Scaling Bitcoin with Inherited IDs

John Law

September 23, 2021

Version 1.4

Abstract

Bitcoin has limited scalability, as all nodes must verify all on-chain transactions. In order to improve scalability, several Layer 2 protocols have been proposed where a fixed set of parties create transactions, most of which are able to remain off-chain in the normal case. One Layer 2 protocol, the Lightning Network, requires no changes to Bitcoin and is currently deployed. A proposed upgrade, eltoo, simplifies and improves upon Lightning, but requires changes to Bitcoin. While these Layer 2 protocols dramatically improve scalability, they do not solve the scalability problem as they require on-chain transactions whenever the set of parties involved changes. In this paper we address the Bitcoin scalability problem by creating new protocols that use a single on-chain transaction in place of thousands, or even millions, of Lightning or eltoo on-chain transactions. These new protocols support arbitrarily large sets of parties, enable dynamic changes to the parties involved, provide significant usability and security improvements, and are completely trust-free. Like eltoo, these new protocols require a change to Bitcoin, but the required change is simultaneously more restrictive and more powerful than that proposed for eltoo.

1 Introduction

Bitcoin is the first and most valuable cryptocurrency, but it has limited scalability **[Nak09] [BDW18] [DRO18]**. Bitcoin operates as a peer-to-peer network in which nodes maintain a globally-shared state consisting of *unspent transaction outputs* called the *UTXO set*. Updates to the shared state are made by *transactions*, each of which spends a set of previously-unspent outputs and redistributes the value from those outputs to a new set of outputs. Updates are kept consistent by miners who put new transactions in *blocks* and order those blocks in a globally-shared *blockchain*. Miners use a proof-of-work consensus protocol to guarantee that all nodes eventually agree on the contents and order of blocks in the blockchain. Because blocks are bounded in size, and because the long-term rate at which blocks are added to the blockchain is fixed, there is an inherent limit to the rate at which transactions can be added to the blockchain.

The most straightforward way to scale Bitcoin would be to increase the size of blocks and/or the rate at which blocks are added to the blockchain. However, scaling in this way increases the memory, computation and bandwidth costs of running a node, thus leading to fewer nodes and a more centralized (and less secure) network.

Another obvious scaling mechanism is to use centralized exchanges that own bitcoin on behalf of users. A centralized exchange can logically transfer bitcoin from one user account to another without updating the blockchain because the exchange (rather than the users) actually owns the bitcoin credited to user accounts. In addition, exchanges can utilize a single Bitcoin transaction to batch transfers from the users of one exchange to the users of another exchange. However, a centralized exchange requires users to trust that the exchange will securely maintain bitcoin on their behalf and will provide bitcoin to them when requested.

A much more promising approach to scaling Bitcoin, while maintaining Bitcoin's decentralized and trust-free nature, is to use a Layer 2 protocol **[GMRMG]**. In a Layer 2 protocol, parties exchange bitcoin by sending messages (including Bitcoin transactions) between themselves. In a well-designed Layer 2 protocol, most bitcoin transfers do not require any *on-chain* transactions (that is, transactions that are put on the blockchain). By keeping most transactions *off-chain* in the normal case (but available to be put on-chain if required), scalability is achieved without requiring trust between the parties. Furthermore, Layer 2 protocols can greatly reduce the latency of bitcoin transfers, thus making them more convenient.

One Layer 2 protocol, the *Lightning Network*, requires no changes to Bitcoin and is currently operational **[PD16] [Rus15]**. The Lightning Network consists of a network of *two-party payment channels* (or simply, *channels*)[1], each of which contains a fixed amount of bitcoin **[BWPC]**. The parties in a channel use the Lightning protocol to update the portion of the channel's bitcoin that is owned by each of them. As long as both parties follow the Lightning protocol, channel updates do not require any on-chain transactions. The Lightning Network also allows parties that do not share a channel to exchange bitcoin by setting up a path of channels, where successive channels in the path include a common party **[BWHTLC] [DW15] [GMRMG] [MMSKM] [PD16]**.

By supporting channel updates and routing across multiple channels without requiring on-chain transactions, the Lightning Network greatly improves Bitcoin's scalability. However, Lightning does not fully solve the scaling problem, as an on-chain transaction is required to create or close a Lightning channel. In addition, the Lightning protocol is complex and has several usability and security challenges. Specifically, Lightning channels require that both parties constantly monitor the blockchain and implement (or pay for) a *watchtower service* **[GMRMG]**. Whenever a watchtower detects that the other party has put an old (and thus invalid) update transaction on-chain, it must respond accordingly within a bounded amount of time. The need to constantly monitor the blockchain is problematic for a casual user running wallet software on a smartphone that may be turned off. In addition, Lightning

1 Throughout this paper (except in Section 3.4 and Appendix B), the word *channel* will be used to refer to a payment channel with only two parties.

channels use *penalty transactions* that put a user's funds at risk in the event that they make an error. Finally, Lightning channels do not support *late determination of transaction fees*, so it is difficult to guarantee that the required Lightning transactions can be put on-chain as quickly as is required.

In order to improve scalability, *channel factories* (or simply, *factories*) have been proposed in which a fixed set of N > 2 parties use a small number of on-chain transactions to implement a large number of channels involving pairs of the N parties **[BDW18]**. Unfortunately, all currently-proposed factories are subject to two limitations: 1) the number of parties in the factory is limited by the fact that all parties must coordinate and agree on the exact set of parties in the factory, and 2) the parties in the factory are fixed and cannot be changed without closing the factory. As a result, the creation of a Layer 2 protocol that does not require tight coordination from all parties, and that supports dynamic changes to the parties involved, has been an important research goal **[Ola18]**.

A proposed Layer 2 protocol, *eltoo*, is simpler than Lightning and improves upon Lightning channels in several ways **[DRO18]**. However, eltoo requires a change to Bitcoin's rules for signing transactions **[BIP118]**. Specifically, eltoo requires a new type of *floating transaction* in which a signed floating transaction does not need to specify which transaction output is being spent[2]. eltoo channels are much simpler than Lightning channels, they eliminate penalty transactions, and they support late determination of transaction fees (but still require watchtowers). Furthermore, eltoo can be used to implement efficient factories[3]. These advantages are substantial, and as a result there is a strong case for adding floating transactions to Bitcoin. However, there is some risk in supporting floating transactions, because they could enable a signed transaction to be used multiple times or in unexpected ways. Therefore, it would be beneficial if the advantages of eltoo could be achieved with fewer risks.

The current paper introduces four new Layer 2 protocols. The first one, called *2Stage*, is a simple channel protocol that eliminates watchtowers and penalty transactions, supports late determination of fees, and has better worst-case performance than eltoo. The second one, *timeout-trees*, is a highly-scalable factory protocol that achieves increased scaling by decoupling the number of parties in the factory from the number of parties cooperating to create the factory (at the expense of requiring an on-chain transaction to update the set of channels created by the factory). Timeout-trees thus remove the existing limits on new users' ability to obtain bitcoin in a trust-free manner. The remaining two factory protocols, *update-forest* and *challenge-and-response*, eliminate both of the limitations of currently-proposed factories listed above. Specifically, these are the first factory protocols that scale to an unbounded number of parties and channels (without having to solve a difficult group coordination problem) and that allow all of the channels in the factory to be bought and sold by anyone (including parties not originally in the factory) with a single on-chain transaction in a trust-free manner[4].

