Scaling Lightning With Simple Covenants

John Law

October 4, 2023

Version 1.2

Abstract

The key challenge in scaling Lightning in a trust-free manner is the creation of Lightning channels for casual users. Signature-based factories appear to be inherently limited to creating at most tens or hundreds of Lightning channels per unspent transaction output (UTXO). In contrast, simple covenants (including those enabled by CheckTemplateVerify (CTV) or AnyPrevOut (APO)) would allow a single UTXO to create Lightning channels for millions of casual users. The resulting covenant-based protocols also 1) support resizing channels off-chain, 2) use the same capital to simultaneously provide in-bound liquidity to casual users and route unrelated payments for other users, 3) charge casual users tunable penalties for attempting to put an old state on-chain, and 4) allow casual users to monitor the blockchain for just a few minutes every few months without employing a watchtower service. As a result, adding CTV and/or APO to Bitcoin's consensus rules would go a long way toward making Lightning a widely-used means of payment.

1 Overview

There has recently been a great deal of interest in understanding the potential benefits of modifying the Bitcoin protocol to support covenants that place certain restrictions on child and descendant transactions. Covenant-based protocols have been created to improve the usability of Bitcoin (e.g., by supporting vaults) and to reduce the fees paid for putting transactions on-chain **[BIP118][BIP119]**. While these protocols solve important problems, the use of covenants to improve Lightning's (and Bitcoin's) scalability has received less attention. This paper shows how covenants can be used to dramatically scale the provision of trust-free Lightning channels by eliminating the scalability bound imposed by the use of signatures. In particular, it shows that covenants, including the restricted covenants enabled by CheckTemplateVerify (CTV) **[BIP119]** or AnyPrevOut (APO) **[BIP118]**, can be used to dramatically improve Lightning's scalability while providing good capital efficiency and meeting the usability requirements of casual users.

The remainder of this paper is organized as follows. Section 2 demonstrates that the key challenge in scaling Lightning is providing Lightning channels to casual users while meeting their usability requirements. Section 3 then shows why no protocol that uses Bitcoin's current consensus rules is likely to meet this challenge. Covenant-based protocols for creating Lightning channels are presented in Section 4, and they are analyzed in Section 5. The remaining sections look at related work and give conclusions and acknowledgments.

2 Casual Users And The Scaling Problem

2.1 Casual and Dedicated Users

If Bitcoin and Lightning are to become widely-used, they will have to be adopted by casual users who want to send and receive bitcoin, but who do not want to go to any effort in order to provide the infrastructure for making payments. Instead, it is reasonable to expect that the Lightning infrastructure will be provided by dedicated users who are far less numerous than the casual users. In fact, there are likely to be tens-of-thousands to millions of casual users per dedicated user. This difference in numbers implies that the key challenge in scaling Bitcoin and Lightning is providing bitcoin and Lightning to casual users. As a result, the rest of this paper will focus on this challenge.

2.2 Usability

Known Lightning protocols **[Law22a][Law22c]** allow casual users to perform Lightning payments without:

- maintaining high-availability,
- performing actions at specific times in the future, or
- having to trust a third-party (such as a watchtower service).

In addition, they support tunable penalties for casual users who attempt to put an old channel state on-chain (for example, due to a crash that causes a loss of state) **[Law22b]**. As a result, these protocols meet casual users' needs and could become widely-used for payments if they were sufficiently scalable.

2.3 Scalability

The Lightning Network lets users send and receive bitcoin off-chain in a trust-free manner **[AOP21] [BOLT]**. Furthermore, there are Lightning protocols that allow Lightning channels to be resized off-chain **[Law23a]**. Therefore, making Lightning payments and resizing Lightning channels are highly scalable operations.

However, providing Lightning channels to casual users is not scalable. In particular, no known protocol that uses the current Bitcoin consensus rules allows a large number (e.g., tens-of-thousands to millions) of Lightning channels, each co-owned by a casual user, to be created from a single on-chain unspent

transaction output (UTXO). As a result, being able to create (and close) casual users' Lightning channels remains the key bottleneck in scaling Lightning.

3 Casual Users And The Signature Bound

Unfortunately, there are good reasons to believe this bottleneck is unavoidable given the current Bitcoin consensus rules. The problem is that in order for a casual user to co-own a Lightning channel, they must co-own an on-chain UTXO **[BDW18]**. Therefore, if a large number of casual users are to each co-own a Lightning channel, all of which are funded by a single UTXO, that UTXO must require signatures from all of those casual users.

In practice, the problem is much harder than just getting signatures from a large number of casual users, as the signatures themselves depend on the exact set of casual users whose signatures are required. For example, if a UTXO requires signatures from a set of 1,000 casual users and if 999 of them sign but one does not, the 999 signatures that were obtained cannot be used. Instead, one has to start all over again, say with a new UTXO that requires signatures from the 999 users that signed the previous time. However, if not all of those 999 users sign, the signatures that were obtained in the second try are also unusable.

The requirement for casual users to sign transactions that specify the exact set of casual users whose signatures are required creates a very difficult group coordination problem that is not well-suited to the behavior of casual users (**[Law21]** Section 2.2). As a result, while a signature-based channel factory could be used to fund channels for perhaps 10 or even 100 casual users, it appears unlikely that any protocol using the current Bitcoin consensus rules can fund tens-of-thousands to millions of channels from a single UTXO.

4 Covenant-Based Protocols For Scaling Lightning

4.1 Simple Covenant Requirements

This paper will only consider protocols that require an extremely simple form of covenant. Specifically, the covenants used here must allow a locking script (scriptPubKey) to define the spending transaction's:

1. number of outputs, and
2. for each output, the amount of the output and its locking script (scriptPubKey).

Note that the covenant does not have to define the spending transaction's nLocktime, number of inputs, or nSequence values[1]. The locking script must be able to allow multiple spending options, where only

1 Some proposals that enable covenants, including CTV, create covenants that **do** define these fields. Such covenants can be used for the protocols given here, although these protocols do not require that these fields be defined by covenants.

one of those options requires meeting the given covenant[2]. Because the covenant can define the spending transaction's locking script, it is possible to create recursive covenants that define a tree of transactions. However, in such a case, the recursion is always bounded. The simple covenants required here could be implemented if CTV and/or APO were added to Bitcoin's consensus rules.

4.2 Timeout-Trees

If the consensus rules are changed to allow even the simple covenants defined above, the scaling bottleneck imposed by signatures is eliminated. The key observation is that with covenants, a casual user can co-own an off-chain Lightning channel without having to sign all (or any) of the transactions on which it depends. Instead, a UTXO can have a covenant that guarantees the creation of the casual user's channel. The simplest way to have a single UTXO create channels for a large number of casual users is to put a covenant on the UTXO that forces the creation of a tree of transactions, the leaves of which are the casual users' channels.

While such a covenant tree can create channels for millions of casual users without requiring signatures or solving a difficult group coordination problem, it is not sufficient for scaling. The problem is that each channel created by a covenant tree has a fixed set of owners, and changing the ownership of a channel created by a covenant tree requires putting the channel on-chain. Therefore, assuming that all casual users will eventually want to pair with different dedicated users (and vice-versa), the covenant tree does not actually provide any long-term scaling benefit.

Fortunately, real long-term scaling can be achieved by adding to the covenant tree 1) leaf transactions that are defined by signatures (rather than covenants), and 2) a deadline after which all non-leaf outputs can be spent by the user who funded the tree. The result is called a *timeout-tree* (**[Law21]** Section 5.3).

Let A_1 ... A_n denote a large number of casual users, let B be a dedicated user, and let E denote some fixed time in the future. User B creates a timeout-tree with expiry E where:

- the timeout-tree consists of an on-chain funding transaction with an output that can be spent by a root transaction (which must satisfy a covenant), internal transactions (which must satisfy a covenant), and leaf transactions (which require signatures but do not have to satisfy a covenant),
- leaf *i* requires signatures from A_i and B, and has a single output that funds a Lightning channel owned by A_i and B, and
- after time E, each non-leaf output in the timeout-tree can also be spent by user B without having to meet any other conditions.

Thus, any time before E, casual user A_i can put the Lightning channel (A_i, B) on-chain by putting all of its ancestors in the timeout-tree on-chain. Once (A_i, B) is on-chain, the expiry E has no effect so A_i and B can continue to use the Lightning channel to send and receive payments from and to A_i.

2 For example, the locking script could allow the spending transaction to either 1) satisfy a given covenant, or 2) provide a signature for a given public key after a specific nLocktime is reached.

On the other hand, sometime shortly before E, casual user A_i can use the Lightning Network to drain all of their balance in the channel (A_i, B) and send it to themselves in some other Lightning channel that is the leaf of some other timeout-tree. More precisely, casual user A_i should rollover their balance by sending it from a given timeout-tree between time E - *to_self_delay_Ai* and time E, where E is the timeout-tree's expiry and *to_self_delay_Ai* is A_i's Lightning channel safety parameter. Note that *to_self_delay_Ai* can be in the range of 1 to 3 months if a watchtower-free channel protocol is used **[Law22a][Law22c]**, so performing the drain within this time window does not put an unreasonable availability requirement on A_i.

If all casual users drain their balances from the timeout-tree before E, then after E dedicated user B can create a new timeout-tree, with leaves that create Lightning channels for a new set of casual users, by putting a single transaction on-chain that spends the UTXO which created the expired timeout-tree. In this case, all *n* of the old Lightning channels are closed and *n* new channels are created with a single on-chain transaction.

Of course, it is possible that some casual users will put their Lightning channel in the old timeout-tree on-chain, while others will drain their balance from the timeout-tree before E. In this case, user B can create a new timeout-tree that is funded by the non-leaf outputs of the old timeout-tree that have been put on-chain. While this results in a larger on-chain footprint than the case in which all casual users drain their balances from the old timeout-tree, it can still provide substantial scaling as long as the number of leaves put on-chain is small (in particular, well below $n/(\log n)$). By creating incentives that reward users who drain their balances from the timeout-tree rather than putting their channels on-chain, almost all leaves will stay off-chain and good scalability will be achieved.

An example of a timeout-tree is shown in Figure 1. In Figure 1 (and throughout the paper):

- **B** denotes B's signature,
- pairs of capital letters indicate signatures from those two parties, and
- $\mathbf{E}_X$ denotes timeout-tree X's expiry.

Relative delays (not present in Figure 1) are shown without parentheses while absolute delays are shown within parentheses. 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,
- **R** for the Root transaction,
- **I** for an Internal transaction, and
- **L** for a Leaf transaction.

Subscripts within transactions denote the name of the timeout-tree (X in this example) and which party can put the transaction on-chain (if only one party can do so).

A line that connects to the right side of a box is an output, while one that connects to the left side of a box in an input. A line that ends with an arrow places a covenant on the transaction that spends it, while a line that does not end in an arrow does not place a covenant. Bold lines (such as all lines in Figure 1) carry channel funds, while thin dashed lines are control outputs that have value equal to the smallest possible (dust) amount.

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.

