Lightning Channels With Tunable Penalties

John Law

September 17, 2022

Version 1.0

Abstract

The Lightning channel protocol uses revocation keys to invalidate old channel states. If one party puts an old Commitment transaction on-chain, the other party can use their revocation keys to take all of the channel funds. While this is an effective penalty mechanism for preventing intentional attempts to steal funds, the use of an old Commitment transaction may not be intentional. For example, a party's node could crash and be restored from a backup that is not current. Therefore, it would be preferable if the penalty for putting an old transaction on-chain were tunable, thus discouraging dishonest parties while limiting the loss for honest parties. This paper presents a modification of the Lightning channel protocol that supports tunable penalties. In addition, it allows untrusted watchtowers to use storage that is logarithmic, rather than linear, in the number of channel states supported. No change to the underlying Bitcoin protocol is required.

1 Overview

In the current Lightning channel protocol **[AOP21][BOLT][PD16]**, if a party puts an old Commitment transaction on-chain, the other party can use their revocation keys to take all of the channel funds. This is an effective penalty mechanism for preventing the intentional use of old Commitment transactions. On the other hand, it severely penalizes honest users who have lost the latest channel state (for example, due to loss of state caused by a crash or loss of a cell phone) and do not have access to an up-to-date backup (this has been called the "toxic-waste" problem **[RZD19]**). It would be preferable if the penalty for putting an old transaction on-chain were tunable, thus discouraging dishonest parties while limiting the loss for honest parties **[Rub22][San22]**.

This paper presents a Tunable-Penalty (TP) channel protocol that allows each party to have an arbitrary penalty for putting an old transaction on-chain. Like the current Lightning protocol, the TP protocol supports $O(1)$ time and $O(1)$ on-chain-transaction unilateral channel closes. Furthermore, if S is the number of channel states supported, the TP protocol reduces the storage requirements for an untrusted

watchtower **[GMRMG]** from $O(S)$ with the current Lightning protocol to $O(\log S)$. The TP protocol does not require any change to the underlying Bitcoin protocol.

The rest of this paper is organized as follows. Section 2 examines the challenges in adding tunable penalties to the Lightning channel protocol. Section 3 presents the TP protocol and proves its correctness. Watchtowers for the TP protocol with $O(\log S)$ storage are given in Section 4. Extensions to the TP protocol are considered in Section 5. The remaining sections present related work and conclusions. The correctness of the TP protocol is proven in Appendix A and the settings of the protocol's timing parameters are given in Appendix B.

2 First Attempt: Tunable Penalties in Lightning Channels

In order to motivate the TP protocol, first consider a straightforward approach to making the penalties in Lightning channels tunable. Let Alice and Bob be the owners of a Lightning channel. The channel's funds come from an on-chain Funding transaction that has a single output that requires both parties' signatures. The channel's funds can be spent with a Cooperative Close transaction (if both parties agree to close the channel), or unilaterally with either party's Commitment transaction. A new channel state is created by having each party sign a new Commitment transaction for the other party, where the new Commitment transaction has a *to_self* output (paying a portion of the channel's funds to the party that put the transaction on-chain), a *to_remote* output (paying a portion of the channel's funds to the other party), and an *HTLC* output for each Hash Time-Locked Contract (HTLC) **[BWHTLC]** that is currently outstanding. The *to_self* output cannot be spent by the party that put the transaction on-chain until a relative delay (namely, the *to_self_delay* parameter set by the other party) has elapsed. If the Commitment transaction is for an old (and thus revoked) state, this delay provides a window during which the other party can use a revocation key to take the entire *to_self* output for themselves. A similar mechanism using revocation keys allows the other party to take the entire balance of each HTLC output, either directly from the revoked Commitment transaction or from an associated HTLC-success or HTLC-timeout transaction that has spent the Commitment transaction's HTLC output.

A straightforward approach to supporting tunable penalties with the Lightning protocol would be to split the value obtained by using each revocation key into a penalty amount (given to the user revoking the transaction) and updated *to_self* and *to_remote* amounts that distribute the output's funds according to the latest channel state. However, this approach runs into two problems, as discussed below.

2.1 Problem 1: Quadratic Blow-up

First, consider the *to_self* output from Alice's old state i Commitment transaction, where the current channel state is state $j > i$. In order to correctly split this *to_self* output, Bob must be able to spend it with a $\text{Penalty}_{\text{Bij}}$ transaction that takes the old state i *to_self* output value and splits it into the required amounts for the current state j. However, Alice must approve of this division of funds when she agrees to make state j the current state. Therefore, Alice must sign a separate $\text{Penalty}_{\text{Bij}}$ transaction for each

pair of states i and j, where $i < j$, so she must sign a total of $O(S^2)$ such penalty transactions and Bob must store $O(S)$ signatures for the current state **[Dec20]**. The current Lightning protocol only requires $O(S)$ signatures and $O(\log S)$ storage for each party **[Rus]**, so support for such tunable penalty transactions would be quite expensive.