---

2 Although the floating transaction must still meet the requirements established by the output that it is spending.

3 The eltoo paper presents the protocol as implementing a multi-party "payment channel" in which all parties update the state of the channel, but that paper also notes that it can be used as a channel factory **[DRO18]**.

4 The timeout-trees, update-forest and challenge-and-response protocols are called "channel factories" in this paper, but there is no existing term that exactly captures their characteristics. On the downside, these protocols require an on-chain transaction to update the ownership of the channels that they create, while that is not required for existing channel

Like eltoo, these new protocols require a change to Bitcoin. Unlike eltoo, they do not require floating transactions. Instead, they rely on transaction signatures that use *Inherited IDs (IIDs)*. A signature that uses IIDs does not specify the exact transaction with the output that it spends. However, it does specify how that output is related to a specific transaction (such as being the first output of the second child of a specific transaction's third child)[5]. Therefore, a transaction that is signed with IIDs cannot be used more than once or in an unexpected location. Surprisingly, despite being more restrictive and safer than floating transactions, IIDs are actually more powerful than floating transactions. In particular, both the update-forest and challenge-and-response protocols appear to be impossible with floating transactions. As a result, we propose that Bitcoin be changed to support IIDs, rather than floating transactions.

Section 2 gives more background on Bitcoin, formalizes the group coordination problem, and presents a high-level description of the changes proposed for supporting eltoo and this paper's protocols. Section 3 presents and analyzes channel protocols, including the new 2Stage protocol. Covenants, which are previously-proposed constructs that are enabled by IIDs, are presented in Section 4. Protocols for factories are given in Sections 5 through 7. Optimizations and extensions to the factory protocols are presented in Section 8, and techniques for implementing off-chain channel networks are given in Section 9. Related work and conclusions complete the paper. Technical results (including details of the IID proposal, bounds on channel protocols, proofs of correctness, and an implementation of vaults using IIDs) are given in appendices.

2 Proposed Changes to Bitcoin

2.1 Bitcoin Background

If transaction C spends an output from transaction P, C is a *child* of P and P is a *parent* of C. The inputs and outputs of a transaction are identified by their index, starting with index 0. When an input of a child transaction spends an output of a parent transaction, the input specifies which output is being spent with an *OutPoint* structure that consists of the pair (transaction ID, output index), where:

- *transaction ID* is a hash of all of the fields in the parent transaction[6], and
- *output index* specifies which output from the parent transaction is being spent **[BWPDTV]**.

---

factories. On the upside, these protocols are more scalable than any existing factory protocols, as they do not require that all parties in the factory solve the group coordination problem defined in Section 2.2. Timeout-trees are valuable because they provide a simple, trust-free and watchtower-free means for large numbers of casual uses to obtain bitcoin. Update-forest and challenge-and-response factories are more scalable and powerful, as they allow millions of channels to be bought and sold in a trust-free manner by arbitrary parties with a single on-chain transaction, and those parties do not have to be fixed or known when the factory is created.

5 This description of IIDs is included in order to give intuition about the concept. In practice, a signature that uses IIDs depends only on the transaction being signed and data contained in (or metadata associated with) the transaction with the output that is being spent. As a result, the use of IIDs does not break Bitcoin's UTXO model. See Appendix A for details.

6 Except the witness fields.

Associated with each transaction output is an *output script* that specifies the conditions that any child transaction must satisfy in order to spend the given output[7]. The output script is a program written in a simple stack-based programming language called *Bitcoin Script*. In order to spend the output, the child transaction provides a sequence of *input stack values* that are used as inputs to the parent's associated output script program[8]. The child transaction is only allowed to be put on-chain if the input stack values provided result in an execution of the output script that completes successfully and results in exactly a single "TRUE" on the stack. If multiple off-chain transactions are designed to spend the same transaction output, those transactions are said to *compete*, and at most one of them can ever be put on-chain (as each output can only be spent once).

Bitcoin Script includes operations that support public key cryptography. In public key cryptography, a party publishes a public key (or *pubkey*) for which only it knows the associated secret *private key*. That party can cryptographically *sign* a message (or a transaction or a part of a transaction) by calculating a *signature* that is a function of the message and private key. Any other party can then verify whether or not the signature is correct for the given pubkey and message, and if so, can conclude that the party has approved of the message. In Bitcoin, output scripts often include a pubkey and an OP_CHECKSIGVERIFY operator that forces the creator of the child transaction to provide a signature for (all or part of) the child transaction and the given pubkey, thus proving that the party with the associated private key approves of the child transaction **[BS]**. In such a case, the party that knows the private key is said to *own* the pubkey and the transaction output that requires the pubkey. It is also possible to create a single pubkey that is associated with a set of private keys, each of which is a secret of a different user **[NRS20]**. In this case, the set of users are the *owners* of the pubkey, the signature that they create is called a *multisig*, and a transaction output that requires a multisig signature is called a *multisig output*. Each user can create a partial signature by signing with the user's private key, and partial signatures from all of the owners can be combined to create a complete multisig signature.

Bitcoin also provides fields and operators that can be used to delay the placement of a transaction on-chain by a specified amount of time. Each transaction has an nLocktime field that gives the earliest time it can be put on-chain, and the OP_CHECKLOCKTIMEVERIFY operator in an output script forces any spending transaction to have at least a specified minimum nLocktime value (thus forcing it to remain off-chain until the specified minimum nLocktime is allowed on-chain). Relative delays are implemented with the nSequence field that is associated with each transaction input. If a child transaction has an input with an nSequence value of S, the child cannot be put on-chain until S time has passed from when the parent creating the output being spent was put on-chain. In addition, the parent transaction's output script can include an OP_CHECKSEQUENCEVERIFY operator that specifies the minimum value for the nSequence field of the input in the child transaction spending the output. Layer 2 protocols often need to use the nLocktime or nSequence fields to force a delay that is long enough to

7 The parent transaction either contains the output script itself, or contains a commitment to the output script in the form of a hash that depends on the output script.

8 Throughout the remainder of this paper, Bitcoin is assumed to support SegWit version 1 or higher with Taproot and Schnorr signatures **[WNR20] [WNT20] [WNT20b]**.

allow a party to put a single transaction (which is not subject to that delay) on-chain. Throughout this paper, the delay that is long enough to allow a party to put one such transaction on-chain will be called a *unit time window*, or simply a *time window*[9].

Other details of the Bitcoin protocol are beyond the scope of this paper, but are available online (see **[Ant17]**).

2.2 The Group Coordination Problem

A challenge that arises in a number of Layer 2 protocols is the coordination of a group of participants. A specific coordination problem, which we call the *group coordination problem*, is formalized as follows:

> **Input:** A set S of N participants, where each participant p, $1 \leq p \leq N$, has a unique value V_p and a public/private key-pair, and a function f() that takes N values as inputs[10].
>
> **Output:** A partial signature from each participant in S for the output of the function $f(V_1, V_2, ..., V_N)$.