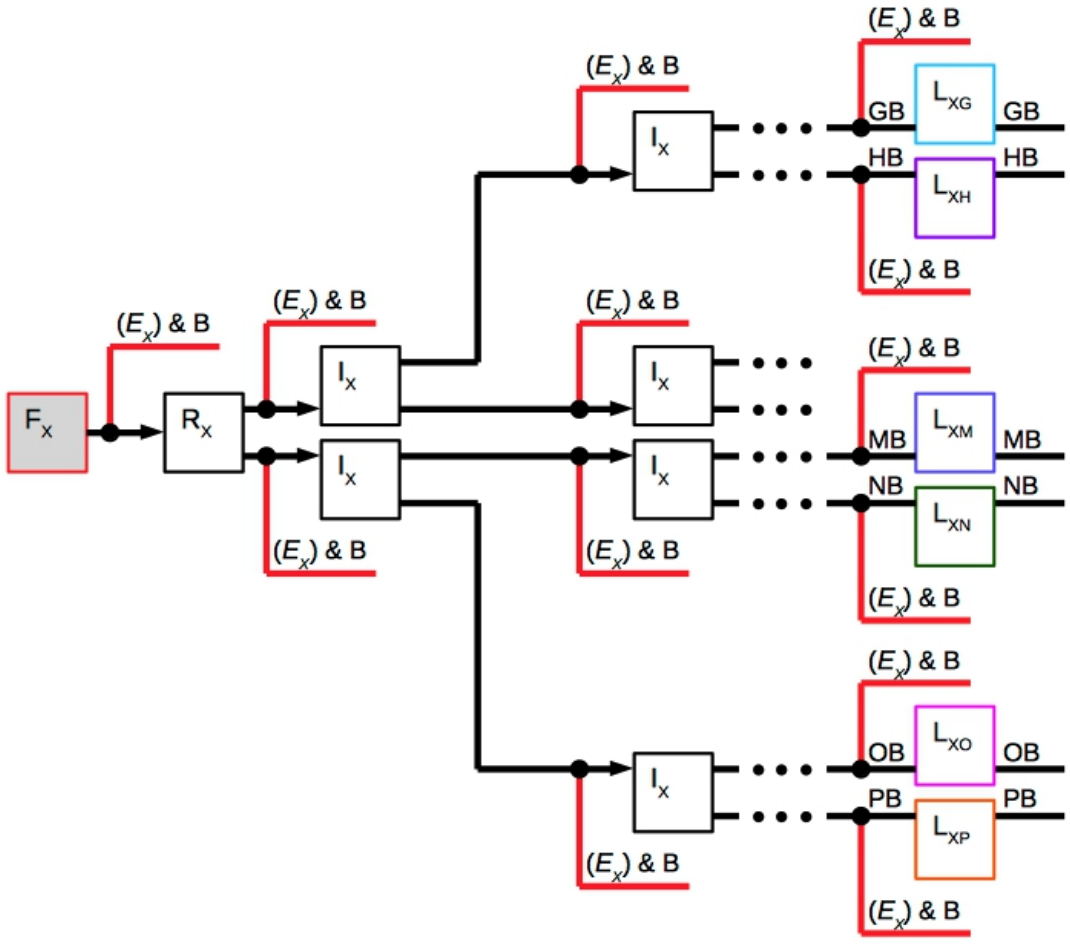

Figure 1. Timeout-tree X with an on-chain Funding transaction (F), where all other transactions in the timeout-tree are off-chain. Timeout-tree X was created by dedicated user B. The Funding, Root (R) and Internal (I) transactions are defined by covenants placed on them by their parent transaction. Several representative Leaf (L) transactions are shown that create channels owned by user B and casual users G, H, M, N, O and P, respectively.

Dedicated user B creates timeout-tree X by putting its Funding transaction (F) on-chain. F's sole output puts a covenant on the Root transaction (R) that spends it, and R in turn has two outputs, each of which puts a covenant on the Internal transaction (I) that spends it. Each Internal transaction has two outputs, each of which can be spent by either another Internal transaction (specified by a covenant) or a Leaf transaction (which requires signatures from B and the casual user that co-owns the Lightning channel created by the Leaf). Each Leaf transaction can only be put on-chain by a casual user as it requires signatures from B and that casual user, and B gives B's signature to the casual user but the casual user does not provide their signature to B. Each output from the Funding, Root and Internal transactions can also be spent by B in any manner (unconstrained by any covenants) after the timeout-tree's expiry E.

The number of levels in the timeout-tree could be reduced by one if the Funding transaction had two outputs (thus making it also play the role of the Root transaction). However, a Funding transaction with only one output can be spent more efficiently by B after expiry E, in the case where no other transaction in the timeout-tree has been put on-chain.

4.3 Lightning Channel Protocols For Timeout-Trees

Any Lightning channel protocol could be used for operating the channels created by a timeout-tree, including the current Lightning channel protocol or the eltoo protocol **[DRO18]** (if APO is supported). The remainder of this paper will focus on variants of the Fully-Factory-Optimized-Watchtower-Free (FFO-WF) protocol (**[Law22c]** Section 4) due to its usability, scalability and efficiency. As will be shown in the following subsections, variants of the FFO-WF protocol also allow casual users to passively rollover their Lightning funds from one timeout-tree to another and they allow casual users to maintain off-chain bitcoin (in addition to their funds within Lightning channels).

There are two changes that will be made in adapting the FFO-WF protocol to timeout-trees. Let A be the casual user and B be the dedicated user.

First, in the FFO-WF protocol B's Commitment transaction does not have any HTLC outputs, as all outstanding HTLCs are assumed to be resolved in B's favor if B's Commitment transaction is put on-chain. This is possible because B's Commitment transaction can only be put on-chain if A does not follow the protocol. In the original version of the FFO-WF protocol, B's Commitment transaction is kept off-chain by including an absolute delay in B's State transaction for the current state which allows A to put their State and Commitment transaction's on-chain first. This approach only works because in the original FFO-WF protocol, both A and B have to update the channel state once per A's *to_self_delay* parameter (e.g., every couple months), as A only pays B for B's cost-of-capital for this long. In contrast, in the FFO-WF protocol for timeout-trees, A must pay B for B's cost-of-capital for the entire life of the timeout-tree, which could be much longer (e.g., a couple years). Therefore, a different technique must be used to ensure that B's Commitment transaction can only be put on-chain if A fails to follow the protocol. The solution is to add a relative delay before B's Commitment transaction can spend the timeout-tree leaf's output that funds the Lightning channel, without having any such relative delay before A's Commitment transaction can spend that same output.

Second, in the original FFO-WF protocol, A's State transaction has a control output with a value that is set to the penalty amount for attempting to put an old (revoked) state on-chain. As a result, if A puts a State transaction for a revoked state on-chain, B (or any other user knowing the required per-commitment key **[Rus][Law22b]**) can spend this control output of A's State transaction, thus obtaining the penalty amount and preventing A from putting their corresponding (revoked) Commitment transaction on-chain. In contrast, in the timeout-tree version of the FFO-WF protocol, the control outputs of A's State transaction all have the smallest possible value. Instead, A is penalized by the agreed upon penalty amount by adjusting the values of the payouts to A and B from B's Commitment transaction (as B will be able to put their Commitment transaction on-chain when A attempts to put a revoked transaction on-chain).

This second change is not strictly necessary, but it does simplify the provisioning of A's State transaction, as will be described later in the paper. For a similar reason, the control outputs of B's State transaction are also set to the smallest possible value.

The timeout-tree version of the FFO-WF protocol that incorporates these two changes is shown in Figures 2 through 6 below, where:

- ***tsd{A|B}*** denotes {A's|B's} *to_self_delay* channel parameter,
- ***L*** denotes the maximum delay from submitting a transaction to its being put in its final position in the blockchain[3],
- ***eAB*** denotes the expiry for the given HTLC,
- ***pckey{A|B}i*** denotes {A's|B's} per-commitment key **[Rus][Law22b]** for channel state *i*, and
- **Preimage(P)** denotes the hash pre-image of P which is the secret of the given HTLC.

In addition to the timeout-tree transactions shown in Figure 1, these figures include transactions designated:

- **In** for an Individual control transaction,
- **St** for a State control transaction,
- **H-s** for an HTLC-success control transaction,
- **H-t** for an HTLC-timeout control transaction,
- **H-p** for an HTLC-payment value transaction, and
- **H-r** for an HTLC-refund value transaction.

3 The L parameter given here matches the L parameter in Appendix 1 of **[Law22a]** and it equals R+S where R and S are defined in the BOLT #2 specification **[BOLT]**.

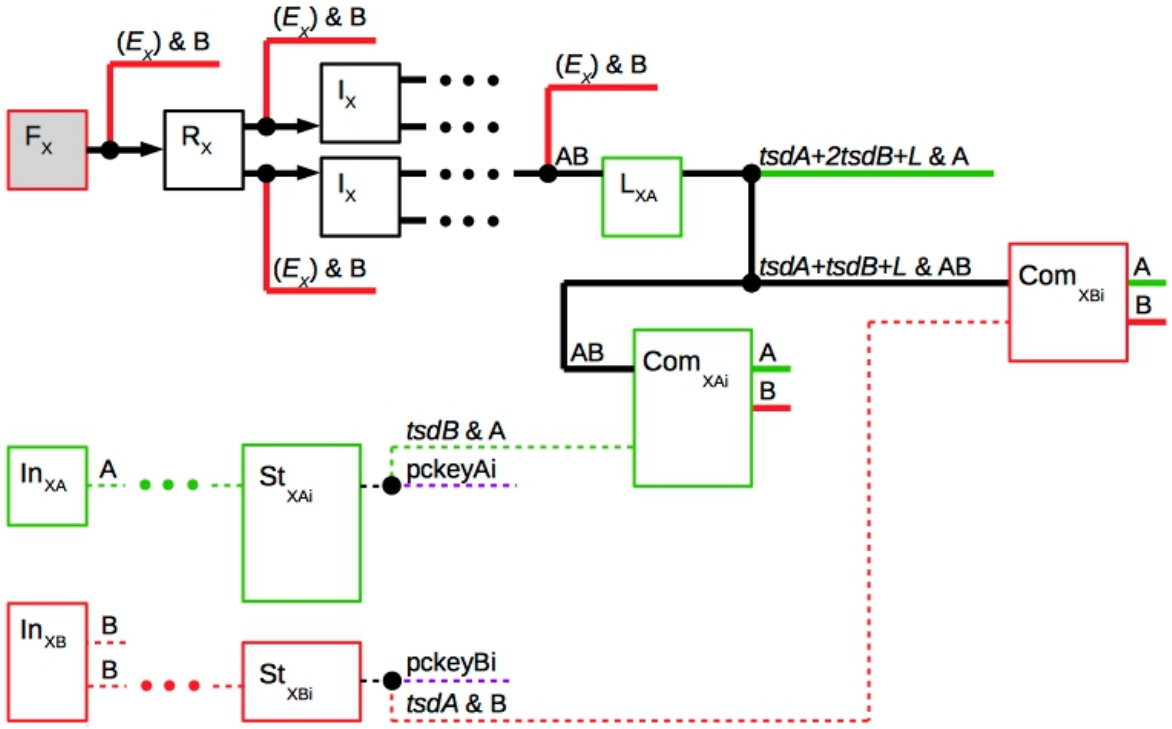

Figure 2. The timeout-tree version of the FFO-WF protocol for a channel owned by casual user A and dedicated user B. No HTLCs are currently outstanding in this channel.

In Figure 2, the relative delay before B can put Com_{XBi} on-chain is set to the sum of A's and B's *to_self_delay* parameters plus the latency parameter *L*. As a result, if A needs to submit transaction L_{XA} (for example, because A and B failed to rollover A's funds to another timeout-tree at least *tsdA* before the timeout-tree's expiry E_x), A can submit L_{XA} (and its ancestors) and St_{XAi} (and its ancestors) at time *T*. Then, by time *T+L* transaction St_{XAi} will be at its final position in the blockchain, so any time after *T+L+tsdB*, A can submit Com_{XAi} which will be at its final position in the blockchain by *T+L+tsdB+tsdA* (from the definition of *tsdA*), which is before B can submit the conflicting transaction Com_{XBi}.

Also, note in Figure 2 that A can spend the output of L_{XA} in any way A chooses after an addition relative delay of tsdB. This case is included in order to force B to put the current transaction Com_{XBi} on-chain after A has tried and failed to put an old state on-chain.

Finally, note that A's and B's Individual control transactions are shown as being off-chain. This is because they could be created by covenants on an on-chain UTXO, as will be described later in the paper.