2.2 Problem 2: Variable HTLC Numbers and Amounts

Second, even if Alice and Bob were willing to compute and store the required penalty transactions for the *to_self* outputs of the Commitment transactions, the HTLC outputs of the Commitment transactions are even more problematic. This is because Commitment transactions for different states have different numbers of HTLC outputs and those outputs have varying values. One could imagine creating a unique Penalty_{Bijk} transaction for each pair of states i and j, where $i < j$, and each HTLC output number k, and forcing every Commitment transaction to have exactly the same number of HTLC outputs with fixed amounts. However, these constraints on the HTLC outputs would reduce the channel's ability to support dynamically varying HTLC requests, thus effectively stranding some of the channel's capital.

3 The Tunable-Penalty (TP) Protocol

3.1 Overview

As a result of these two problems, the straightforward approach of adding penalty transactions to the Lightning protocol is not attractive and a more efficient technique is required.

First, consider the quadratic blow-up problem. This problem is caused by the fact that the old transaction, which needs to be revoked, has already spent the output of the Funding transaction before it can be revoked. As a result, if a penalty transaction is added that corrects the distribution of funds, that penalty transaction will depend on both the revoked state and the current state, creating a quadratic number of penalty transactions.

The TP protocol avoids this quadratic blow-up by using separate value transactions and control transactions. In addition to an on-chain Funding transaction that provides the channel's funds, each party in the TP protocol has an on-chain Individual transaction that they use to control how the channel's funds are distributed. Each Individual transaction has a single output with a value slightly larger than the penalty amount charged for putting an old transaction on-chain. Before either party can put their Commitment transaction on-chain, they must first put their State transaction on-chain that spends the output of their Individual transaction. The State transaction encodes the channel's state number in its nLocktime and nSequence fields, just like the Commitment transaction encodes the state number in the Lightning protocol. The first output of the State transaction has a value equal to the desired penalty amount.

If a State transaction for the current state is put on-chain, after a relative delay equal to the maximum of the parties' *to_self_delay* parameters, the State transaction's first output can be spent by the same party's Commitment transaction for the current state[1].

On the other hand, if a State transaction for an old state is put on-chain, it can be revoked by the other party (by spending the State transaction's first output and thus taking the penalty amount) before it can be used as an input to the Commitment transaction.

If one party's State transaction is revoked, the other party can still put their correct State transaction on-chain, as the two parties' State transactions spend outputs from different Individual transactions, and thus do not conflict. Because the old State transaction is revoked before it can affect the distribution of the channel's funds, the quadratic blow-up problem is avoided.

Next, consider the problem of variable HTLC numbers and amounts. The TP protocol also solves this problem by revoking an old State transaction before it can affect the distribution of channel funds to the Commitment transaction's HTLC outputs. However, there is still the issue of efficiently resolving the outcome of each HTLC. Recall that the Commitment transaction cannot be put on-chain until the maximum of the parties' *to_self_delay* parameters after the corresponding State transaction is put on-chain. Therefore, if each HTLC were resolved at the Commitment transaction's corresponding HTLC output, the channel's *to_self_delay* parameters would contribute to the latency of resolving the channel's HTLCs, thus resulting in excessive delays in freeing channel capital in earlier hops and in producing payment receipts at the sender. The TP protocol makes the resolution of HTLCs independent of the channel's *to_self_delay* parameters by adding minimum-value HTLC control outputs (one per HTLC) to the State transactions. These HTLC control outputs are where the HTLC timeout and success results are determined, using conflicting HTLC-timeout and HTLC-success transactions. Then, the results of these HTLC resolutions are used as control inputs to HTLC-refund and HTLC-payment transactions that spend the Commitment transaction's HTLC outputs (in a manner analogous to how the State transaction's first output is used as a control input to the Commitment transaction).

3.2 Protocol Specification

Protocol Transactions

The resulting TP protocol for a channel shared by Alice and Bob is shown in Figure 1. This figure shows a state *i* in which Alice has an HTLC offered to Bob for a multi-hop payment, with the payment's next hop being in a channel shared by Bob and Carol.

1 Note that the penalty amount provided by the State transaction's first output can be refunded to the party that paid for it by increasing the value of the *to_self* output in this Commitment transaction.