The group coordination problem is challenging because a single non-responsive or malicious user prevents the entire group from solving the problem, and removing such a user from the group creates a new problem that must be solved from scratch.

2.3 BIP 118

As was described above, an output script can force a child transaction to provide a signature showing that the transaction is approved by the owner of the output being spent. The signature provided with the child transaction includes a 1-byte flag, called the *SIGHASH* flag, that specifies which parts of the child transaction are covered by the signature. All existing options for the SIGHASH flag require the signature to cover the transaction ID and index of the output being spent, as well as (at least part of) the script associated with that output.

In order to support the eltoo protocol, two new types of SIGHASH flags are proposed in BIP 118 **[BIP118]**. The first one, SIGHASH_ANYPREVOUT, eliminates coverage of the transaction ID and index of the output being spent. The second one, SIGHASH_ANYPREVOUTANYSCRIPT, further eliminates coverage of the associated output script and of the amount of the output being spent.

9 In reality, no fixed amount of time (or number of blocks) can be sufficient to guarantee that a given party will succeed in putting a transaction on-chain. Still, the time window concept is an abstraction that is helpful in designing and analyzing Layer 2 protocols. The value of the time window parameter that is used in practice represents a trade-off between security and latency, and should be selected conservatively if significant funds are at stake.

10 For example, f() could be a hash function or a function that creates a transaction that has an output for each participant.

2.4 This Paper's Proposed Change: Inherited IDs (IIDs)

This paper proposes a single change to Bitcoin. Like the change proposed for eltoo, the change proposed here affects the construction of signatures. It is first presented in an intuitive manner that ignores backward-compatibility issues, and then as the actual change with backward-compatibility supported.

Intuitive Description of the Proposed Change

The simplest way to understand the proposed change is to imagine what the change would be if it did not have to be compatible with existing Bitcoin software. Ignoring such compatibility issues, the change would allow an output O to specify that the child spending O must reference O with an OutPoint of the form (Inherited ID, output index), where:

- *Inherited ID (IID)* is a hash of the OutPoint in input 0 of the parent transaction (rather than the transaction ID, which is a hash of all of the fields in the parent transaction), and
- *output index* specifies which output from the parent transaction is being spent.

By using an IID rather than a transaction ID to reference the output being spent, the spending transaction (and any signature for the spending transaction) does not depend on the exact values of most of the fields in the parent transaction. However, the spending transaction still uniquely identifies which transaction output is being spent, because only the specified parent can have the given IID (assuming no hash function collisions). Furthermore, note that a child that spends an output of a parent that is referenced using an IID can itself have outputs that are referenced using IIDs. In this manner, a chain or a tree of transactions with IIDs can be created.

In order to facilitate the description of protocols that use IIDs, the notation T.a will be used to represent hash(ID(T) || a), which is the IID of child "a" of transaction T (where "||" denotes concatenation and ID(T) is either the transaction ID or IID of T, depending on how T is referenced by its child "a"). This notation can be used iteratively, so T.a.b.c equals hash(hash(hash(ID(T) || a) || b) || c) and is the IID of child "c" of child "b" of child "a" of transaction T.

Examples of IIDs (based on this intuitive description of IIDs) are shown in Figure 1. In that figure, and throughout this paper, transactions are shown as boxes with on-chain transactions being shaded and off-chain transactions being empty. Transaction outputs leave the transaction from the right side of the transaction box and inputs enter from the left. Transaction inputs and outputs are ordered from top to bottom, with input 0 and output 0 being topmost. Outputs that are referenced via transaction IDs are shown as solid lines, while those that are referenced via IIDs are dashed. When a single output branches to become an input to multiple transactions, those transactions that receive the input compete (and thus at most one of them can ever be put on-chain).

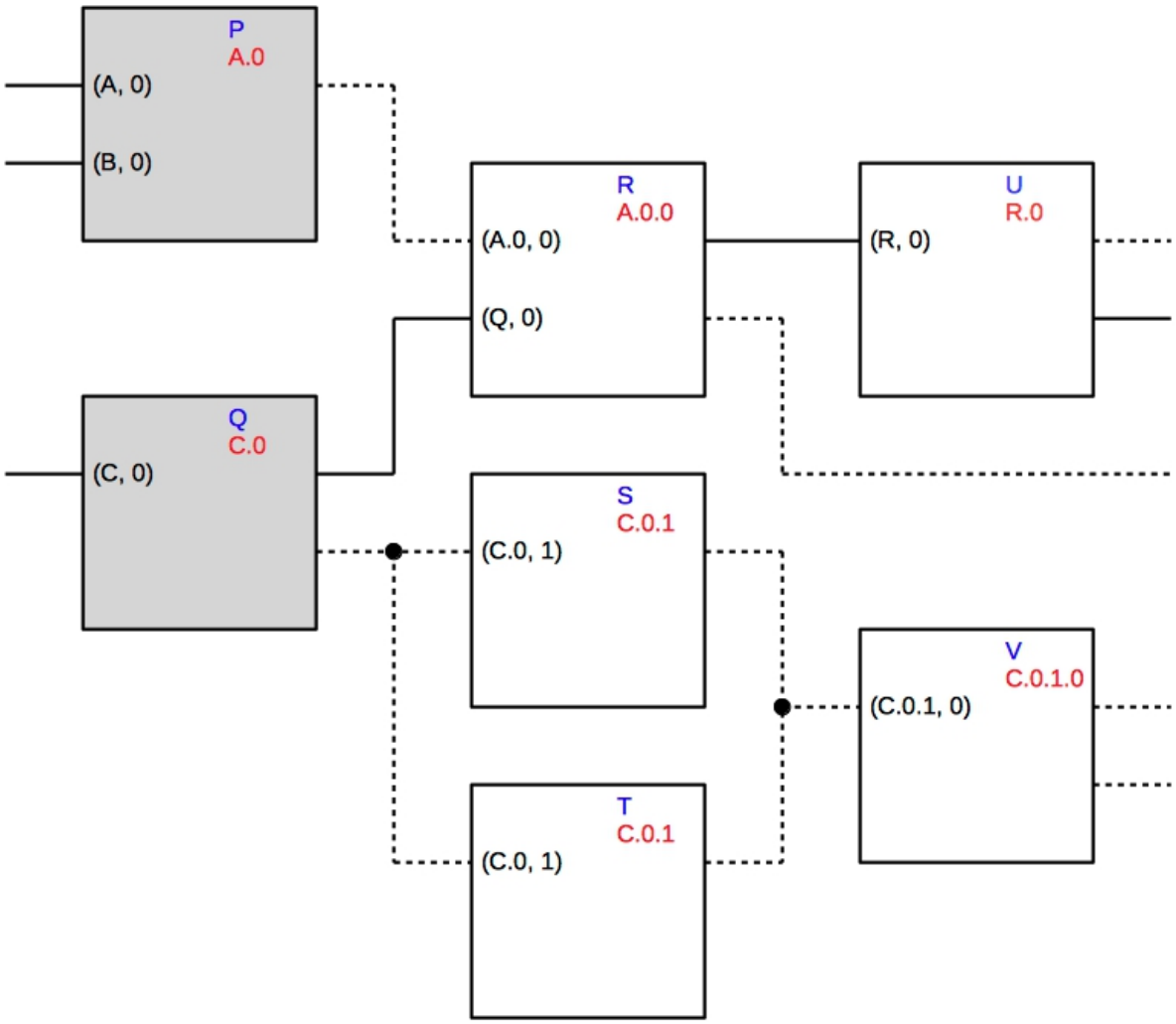

Figure 1: Transaction IDs and IIDs[11]. On-chain transactions P and Q are shown with some potential descendent transactions. Transaction IDs are shown in blue, IIDs are shown in red, and OutPoint values are shown in black. Note the relationship between a transaction's input 0 OutPoint and its IID.

