Reaching The Ground With Lightning (draft 0.1)

Rusty Russell <rusty@blockstream.com>

July 20, 2015

Abstract

The Lightning Network (as proposed by Joseph Poon and Thaddeus Dryja[5]) requires some new sighash modes in order to work with Bitcoin. This paper proposes a simplified variant which requires only modifications which are already proposed for bitcoin, and slightly simplifies the revocation of existing contracts.

Keywords: bitcoin, lightning, revocation hash, HTLC

1 Introduction

The Bitcoin network[3] allows the transfer of value between peers using *transactions*. Each bitcoin transaction consists of one or more *outputs* (typically specifying the hash of the recipient's key), and one or more *inputs* (typically containing the recipient's key and a signature of the transaction). Thus one transfers value to another peer by creating a transaction which *spends* one or more outputs and creates an output which the recipient can spent using their private key.

While such cryptographic transfer of value is near-instantaneous, ensuring that the transaction has been included in the consensus of the shared ledger (aka. *blockchain*) creates delays ranging from a few minutes to hours, depending on the level of reliability required. Inclusion in the blockchain is performed by miners, who preferentially include transactions paying greatest fee per byte.

Thus using the blockchain directly is slow, and too expensive for genuinely small transfers (typical fees are a few cents).

2 Previous Work

To work around the bitcoin network's delays and fees, several forms of *off-chain* transaction patterns have been developed, where series of transactions are sent directly between two parties, with only the initial opening transaction and final redemption transaction being included in the bitcoin blockchain.

The Lightning Network paper proposed a solution, but at the cost of introducing new signature variants (sighash ops). Adding a new signature opcode would allow many other improvements[1][2] but that is precisely why it's a matter for longer term research and unlikely to be deployed in Bitcoin in the immediate future.

2.1 Payment Channels

The concept of *payment channels* (sometimes called micropayment channels) has existed in various forms for several years[1]. The simplest form is as follows, and allows A to quickly and cheaply pay B a stream of slightly increasing amounts:

1. A creates an *anchor* transaction to *open the channel* which:
   (a) Outputs $1,
   (b) Requires the signatures of both A and B to redeem.
2. A sends the transaction ID of the anchor, which output to spend, and the amount of that output to B.
3. B signs a "refund" transaction which:
   (a) spends that anchor output,
   (b) outputs the $1 to an address controlled by A, and
   (c) can only be spent in 24 hours (using the *locktime* field)
4. B sends A the refund transaction.
5. A broadcasts the anchor transaction, knowing she can get the funds back in 24 hours using the refund if B vanishes.

A can now pay B 1 cent by signing a new *commitment* transaction to send to B, which spends the anchor output and has two outputs: one pays A 99c, and the other pays B 1 cent. A can later pay B another cent by signing another transaction ("updating the commitment") for B which pays A 98c and B 2c, etc.

At any point, B can "close the channel" by signing and broadcasting the latest commitment transaction to collect the money. B should do this before 24 hours pass, otherwise A can use the refund transaction.

2.1.1 Limitations Of Simple Payment Channels

Simple channels have several limitations:

Single recipient. A new recipient requires a new channel, which must wait for consensus on the anchor transaction.

[1] Schnorr signatures offer faster batch validation, according to https://github.com/ElementsProject/elementsproject.github.io#schnorr-signature-validation

[2] DER encoding adds unnecessary bytes and is a cause of malleability

One way. They cannot be reversed: A can sign a transaction which pays B less money than the last, but B could still broadcast the older transaction.

Vulnerable to malleability. The anchor transaction could be altered in several ways (without invaliding it completely) before inclusion in the blockchain: this alters its transaction id and thus makes the refund transaction unusable.

This last issue is a common one with complex bitcoin transactions, and BIP62[7] is proposed to prevent non-signing parties from being able to malleate transactions.

2.2 Generalized Payment Channels Using Revocable Transactions

The Lightning network introduced generalized, bi-directional payment channels, referred to here as *Poon-Dryja channels*. These use a mutual anchor, which both create to provide the channel funding, and a symmetrical *pair* of updatable commitment transactions rather than the single transaction used in the one-way channel case, as shown in figure 1.