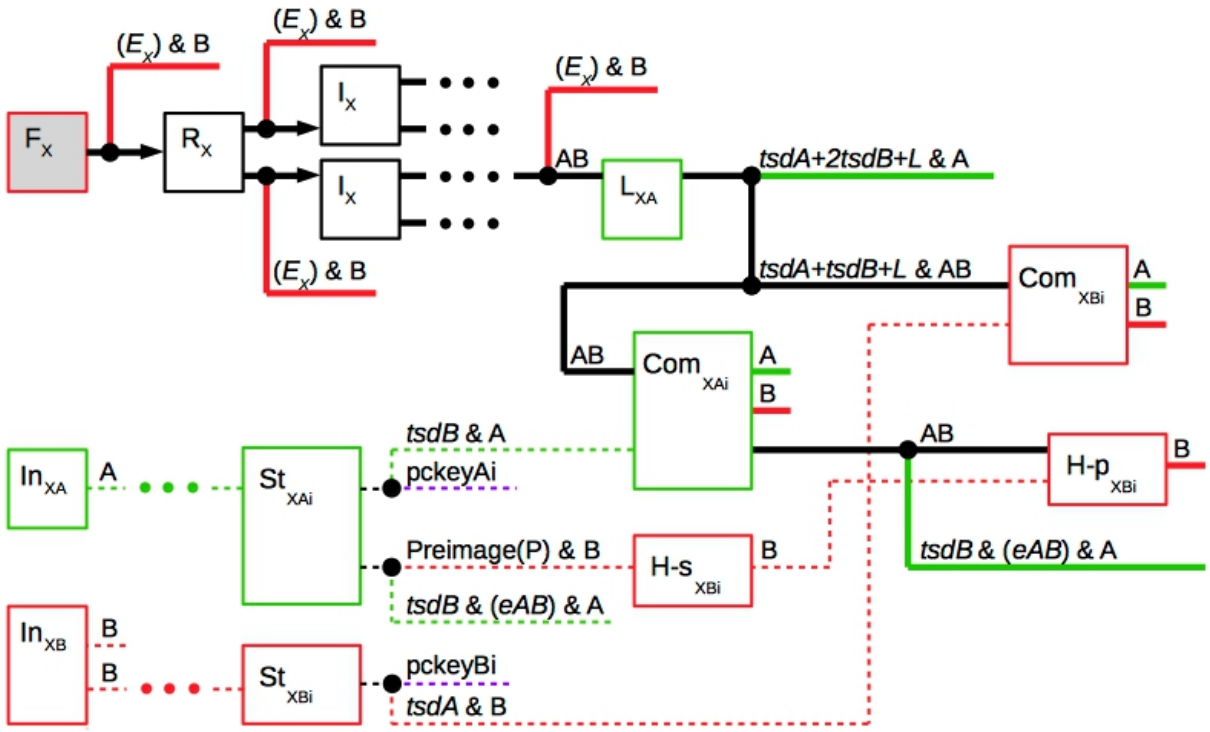

Figure 3. The timeout-tree version of the FFO-WF protocol with one HTLC offered by casual user A to dedicated user B.

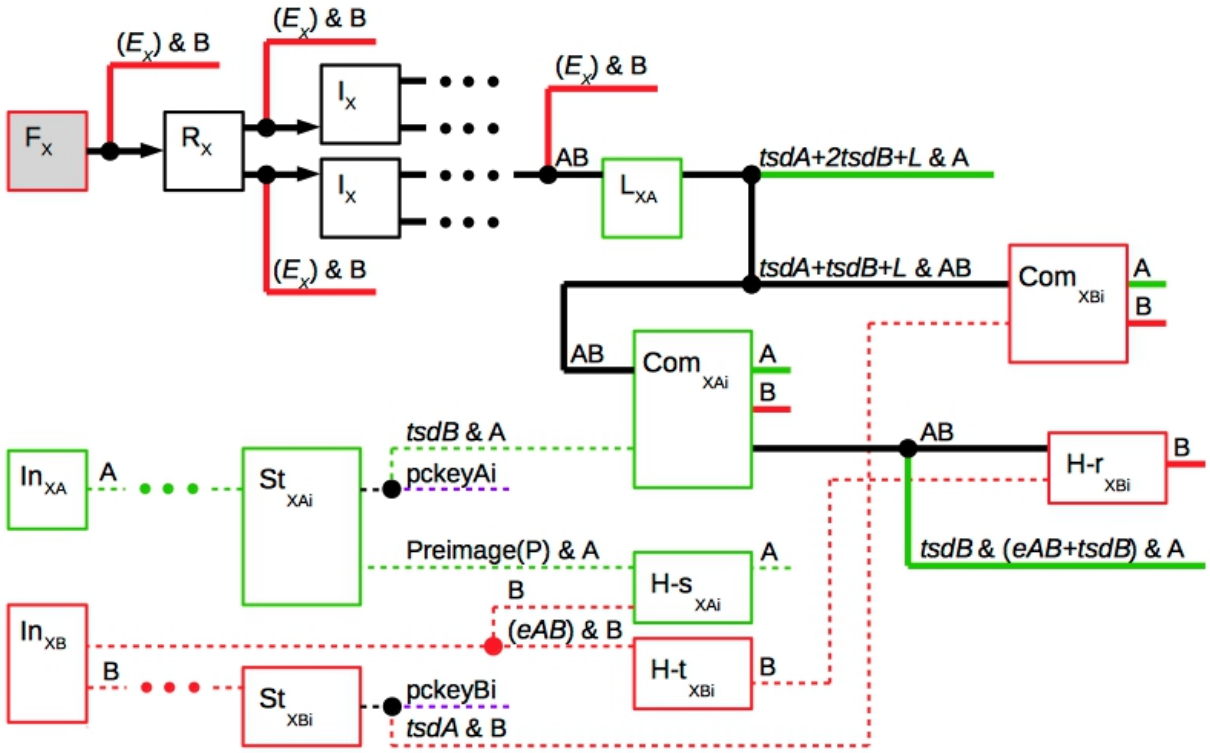

Figure 4. The timeout-tree version of the FFO-WF protocol with one HTLC offered by dedicated user B to casual user A.

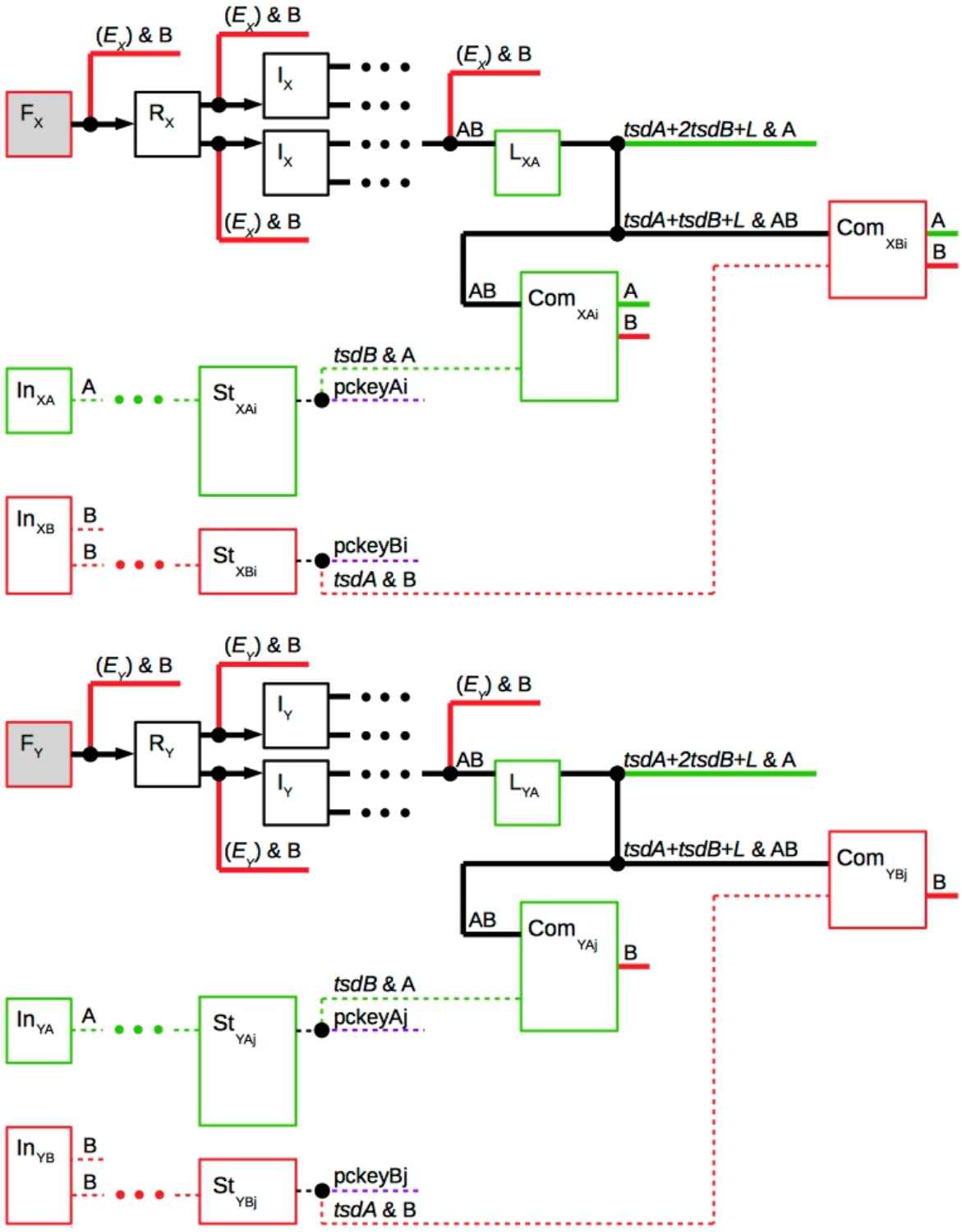

Figure 5. Timeout-trees X and Y before rolling-over A's balance from old timeout-tree X to new timeout-tree Y. Note that A has no balance in Y. Also note that X and Y have independent control transactions and independent state numbers.