As can be seen in Figure 1, each transaction's IID can be obtained by hashing the transaction's input 0 OutPoint. For example, transaction R's input 0 has the OutPoint value (A.0, 0) (that is, its input 0 spends output 0 of the transaction with IID A.0), and transaction R's IID is A.0.0 = hash(hash(ID(A) || 0) || 0). Note that transaction U's input 0 has the OutPoint value (R, 0) (that is, its input 0 spends output 0 of the transaction with transaction ID R), as R's output 0 is referenced via transaction ID.

Note that transactions S and T compete to spend Q's output 1. However, because S's output 0 and T's output 0 both use IIDs, V can spend the output of whichever transaction (S or T) is put on-chain without changing its OutPoint 0 value of (C.0.1, 0).

11 Note that Figure 1 illustrates a simple form of IIDs that would require a hardfork. See Appendix A for the actual IID proposal, which can be implemented with a softfork.

Actual Description of the Proposed Change

In reality, the change described above is not attractive, as it is not backward-compatible and thus would create a hardfork **[BWHF]**. The actual proposed change leaves the OutPoint data structure as it is, and simply modifies how signatures are calculated to simulate the effect of using the modified OutPoint data structure. The details of the actual proposed change, which is backward-compatible and can be introduced as a softfork **[BWSF]**, are presented in Appendix A. For ease of explanation, the remainder of this document (except in Appendix A) will assume the simpler (non-backward-compatible) change described above.

3 Channels

3.1 Basic Concepts

As described previously, a channel is a Layer 2 protocol in which two fixed parties assign, and then re-assign, a specific amount of bitcoin between them **[BWPC]**. While many techniques have been used to create channels, several have the following general form.

First, a *funding transaction* is created that assigns a certain amount of bitcoin to a multisig address owned by both parties and a *refund transaction* is created that spends the output of the funding transaction and distributes it to the parties in a fair manner. The refund transaction is signed by both parties before the funding transaction is put on-chain, thus guaranteeing that the funds will be spendable.

The distribution of funds from the funding transaction can be updated via off-chain transactions that are signed by both parties. For example, the parties could create new *settlement transactions* that provide new distributions of funds, each of which is the child of one or more *update transactions* that serve as a pipe for moving funds from the funding transaction to the desired settlement transaction. The protocol must provide some means for guaranteeing that the desired update transaction(s) and settlement transaction, rather than the refund transaction, can be put on-chain. For example, the refund transaction can have a timelock which delays its placement on-chain, while the desired update transaction has no such delay (or a smaller delay) and competes with the refund transaction by spending the same output from the funding transaction. Of course, the protocol must also guarantee that only the latest settlement transaction, rather than some earlier settlement transaction, can be put on-chain, and a number of techniques have been created for providing this guarantee (see below) **[GMRMG]**.

In addition, the protocol can include a *close transaction* that provides the same distribution of funds as the final settlement transaction, but can be put on-chain earlier and/or more efficiently.

Finally, if the delays of the refund and update transactions are absolute (or relative to when the funding transaction is put on-chain), the channel will have a bounded lifetime, after which the refund transaction could be put on-chain. In order to extend the lifetime, it is possible to introduce a *trigger*

transaction that acts as a pipe between the output of the funding transaction and the input of the refund and update transactions. By making the time delays of the refund and update transactions relative to when the trigger transaction is put on-chain, an unbounded channel lifetime can be achieved (provided no party puts the trigger transaction on-chain until both parties want to close the channel).

3.2 Channel Protocols

A number of channel protocols have been proposed.

Invalidation Trees

Decker and Wattenhofer created *invalidation trees*, in which a unique sequence of update transactions leads to each settlement transaction **[DW15]**. In order to create a new settlement transaction and invalidate the current one, the parties sign the new settlement transaction and then sign one or more new update transaction(s) leading to the new settlement (by having the new settlement spend the output of the last new update). The first new update transaction is given a smaller time delay than any previous update transaction with which it competes, thus guaranteeing that the new update transaction(s) will be able to be put on-chain. By structuring the update transactions in a tree, the total number of transactions that have to be put on-chain in the worst case is reduced. Because each transaction requires a signature from both parties, no change to the channel's state (as expressed by the latest settlement transaction) can be made without both parties' consent.

The Lightning Network

Poon and Dryja introduced the Lightning Network, which includes a channel protocol. Each party maintains its own set of off-chain update and settlement transactions **[PD16] [Rus15]**. New public and private keys are used for each new update and settlement transaction. Old update and settlement transactions are invalidated by having one party give the private keys which allow the other party to sign new transactions that compete with (and can be placed on-chain earlier than) the old update or settlement transaction.

eltoo

Decker, Russell and Osuntokun proposed the eltoo protocol **[DRO18]**. In eltoo, update and settlement transactions are floating transactions that are signed with the SIGHASH_ANYPREVOUTANYSCRIPT and SIGHASH_ANYPREVOUT flags, respectively. As a result, the signatures on these transactions do not depend on the transaction ID of their parent transaction, so they must use some other mechanism to constrain the output that they spend. Because the signature in each settlement transaction uses the SIGHASH_ANYPREVOUT flag which *does* cover its parent's output script, and because each update transaction has a unique output script, each settlement transaction binds to a unique parent update transaction[12].

12 The description given here matches the most recent proposal for BIP 118. The original eltoo paper used separate public/private key-pairs for each settlement transaction and its associated update transaction.

In contrast, each update transaction is signed with the SIGHASH_ANYPREVOUTANYSCRIPT flag, which would allow a signed update transaction to be the child of any other update transaction (or of the trigger transaction). In order to constrain the ordering of updates, eltoo makes novel use of the nLocktime field. As described above, this field was intended to bound the earliest time that a transaction can be put on-chain. However, in the eltoo protocol all update transactions have nLocktime values that are in the past. Rather than constraining when the update transactions can be put on-chain, the nLocktime values represent the state number of the update transaction. Each update transaction has an output script that forces any child update transaction to have a larger nLocktime value (by use of the OP_CHECKLOCKTIMEVERIFY operator), thus preventing an earlier (lower state number) update transaction from following a given update transaction. In the ideal case, only the final (highest state number) update transaction will be put on-chain. Even in the worst case, in which a malicious party continually puts the lowest-possible update transaction on-chain, the final update transaction (and its associated settlement transaction) will eventually be put on-chain.