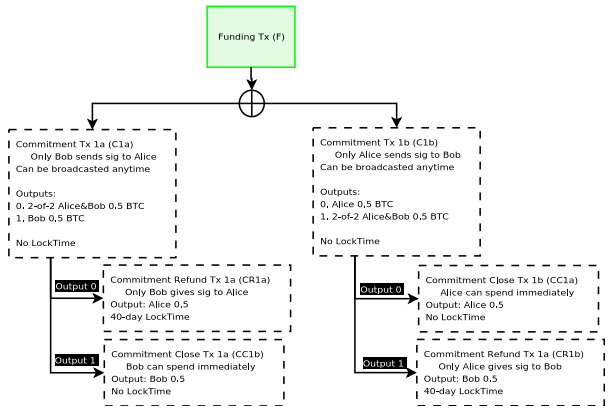

Figure 1: Figure 1 from the Lightning Network Draft 0.5

To update the commitment, A sends B a signature for B's new commitment transaction, and B sends A a signature for A's new commitment transaction.

As before, each commitment transaction contains two outputs, one for A and one for B; but A's commitment transaction output to itself is encumbered by an additional restriction (as is B's output to itself). Instead of paying A directly, needs both A and B's signature. B provides such a signature, but on a "commitment refund" transaction which can only be spent after a delay (40 days in the paper). Thus if A closes the channel by signing and broadcasting its commitment transaction, B can collect its output immediately, but A must wait 40 days.

This delay encumbering the output is what makes the commitment transaction *revocable;* once an updated commitment transaction is agreed upon, the previous commitment transaction pair is revoked by sharing the private keys needed to redeem those encumbered outputs. Thus, A shares its (throwaway) private key, and B shares its throwaway private key. If A were to sign and broadcast a revoked commitment transaction, B could not only immediately spend its own output, but it has both A's key and its own to generate a transaction which can spend the output which would normally go to A after a delay.

2.3 Hashed Timelock Contracts (HTLCs)

The Lightning Network paper used a set of 4 transactions to implement a *hashed timelock contract,* which guarantees payment of a given amount on presentation of a secret value R within a certain timespan. Any number of these could be active within a generalized channel, and this is what allows a network to form: Node A offers node B $1 for the secret within 2 days, node B offers node C 99c for the secret within 1 day, etc.

This arrangement for one side of a single node is shown in figure 2.

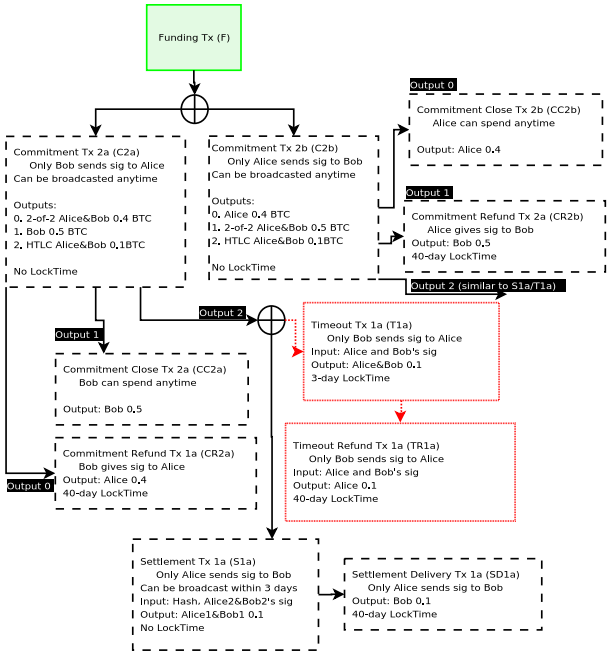

Figure 2: Figure 2 from the Lightning Network Draft 0.5

3 Enhancements To Lightning

This paper proposes various modifications.