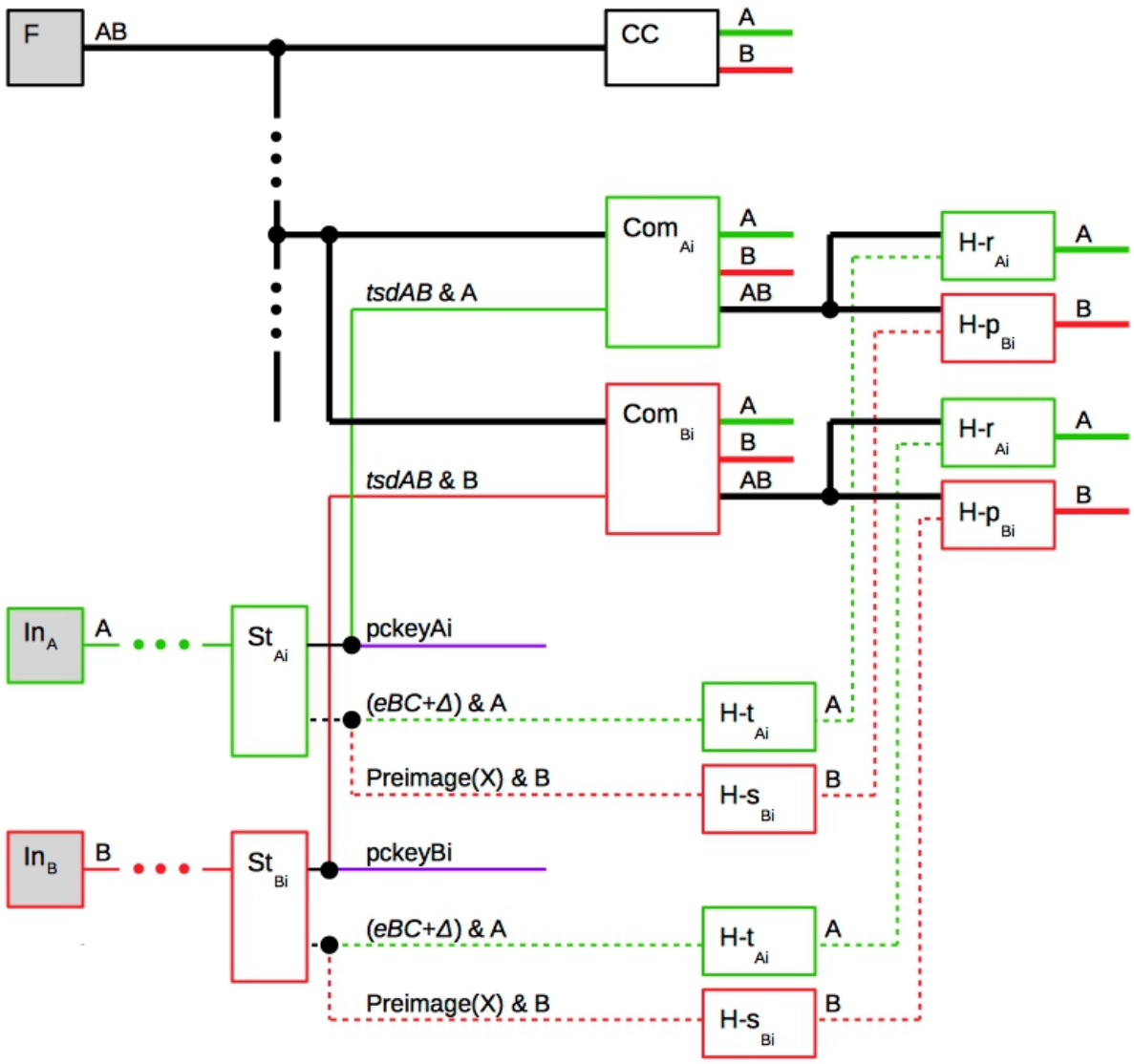

Figure 1. The TP Channel Protocol. The State (St) control transactions determine which Commitment (Com) value transaction can be put on-chain. The HTLC is resolved by the HTLC-timeout (H-t) and HTLC-success (H-s) control transactions, which control whether an HTLC-refund (H-r) or HTLC-payment (H-p) value transaction can be put on-chain.

In Figure 1 (and throughout the paper):

- **A** denotes Alice's signature,
- **B** denotes Bob's signature,
- **AB** denotes both Alice's and Bob's signature,

- **pckey{A|B}i** denotes a signature using a per-commitment key for revoking {Alice's|Bob's} state *i* transaction,
- ***eBC*** denotes the expiry for this payment in the next hop shared by Bob and Carol,
- ***Δ*** denotes the *cltv_expiry_delta* parameter set by Bob,
- ***tsdAB*** denotes the maximum of the *to_self_delay* channel parameters set by Alice and Bob, and
- **Preimage(X)** denotes the payment secret (and payment receipt) which is the preimage of X.

Shaded boxes represent transactions that are on-chain, while unshaded boxes represent off-chain transactions. Each box includes a label showing the transaction type, namely:

- **F** for the Funding transaction,
- **CC** for a Cooperative Close transaction,
- **In** for an Individual transaction,
- **St** for a State transaction,
- **Com** for a Commitment transaction,
- **H-t** for an HTLC-timeout transaction,
- **H-s** for an HTLC-success transaction,
- **H-r** for an HTLC-refund transaction, and
- **H-p** for an HTLC-payment transaction.

Subscripts denote which party can put the transaction on-chain (if only one party can do so) and which channel state the transaction is associated with (namely state *i* in the figure). Transactions that can only be put on-chain by Alice are green, those that can only be put on-chain by Bob are red, and those that can be put on-chain by either party are black.

Bold lines carry channel funds, thin solid lines have value equal to, or slightly larger than[2], the tunable penalty amount, and dashed lines have the minimal allowed value (as they are used for control). When a single output can be spent by multiple off-chain transactions, those transactions are said to *conflict*, and only one of them can be put on-chain. A party will be said to *submit* a transaction when they attempt to put it on-chain.

2 The value of the first output of each State transaction equals the tunable penalty amount, while the output of the Individual transaction has a slightly larger value in order to provide funds for the State transaction's HTLC control outputs.

