds in case of commit
- The address where to send the funds in case of rollback

Now, it is necessary to cryptographically link the TCD to a specific output of W. An initial thought would be to include W and the output index in the TCD, but that would mean that the TCD needs to be updated quite often when W is updated. If E's *promise* also signs TCD, then this would also invalidate old promises.

Instead, linking is done the other way around: W contains information about which TCDs apply to which of its outputs. This information is encoded in an extra OP_RETURN output of W, which contains the secure hash of all TCDs concatenated together (in the same order as the outputs): $L = [TCD_0, TCD_1, ...]$ and $h(L)$ is included in W. This has the additional advantage that, by signing W, Alice and Bob automatically also sign the contents of the TCDs.

4 Data structures summary

The following diagram summarizes the data structures that follow from sections 2 and 3. Note that the Commit and Rollback transactions (C and R) have been added: on request by Alice or Bob, the escrow service will sign exactly one of these.



5 Transaction sequences

5.1 Deposit into the channel

Alice wants to use some of her funds to create a channel to Bob. This sequence is the same as in a traditional microtransaction-channel.

W_0 is the first version of W, and it sends all funds back to Alice. D is a deposit transaction that spends some funds from Alice; therefore, it is signed by Alice.

From	To	Data
A	B	W_0
B	A	$sig_B(W_0)$
A	block chain	D

5.2 Lock funds for a microtransaction

Alice wants to lock funds for a transaction to Bob.

W_L is an updated version of W, which has the funds locked. L is the list of all TCDs of W_L.

From	To	Data
A	B	W_L,L,$sig_A(W_L)$

5.3 Commit by settling

Alice , who has previously locked funds for a transaction to Bob, receives the transaction token T and wants to settle the microtransaction without involvement of the escrow service.

W_S is an updated version of W, in which the locked funds are unlocked and assigned to Bob. L is the list of all TCDs of W_S.

From	To	Data
B	A	T
A	B	W_S,L,$sig_A(W_S)$

This is the normal way of unlocking a transaction. If this way is followed, the escrow service is not involved: it does not even know about the existence of the transaction.

5.4 Rollback by settling

Alice has previously locked funds for a transaction to Bob, but Bob has not received the transaction token T before the time-out. Bob wants to settle the microtransaction without involvement of the escrow service.

W_S is an updated version of W, in which the locked funds are unlocked and assigned to Alice. L is the list of all TCDs of W_S.

From	To	Data
B	A	W_S,L,$sig_B(W_S)$

This is a common way of unlocking a transaction in case something downstream (on Bob's side) has gone wrong. In this case, the escrow service is not involved in the transaction, at least not for this channel: it might be involved in dispute resolution for the downstream channel where the problem occurred.

5.5 Commit by escrow

Alice has previously locked funds for a transaction to Bob. Bob receives the transaction token T and sends it to Alice before the time-out, but Alice does not immediately settle the transaction (she is uncooperative).

TCD describes the conditions of the microtransaction.

From	To	Data
B	A	T
B	E	T,TCD
E	B	$sig_E([\text{"}B\text{"}, TCD])$
B	A	$sig_E([\text{"}B\text{"}, TCD])$

After this, Bob blocks new transactions on the channel until Alice cooperates with settling existing transactions correctly. Normally, the promise by the escrow service will convince Alice to settle, so that no further involvement by the escrow service is required.

If Alice remains uncooperative until the lock time of W expires, Bob executes the part "withdraw with uncooperative neighbor" (section 5.8).

Note that, if the escrow service determines that it receives Bob's T after the time-out *and* it has not previously received T before the time-out, it will refuse Bob's request by sending him the opposite promise:

From	To	Data
B	A	T
B	E	T,TCD
E	B	$sig_E([\text{"}A\text{"}, TCD])$

5.6 Rollback by escrow

Alice has previously locked funds for a transaction to Bob. Alice has not received the transaction token T before the time-out, and Bob does not settle the transaction (Bob is uncooperative).

TCD describes the conditions of the microtransaction.

From	To	Data
A	E	TCD
E	A	$sig_E([\text{"}A\text{"}, TCD])$
A	B	$sig_E([\text{"}A\text{"}, TCD])$

After this, Alice blocks new transactions on the channel until Bob cooperates with settling existing transactions correctly. Normally, the promise by the escrow service will convince Bob to settle, so that no further involvement by the escrow service is required.