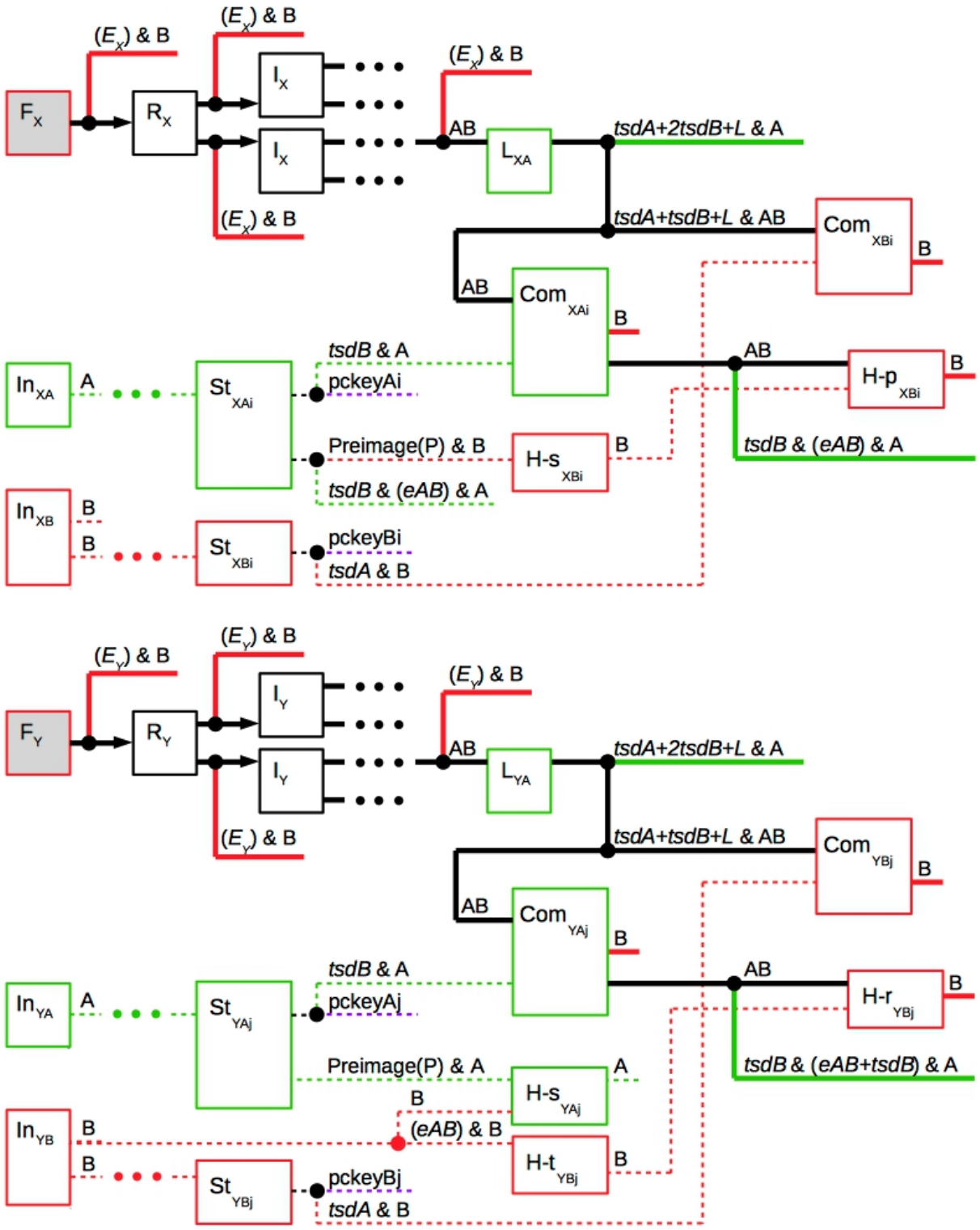

Figure 6. Timeout-trees X and Y during the rollover of A's funds from X to Y. A is sending all of their funds in X to themselves in Y via a Lightning Network payment.

4.4 Passive Rollovers For Casual Users

The timeout-trees defined above do not place unreasonable availability requirements on casual users and they allow a very large number of casual users to obtain a Lightning channel with a single on-chain transaction. However, there are two problems with forcing casual users to drain their balances from an old timeout-tree to a new timeout-tree before the old timeout-tree's expiry:

1. if a casual user fails to perform the required drain before the old timeout-tree's expiry (due to unexpected unavailability), they lose all of their funds in the timeout-tree, and
2. if the dedicated user B is unavailable when a casual user attempts to drain their funds prior to the timeout-tree's expiry, the casual user will put their timeout-tree leaf on-chain (thus increasing the on-chain footprint and limiting scalability).

This second problem matters, as a casual user should only have to devote a short period (e.g., 10 minutes) every few months to performing the drain, so even a short period of unavailability by the dedicated user could force the casual user to go on-chain. Instead, it would be preferable if the dedicated user could facilitate the rollover of the casual user's funds from a timeout-tree that is about to expire to another one without requiring input from the casual user.

In order to support a passive rollover for a casual user from an old timeout-tree X to a new timeout-tree Y, the Lightning protocol cannot be used to send the casual user's balance from X to Y (as such a Lightning transfer requires the casual user's cooperation). Instead, the dedicated user has to (somehow) create a Lightning channel in Y that has the casual user's Lightning channel balance from X without any cooperation from the casual user.

There are four problems that must be overcome to accomplish this:

1. the casual user must be prevented from obtaining their funds from their Lightning channels in both X and Y (as this would let them double their Lightning funds),
2. in the new channel in Y, the dedicated user's Commitment transaction cannot require the casual user's signature,
3. in the new channel in Y, the dedicated user cannot require any HTLC-payment transactions for successful payments from the casual user (because such HTLC-payment transactions require the casual user's signature, as is shown in Figure 3), and
4. in the new channel in Y, the dedicated user cannot require any HTLC-refund transactions for timed-out payments to the casual user (because such HTLC-refund transactions require the casual user's signature, as is shown in Figure 4).

These problems are solved as follows. As before, let A denote a casual user and let B denote a dedicated user.

Problem 1) is solved by having A and B use the same set of control transactions for their channels in both X and Y, and by adding a new output from In_A that is spent by either L_{XA} (their leaf in timeout-tree X) or L_{YA} (their leaf in timeout-tree Y). Because this new output from In_A can only be spent by one of those leaf transactions, A is prevented from obtaining Lightning funds from both X and Y.

Problem 2) is solved by giving B a special initial Commitment transaction, $Com_{YBi,}$ that is defined by a covenant and that pays the channel's initial balances to A (without any delay) and to B (after a delay of *tsdA* that allows A to claim all of the channel's funds if the channel's initial state has been revoked). Because Com_{YBi} is defined by a covenant, A's signature is not required in order to create the initial channel state in timeout-tree Y.

Figure 7 shows how Problems 1) and 2) are solved in rolling-over funds from an old channel in timeout-tree X to a new channel in timeout-tree Y. In Figure 7, timeout-tree X had an initial state *h* that has been revoked by B's revealing their per-commitment key for state *h* to A. In order to rollover the channel funds from timeout-tree X to timeout-tree Y, just prior to E_X - *tsdA*, user B puts the Funding transaction F_Y on-chain and sends B's partial signatures for L_{YA} and Com_{YAi} to A (along with the transactions in timeout-tree Y that are ancestors of L_{YA}).

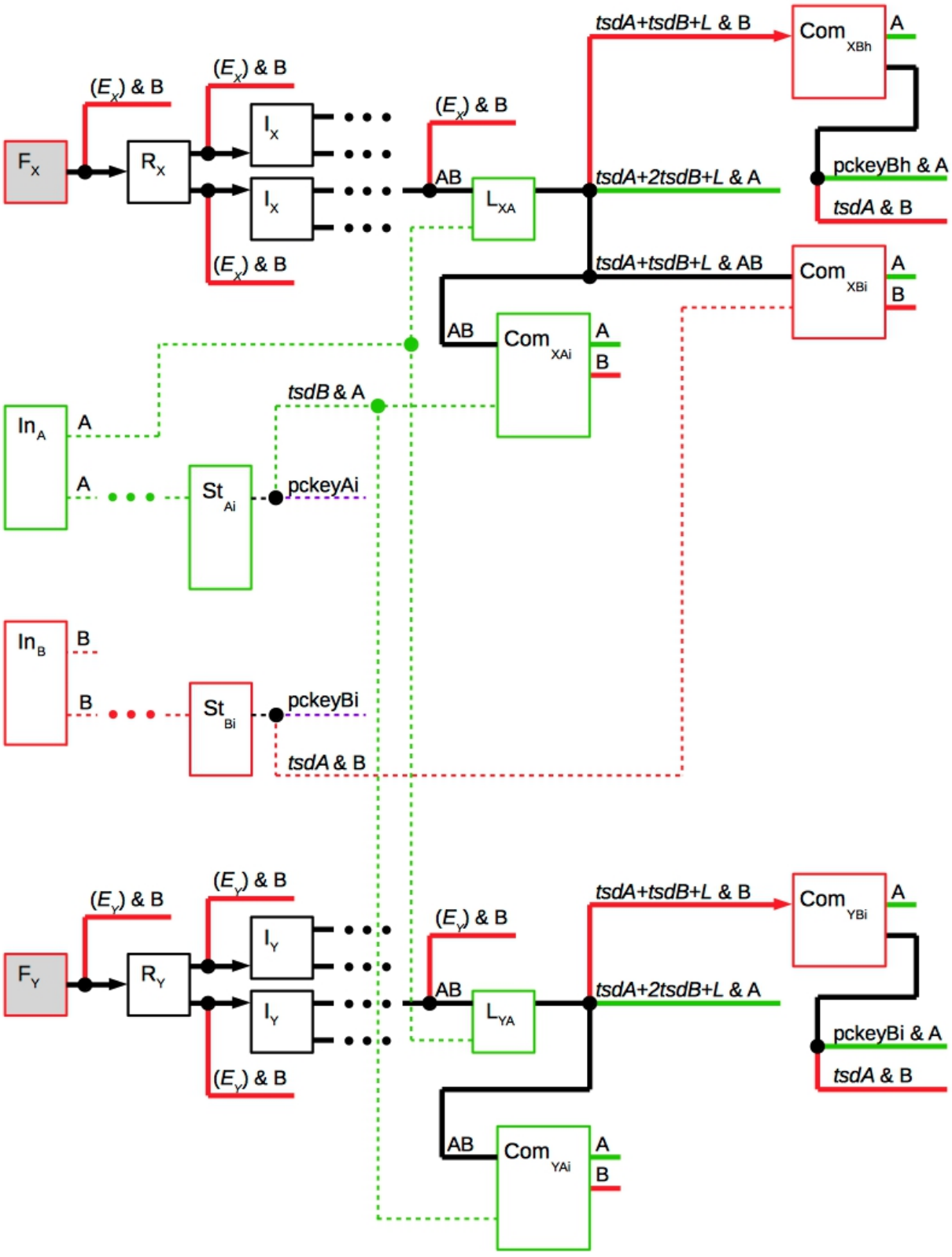

Figure 7. Funds from a channel in timeout-tree X are rolled-over to timeout-tree Y without requiring any signatures from A. L_{XA} and L_{YA} conflict, so A cannot obtain funds from both timeout-trees.

Problem 3) is solved by making the outputs from A's Commitment transaction for HTLCs offered by A pay to an HTLC-refund transaction (that refunds the HTLC to A if A has put a corresponding HTLC-timeout transaction on-chain) or to B after a suitable delay. Of course, in order to rollover funds to timeout-tree Y, B has to send B's partial signatures for HTLC-r_{YAi} to A (along with the other partial signatures and transactions given in the description of Figure 7 above). This protocol for an HTLC offered by A is shown in Figure 8, and an example of rolling-over a channel with an outstanding HTLC offered by A is shown in Figure 9.