Each output is labeled with the requirements that must be met to spend the output, with multiple cases being shown as outputs that branch. Multiple requirements are indicated with "&" between them and are listed with knowledge of a secret first, relative delays next, then absolute timelocks (in parentheses) and signature requirements last.

Protocol Operation

In order to establish a new channel state, both parties:

1. calculate the State, Commitment, HTLC-timeout, HTLC-success, HTLC-refund and HTLC-payment transactions for the new state[3],
2. exchange partial signatures for the new state's HTLC-refund, HTLC-payment and Commitment transactions (in that order)[4], and
3. exchange per-commitment keys for the old state, thus revoking it.

Each party has a *fulfillment_deadline* parameter, which defines how long before the expiry of an HTLC offered to them that they will go on-chain if they have the secret for the HTLC but have been unable to update the channel state off-chain to reflect the HTLC's fulfillment. Similarly, each party has a *timeout_deadline* parameter, which defines how long **before** the expiry of an HTLC offered by them that they will go on-chain if the channel state has not been updated off-chain to reflect the resolution (either fulfillment or timeout) of the HTLC[5].

Consider the case of an HTLC offered by Alice to Bob (as shown in Figure 1). When Bob receives the secret for the HTLC, he shares it with Alice and attempts to update the channel state off-chain. If he is unable to update the state off-chain by his *fulfillment_deadline* prior to the HTLC's expiry, he puts his current State and associated HTLC-success transactions on-chain. If Alice has not received the HTLC's secret by her *timeout_deadline* prior to the HTLC's expiry, she puts her current State transaction on-chain. Upon the expiry of the HTLC, she submits her HTLC-timeout transaction spending an output of her State transaction.

Both parties constantly look for an old (revoked) State transaction put on-chain by their partner, and if they find such a transaction they use the corresponding per-commitment key to spend its first output, thus obtaining the penalty funds and revoking the old state. They also monitor the blockchain for current (not revoked) State transactions put on-chain by their partner, and whenever such a State transaction is found, they attempt to spend its HTLC control output(s) by submitting their HTLC-timeout and HTLC-success transactions as soon as possible.

---

3 This step requires that each party shares their per-commitment pubkey for the new state with the other party.

4 Specifically, each party sends their partial signature for every transaction input that requires it, but only if the transaction can be put on-chain by the other party.

5 The current Lightning protocol also has a *timeout_deadline* parameter, but that parameter specifies the amount of time **after** the HTLC's expiry that the given party goes on-chain.

If a State transaction has been on-chain for *tsdAB* without its first output being spent, the party that put the State transaction on-chain submits their corresponding Commitment transaction as soon as possible. Once a Commitment transaction is on-chain, each party puts their HTLC-refund and HTLC-payment transactions that spend its HTLC outputs on-chain (as determined by whether the corresponding HTLC-timeout or HTLC-success transaction is on-chain[6]).

Fees

If the fees in a signed off-chain transaction are insufficient when that transaction is submitted, the transaction can be submitted as part of a package where the transaction's child provides the required additional fees **[Zha22]**.

3.3 Correctness

Two-Input Transactions

All transactions in the current Lightning protocol have a single input, while the Commitment, HTLC-refund and HTLC-payment transactions in the TP protocol each have two inputs. The first input of these two-input transactions is a value input that carries channel funds and requires both parties' signatures (using the SIGHASH_ALL flag). The second input is a control input that only requires one party's signature. These two-input transactions are quite different from the Lightning protocol's one-input transactions, so it is worth examining them in detail.

First, because the first input requires both parties' signatures, only two-input transactions that have been signed by both parties can be used. Second, because the signatures for the first input use the SIGHASH_ALL flag, they force the source of the second input to have the specified transaction ID. Third, because each transaction's ID is a function of the transaction IDs of all of its parents (and indirectly, of all of its ancestors), all of the ancestors of the two-input transactions must have the specified transaction IDs. For example, Alice's partial signature on the first input of Bob's Commitment transaction forces its second input to spend the first output of Bob's State transaction for the same state, and indirectly requires Bob's State transaction for the same state to spend the output of his Individual transaction. Therefore, while Bob could spend the output of his Individual transaction in any way he chooses (without any constraints from Alice), spending it with anything other than the correct State transaction will prevent him from ever putting his correct Commitment transaction on-chain. Similarly, if the first output of Bob's State transaction is spent by anything other than his corresponding Commitment transaction, that Commitment transaction can never be put on-chain.

Per-Commitment Keys

In this way, Bob's Commitment transaction for state i, as well as the HTLC-refund and HTLC-payment transactions that spend the HTLC outputs from his Commitment transaction for state i, all require that

6 If neither the corresponding HTLC-timeout or HTLC-success transaction is on-chain, the HTLC-refund or HTLC-payment transaction is put on-chain once the corresponding HTLC-timeout or HTLC-success transaction is on-chain.

the first output of Bob's State *i* transaction be spent by his Commitment transaction for state *i*. Therefore, revoking an old State transaction by spending its first output also revokes all of the same party's Commitment transactions (for all states) and their associated HTLC-refund and HTLC-payment transactions. This is why there is no need to separately revoke any of the outputs from the Commitment, HTLC-refund, HTLC-payment, HTLC-timeout or HTLC-success transactions.