If Bob remains uncooperative until the lock time of W expires, Alice executes the part "withdraw with uncooperative neighbor" (section 5.8).

Note that, if the escrow service has previously received T before the time-out (e.g. from Bob), it will refuse Alice's request by sending her the opposite promise. In that case, Alice will also receive T from the escrow service, so that she can cause a commit on her upstream channel:

From	To	Data
A	E	TCD
E	A	$sig_E([\text{"}B\text{"}, TCD])$,$T$

5.7 Withdraw from the channel (with cooperative neighbor)

Alice wants to close the channel and withdraw all her funds. It is assumed that there are no remaining on-going microtransactions on the channel.

W_f is a final version of W; W_f has no lock time.

From	To	Data
A	B	W_f, $sig_A(W_f)$
B	A	$sig_B(W_f)$
A, B	block chain	$W_{f,signed}$

5.8 Withdraw from the channel (with uncooperative neighbor)

Alice wants to close the channel and withdraw all her funds.

W_A is the last version of W for which Alice has received Bob's signature ($sig_B(W_A)$). This is either the last version of W, or an earlier version in which Alice has more funds.

Since W_A has a lock time, Alice can only start this sequence as soon as the lock time has expired. Note that, if there are more recent versions of W in which Bob has more funds, then Bob has the ability and the incentive to publish a more recent version before Alice publishes W_A. This is a good thing, since Bob's version is better (and hence more correct) anyway, and Alice can continue with Bob's version just as well as with W_A.

From	To	Data
A	block chain	$W_{A,signed}$

After this step, Alice can continue by redeeming the funds locked in W_A for any on-going transactions that belong to her. So, assuming there is such a transaction, and she has previously received a signed promise by E:

L is the list of all TCDs of W_A. The Commit transaction (C) spends the locked transaction output of W_A and sends the funds to Alice.

From	To	Data
A	E	$W_{A,signed}$, L,TCD,$sig_E([\text{"}A\text{"},TCD])$
E	A	C,$sig_E(C)$
A	block chain	C_{signed}

6 Remaining vulnerabilities

6.1 Positive escrow failure

There are no cryptographic guarantees that the escrow service will behave as described in section 5. The escrow service is not only able to not perform actions it is supposed to do, it is also able to perform actions it is not supposed to do. Specifically, it is able to:

1. sign a commit promise, while it has not received T before the time-out.

2. sign a rollback promise, while it has received T before the time-out.
3. sign both a commit promise and a rollback promise.
4. sign a commit transaction, while it has signed a rollback promise.
5. sign a rollback transaction, while it has signed a commit promise.
6. sign both a commit transaction and a rollback transaction (though, obviously , only one of these will end up in the block chain, since they are double spends of each other).

These can be addressed in the following ways (by convention, let's call the paying side Alice and the receiving side Bob):

1. E can offer a public interface (e.g. a website) on which E returns T when presented with a valid commit promise. If E signs a commit promise without having received T, this public interface will fail when the incorrect promise is given as input. If Alice receives the incorrect promise, she can publish it, so that other parties can witness this failure (this can be automated!). So, by performing this kind of failure, E risks a loss of reputation. Even if the failure to return T looks like it is caused by involuntary down-time of E, sustained down-time will lead to loss of reputation.
2. If the rollback promise is signed *before* the time-out, and Bob receives it, Bob can publish it before the time-out, so that E's failure can be verified publicly, leading to an immediate loss of reputation of E. The scenario where the rollback promise is signed *after* the time-out is more problematic. Bob can protect himself against this scenario by publishing T on the block chain before the time-out (e.g. in an OP_RETURN output). E should be able to detect this, and as a result E should sign a commit promise and not a rollback promise. If E signs a rollback promise anyway, and Bob receives this promise, Bob can publish the rollback promise. Together with T being in the block chain before the time-out, this proves E's failure publicly, so E risks an immediate loss of reputation.
3. If both are published, this proves E's failure publicly, so E risks an immediate loss of reputation.
4. If both are published, this proves E's failure publicly, so E risks an immediate loss of reputation.
5. If both are published, this proves E's failure publicly, so E risks an immediate loss of reputation.
6. If both are published, this proves E's failure publicly, so E risks an immediate loss of reputation.

So, all modes of mis-behavior have methods through which third parties can witness the mis-behavior in an automated way. As a result, all modes of mis-behavior expose E to a risk of loss of reputation. Since there is no inherent lock-in to E's business, a single failure will probably end E's business as an escrow service.