3.3 2Stage Channels

We now introduce a new channel protocol, 2Stage, that uses Inherited IDs (IIDs) as described in Section 2.4. It borrows ideas from both Lightning channels and eltoo. Like Lightning channels, 2Stage is designed for two parties, each of whom privately maintains certain transactions that are partially-signed by the other party. Like eltoo, 2Stage uses values of the nLocktime field that are in the past to represent channel state numbers and OP_CHECKLOCKTIMEVERIFY to prevent an earlier state from invalidating a later state.

The protocol has a very simple two-stage structure following the placement of the funding transaction on-chain. In the first stage, one party puts an *update transaction* for some state X on-chain. In the second stage either the same party puts a *settlement transaction* for the same state X on-chain, or the other party puts a *remedy transaction* for some state $Y \geq X$ on-chain. The settlement transaction or the remedy transaction provides the payout to the two parties that corresponds to the given channel state. In order to guarantee that the other party has a chance to put a remedy transaction on-chain, the settlement transaction has a delay before it can be put on-chain. The update, settlement and remedy transactions held by one party differ trivially from those held by the other party, so that only the same party that put an update transaction on-chain can use a settlement to spend the output of that update, and only the other party can use a remedy to spend the output of that update. There is no trigger or refund transaction. The overall structure of the 2Stage protocol is shown in Figure 2.

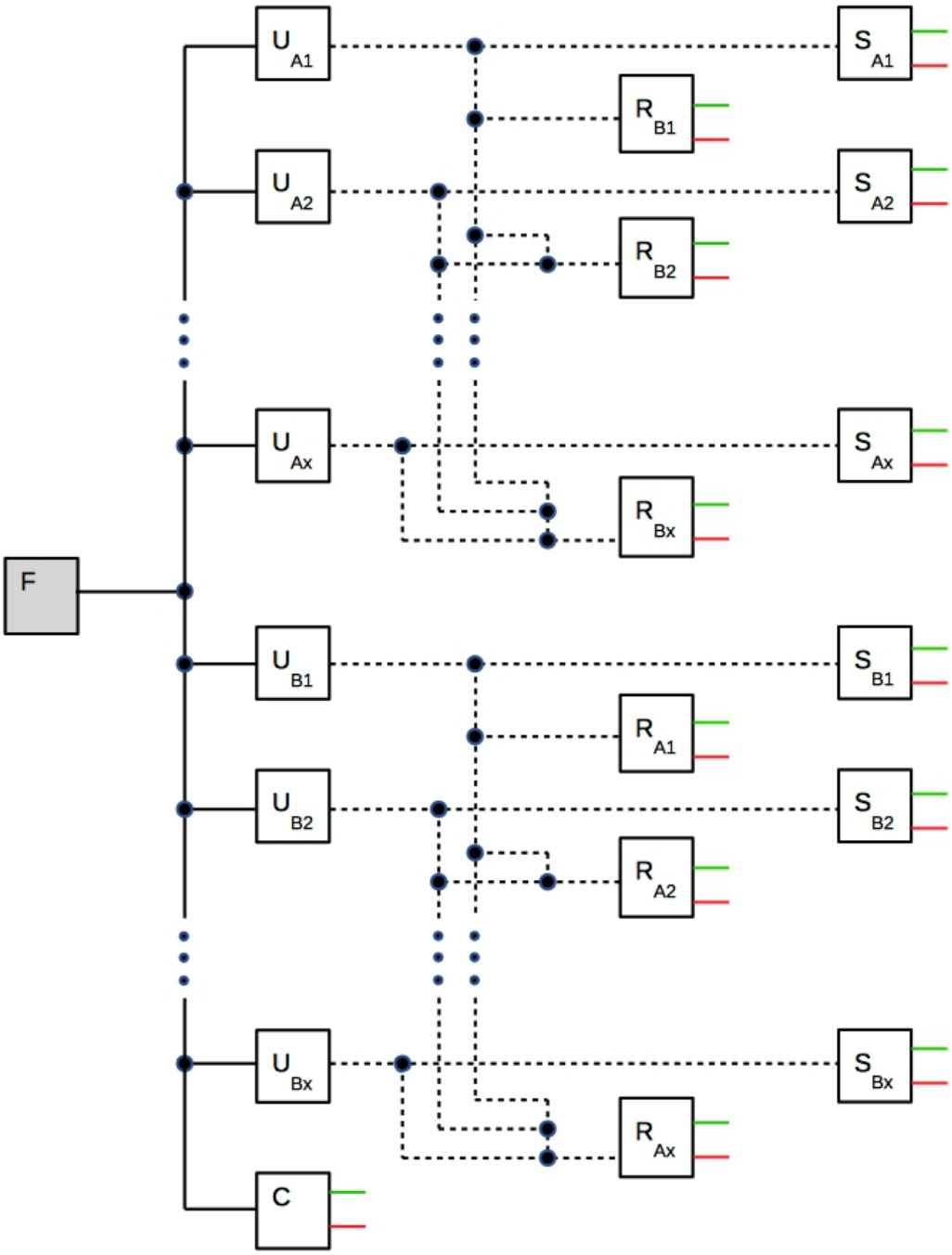

Figure 2: Overall structure of the 2Stage protocol. The type of each transaction (Funding, Update, Remedy, Settlement or Close) is indicated by a single letter. The first subscript gives the party (A = Alice, B = Bob) that can put the transaction on-chain and the second subscript gives the channel state. Line colors denote who must sign to spend the output: green = Alice, red = Bob, black = Alice & Bob.

In order to allow the output of a state-X update transaction to be spent by a pre-signed state-Y remedy transaction, where Y ≥ X, the output of the update must be referenced using IIDs. Remedy transactions for state Y < X are prevented from spending the output of a state-X update by using nLocktime values from the past to represent the state number and OP_CHECKLOCKTIMEVERIFY to prevent a child from having a lower state number than its parent (as in eltoo). Settlement transactions for state Y ≠ X are prevented from spending the output of a state-X update by having the signature in the settlement cover the state number X in the output script of the update. Of course, if the signature in a remedy transaction also covered the state number in the output script of the update, the remedy could only be used to spend the output of a single specific update (thus defeating the purpose of using IIDs). This problem is avoided by placing an OP_CODESEPARATOR[13] in the part of the output script of the update transaction that is executed when it is spent by a remedy transaction, thus blocking the signature in the remedy transaction from covering the state number in the output script.

When the parties want to update the channel to a given state X (or to set the initial channel state), each party partially-signs both a state-X settlement transaction and a state-X remedy transaction and gives them to the other party. Next, each party partially-signs a state-X update transaction and gives it to the other party. Finally, if state X is the initial state, each party signs the funding transaction and one of the parties puts it on-chain.

Once both parties have received their partially-signed transactions for state-X, either party can fully-sign and put their state-X update transaction on-chain, wait for one time window, and then put their fully-signed state-X settlement transaction on-chain.

However, if Alice puts her state-X update on-chain and Bob has a partially-signed remedy for a state Y ≥ X, Bob can fully-sign and put his state-Y remedy on-chain before Alice can put her state-X settlement on-chain. In fact, if Bob has multiple such state-Y remedy transactions where Y ≥ X, he can choose to put on-chain the one that is most advantageous to him. The existence of such remedy transactions prevents either party from cheating and attempting to settle the channel in an old state.