These dependent revocations also allow the TP protocol to use a different type of key for revoking old states. In the Lightning protocol the party that puts an old transaction on-chain cannot know the key for revoking the transaction. This is necessary in Lightning because if that party did know the revocation key, they could intentionally put an old Commitment transaction on-chain and then "revoke" the outputs of that transaction, thus taking the value for themselves. In contrast, in the TP protocol, if a party puts an old State transaction on-chain and then revokes it by spending its first output with a transaction other than its associated Commitment transaction, that party is actually blocking their ability to ever put a Commitment transaction on-chain. Furthermore, the funds that party obtains by spending the State transaction's first output are the same penalty funds that they provided from their Individual transaction. As a result, it is safe in the TP protocol to allow a party to revoke their own State transaction.

In the Lightning protocol, revocation pubkeys for the first party's transactions are created by taking a linear combination of a revocation basepoint, with a private key known only to the second party, and a per-commitment point, with a private key known only to the first party (until it is revealed to the second party, thus revoking the transaction)[7]. In contrast, in the TP protocol the first party's transactions are revoked using per-commitment keys that are exactly the private keys of the per-commitment points used by the Lightning protocol **[Rus]**. The ability to use per-commitment keys that are known to the party putting the revocable transaction on-chain has important implications for untrusted watchtowers, as will be shown in Section 4.

Establishing the Correct Commitment Transaction

Commitment transactions require both parties' signatures, so only Commitment transactions for mutually-agreed channel states can be put on-chain. Each party prevents the other party from putting an old Commitment transaction on-chain by monitoring the blockchain for revoked State transactions. When they find a revoked State transaction, they spend its first output, thus preventing the corresponding Commitment transaction from being put on-chain. Because there is a relative delay of *tsdAB* (the maximum of both parties' *to_self_delay* parameters) before that Commitment transaction can be put on-chain, the party revoking the transaction has sufficient time to do so. Finally, each party can get the current Commitment transaction on-chain by putting their current State transaction on-chain, waiting for the *tsdAB* relative delay, and then submitting their current Commitment transaction.

7 In addition, the per-commitment points for successive states are defined in a manner that allows the second party to use only $O(\log S)$ storage to recalculate the private key for any of a maximum of S previously-revoked states (see **[Rus]**).

Either their current Commitment transaction, or the other party's current Commitment transaction, will then appear on-chain.

Correctly Resolving HTLCs

The correct Commitment transaction has the correct *to_self* and *to_remote* outputs, as well as HTLC outputs for all currently-outstanding HTLCs. We will now show that the TP protocol correctly resolves each of those HTLCs.

While having separate, non-conflicting State transactions enables tunable penalties, it makes HTLC resolution more complex. As can be seen in Figure 1, the parties' Commitment transactions conflict and the party whose Commitment transaction appears on-chain (called the *winning party*) also put the State transaction with the HTLC control outputs that determine whether HTLCs succeed or time out. Therefore, in order to show that an HTLC is resolved correctly, we must show that the corresponding HTLC control output in the winning party's State transaction is resolved correctly. Two problems are possible when resolving an HTLC.

First, consider the HTLC offered by Alice to Bob in Figure 1. If Bob revealed the HTLC's secret to Alice well before the HTLC's expiry but cannot update the channel state off-chain, he can put his State transaction and its associated HTLC-success transaction on-chain. However, there is a risk that sometime after the HTLC's expiry Alice will put her State transaction and its associated HTLC-timeout transaction on-chain. Therefore, if Alice is the winning party, Bob will not be paid by the HTLC despite having revealed its secret on time.

Second, consider the case where Alice puts her State and its associated HTLC-timeout transactions on-chain after the HTLC's expiry. In this case, there is a risk that much later Bob will put his State and its associated HTLC-success transactions on-chain (thus revealing the HTLC's secret far later than was agreed to in the HTLC). Therefore, if Bob is the winning party, Alice will have to pay the HTLC despite not receiving its secret on time.

The TP protocol avoids both of these problems by using racing Commitment transactions. Note that the relative delay from each party's State transaction to their associated Commitment transaction is the same (namely *tsdAB*), so if one party's State transaction appears earlier in the blockchain than the other party's State transaction, the first party has an advantage in winning the race to get their Commitment transaction on-chain. In fact, it can be proven that if one party's State transaction appears in the blockchain a sufficient number of blocks earlier than the other party's State transaction, the party with the earlier State transaction is guaranteed to be able to win the race in getting their Commitment transaction on-chain.

This is fortunate, because having the winning party's State transaction on-chain early relative to the HTLC's expiry guarantees that:

- Bob can force payment of the HTLC by putting his HTLC-success transaction (spending an output of the winning party's State transaction) on-chain before the HTLC's expiry (provided he knows the HTLC's secret), and
- Alice can force the prompt resolution of the HTLC[8] by submitting her HTLC-timeout transaction (spending an output of the winning party's State transaction) upon the HTLC's expiry.

As a result, each party is able to force the winning party's State transaction to be on-chain early relative to the HTLC's expiry (by submitting their State transaction even earlier relative to the HTLC's expiry), thus guaranteeing that the HTLC is resolved correctly. A detailed proof of correctness, which formalizes this reasoning, is given in Appendix B.