3.1 Poon-Dryja Generalized Payment Channel Modifications

This paper proposes three of these.

3.1.1 Placing Timeout in Output Script

Rather than using a separate transaction to enforce the delay, BIP65[6] proposes an OP_CHECKLOCKTIMEVERIFY which allows an output to specify the minimum time at which it can be spent. With this enhancement, we no longer need a separate "commitment refund" transaction. The commitment transaction to-self output script would be a little more complex:

- A and B's signature, OR
- A's signature and OP_CHECKLOCKTIMEVERIFY <40 days>

3.1.2 Using Relative Locktime

The Poon-Dryja channel uses a 40 day locktime, because transaction locktime is absolute. Before 40 days the channel must be closed otherwise spending a revoked transaction and immediately following it with the commit refund transaction is possible.

A proposal to extend output scripts to specify a minimum *relative* time before they can be spent[2] can reduce this timeout (say, to 1 day) and avoid placing a lifetime limit on the channel, like so:

- A and B's signature, OR
- A's signature and OP_CHECKSEQUENCEVERIFY <1 day>

3.1.3 Using Revocation Preimages Instead of Private Keys

There's a slightly more intuitive and more efficient method than exchanging private keys, which is to reuse a technique of hash preimages which is already needed for HTLCs (as we see later).

Instead of using a private keys, B uses knowledge of a hash preimage as well as its signature to steal funds from a revoked commitment transaction. Thus, to create a commitment transaction each side provides a hash value; to revoke a commitment transaction it provides the prehash image.

The resulting commitment transaction to-self output now looks like:

- B's signature and a preimage which hashes to <revocation-hash>, OR
- A's signature and OP_CHECKSEQUENCEVERIFY <1 day>

This can be expressed fairly easily in bitcoin's script-based scripting language, as annotated in Commitment Transactions For Generalized Channels. The final pair of commitment transaction outputs is shown in Figure 3.

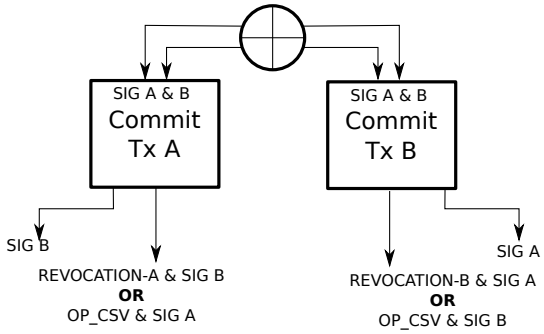

Figure 3: Commitment Transaction Outputs

3.2 Channel Opening Modifications

The method of creating the first commitment transaction before signing the anchor transaction (as proposed in the paper) presents two problems in practice:

1. The anchor transaction id required for the commitment input will only be known one the anchor is signed, and

2. The anchor transaction can be malleated by either party before entering the blockchain, rendering the commitment input unusable.

The last of these is particularly pernicious, as BIP62 doesn't solve it: signatories can always re-sign a transaction, hence altering its transaction ID. The paper proposes new SIGHASH flags which mitigate this problem, but we are attempting to avoid that.

For ease of understanding, we develop the protocol in stages. Please note that the intermediary proposals are insecure!

3.2.1 Separate Anchor Transactions

To avoid the problem of needing all anchor signatures to derive the anchor transaction ID to create the commitment transaction input, we split the anchor into two transactions; thus A knows its anchor transaction ID, and B knows its anchor transaction ID as shown in Figure 4.