If both parties want to close the channel, they can sign a close transaction that competes with the update transactions and provides the final channel payout to both parties.

Some details of the funding, update, settlement, remedy and close transactions are given below:

Funding transaction:

Output script (referenced via transaction ID):

OP_DUP <pubkey A&B> OP_EQUALVERIFY OP_CHECKSIG check signature for pubkey A&B

13 OP_CODESEPARATOR is a Bitcoin Script operator that blocks the signature in a spending transaction from covering the portion of the output script that precedes the most recently executed OP_CODESEPARATOR. The signature is also prevented from covering the entire output script by changes described in Appendix A.

Update transaction for state X:

Input stack values: **<signature> <pubkey A&B>**

Output script (referenced via IID):

<state X> OP_CHECKLOCKTIMEVERIFY OP_DROP child transaction has nLocktime ≥ state X

OP_IF settlement transaction X

> **OP_DUP <pubkey A&B> OP_EQUALVERIFY OP_CHECKSIG** check signature for pubkey A&B

OP_ELSE remedy transaction Y ≥ X

> **OP_CODESEPARATOR** prevent the following checksig from covering the <state X> value above, thus allowing the signature for the child transaction to be independent of the value of X
>
> **OP_DUP <pubkey A&B> OP_EQUALVERIFY OP_CHECKSIG** check signature for pubkey A&B

OP_ENDIF

<OP_0 if Alice's, OP_1 if Bob's> push 0 (if Alice's transaction) or 1 (if Bob's transaction), which is included to distinguish signatures for children of Alice's update transactions from signatures for children of Bob's update transactions

OP_DROP remove previous 0 or 1 from stack, as it was only included to differentiate signatures

Settlement transaction for state X:

nLocktime field: **<state X>**

nSequence field: **<1 time window>**

Input stack values: **<signature covering same party's update transaction X output script> <pubkey A&B> <TRUE>**

Remedy transaction for state X:

nLocktime field: **<state X>**

Input stack values: **<signature covering other party's update transaction output script after OP_CODESEPARATOR> <pubkey A&B> <FALSE>**

Close transaction:

Input stack values: **<signature> <pubkey A&B>**

3.4 Analysis

The following table summarizes key properties of invalidation trees (IT), the Lightning Network (LN), eltoo, and 2Stage channels. It uses asymptotic ("Big-O"[14]) notation to quantify the performance of a channel that goes through a sequence of N states. The entries marked "GCP" indicate that a potentially unbounded number of parties can participate in the protocol, but that the parties must solve the group coordination problem (see Section 2.1), thus creating a practical limit to the number of parties supported.

	IT	LN	eltoo	2Stage
# of parties supported	GCP	2	GCP	2
# on-chain txs & time: Cooperative	O(1)	O(1)	O(1)	O(1)
# on-chain txs & time: Non-responsive	O(log N)	O(1)	O(1)	O(1)
# on-chain txs & time: Malicious	O(log N)	O(1)	O(N)	O(1)
Error causes loss of all funds?	No	Yes	No	No
Supports late determination of fees?	No	No	Yes	Yes
Requires changes to Bitcoin?	No	No	Yes	Yes
Requires floating transactions?	No	No	Yes	No
Requires IIDs?	No	No	No	Yes

Table 1. Comparison of Channel Protocols.

The analysis of the number of on-chain transactions (and number of time windows elapsed) uses the following definitions for the parties implementing the channel protocol:

- **Cooperative:** every party attempts to implement the protocol fairly and is always responsive,
- **Non-responsive:** at least one party stops participating in the protocol, and
- **Malicious:** at least one party attempts to delay or thwart other parties, even at their own expense.

When all of the parties are cooperative, every protocol supports the use of a close transaction that finalizes the channel state with a constant number of on-chain transactions and in a constant number of time windows. However, if a party is non-responsive, the close transaction cannot be used, thus forcing invalidation trees to use a logarithmic number of time windows. With eltoo, a self-interested party will not attempt to put old update transactions on-chain, as they will never be able to steal funds (assuming the time window is sufficiently large and some party attempts to put the final update transaction on-

14 https://en.wikipedia.org/wiki/Big_O_notation

chain). However, eltoo does permit a malicious party to pay the fees to put up to N update transactions on-chain, thus significantly delaying a cooperative party from getting its funds.

It is interesting to note that the two protocols that support a potentially unbounded number of parties (namely invalidation trees and eltoo) are also the ones that have a non-constant worst-case number of on-chain transactions and amount of time required. It turns out that this is not coincidental, as any channel protocol that uses a trigger transaction for unbounded channel lifetime, uses time window delays to guarantee the ability to put transactions on-chain, and supports an arbitrary number of parties must require log N time windows in the worst case. The details are given in Appendix B.

Finally, if a party accidentally puts an old transaction on-chain with the Lightning Network, they could lose all of their funds in the channel, even if they always had a positive balance of funds in the channel. This is because the Lightning Network forces each party to reveal the private keys that protected the outputs of their old transactions. This can be a problem in practice, as such an error could be made due to an honest mistake (such as restoring a backup of an earlier channel state).

3.5 Late Determination of Fees

Each Bitcoin transaction includes a fee (defined as the total bitcoin in the inputs minus the total bitcoin in the outputs) that pays miners to put the transaction on-chain. When attempting to put a transaction on-chain within a fixed time window, the fee must be large enough to convince miners to put the transaction on-chain before the time window expires, or funds could be lost. However, when signing a transaction that can be held off-chain for an extended period as part of a Layer 2 protocol, it can be difficult to estimate the required fee, as network congestion, and hence fee requirements, vary over time. As a result, it is beneficial if the fees for a transaction can be determined when attempting to put it on-chain, rather than when signing it.

Bitcoin's existing SIGHASH flags, which specify the portions of the transaction that are covered by a signature on the transaction, can be used to allow a single signature to cover multiple versions of a transaction having different fees. For example, SIGHASH_ANYONECANPAY allows new inputs to be added to a transaction after it has been signed, and SIGHASH_SINGLE allows the amount of bitcoin that is refunded to the party putting the transaction on-chain to be varied after signing.

More problematic is signing a transaction whose parent is off-chain and can be kept off-chain for an extended period, because the existing Bitcoin protocol forces the signature on the child to cover the transaction ID of the parent, which in turn depends on the fee in the parent. As a result, it is impossible to update the fee in the parent without invalidating the signature in the child. Any channel protocol that uses Bitcoin as it currently exists (including invalidation trees and the Lightning Network) is thus subject to this limitation[15].

15 Although some flexibility can be obtained by signing multiple versions of transactions with different fees or by using replace-by-fee or child-pays-for-parent **[SHMB20]**.

Because eltoo uses floating transactions which do not require the signature of a child to cover the transaction ID of the parent, fees can be added to eltoo transactions when they are put on-chain without invalidating signatures on child transactions. Similarly, because 2Stage uses IIDs that do not change when a new input is added, fees can be added to 2Stage update transactions (by adding a new input to the update transaction) without invalidating signatures on child settlement and remedy transactions. Because the output of the funding transaction in 2Stage is referenced using its transaction ID, its fee cannot be modified without invalidating previously-signed child transactions. Fortunately, the funding transaction can be put on-chain immediately after signing its child update transactions for the initial channel state, so this should not be a problem.