4 Efficient Watchtowers

As was noted in Section 3.3, in the Lightning protocol the party that puts a revocable transaction on-chain cannot know the revocation key for that transaction, as such knowledge could lead to the theft of the channel's funds. This constraint affects the use of an untrusted watchtower for the Lightning protocol. If the watchtower knew the revocation key for a given transaction, the watchtower could give the revocation key to the party that puts the revocable transaction on-chain[9], thus allowing it to steal funds. As a result, with the Lightning protocol, when a party uses an untrusted watchtower they give the watchtower signatures for penalty transactions that revoke their channel partner's old transactions. There is no efficient compaction scheme known for these signatures, so the watchtower must use $O(S)$ storage to be able to revoke S old transactions.

Because the TP protocol revokes old transactions by using per-commitment keys, rather than revocation keys, a party using the TP protocol can give an untrusted watchtower the per-commitment keys for all states that have been revoked by their partner. The watchtower can use the Lightning protocol's compact storage technique for revocation keys to consume only $O(\log S)$ storage to revoke a maximum of S old transactions **[Rus]**. Conveniently, the watchtower can also take the penalty amount as payment for the service it provided. The watchtower must be given the UTXO of the partner's Individual transaction's output in order to detect a revoked transaction[10], but the watchtower will see no association between that UTXO and the Funding transaction or any other channel state.

8 The resolution of the HTLC is either that it is timed out (if her HTLC-timeout transaction spending an output of the winning party's State transaction appears on-chain) or the HTLC's secret is revealed (if Bob's conflicting HTLC-success transaction appears on-chain).

9 In fact, the watchtower and the party putting the revocable transaction on-chain could be the same party.

10 In addition, the watchtower must be given the state-independent portion of the redeem script or Taproot witness for the State transaction's first output, including the *tsdAB* delay value (if included in the script), the other party's public key and the tapleaf version number.

5 Extensions

Unilateral Close after an Old Transaction is Put On-Chain

If a party accidentally puts on old State transaction on-chain, they only lose the penalty amount that is the output of that transaction (and potentially some of the minimal values of that transaction's HTLC control outputs). However, once their State transaction has been revoked, they have lost the ability to force a unilateral close of the channel.

To address this, it is possible to add a Trigger (or Kickoff **[BDW18]**) transaction that spends the output of the Funding transaction and to modify the Commitment transactions to spend the output of the Trigger transaction (rather than the Funding transaction). If the Trigger transaction's output has not been spent after a suitable delay, either party can initiate the Decker-Wattenhofer protocol **[DW15]** for settling the channel. In particular, the first transaction from the Decker-Wattenhofer protocol should have a relative delay of 3*tsdAB* in order to guarantee that the party that did not put the old State transaction on-chain has time to put their correct State and Commitment transaction on-chain, and thus resolve the HTLCs correctly[11].

Note that while this change allows the penalized party to unilaterally close the channel, the penalized party cannot force it to close quickly enough to guarantee that the HTLCs will be resolved correctly.

Off-Chain Control Outputs

In Figure 1, each party has an on-chain Individual transaction with an unspent output (UTXO) that is used as the input to their State transaction. However, if a party operates multiple channels, they do not have to have multiple UTXOs to fund the State transactions for each of them. Instead, they could have a single UTXO that can be spent by their (currently off-chain) Fan-Out transaction which has a separate output for each channel operated by that party[12]. Of course, the single UTXO must have sufficient funds to provide the penalty amount and HTLC control amounts for all of their channels' State transactions.

Watchtower-Freedom for Casual Users

A recent paper introduced a Watchtower-Free (WF) protocol for Casual Lightning Users (CLUs) who do not want to use a watchtower **[Law22]** or route Lightning payments for others. While that paper modified the current Lightning protocol to obtain a watchtower-free protocol, an analogous change can be made to the TP protocol to make it watchtower-free. Specifically, a relative delay of the Dedicated-Lightning-User's (DLU's) *to_self_delay* is added before each of the CLU's HTLC-timeout transactions can be put on-chain.

---

11 In particular, their correct State transaction can be put on-chain less than *tsdAB* after the Trigger transaction is put on-chain, and their Commitment transaction can be put on-chain sometime between 2*tsdAB* and 3*tsdAB* after their State transaction is put on-chain.

12 Or, more generally, they could have an off-chain tree of Fan-Out transactions that have the required outputs for all of their channels' State transactions at the leaves.

While the resulting protocol is watchtower-free for casual users, it is not suitable for them as it can require that users submit their Commitment transaction exactly *tsdAB* (e.g., weeks or months) after the expiry of an HTLC.

6 Related Work

The protocol presented here makes extensive use of previously-published protocols, namely the Poon-Dryja Lightning channel protocol **[PD16]** and the BOLT specifications **[BOLT]**. The compact storage technique for per-commitment keys comes from the compact storage technique for revocation keys created by Russell **[Rus]**.

Riard, ZmnSCPxj and Decker examined the idea of adding penalties to the eltoo protocol **[RZD19]**, but the techniques used there were different, as they did not use separate value and control transactions.