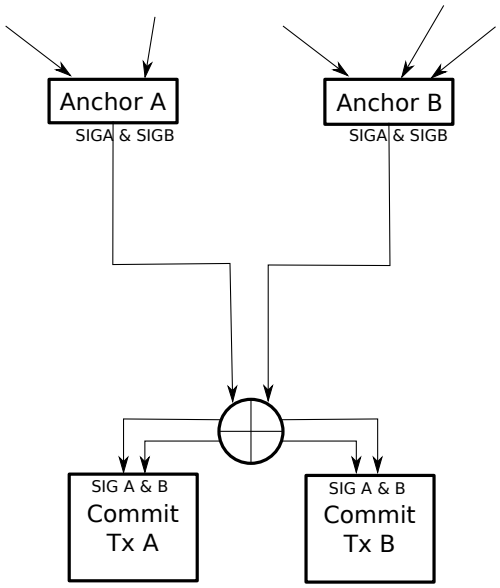

Figure 4: Simplistic Dual Anchor Design

This form allows A and B to create commitment transactions which spends the anchors outputs by exchanging anchor transaction IDs. It has the problem that if the other party does not then broadcast its anchor transaction, we cannot spend the commitment transaction, and our own anchor funds are stuck.

Thus we introduce an *escape* transaction, which lets us regain our anchor funds in that case, as shown in Figure 5.

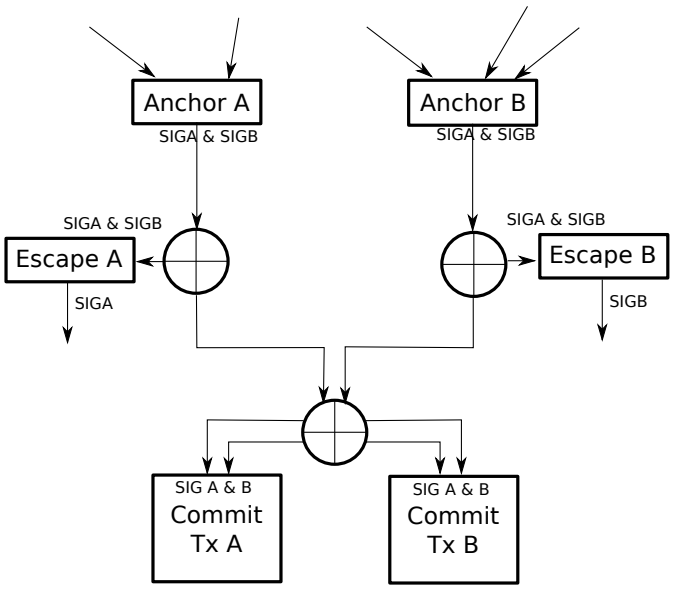

Figure 5: Dual Anchor With Simple Escape Transactions

However, this escape transaction would let either side remove its funds from the channel at any time, which would make the channel insecure. Thus, after the commitment transactions have been established, we want to revoke the escape transactions. We can do the same way we did for the commitment transaction revocation; by placing restrictions on the "to-me" output. In particular, adding a delay if paying back to the anchor owner, and allowing it to be spent by the other party immediately if they possess the revocation preimage, as shown in Figure 6.

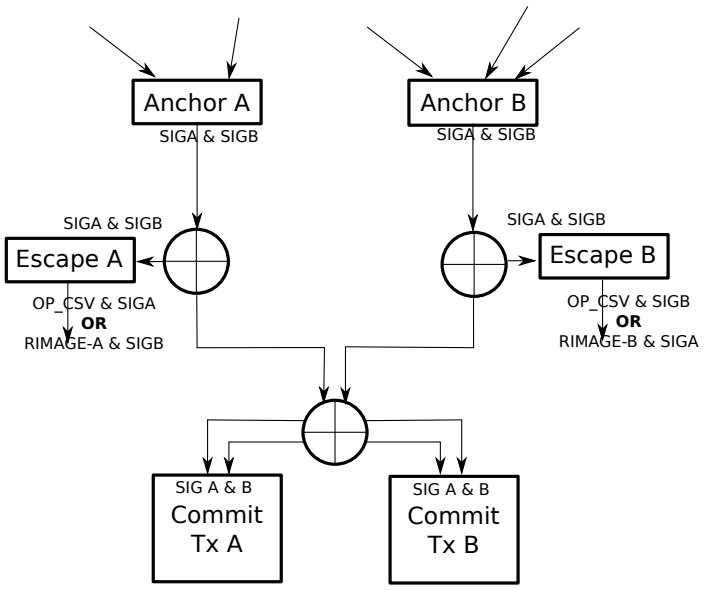

Figure 6: Dual Anchors With Revocable Escape Transactions

Unfortunately, this revocation is not a complete solution; if B uses its escape transaction, A can collect B's anchor funds, but it has no way of collecting its own! The commitment transaction cannot be used, as one of its inputs has been spent by B's escape transaction. A's own escape transaction has been revoked, so B would simply steal the funds.