3.6 Elimination of Watchtowers

As described so far, 2Stage requires that both parties constantly monitor the blockchain and implement (or pay for) a watchtower service **[GMRMG]**. Fortunately, the requirement for monitoring the blockchain can be eliminated for one or both parties.

First, consider the case where Alice operates multiple channels and is constantly monitoring the blockchain, but Bob is a casual user who cannot monitor the blockchain and does not want to pay (or trust) a separate watchtower service. In this case, Alice and Bob can agree to a fixed maximum channel lifetime of M time windows. Alice's update transactions are all signed with nSequence values that prevent them from being put on-chain until M time windows have passed since the funding transaction was put on-chain. Bob's update transactions do not have such nSequence values, and thus can be put on-chain at any time, and the close transaction can be signed and put on-chain at any time. As a result, if both parties agree, they can close the channel and get their funds at any time. Even if Alice is non-responsive, at any time during the life of the channel Bob can put his latest update transaction on-chain and then put his latest settlement transaction on-chain one time window later (unless Alice has already put her latest remedy transaction on-chain, in which case the channel is closed and Bob's funds are already available). Thus, while Bob does have to act within the lifetime of the channel, he never has to monitor the blockchain. The only time Bob's actions are affected by what Alice puts on-chain is when Alice puts her final remedy transaction on-chain before Bob puts his final settlement transaction on-chain. In this case, Alice has correctly closed the channel for Bob, and Bob does not need to take any further action. Furthermore, there is no time bound during which Bob must detect that Alice has already closed the channel.

This version of the protocol can be simplified by eliminating Bob's remedy transactions and by merging Alice's update transaction for state X with her settlement transaction for state X, so that the settlement transaction directly spends the output of the funding transaction (after M time windows have passed). This simplified form of the protocol is shown in Figure 3.

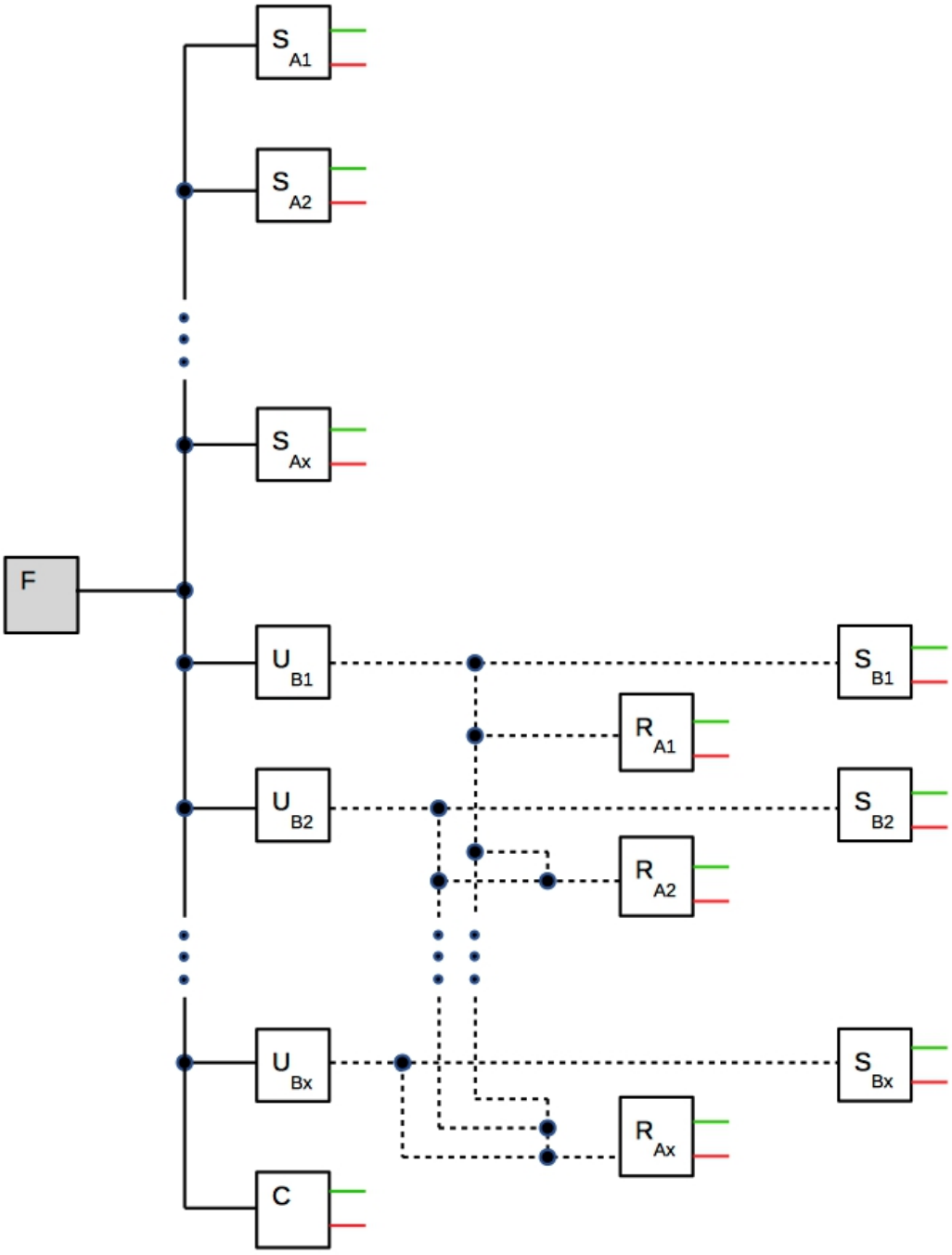

Figure 3: Simplified 2Stage protocol. The protocol is identical to the one shown in Figure 2, except Alice's settlement transactions have an nSequence value that delays them M time windows and Alice's settlement transactions directly spend the output of the funding transaction (thus eliminating both her update transactions and Bob's remedy transactions).

Next, consider the case where neither Alice nor Bob wants to implement or pay for a watchtower service. In this case, the simplified protocol from Figure 3 can be used, with the following two modifications:

1. Alice's settlement transactions have nSequence values which delay them M+1 (rather than M) time windows, and
2. Bob's update transactions have nSequence values which delay them M time windows.

With these modifications, Alice and Bob can still cooperatively close the channel at any time. If M time windows have passed without closing the channel, Bob must put his latest update transaction on-chain within the next time window. Then, either Alice puts her final remedy transaction on-chain or Bob puts his final settlement transaction on-chain. In any case, Bob never needs to monitor the blockchain.

Alice only has to put either her final settlement transaction or her final remedy transaction on-chain after M+1 time windows. Her choice of which transaction to put on-chain is based on whether or not Bob has put an update transaction on-chain. In fact, she can attempt to put both her final settlement transaction and her final remedy transaction on-chain at the same time, and whichever one succeeds will guarantee that the channel reaches the correct final state. In any case, Alice never needs to monitor the blockchain.

The above modifications limit the lifetime of the channel and thus force at least one transaction to be put on-chain during that lifetime. However, as will be shown in later sections on factories, this cost can be amortized over thousands or millions of channels, thus making it trivial.