Rubin (who also credited nullc and sipa) **[Rub21]** showed how a two-input transaction with separate value and control inputs can be used to delegate the control of a UTXO (the value input) to another UTXO (the control input). This two-input transaction is similar to the TP protocol's Commitment, HTLC-refund and HTLC-payment transactions, but its value input does not require a multi-signature, so the delegation can be revoked by the delegator via a double-spend. In contrast, the TP protocol's Commitment, HTLC-refund and HTLC-payment transactions have multi-signature value inputs, so they cannot be revoked by one party without the other party's agreement.

The idea of using separate value and control transactions was presented by Law in the update-forest and challenge-and-response protocols **[Law21]**, but those protocols were for channel factories as opposed to channels, and they assumed a change to the underlying Bitcoin protocol.

7 Conclusions

This paper presents a new channel protocol that allows one to select the penalty for putting an old transaction on-chain. In addition, it reduces the storage costs for untrusted watchtowers from $O(S)$ to $O(\log S)$, where S is the number of channel states supported.

References.

AOP21 Andreas Antonopoulos, Olaoluwa Osuntokun and Rene Pickhardt. Mastering the Lightning Network, 1st. ed. 2021.

BDW18 Conrad Burchert, Christian Decker and Roger Wattenhofer. Scalable Funding of Bitcoin Micropayment Channel Networks. In Royal Society Open Science, 20 July 2018. See http://dx.doi.org/10.1098/rsos.180089.

BOLT BOLT (Basis Of Lightning Technology) specifications. See https://github.com/lightningnetwork/lightning-rfc.

BWHTLC Bitcoin Wiki: Hash Timelocked Contracts. See https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts.

Dec20 Christian Decker. Re: Simulating Eltoo Factories using SCU Escrows (aka SCUE'd Eltoo). See https://www.mail-archive.com/lightning-dev@lists.linuxfoundation.org/msg02046.html.

DW15 Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In Proc. 17th Intl. Symposium on Stabilization, Safety, and Security of Distributed Systems, August 2015. pp. 3-18. See https://tik-old.ee.ethz.ch/file/716b955c130e6c703fac336ea17b1670/duplex-micropayment-channels.pdf.

GMRMG Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry and Arthur Gervais. SoK: Layer-Two Blockchain Protocols. See https://eprint.iacr.org/2019/360.pdf.

Law21 John Law. Scaling Bitcoin With Inherited IDs. See https://github.com/JohnLaw2/btc-iids.

Law22 John Law. Watchtower-Free Lightning Channels For Casual Users. See https://github.com/JohnLaw2/ln-watchtower-free.

PD16 Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments (Draft Version 0.5.9.2). January 14, 2016. See https://lightning.network/lightning-network-paper.pdf.

RZD19 Antoine Riard, ZmnSCPxj, Christian Decker. Using Per-Update Credential to enable Eltoo-Penalty. July 13, 2019. See thread at https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-July/002064.html.

Rub21 Jeremy Rubin. Delegated signatures in Bitcoin within existing rules, no fork required. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-March/018615.html

Rub22 Jeremy Rubin. Re: 'OP_EVICT': An Alternative to 'OP_TAPLEAFUPDATEVERIFY'. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-February/019945.html.

Rus Rusty Russell. Efficient Per-Commitment Secret Storage. See https://github.com/lightning/bolts/blob/master/03-transactions.md#efficient-per-commitment-secret-storage.

San22 Greg Sanders. Re: 'OP_EVICT': An Alternative to 'OP_TAPLEAFUPDATEVERIFY'. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-February/019947.html.

Zha22 Gloria Zhao. Package Relay Proposal. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-May/020493.html.

Appendix A: Proof of Correctness

The proof of correctness of the TP protocol will use the timing model from Appendix A of **[Law22]**, including its parameters *R*, *S*, *G*, *B*, *U* and *L*.

Theorem 1: If the first party has not signed a Cooperative Close transaction, and the first party's current State transaction is on-chain more than L blocks before the second party's State transaction (or if the second party's State transaction is never on-chain), the first party is guaranteed to be able to get their associated Commitment transaction on-chain and thus become the winning party.

Proof: Let T_1 denote the block containing the first party's current State transaction. At time $T_1 + tsdAB$, the first party submits their associated Commitment transaction[13]. Therefore, by time $T_1 + tsdAB + L$, either the first party's Commitment transaction is fixed, or a transaction that conflicts with that Commitment transaction is fixed. Assume for the sake of contradiction that the first party's Commitment transaction is not fixed. Because no Cooperative Close transaction has been signed, it follows that the second party's Commitment transaction is fixed by time $T_1 + tsdAB + L$. Let $T_2 \leq T_1 + tsdAB + L$ denote the block containing the second party's Commitment transaction. The second party's Commitment transaction has a relative delay of $tsdAB$ after its associated State transaction, which implies that the second party's State transaction is fixed in block $T_3 \leq T_2 - tsdAB \leq T_1 + tsdAB + L - tsdAB = T_1 + L$, which contradicts the theorem's assumptions. □

We will examine an HTLC offered by Alice to Bob (as in Figure 1) with an expiry of $eAB = eBC + \Delta$ where eBC is the expiry of the corresponding HTLC in the channel Bob shares with Carol.