6.2 Negative escrow failure

Another type of failure is that E does *not* perform a certain action, while it *should* do that action (before a certain time-out). This is probably a more serious threat: not only does not performing an action provide no evidence, it is also possible to happen accidentally (e.g. by hardware down-time) or by Denial of Service attacks by third parties. Specifically, it is possible that E:

1. does not sign a commit promise, while it has received T before the time-out.
2. does not sign either a commit promise a roll-back promise after the time-out.
3. does not sign a commit transaction after W has been placed in the block chain, while a commit promise was signed.
4. does not sign a rollback transaction after W has been placed in the block chain, while a rollback promise was signed.

All these modes can be addressed by letting E have a public interface (e.g. a web interface) on which these signatures can be requested. If the required input data is published, third parties can verify that E is "down". Sustained down-time will lead to a loss of reputation of E; while E is "down", no one should initiate transactions that use E as an escrow service.

The first failure mode is a special one, since E must witness T before a time-out. If E is "down" and does not immediately return a commit promise to Bob (who is the receiving end of the transaction), Bob should publish T on the block chain. When E becomes "up" again, it should witness this timely publication of T, and produce a commit promise, even after the time-out. Now, there is no more time-pressure on the first failure mode, so it becomes the same as the other three.

6.3 Malleability of the deposit transaction

Malleability of the withdraw transaction (W) is not a problem, since its follow-up transactions are only created *after* W is included in the block chain. However, malleability of the deposit transaction (D) is still problematic: Between the moment when Alice publishes D on the Bitcoin network and the moment D is confirmed on the block chain, malleability of D allows other parties to publish a modified version of D, which has a different transaction ID. If this modified

version is accepted on the block chain instead of the original one, this invalidates Bob's signature of W_0. As a result, Bob can hold the deposited funds hostage.

As long as transaction malleability continues to exist, this problem will continue to exist for all[4] types of micro-transaction channels. Luckily, in the case of the deposit transaction, the attack is not guaranteed to succeed (in fact, since the original D is published first, it is probably more likely to fail than to succeed), and it is very likely to be detected. Nevertheless, it is still an existing vulnerability in this design, and in all microtransaction channel designs.

7 Conclusions and recommendations

7.1 Comparison with the Lightning design

By using the methods described in this document, a precursor the Lightning network can be created, using only Bitcoin functionality that already exists right now. Since, on a high level, the behavior of Lightning's Hash-Time-Locked Contracts is emulated, such a precursor network can be gradually converted to a true Lightning network, once true Lightning-style HTLCs become available.

While the escrow service in this design is publically auditable to a high degree, it can still cause damage by misbehaving. It is therefore recommended that users only use escrow services that have a strong reputation, and to continuously keep monitoring this reputation. The possibility of escrow service misbehavior makes this design less robust than the original Lightning design. It is therefore still recommended to add the missing features, required by Lightning, to Bitcoin as soon as possible.

A potential danger exists when the network becomes popular, while the Lightning features are not enabled: in that case, escrow services can become an established power in the Bitcoin ecosystem, that has everything to lose if the Lightning features are ever enabled. They might be be very motivated to stop these features from being enabled, thereby preventing progress from happening.

7.2 Bitcoin improvements

It was a bit of a surprise discovery during the design of this system that an opcode like OP_CHECKSIGDATAVERIFY does not exist yet. If the aim is for the Bitcoin scripting language to provide a general-purpose cryptographic toolset, then it is strange to lack the ability to verify signatures that sign arbitrary data. For the design described here, such a feature would remove the need for the separate promising phase. There are probably other applications as well that could benefit from such a scripting feature.

Finally, like all micro-transaction concepts, this design requires a solution for the transaction malleability problem. Luckily, there seems to be a wide

[4] Maybe someone can come up with a smart microtransaction channel design that does not suffer from this issue, but I have never seen one, and I strongly doubt it is possible.

consensus in the Bitcoin community that the malleability problem needs to be solved.

References

[1] Joseph Poon, Thaddeus Dryja (2015). *The Bitcoin Lightning Network* (DRAFT Version 0.5). http://lightning.network/lightning-network-paper-DRAFT-0.5.pdf

[2] Alex Akselrod (username blueadept) (2014). https://bitcointalk.org/index.php?topic=814770