Thus we need an additional construction, such that using one escape transaction immediately unlocks the other anchor funds for its owner. To do this, we ensure that the escape transaction is forced to reveal a secret, which is a fairly well-established technique[4]. The anchor transaction is modified to either require both signatures (for the commitment transaction), or both signatures and the secret (for the escape transaction), as shown in Figure 7. Note that this requires the other party to provide an alternate key (denoted here using A' and B'), otherwise there is no way to force the escape transaction to provide the secret.

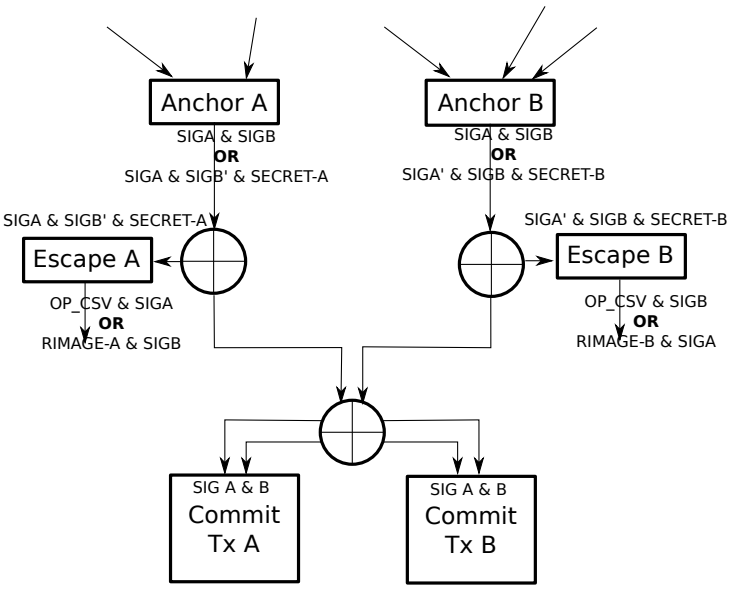

Figure 7: Secret Revelation by Escape Transactions

That revealed secret can be used with the other alternative: the *fast escape transaction*. This reveals the secret just like the escape transaction, but its output is immediately usable if one knows the other side's secret. This is shown in Figure 8. Thus, if the B broadcasts its escape transaction after it has been revoked, A can (after ensuring escape B is sufficiently deep in the block chain) broadcast its fast escape transaction and use B's secret to immediately spend the output.

On the other hand, if B broadcasts its fast escape transaction without knowing A's secret, A can simply wait for the timeout and spend the fast escape output, then use its own fast escape transaction and B's secret to recover its own anchor funds as well.

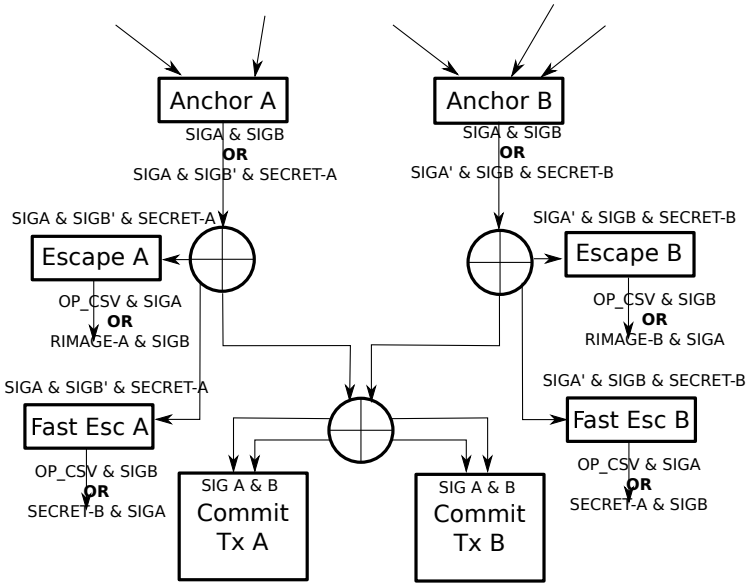

Figure 8: Final Dual Anchor Design

The final scripts are shown in Appendix A: Transaction Scripts.