If Bob knows the HTLC's secret by $eAB - fulfillment\_deadline$, he will submit his State transaction by $eAB - fulfillment\_deadline$, his State transaction will be fixed by $eAB - fulfillment\_deadline + L$, and it follows from Theorem 1 that the winning party's State transaction will be fixed by $eAB - fulfillment\_deadline + 2L$. Bob will submit his HTLC-success transaction that spends the corresponding HTLC control output in the winning party's State transaction by $eAB - fulfillment\_deadline + 2L$ and it will be fixed by $eAB - fulfillment\_deadline + 3L$. Therefore, if $fulfillment\_deadline \geq 3L$, Bob's HTLC-success transaction spending an output in the winning party's State transaction will be fixed by eAB at the latest (as Alice is therefore unable to submit her conflicting HTLC-timeout transaction), and the HTLC will be resolved correctly.

If Bob attempts to time out the HTLC in the channel with Carol by submitting his State transaction by $eBC - timeout\_deadline$, his State transaction will be fixed by $eBC - timeout\_deadline + L$, and it follows from Theorem 1 that the winning party's State transaction will be fixed by $eBC - timeout\_deadline + 2L$. Therefore, if $timeout\_deadline \geq 2L$, the winning party's State transaction will be fixed by eBC, so Bob will submit his HTLC-timeout transaction spending an output in the winning party's State transaction by eBC, and this HTLC-timeout transaction or Carol's conflicting HTLC-success transaction will be fixed by $eBC + L$.

Therefore, Bob sets his $timeout\_deadline$ to $2L$, his $fulfillment\_deadline$ to $3L$, and his $cltv\_expiry\_delta$ to $L + G + 3L = G + 4L$, where G is a grace period that is long enough to update the channel state off-chain. Given these parameters, if Bob pays the HTLC in the channel with Carol, he

13 Note that the first party's State transaction is current, so it has not been revoked, which implies that its first output cannot be spent by any other party.

will know the HTLC's secret by $eBC + L = eAB - G - 4L + L = eAB - G - 3L = eAB -$ *fulfillment_deadline* $- G$. Thus, he will either update the channel state with Alice within time G, or he will go on-chain by $eAB -$ *fulfillment_deadline*, so in either case he will receive the HTLC payment in the channel with Alice.

Finally, it will be required that Alice's *timeout_deadline* is no larger than Bob's *fulfillment_deadline*, so Alice and Bob will have at least G time (from $eBC + L$ to $eBC + L + G$) to update the channel state off-chain, so the channel will remain off-chain if both parties follow the TP protocol.

Appendix B: Timing Parameters

The timing model from Appendix A of **[Law22]** and the analysis from Appendix A above will be used to determine the timing parameters for the TP protocol as follows:

timeout_deadline

In the TP protocol, the *timeout_deadline* parameter gives the number of blocks **before** the HTLC expiry when the party offering the HTLC goes on-chain unless both parties' Commitment transactions are updated to reflect that the HTLC has timed out. The *timeout_deadline* is set to $2L$, as was explained above in Appendix A[14].

cltv_expiry_delta

The *cltv_expiry_delta* parameter gives the number of blocks between the HTLC expiry in the next hop and the HTLC expiry in the current hop. The *cltv_expiry_delta* must be large enough to guarantee that if the HTLC is fulfilled in the next hop, it will also be fulfilled in the current hop. The *cltv_expiry_delta* is set to $4L + G$, as was explained above in Appendix A.

fulfillment_deadline

The *fulfillment_deadline* parameter gives the number of blocks before the HTLC expiry when the party receiving the HTLC goes on-chain[15] unless both parties' Commitment transactions are updated to reflect that the HTLC has been fulfilled. This parameter is set to $3L$, as was explained above in Appendix A.

min_final_cltv_expiry

The *min_final_cltv_expiry* parameter gives the number of blocks from the current block height to the HTLC expiry in the last hop. This parameter needs to support a grace period of length G followed by a *fulfillment_deadline*, so it will be set to $G +$ *fulfillment_deadline* $= G + 3L$.

---

14 In addition, each party's *timeout_deadline* is required to be no larger than the other party's *fulfillment_deadline*. This is enforced in the TP protocol by having each party send their *timeout_deadline* to the other party during channel set-up. If the received *timeout_deadline* is too large, the party receiving it fails the channel set-up.

15 Assuming that party has fulfilled the HTLC.

to_self_delay

The *to_self_delay* parameter gives the relative delay before the other party in a channel can receive the payout from a transaction that the other party has put on-chain. This parameter is required to allow the current party to put a conflicting transaction on-chain (thus preventing payout to the other party) in the case where the other party's transaction violates the channel protocol (e.g., it is for an old state). In the worst case:

1. The current party first detects the other party's transaction when it is R+1 blocks deep in the current party's version of the blockchain.
2. The current party then experiences U unintentional unavailability before they are able to submit their conflicting transaction.
3. This conflicting transaction then takes L blocks before it is permanently in every version of the blockchain, thus preventing the other party from performing their attempted spend.

The above analysis establishes the following inequality:

$$to\_self\_delay \geq U + L + R + 1.$$