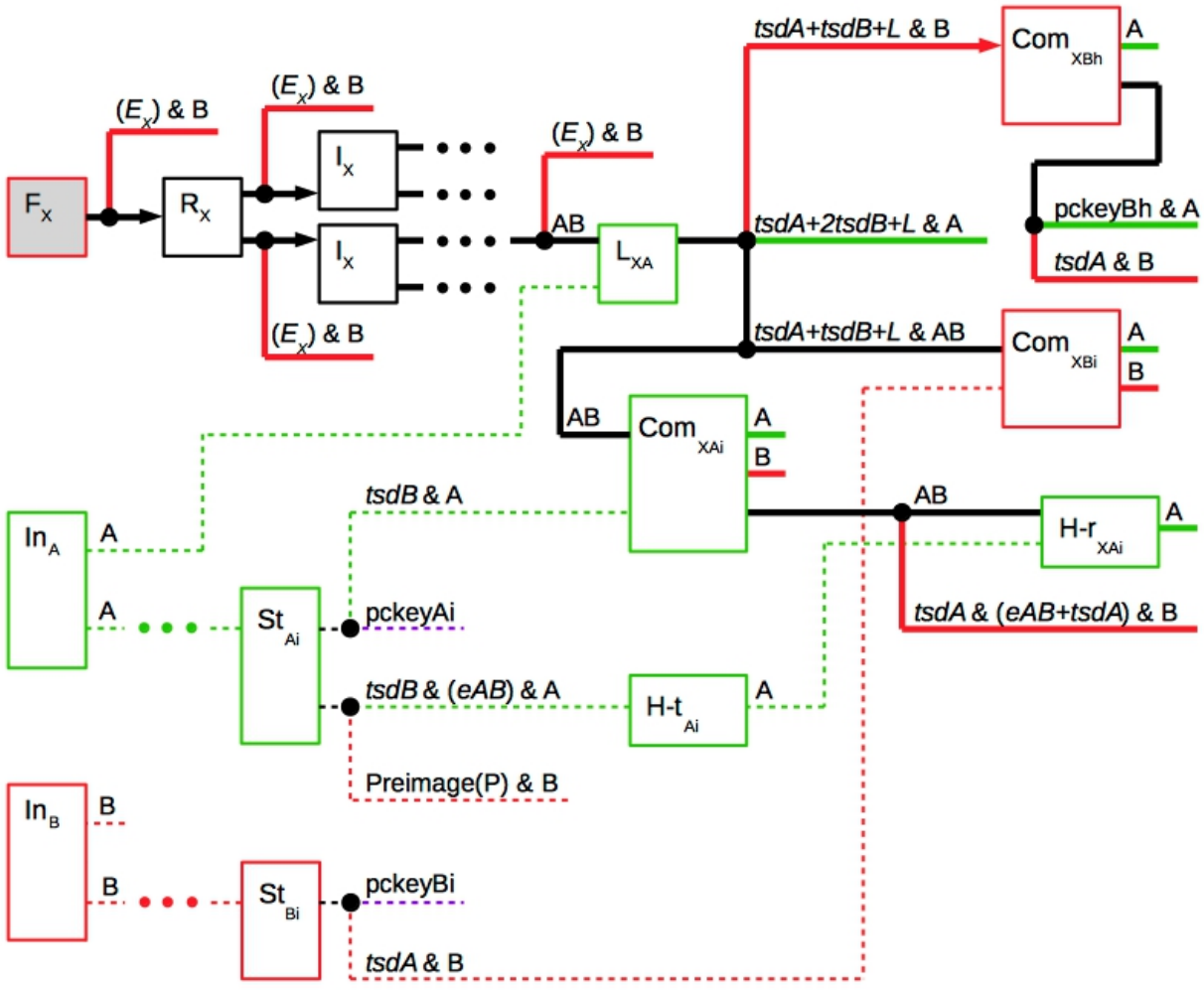

Figure 8. Protocol for an HTLC offered by casual user A to dedicated user B that solves Problem 3) above.

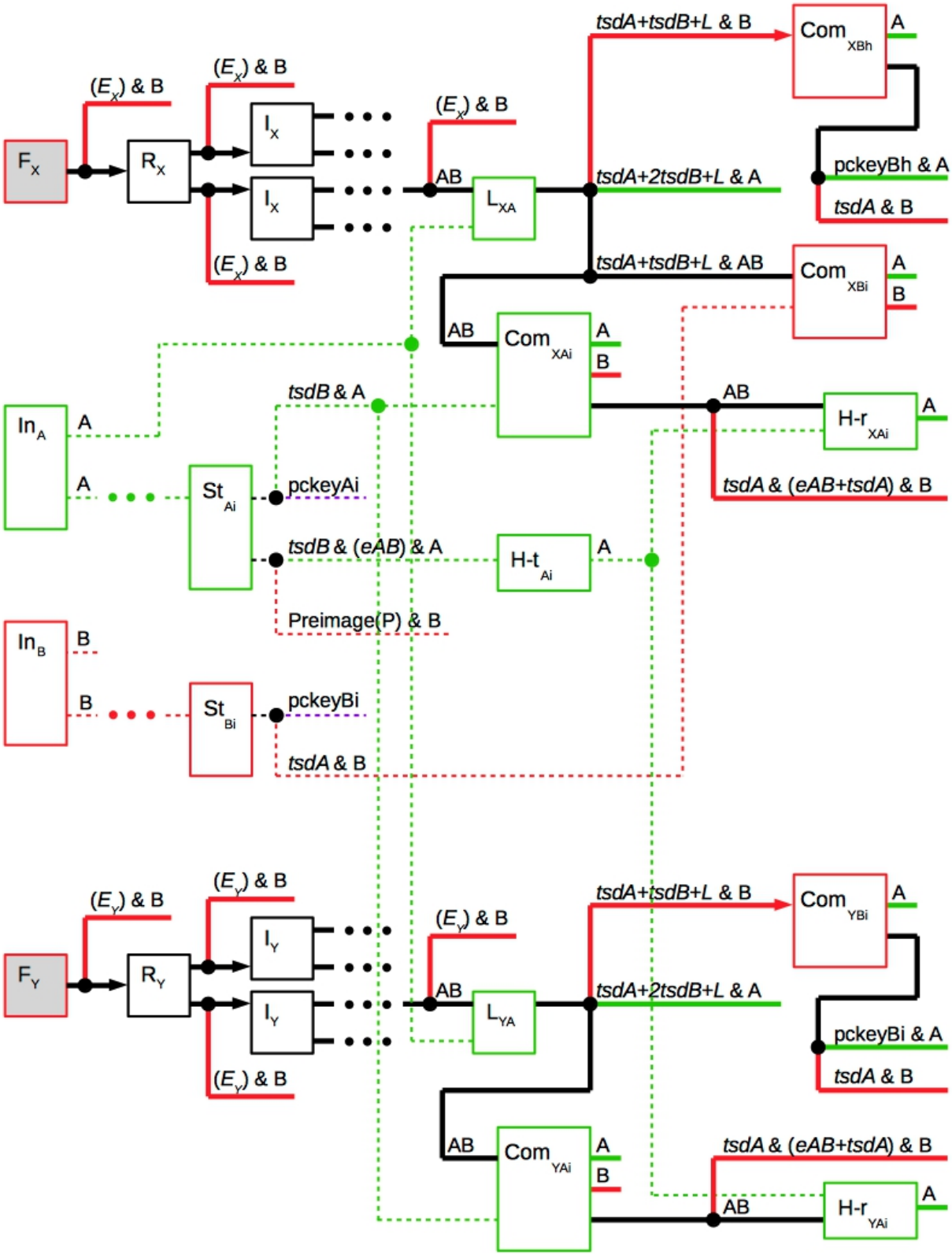

Figure 9. Rollover of funds from timeout-tree X to timeout-tree Y for a channel that has an outstanding HTLC offered by casual user A to dedicated user B. The rollover does not require a signature from A.

Problem 4) is solved by disallowing the rollover of a channel from one timeout-tree to another whenever there is an outstanding HTLC offered by B to A. This restriction is possible because B must be the final destination of the payment, so B must know the secret for the HTLC.

Finally, note that the passive rollover protocol does not use the Lightning network to send funds from the old timeout-tree to a new timeout-tree. Therefore, the passive rollover protocol is immune to HTLC-withholding or channel-jamming attacks on the Lightning network.

4.5 Off-Chain bitcoin

The Lightning Network lets casual users send and receive bitcoin entirely off-chain However, the casual user has to wait (for a period of time specified by their Lightning partner's *to_self_delay* parameter) before they can access their Lightning funds on-chain. This is problematic, as accessing one's Lightning funds on-chain requires paying fees to put transactions on-chain, and those fees cannot be paid using one's Lightning funds (due to the delay mentioned above). Thus, while Lightning can be used for most of a user's funds, the user must also be able to access some bitcoin (enough to pay transaction fees) without any delays.

Fortunately, timeout-trees can be used to provide casual users with immediately-accessible off-chain bitcoin in addition to bitcoin in Lightning channels. Furthermore, it is possible to modify the protocols given in the previous subsection to rollover the casual user's immediately-accessible bitcoin from one timeout-tree to the next along with their Lightning funds. In fact, this rollover can also be done without requiring any actions from the casual user and it can be used to rebalance the fraction of the user's funds that are immediately-accessible versus within Lightning.

The key is to add a second output from the leaf transaction that pays directly to A without any delay. Because the leaf transaction spends a control output from In_A, as required to solve Problem 1) and as shown in Figure 7, A cannot obtain immediately-accessible bitcoin from both the old and new timeout-trees. An example is shown in Figure 10.

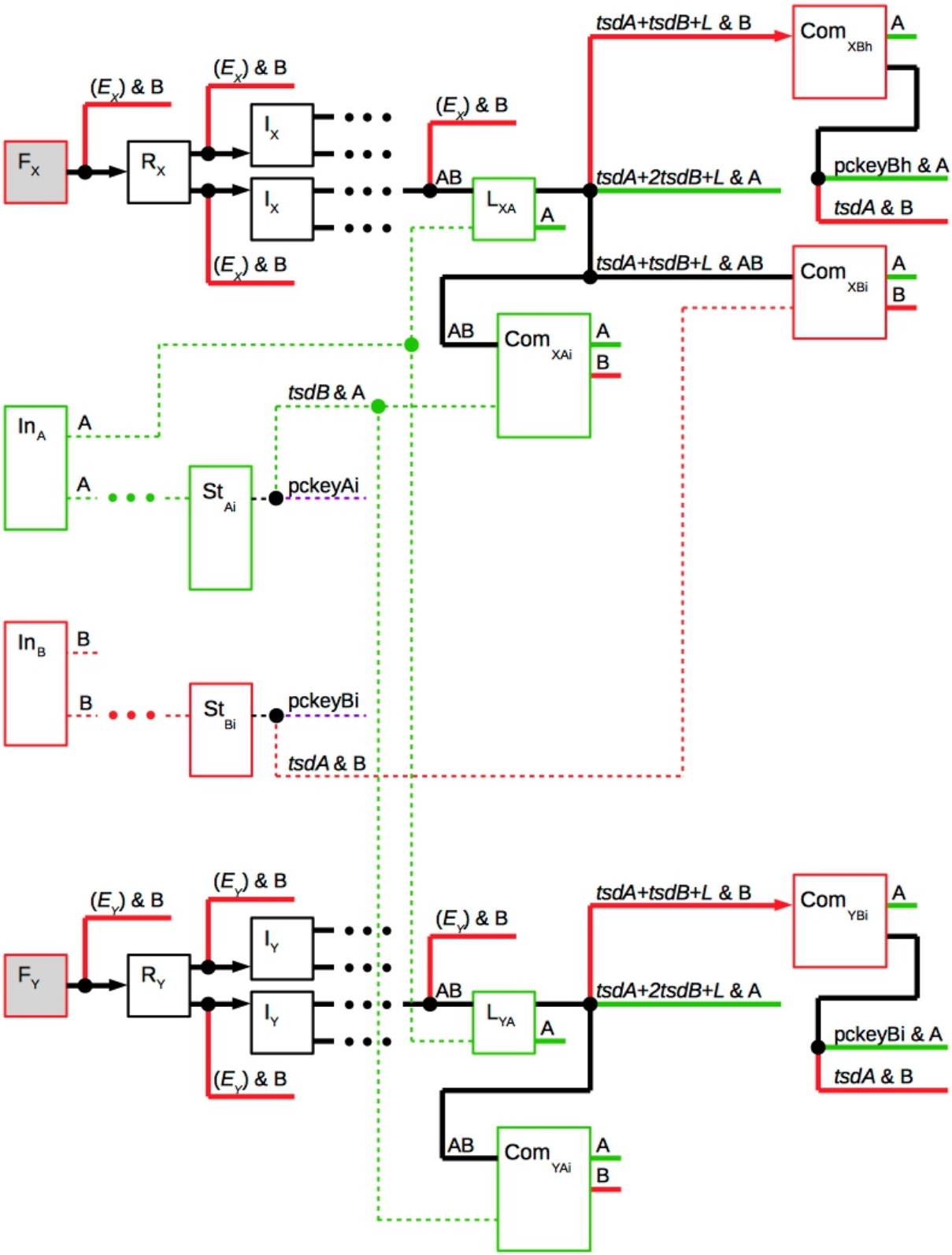

Figure 10. Rollover of immediately-accessible bitcoin from timeout-tree X to timeout-tree Y.

4.6 Control UTXOs

The channel protocols presented above require that each user own a UTXO that is spent by that user's Individual control transaction. Creating an on-chain UTXO for every user could require a significant on-chain footprint, thus limiting scalability. Instead, each user can be given an off-chain UTXO that is created by a leaf of a tree of off-chain transactions defined by covenants. For example, the Funding transaction that creates a timeout-tree could be given a second output that creates a tree with leaves that are Individual control transactions. An example is shown in Figure 11.