3.2.2 Disadvantages of The Dual Anchor Approach

Unlike the mutual anchor approach, use of escape transactions is not outsourcable: you cannot have an untrusted third party which can monitor the network for the other sides' revoked escape transaction and respond with your own escape transaction. If you were to provide a third party with your fast escape transaction, you would necessarily provide it with the secret, which it could give to B.

3.3 Hashed Timelock Contract (HTLC) Modification

Using the same techniques used above, we can condense each HTLC into a single output script on the commitment transaction. This output is spendable under three conditions:

1. Recipient knows the R value (funds go to recipient), or
2. The HTLC has timed out (funds return to sender), or
3. The Commit transaction has been revoked (funds to go other side).

Unlike the original paper, we use revocation preimages instead of sharing temporary private keys. If we also use OP_CHECKLOCKTIMEVERIFY and OP_CHECKSEQUENCEVERIFY it is fairly simple to express these conditions in a single output script.

For each direction the HTLC could transfer funds, there are two scripts required; one for A's commitment transaction and one for B's commitment transaction. It's also a requirement that the conditions which allow payment to oneself be delayed, to give the other side an opportunity to take the funds in case of revocation. This is shown in figure 9.

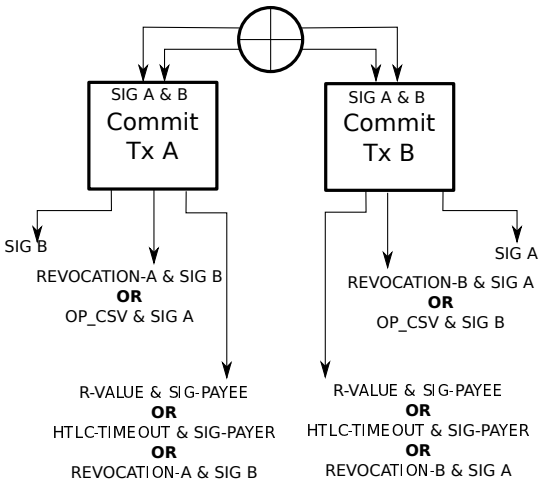

Figure 9: HTLC Using Revocation Preimages, OP_CLV and OP_CSV

The scripts for this can be found in 4.

4 Conclusions

Secret preimages can replace exposure of temporary private keys in the Lightning Network constructs with no loss of generality, and a slight gain in simplicity.

The use of script conditionals to enforce timeouts instead of using separate pre-signed transactions reduces an HTLC from a set of four dual-signed transactions to a single (more complex) output script, and additionally avoids any requirement for new CHECKSIG flags for HTLCs.

By using a dual anchor and escape transactions, channel establishment can also avoid new CHECKSIG flags, though it loses the important ability to outsource the enforcement of channel contract terms.

Acknowlegments

Thanks to mmeijeri on Reddit's r/Bitcoin for pointing out a flaw in escape transactions reusing the same A and B keys as the commitment transaction[3].

[3] https://www.reddit.com/r/Bitcoin/comments/3dlxw4/reaching_the_ground_with_lightning_lightning/ct80xpp

Thanks to Joseph Poon for designing the escape/fast-escape dual-anchor method, as well as finding a flaw in my original formulation of the dual anchor construct and reviewing an earlier draft of this paper. Also thanks to him and Thaddeus Dryja for the initial eye-opening Lightning Network paper.