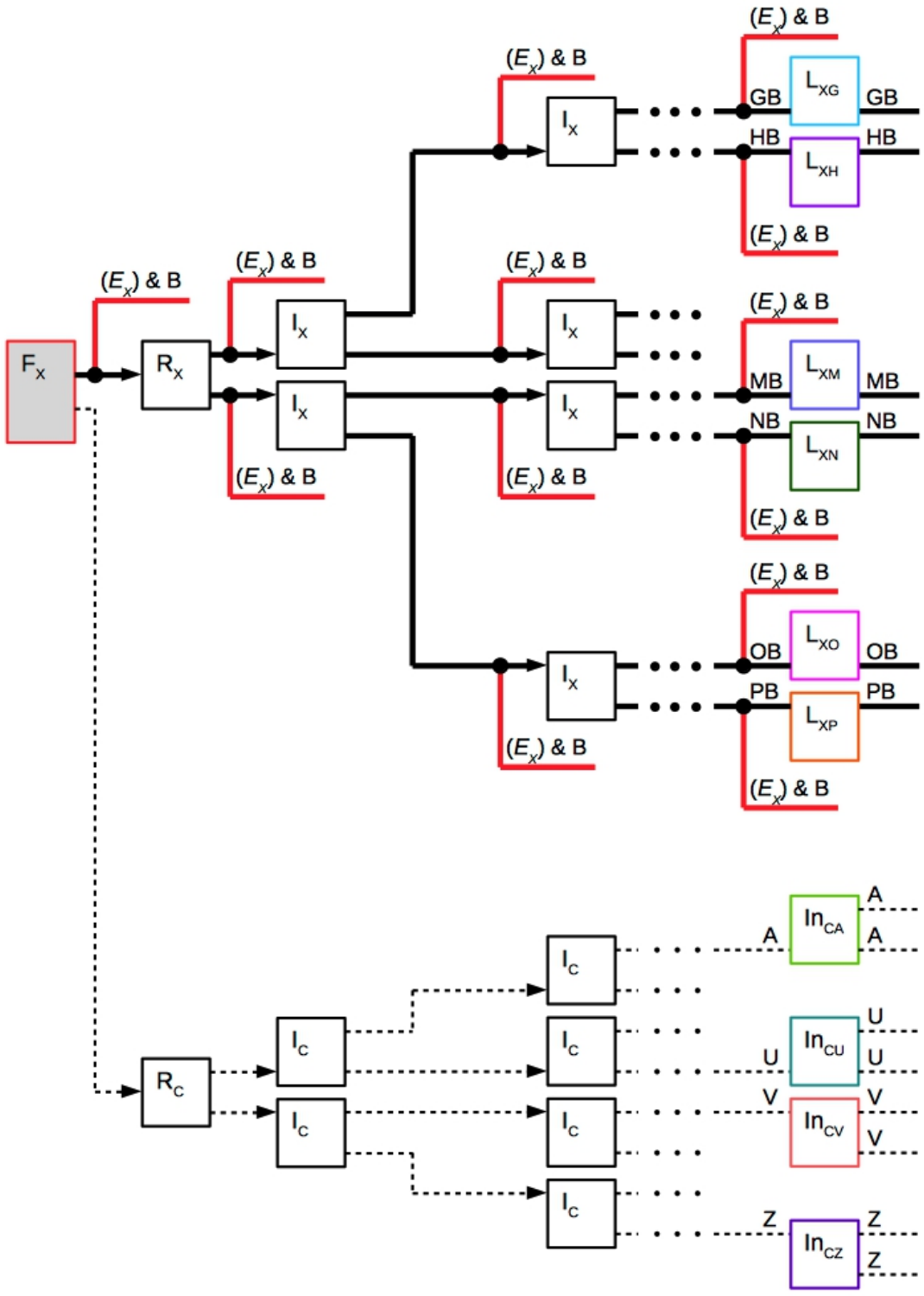

Figure 11. Covenant tree creating Individual control transactions that is funded by the second output of a timeout-tree's Funding transaction.

4.7 The Dust Limit

The control transactions used above do not provide any value themselves; instead, they control the ability to use other (valuable) transactions. Therefore, it would be ideal if the amount of bitcoin associated with each output of a control transaction were exactly zero. However, the Bitcoin protocol currently has a small nonzero limit, called the *dust limit*, on the value of a transaction output.

The concept of a dust limit makes sense when the only reason to have an output is to be able to spend the value associated with that output. However, if protocols that use control transactions become important in practice, it may make sense to revisit the concept of a dust limit.

If zero-valued outputs are allowed, then control transactions (such as State, HTLC-success and HTLC-timeout transactions) could be given an extra zero-valued output that could be used to as the parent of another Individual transaction. In this manner, once a user has put an Individual transaction on-chain, they would be able to put future Individual transactions on-chain without using a new covenant tree to create them. Also, rather than having a covenant tree create Individual transactions directly (as shown in Figure 11), the first output that requires a user's signature could fund a subtree with multiple Individual transactions for that user. As a result, a user could fund the Individual transactions for their Lightning channels in all of their timeout-trees from a single on-chain UTXO.

4.8 Staggered Timeout-Trees

In order to (actively or passively) rollover funds from one timeout-tree to another, both the old and new timeout-trees must be on-chain simultaneously. However, the total value of the funds being rolled over cannot exceed the value of either the old or the new timeout-tree.

For example, consider the case where all rollovers are passive and leave the capacity of the rolled-over channel unchanged. Let V denote the total value of the channels in each timeout-tree. Assume dedicated user B creates a single timeout-tree X, waits until X is near its expiry, and then creates a new timeout-tree Y to which X's channels are rolled over. In this case, B must have $2V$ capital in order to fund channels with a total value of V, thus resulting in low (50%) utilization of B's capital.

However, a better utilization of B's capital can be achieved if B funds a large number of timeout-trees which expire, and thus rollover, at different times. For example, if B funds t different timeout-trees, at most one of which is being rolled over at any given time, B must have $(t+1)*V$ capital in order to fund channels with a total value of $t*V$, thus resulting in a higher (t/(t+1)) utilization of B's capital.

Similarly, it would be possible for each casual user to only have a single Lightning channel with a single dedicated user, where the dedicated user repeatedly rolls over the funds in that channel to a new timeout-tree prior the expiration of the current timeout-tree. However, most casual users would want to divide their funds between multiple Lightning channels, each of which is owned by a different dedicated user, thus increasing the likelihood that at least one of the casual user's partners is available to help send or receive a Lightning payment at any given time.

Furthermore, the timeout-trees that create the multiple Lightning channels for a given casual user could be staggered so that they expire at (roughly) even time intervals. As a result, the casual user will never have to wait too long in order to have a rollover of an about-to-expire timeout-tree, with each rollover giving the user a chance to rebalance their immediately-available bitcoin and their Lightning funds, and to pair with a new dedicated user (if desired).

4.9 Timeout-Trees With Hierarchical Channels

The above protocols require that dedicated users assign their capital to fund channels with casual users. Because casual users may send and receive payments infrequently, this capital may generate few routing fees for the dedicated users. As a result, casual users may have to pay significant fees for the creation of their Lightning channels (and/or for payments to or from those channels).

However, the fees that casual users have to pay could be reduced if the capital in their channels could also be used for routing payments between other users. This can be accomplished by having the timeout-trees create hierarchical channels, each of which is owned by a single casual user and a pair of dedicated users **[Law23a]**. When the casual user is not sending or receiving a payment, the pair of dedicated users can use all of their funds in the hierarchical channel to route payments for other users. As a result, the funds owned by the dedicated users can be used either for routing payments for the hierarchical channel's casual user, or for unrelated users (but not both simultaneously).

In fact, by using an idea created by Towns **[Tow23][Law23b]**, it is possible to use the same funds to route a payment for the hierarchical channel's casual user, and for unrelated users, simultaneously. The key observation is that the payment for the hierarchical channel's casual user may shift the source of the dedicated users' funds, but it does not decrease those funds. As a result, the dedicated users can route unrelated payments by allowing for two possible sources of their funds.

Assume casual user A shares a hierarchical channel with dedicated users B and C, that this hierarchical channel is created by a timeout-tree X funded by C, and that *tsdA* and *tsdB* denote A's and B's *to_self_delay* parameters[4]. In this case, C should create a new timeout-tree with a new hierarchical channel just before E_X - *tsdA* - *tsdB*, thus allowing A to verify the rollover to the new timeout-tree anytime between E_X - *tsdA* - *tsdB* and E_X - *tsdB*. Then, between E_X - *tsdB* and E_X, B and C can use the Lightning network to actively rollover B's funds to the new timeout-tree.

If the rollover of A's funds is done passively, then between E_X - *tsdA* - *tsdB* and E_X - *tsdB*, A could put their funds on-chain from either the old or the new timeout-tree. The fact that A's funds could be in either of two places (the old or the new timeout-tree) complicates the use of the remaining funds owned by B and C for routing unrelated payments. However, by once again using the idea created by Towns **[Tow23][Law23b]** to fund an HTLC from one of two sources (in this case, either the old or the new timeout-tree), B and C can continue to use their funds for routing unrelated payments during the period E_X - *tsdA* - *tsdB* through E_X - *tsdB*.

4 A's leaf transaction conflicts with transactions signed by B and C after E_X - *tsdB* and transactions signed by C after E_X.

4.10 Improving Scalability And Reducing Costs With Fee Penalties

While it is essential that a casual user be able to put a timeout-tree leaf on-chain, actually choosing to do so hurts scalability. The rate at which casual users put timeout-tree leaves on-chain can be minimized by:

1. reducing the rate at which unintentional dedicated user unavailability forces casual users to put timeout-tree leaves on-chain,
2. allowing casual users to accomplish what they want to accomplish (e.g., making and receiving Lightning payments, resizing Lightning channels, pairing with new Lightning partners, and rebalancing between immediately-available bitcoin and Lightning) without putting a timeout-tree leaf on-chain, and
3. making casual users pay a penalty for putting a timeout-tree leaf on-chain.

Item 1) is helped by performing passive rollovers of timeout-trees, thus preventing the casual user from having to go on-chain when their dedicated user channel partners are unavailable for using Lightning to drain the casual user's funds from an about-to-expire timeout-tree.

Item 2) is largely addressed by the protocols presented above, as they allow all of the actions listed to be performed without putting a timeout-tree leaf on-chain. However, not all types of channel resizing can be performed at any given time, and pairing with a new Lightning partner requires waiting for that partner to create a new timeout-tree. In addition, obtaining immediately-available bitcoin in order to pay fees (e.g., to resolve an HTLC at a payment's destination before it times out) using the above protocols requires putting a timeout-tree leaf on-chain. As a result, it may be beneficial to include some immediately-available bitcoin within the covenant tree presented in Figure 11. While putting leaves of a covenant tree on-chain obviously impacts scalability, doing so helps the other users of the covenant tree by making it cheaper for them to eventually put their leaves on-chain. This is in contrast to putting a leaf of a timeout-tree on-chain, which requires the funder of the timeout-tree to pay additional fees to spend the outputs of the leaf's ancestors when the timeout-tree reaches its expiry.

Item 3) is partially addressed just by the fact that putting a timeout-tree leaf on-chain requires paying fees for putting transactions on-chain. However, putting a timeout-tree leaf on-chain imposes additional costs on the funder of the timeout-tree when the funder spends the previously-unspent outputs of the leaf's ancestors. If the casual user were made to pay a penalty equal to the costs they imposed on the funder, the casual user would be less inclined to put the leaf on-chain (thus improving scalability) and the funder would be able to charge the other casual users lower fees (thus reducing their costs).