References

[1] Rapidly-adjusted (micro)payments to a pre-determined party. https://en.bitcoin.it/wiki/Contract#Example_7:_Rapidly-adjusted_.28micro.29payments_to_a_pre-determined_party.

[2] Mark Friedenbach. [bitcoin-development] [BIP draft] consensus-enforced transaction replacement signalled via sequence numbers. http://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-June/008452.html.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[4] Tier Nolan. Alt chains and atomic transfers. https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

[5] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network draft version 0.5, 2015. http://lightning.network/lightning-network-paper-DRAFT-0.5.pdf.

[6] Peter Todd. OP_CHECKLOCKTIMEVERIFY. https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki.

[7] Pieter Wuille. Dealing with malleability. https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki.

Appendix A: Transaction Scripts

All outputs are expressed as pay-to-scripthash outputs, where the redeeming input provides the redeemscript. Where a redeem-hash value is optional, it is generally supplied: for example, if we want to pay to A if a preimage is supplied and B if no preimage is supplied, we expect the input scriptsig to provide two arguments in both cases (generally a zero in the second case). This saves an extra test (of form "OP_DEPTH <N> OP_EQUAL"), at cost of a single byte in the input script.

Anchor Transaction

The anchor inputs are whatever the node chooses.

Anchor Output Redeemscript

The anchor output is a pay to script hash, with a redeemscript as follows:

OP_HASH <SECRET-A-HASH> OP_EQUAL If the secret is supplied,

OP_IF

2 <KEY-B'> Should be signed by B's escape key.

OP_ELSE

2 <KEY-B> Should be signed by B's commitment key.

OP_ENDIF

<KEY-A> 2 OP_CHECKMULTISIG Make sure A and B have signed.

Escape Transaction

The escape transaction for A spends A's anchor output and reveals A's secret. Similarly for B.

Escape Input Script

<SIG-A> <SIG-B'> SECRET 1 {<ANCHOR-REDEEMSCRIPT>}

Escape Output Redeemscript

This allows two paths: one for the other side to use the revocation image, and one for this side to get their funds back after a delay. This show's A's script, but B's is the same with A and B exchanged.

OP_HASH160 <RHASH-A> OP_EQUAL Check if the top of stack is the revocation image.

OP_IF

<KEY-B> Funds for B.

OP_ELSE It's A getting their funds back

<DELAYTIME> OP_CHECKSEQUENCEVERIFY OP_DROP Ensure delay.

<KEY-A> Needs to be signed by A.

OP_ENDIF

OP_CHECKSIG Make sure it's signed correctly.

Spending The Escape Output

Either B using a revocation preimage:

<SIG-B> <REVOCATION-IMAGE-A> {<ESCAPE-REDEEMSCRIPT>}

Or A using after a timeout:

<SIG-A> 0 {<ESCAPE-REDEEMSCRIPT>}

Fast-Escape Transaction

Fast-Escape Input Script

This is identical to the normal escape input script.

<SIG-A> <SIG-B'> SECRET 1 {<ANCHOR-REDEEMSCRIPT>}

Fast-Escape Output Redeemscript

This allows two paths: one for this side to use the other side's secret (revealed by them using an escape transaction), and one for the other side to claim this side's anchor funds after a delay. This show's A's script, but B's is the same with A and B exchanged.

OP_HASH <SECRET-B-HASH> OP_EQUAL If top argument is B's secret

OP_IF

<KEY-A> For A

OP_ELSE B gets it if A doesn't know the secret.

<DELAYTIME> OP_CHECKSEQUENCEVERIFY OP_DROP Ensure delay.

<KEY-B> Needs to be signed by B.

OP_ENDIF

OP_CHECKSIG Make sure it's signed correctly.

Spending The Fast-Escape Output

Either A using a B's secret revealed by B using its own escape transaction:

<SIG-A> <SECRET-B> {<FAST-ESCAPE-REDEEMSCRIPT>}

Or B using after a timeout:

<SIG-B> 0 {<FAST-ESCAPE-REDEEMSCRIPT>}

Commitment Transactions For Generalized Channels

These examples are for A's Commitment Transaction; switch A and B to get B's commitment transaction.