The penalty imposed should be the product of the number of virtual bytes (vbytes) the funder is forced to put on-chain and the estimated per-vbyte feerate the funder will have to pay. The number of vbytes that must be put on-chain by the funder depends on how many other casual users (and which ones) also put their leaves on-chain. For example, if only one casual user puts their leaf on-chain, that leaf forces the funder to spend outputs from a number of non-leaf timeout-tree transactions that is logarithmic in

the number of leaves. On the other hand, if casual users put all of the leaves on-chain, the funder will not have to spend any outputs from non-leaf timeout-tree transactions.

A penalty can be implemented by adding a third output to each timeout-tree leaf, where the value of this third output is at least the maximum cost imposed by the casual user on the funder. The value of this third output, called the *fee-penalty output*, is then split between the casual user and the funder according to the estimated costs that the casual user imposed on the funder. These costs are determined by making the casual user reveal a secret whenever they put a timeout-tree Root, Internal or Leaf transaction on-chain. Then, the funder must provide one of the casual user's secrets when using a covenant-defined *fee-splitting* transaction to spend the fee-penalty output. While the casual use may have revealed multiple secrets, the first such secret represents the largest number of vbytes that the funder is forced to put on-chain, so the funder will naturally use that secret in order to obtain the largest payout from the fee-splitting transaction.

This protocol forces a casual user who puts a timeout-tree leaf on-chain to reveal secrets that pay for the estimated value of the fees imposed by that casual user on the timeout-tree's funder. However, a casual user would still be able to maliciously put some timeout-tree transactions on-chain without putting their leaf transaction on-chain, thus resulting in additional fees imposed on the funder that would not be reimbursed by the casual user. Such a possibility is eliminated (assuming honest miners) by forcing the Root and Internal transactions to pay 0 fees, and to add an additional (anchor) output to each Leaf transaction for paying fees.

The use of casual users' secrets to determine the correct fee-splitting transaction is shown in Figure 12 below. In Figure 12:

- S_{lu} denotes casual user *u*'s level-*l* secret (hash preimage) and
- | denotes that only one of the listed secrets must be provided.

For example, $S_{nD}|...|S_{nW}$ indicates that a level-*n* secret known by any of the casual users D..W (that is, any of the casual users in the given timeout-tree) must be provided, while S_{0W} indicates that casual user W's level-*0* secret must be provided.

If L_{XW} is the only leaf transaction put on-chain, W must provide secrets S_{nW} through S_{0W}. In this case, dedicated user B will choose to put Fee-Splitting transaction F-s_{XBn} on-chain, as it provides B with the largest fee penalty from W. On the other hand, if casual user V put L_{XV} on-chain before casual user W put L_{XW} on-chain, W only needs to reveal secret S_{0W}, and B will only be able to obtain the smaller fee penalty provided by F-s_{XB0}. Finally, if B fails to put any Fee-Splitting transaction on-chain, W is able to spend all of the fee-penalty output.

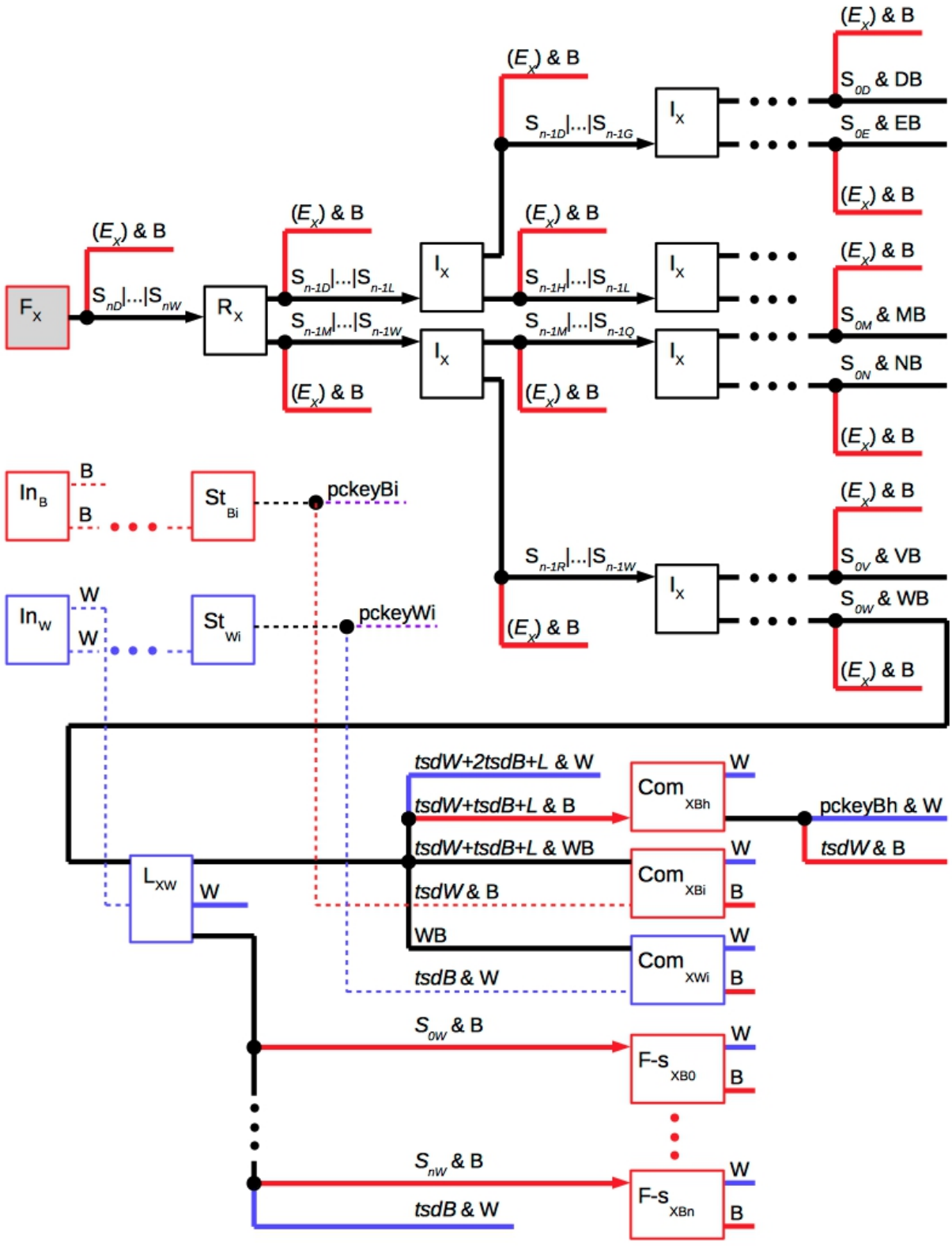

Figure 12. Timeout-tree X showing only casual user W's leaf. Root (R), Internal (I) and Leaf (L) transactions require a secret (hash preimage) of a casual user within the transaction's subtree and unique to that transaction. B uses these revealed secrets to select a Fee-Splitting (F-s) transaction.

Note that if a casual user's funds are rolled-over passively from timeout-tree X to timeout-tree Y, the casual user can use the same secrets for both X and Y (as the casual user will only need to put a leaf from at most one of those trees on-chain).

Also, note that if all timeout-tree outputs use Taproot, the witness for a level-l secret will contain a control block with $1+32(l+1)$ bytes **[WNT20]**. Therefore, if a single leaf in a timeout-tree with 2^n leaves is put on-chain, that single leaf and its ancestors will have control blocks with

$$\sum_{i=0}^{n} 1+32(i+1) = n+1+32\sum_{i=1}^{n+1} i = n+1+32(n+1)(n+2)/2 = 16n^2+49n+33 = O(n^2)$$

vbytes. However, if all 2^n leaves are put on-chain, the timeout-tree control blocks will have a total of

$$\sum_{i=0}^{n} 2^{n-i}(1+32(i+1)) < \sum_{i=0}^{\infty} 2^{n-i}(1+32(i+1)) = 2^{n+1}+32\sum_{i=0}^{\infty} 2^{n-i}(i+1) = 2^{n+1}+2^{n+5}\sum_{i=0}^{\infty}(i+1)(1/2)^i$$

vbytes. Because (from **[GKP94]**, p. 335) for any z, $-1 < z < 1$:

$$\sum_{i=0}^{\infty}(i+1)z^i = 1/(1-z)^2$$

it follows that

$$\sum_{i=0}^{n} 2^{n-i}(1+32(i+1)) < 2^{n+1}+2^{n+5}(1/(1/2)^2) = 2^{n+1}+2^{n+7} = 2^n * 130$$

which means that each of the 2^n leaves contributes an average of fewer than 130 control block vbytes.

The goal so far has been to make casual users pay the fees that their on-chain transactions impose on the funder. However, if all or nearly all of the casual users put their leaves on-chain, that almost certainly indicates a failure by the funder. As a result, in such a case it would be preferable if the funder's fees were not reimbursed. One way to achieve this goal is to eliminate the secrets and Fee-Splitting transactions corresponding to levels 0 and 1 of the timeout-tree. In the normal case, it should be extremely rare for two casual users within a single 4-leaf subtree to both put their leaves on-chain, so this change will have little or no impact on the funder. On the other hand, if all of the casual users put their leaves on-chain, three-quarters of the leaves will impose fees upon the funder without the funder being reimbursed, thus penalizing the funder.

Finally, a reduction in the number of control block vbytes put on-chain can be achieved by eliminating the need for secrets at certain levels of the timeout tree (in addition to eliminating secrets for levels 0 and 1). For example, only requiring secrets at every fourth level would approximately quarter the control block vbytes required while still providing a reasonable estimate of the fees imposed on the funder.

5 Analysis Of Covenant-Based Protocols

5.1 Scalability

Timeout-trees with hierarchical channels can perform the following actions completely off-chain:

- Lightning sends and receives, and
- resizing of Lightning channels.

Assuming:

- 1 million hierarchical Lightning channels per timeout-tree,
- a 1,000-block (about a week) *to_self_delay* parameter for dedicated users,
- a 10,000-block (about 69 days) *to_self_delay* parameter for casual users, and
- 121,000 blocks (about 2.3 years) from the creation of each timeout-tree to its expiry,

a single 1-input/2-output transaction (such as the Funding transaction shown in Figure 11) per block provides:

- 11 Lightning channels per casual user to each of 10 billion casual users.

The hierarchical Lightning channels owned by each casual user could be staggered so that their expiries are 10,000 blocks apart. As a result, given the above assumptions, a single 1-input/2-output transaction per block allows each casual user to:

- close an existing Lightning channel,
- open a new Lightning channel with a new partner, and
- rebalance funds between Lightning and immediately-accessible off-chain bitcoin

once every 10,000 blocks (about 69 days).

Of course, the above calculations do not mean that 10 billion casual Lightning users would create only 1 on-chain transaction per block. In reality, their on-chain footprint would be dominated by users who do not follow the protocol due to errors, unavailability, or malicious intent. The rate of such protocol violations is hard to predict, but it is likely that casual users' unavailability would be the most significant problem.

5.2 Capital Efficiency

Given the assumptions in Section 5.1, each dedicated user could create a new timeout-tree every 11,000 blocks, with every tenth timeout-tree created by a given dedicated user being used by the same set of casual users as is shown in Figure 13.