Commitment Input Script

The commitment transaction has two inputs; one which spends each anchor output. The two zeroes indicate it is not revealing the secret:

<SIG-A> <SIG-B> 0 0 {<ANCHOR-REDEEMSCRIPT>}

Commitment Transaction Output Redeemscripts

One output pays B's funds to B as normal (eg. pay to scripthash "<KEY-B> OP_CHECKSIG"). The other output pays A's funds: either to B if they supply the revocation preimage, or to A after a delay. This is the redeemscript:

OP_HASH160 <COMMIT-REVOCATION-HASH> OP_EQUAL Did they supply revocation preimage?

OP_IF

  <B-KEY> To B.

OP_ELSE

  <LOCKTIME> OP_CHECKSEQUENCEVERIFY OP_DROP Spending transaction must be after timeout

  <A-KEY> To A.

OP_ENDIF

OP_CHECKSIG Signature must be correct.

Spending Commitment Output

Either B using a revocation preimage:

<SIG-B> <COMMIT-REVOCATION-IMAGE-A> {<COMMITMENT-REDEEMSCRIPT>}

Or A using after a timeout:

<SIG-A> 0 {<COMMITMENT-REDEEMSCRIPT>}

Hash Locked Transaction Commitments

There are two styles of commitment transaction outputs for HTLCs: a "sender" and "receiver" case. The output is a pay-to-script-hash, so the redeemscripts are shown below.

These scripts show A as the sender, and B as the receiver: exchange A and B for the reverse.

HTLC Sender Redeemscript

OP_HASH160 OP_DUP Replace top element with two copies of its hash

<R-HASH> OP_EQUAL Test if they supplied the HTLC R value

OP_SWAP <COMMIT-REVOCATION-HASH> OP_EQUAL OP_ADD
Or the commitment revocation hash

OP_IF If any hash matched.

<KEY-B> Pay to B.

OP_ELSE Must be A, after HTLC has timed out.

<HTLC-TIMEOUT> OP_CHECKLOCKTIMEVERIFY OP_DROP
Ensure (absolute) time has passed.

<DELAY> OP_CHECKSEQUENCEVERIFY OP_DROP Delay gives B enough time to use revocation if it has it.

<KEY-A> Pay to A.

OP_ENDIF

OP_CHECKSIG Verify A or B's signature is correct.

HTLC Receiver Redeemscript

OP_HASH160 OP_DUP Replace top element with two copies of its hash

<R-HASH> OP_EQUAL B redeeming the contract, using R preimage?

OP_IF

OP_DROP Remove extra hash

<KEY-A> Pay to B

OP_ELSE

<COMMIT-REVOCATION-HASH> OP_EQUAL If the commit has been revoked.

OP_NOTIF If not, you need to wait for timeout.

<HTLC-TIMEOUT> OP_CHECKLOCKTIMEVERIFY OP_DROP
Ensure (absolute) time has passed.

OP_ENDIF

<KEY-A> Pay to A

OP_ENDIF

OP_CHECKSIG Verify A or B's signature is correct.

Redeeming A HTLC Output

To redeem an HTLC, the recipient one provides the preimage R, and their signature. In our example above, B can redeem the HTLC:

<SIG-B> <HTLC-R-VALUE> {<HTLC-REDEEMSCRIPT>}

Claiming a Timed-out HTLC

To claim a timed-out HTLC, the sender supplies a zero value (which is nice and short, but fails to hash to any of the revocation hashes), and their signature. In our example above, A can claim the timed-out HTLC:

<SIG-A> 0 {<HTLC-REDEEMSCRIPT>}

Claiming A HTLC Output For A Revoked Commitment Transaction

If either side publishes a commitment transaction which has been revoked, we can use the revocation preimage they supplied to spend all the outputs. This example shows A claiming the HTLC output if B broadcasts a revoked commitment transaction:

<SIG-A> <COMMIT-REVOCATION> {<HTLC-REDEEMSCRIPT>}