	00	10	20	30	40	50	60	70	80	90	100	110	120	130
0	C_0	M_0	L_1	K_2	J_3	I_4	H_5	G_6	F_7	E_8	D_9	C_{10}	M_{10}	L_{11}
1	D_0	C_1	M_1	L_2	K_3	J_4	I_5	H_6	G_7	F_8	E_9	D_{10}	C_{11}	M_{11}
2	E_0	D_1	C_2	M_2	L_3	K_4	J_5	I_6	H_7	G_8	F_9	E_{10}	D_{11}	C_{12}
3	F_0	E_1	D_2	C_3	M_3	L_4	K_5	J_6	I_7	H_8	G_9	F_{10}	E_{11}	D_{12}
4	G_0	F_1	E_2	D_3	C_4	M_4	L_5	K_6	J_7	I_8	H_9	G_{10}	F_{11}	E_{12}
5	H_0	G_1	F_2	E_3	D_4	C_5	M_5	L_6	K_7	J_8	I_9	H_{10}	G_{11}	F_{12}
6	I_0	H_1	G_2	F_3	E_4	D_5	C_6	M_6	L_7	K_8	J_9	I_{10}	H_{11}	G_{12}
7	J_0	I_1	H_2	G_3	F_4	E_5	D_6	C_7	M_7	L_8	K_9	J_{10}	I_{11}	H_{12}
8	K_0	J_1	I_2	H_3	G_4	F_5	E_6	D_7	C_8	M_8	L_9	K_{10}	J_{11}	I_{12}
9	L_0	K_1	J_2	I_3	H_4	G_5	F_6	E_7	D_8	C_9	M_9	L_{10}	K_{11}	J_{12}

Figure 13. Staggered funding of timeout-trees. Each square represents 1,000 blocks in the Bitcoin blockchain, with time running top-to-bottom and then left-to-right. Each square is labeled with the dedicated user (C, D, E ... M) that creates a timeout-tree at the first block represented by the square. Subscripts number the timeout-trees created by a given dedicated user. Each row represents a group of casual users, each of whom receives a Lightning channel in each timeout-tree that is created in that user's row. Shaded boxes show the lifetime of timeout-tree C_0 which is created at the start of square 000 (block 0) and expires at the start of square 121 (block 121,000).

As can be seen from Figure 13, each casual user that obtains a channel in timeout-tree C_0 verifies that their channel is rolled-over to timeout-tree C_{10} between blocks 110,000 and 120,000. Then, between blocks 120,000 and 121,000, dedicated user C and C's dedicated user partner actively rollover their funds from their hierarchical channel in C_0 to their hierarchical channel in C_{10}. As a result, C has to fund 11 timeout-trees in order to create 10 timeout-trees' worth of hierarchical channels that are co-owned by casual users, yielding over 90% utilization of C's capital by casual users[5].

5 For simplicity, the few-block latency required to create a timeout-tree is not included in this analysis.

5.3 Usability

The above protocols have the following properties for casual users:

- watchtower-freedom (that is, they accommodate months-long unavailability without requiring a watchtower service to secure the user's funds) (**[Law22a]** Section 3.1),
- one-shot receives (that is, receiving a payment does not require performing actions at multiple blockheights) (**[Law22a]** Section 3.4),
- asynchronous receives (that is, it is possible to receive a payment when the sender is offline) (**[Law22a]** Section 3.6), and
- tunable penalties for attempting to put an old state on-chain (**[Law22b]**).

5.4 Limitations

The above protocols provide a great deal of functionality and scalability while meeting the requirements of casual users. However, there are some limits to their functionality. For example, the hierarchical channel protocols allow Lightning channels to be resized off-chain, but the resizing they enable is not completely arbitrary **[Law23a]**. On the other hand, staggered rollovers of timeout-trees provide many additional opportunities to resize hierarchical channels without increasing the on-chain footprint.

In addition, the above results depend on the following assumptions:

1. the cost of resolving an HTLC on-chain is less than the value of the HTLC,
2. transaction packages are relayed reliably from users to miners, and
3. there is a known upper bound on the delay from when a package is submitted to when it is included in the blockchain.

If zero-valued outputs are allowed, one way to address the first limitation is to add *short-cut transactions*. A short-cut transaction is a control transaction that a user **may** (but may not) be able to put on-chain more cheaply than their standard control transaction. For example, consider user Z's Individual transaction In_{CZ} shown in Figure 11. If the covenant tree is a complete binary tree with 2^{20} Individual transactions, putting In_{CZ} on-chain requires putting its 20 currently-off-chain ancestors on-chain. However, it would be possible to modify the covenant tree so that it contains one or several short-cut (SC) transactions that are closer to the root, do not have any covenants placed on them, and do not require a signature. As a result, any user can put any SC transaction on-chain at any time, and that SC transaction can have any form. An example of short-cut transactions is shown in Figure 14.

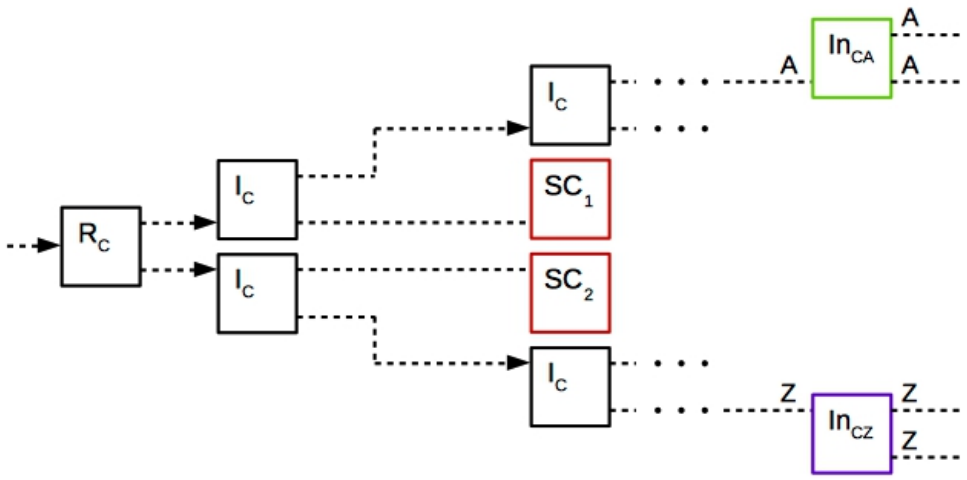

Figure 14. A covenant tree with short-cut transactions SC_1 and SC_2.

In Figure 14, each short-cut transaction is shown as a single box with a single label and no outputs, but in reality the form of each short-cut transaction is arbitrary. Thus, a short-cut transaction could have any number of outputs, each output could have any locking script, and the short-cut transaction could be put on-chain by any user.

Given the covenant tree in Figure 14, user Z could maintain two versions of their channel state, one of which uses SC_2 as their Independent transaction, and the other of which uses In_{CZ} as their Independent transaction. If no other user has put a conflicting transaction on-chain, Z can put their desired form of SC_2 on-chain and use it as their Independent transaction, thus greatly reducing the on-chain footprint for resolving their HTLC.

The use of a short-cut transaction has some downsides, including:

- doubling the number of HTLC transactions that the user must manage and obtain signatures for,
- increasing the overall depth of the covenant tree (and thus the cost of putting the leaf transaction In_{CZ} on-chain) and
- complicating the casual user's ability to resolve an HTLC in a single-shot manner (as attempting to use a short-cut transaction could fail if it conflicts with another user's version of the same short-cut transaction).

However, just having the possibility of using a short-cut transaction may be sufficient to allow a casual user to implement HTLCs for small payments, as the casual user's channel partner would know that the HTLC may be resolved cheaply on-chain. In fact, game-theoretic considerations may lead a casual user to resolve an HTLC on-chain, even in the case where the cost of doing so exceeds the value of the HTLC.

6 Related Work

A number of researchers have attempted to improve the scalability of Lightning by allowing a single UTXO to create a large number of Lightning channels.

Burchert, Decker and Wattenhofer introduced the concept of a Lightning channel factory and created protocols for creating and modifying channels using factories **[BDW18]**. Law presented alternative, more efficient, protocols for channel factories **[Law22d]** using Bitcoin's current consensus rules. Decker, Russell and Osuntokun showed how adding APO to Bitcoin's consensus rules would allow the creation of a particularly clean and simple factory protocol (without using APO to create covenants) **[DRO18]**. ZmnSCPxj proposed changes to Bitcoin's consensus rules **[Zmn22]** that would improve the ability of users in a factory to update the factory's state without obtaining signatures from all users in the factory. All of these factory protocols could improve Lightning's scalability, but they are constrained by their requirement to obtain signatures from all users in the creation of the factory. As a result, their scalability is very limited compared the results presented here.

Law gave protocols for covenant-based factories that rely on changes to Bitcoin's consensus rules to support Inherited IDs (IIDs) (**[Law21]** Sections 6 and 7). These protocols are highly scalable, but getting agreement for supporting IIDs in Bitcoin is likely more difficult than getting agreement for supporting simple covenants enabled by CTV or APO..

Rubin proposed using covenants to create trees, the leaves of which are payment channels **[BIP119]**. This idea is similar to the idea of a timeout-tree, but it does not have an expiry after which the funding user can spend any non-leaf output. As a result, such covenant trees could delay, but not eliminate, the placement of the tree's transactions on-chain, and thus do not provide a long-term scaling improvement.

Finally, Law presented timeout-trees for scaling Lightning using simple covenants (**[Law21]** Section 5.3). The results presented here build on that work by creating variants of the FFO-WF protocol **[Law22c]** for timeout-trees, improving capital efficiency by staggering the funding of timeout-trees and using hierarchical channels, supporting passive rollovers for casual users, giving users immediately-available bitcoin, and allowing users to rebalance their off-chain bitcoin and Lightning.

7 Conclusions

With the current Bitcoin consensus rules, there are reasons to believe that the scalability of Lightning is inherently limited. However, simple covenants and timeout-trees can overcome these scalability limitations. In particular, CheckTemplateVerify (CTV) and/or AnyPrevOut (APO) could be used to dramatically increase the number of casual users who send and receive bitcoin in a trust-free manner.

As a result, it is hoped that CTV, APO or a similar mechanism that enables simple covenants will be added to Bitcoin's consensus rules in order to allow Lightning to become a widely-used means of payment.

8 Acknowledgments

Thanks to ZmnSCPxj for his corrections to the initial text definition of timeout-trees in version 1.0 of this paper. Also, thanks to Antoine Riard for his highlighting the risk of HTLC-withholding attacks and channel jamming attacks when performing active rollovers of timeout-trees.

References

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

BIP118 BIP 118: SIGHASH_ANYPREVOUT. See https://anyprevout.xyz/ and https://github.com/bitcoin/bips/pull/943.

BIP119 BIP 119: CHECKTEMPLATEVERIFY. See https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki.

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.

DRO18 Christian Decker, Rusty Russell and Olaoluwa Osuntokun. eltoo: A Simple Layer2 Protocol for Bitcoin. 2018. See https://blockstream.com/eltoo.pdf.

GKP94 Ronald Graham, Donald Knuth and Oren Patashnik. Concrete Mathematics, Second Edition. 1994. Addison-Wesley: Upper Saddle River, NJ.

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

Law22a John Law. Watchtower-Free Lightning Channels For Casual Users. See https://github.com/JohnLaw2/ln-watchtower-free.

Law22b John Law. Lightning Channels With Tunable Penalties. See https://github.com/JohnLaw2/ln-tunable-penalties.

Law22c John Law. Factory-Optimized Channel Protocols For Lightning. See https://github.com/JohnLaw2/ln-factory-optimized.

Law22d John Law. Efficient Factories For Lightning Channels. See https://github.com/JohnLaw2/ln-efficient-factories.

Law23a John Law. Resizing Lightning Channels Off-Chain With Hierarchical Channels. See https://github.com/JohnLaw2/ln-hierarchical-channels.

Law23b John Law. Re: Resizing Lightning Channels Off-Chain With Hierarchical Channels. See https://lists.linuxfoundation.org/pipermail/lightning-dev/2023-April/003917.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.

Tow23 Anthony Towns. Re: Resizing Lightning Channels Off-Chain With Hierarchical Channels. See https://lists.linuxfoundation.org/pipermail/lightning-dev/2023-April/003913.html.

WNT20 Peter Wuille, Jonas Nick and Anthony Towns. BIP 341: Taproot: SegWit Version 1 Spending Rules. See https://en.bitcoin.it/wiki/BIP_0341.

Zmn22 ZmnSCPxj. Channel Eviction From Channel Factories By New Covenant Operations. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-February/003479.html.