4 Covenants

4.1 Basic Concept

A number of authors have proposed modifications to Bitcoin that allow an output script to place constraints on the form of the transaction that spends the output. Such constraints are called *covenants*, and they have been shown to be very powerful **[McE19] [MES16] [Rub19] [SHMB20] [Som20] [Tow19]**. For example, covenants have been proposed to implement *vaults*, which allow for increased security (see Appendix D) **[McE19] [MES16] [SHMB20]**. Next, we will show how covenants can be implemented using IIDs.

4.2 Implementation with Signature Values

One natural attempt to implement a covenant is to require that the child transaction not only have *some* signature (provided as part of the child transaction) corresponding to a specified public key, but that the child transaction have a *specific* signature value (provided as part of the output script) corresponding to a specified public key. Such a requirement is equivalent to requiring that the portion of the child

transaction that is covered by the signature (as determined by its SIGHASH flag) has a specified hash value, thus fixing the values within that portion of the transaction.

This approach cannot be used with Bitcoin as it is today, because all SIGHASH flag values force the signature on the child to cover the transaction ID of the parent whose output is being spent, and the transaction ID of the parent is a hash function that covers the entire parent transaction, including the output script (or a hash dependent on the output script) containing the specified signature value. This circularity prevents one from selecting a signature value to put in the output script.

Signature-Value Covenants with Floating Transactions

However, if a change is made to Bitcoin that allows this circularity to be broken, then Bitcoin can be used to implement covenants. For example, the new SIGHASH flags proposed for supporting eltoo via floating transactions allow the signature to not cover the transaction ID of the parent transaction **[BIP118]**. As a result, floating transactions would support covenants in Bitcoin **[Tow19]**.

Signature-Value Covenants with IIDs

The alternate change proposed here, namely the use of IIDs, also supports covenants. The key idea is to have the transaction output that places the covenant be referenced via an IID, and to have the output script contain an OP_CODESEPARATOR after it pushes the required signature value on the stack. Examples of output scripts that place covenants are given below:

Output script placing a covenant (referenced via IID):

<signature covering (portion of) child transaction and this output script after OP_CODESEPARATOR that places the desired covenant on the child transaction>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

Output script placing a covenant that also requires a signature for pubkey P (referenced via IID):

OP_DUP <pubkey P> OP_EQUALVERIFY OP_CHECKSIGVERIFY check signature for pubkey P

<signature covering (portion of) child transaction and this output script after OP_CODESEPARATOR that places the desired covenant on the child transaction>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

The first example above can be spent using a transaction that implements the required covenant and has an empty input stack, while the second example above requires an input stack that consists of the

signature for pubkey P and pubkey P. Note that the covenant can be implemented using a standard, widely-known pubkey with a widely-known "private key", as it is the value of the signature (rather than knowledge of the "private key") that is being enforced.

4.3 Types of Covenants with Signature Values

While the previous subsection shows how floating transactions or IIDs can be used to eliminate the circularity from a signature-value covenant on a transaction output, different types of covenants can be achieved by selecting the SIGHASH flag value that is included in the required signature value (regardless of whether floating transactions or IIDs are used). The types of signature-value covenants are described next[16].

- **Base:** A base covenant only specifies values for those fields that are covered by all SIGHASH flags. It does not place any additional constraints on the child transaction's inputs or outputs. It is implemented using a SIGHASH_NONE|ANYONECANPAY flag.
- **Input-only:** An input-only covenant specifies the required number of inputs along with the source, amount and nSequence value of each input in the child transaction and the index of the covenant-carrying input. It does not place any constraints on the child transaction's outputs. It is implemented using a SIGHASH_NONE flag.
- **Output-only:** An output-only covenant specifies the required number of outputs along with the amount and output script for each output in the child transaction. It does not place any constraints on the child transaction's inputs. It is implemented using a SIGHASH_ALL|ANYONECANPAY flag.
- **Input-output:** An input-output covenant specifies the number of inputs along with the source, amount and nSequence value of each input, the index of the covenant-carrying input, and the number of outputs along with the amount and output script of each output in the child transaction. It is implemented using a SIGHASH_ALL flag.
- **Input-single-output:** An input-single-output covenant specifies the number of inputs along with the source, amount and nSequence value of each input, and the index, amount and output script of the single output in the child transaction that has the same index as the input that spends the covenant-carrying output. It is implemented using a SIGHASH_SINGLE flag.
- **Single-output:** A single-output covenant specifies the amount and output script of the single output in the child transaction that has the same index as the input that spends the covenant-carrying output. It is implemented using a SIGHASH_SINGLE|ANYONECANPAY flag.

16 All types of signature-value covenants specify the required values for those fields that are covered by all SIGHASH flags, namely the nLocktime and nVersion fields of the child transaction, the source, amount and output script (after the most recently executed OP_CODESEPARATOR) of the output being spent and the nSequence value of the corresponding input (unless the child is a floating transaction using the SIGHASH_ANYPREVOUTANYSCRIPT flag), the spend type and annex, and the SIGHASH flag value.

4.4 Covenant Trees with Signature Values

Because a signature-value covenant can specify the required value of one or more output scripts in the child transaction, and because the required output script can itself include a signature-value covenant, it is possible to create covenants that affect multiple generations of descendent transactions. In particular, a covenant can force the creation of a specific tree[17] by having each transaction output that is internal to the tree create an output-only covenant defining the subtree that must follow. In general, the signature values that implement the desired covenants must be calculated in a bottom-up order, starting with the leaves and ending with the root[18].

If IIDs (rather than floating transactions) are used to implement covenants, the required signature values of the leaf transactions will depend on the IIDs of their parents. As a result, the IIDs of the transactions in the desired tree must be calculated first (in a top-down order starting with the root), followed by the desired signature values that are calculated in a bottom-up order.

Because signature-value covenant trees can be implemented using output-only covenants, the number of inputs and the amount associated with each input remains unspecified by the covenant. As a result, it is possible to add one or more new inputs to a transaction in a signature-value covenant tree at any time, and these new inputs can be used for the late determination of fees. Of course, any new input should not be input 0, as that would change the IID of the transaction.

While a signature-value covenant on the output of a root node constrains the output to create the desired tree, a party that does not know that tree will in general not be able to put it on-chain, as the exact transactions that comprise the tree are not obvious from the covenant's signature value. On the other hand, if a party has some idea of the structure of the tree, such as its size and shape and the amounts and output scripts of its leaves, it may be possible for them to guess and determine the required transactions. There are cases where the creator of the covenant would like to keep the structure of the tree private from others, or to only reveal the transactions on the path from the root to a leaf to the parties that own the outputs from that leaf. This is easily accomplished by adding an irrelevant detail to each output script (such as a randomly-selected number that is pushed on the stack and then popped off the stack), thus making it impractical to guess. A more efficient approach is to use a random nonce (signature "R" value) **[Ant17]** when creating the required signature values in the output scripts of the non-leaf transactions.

5 Factories

5.1 Motivation

As noted above, channels do not solve Bitcoin's scalability problem, as they require one or more on-chain transactions whenever a new channel is created or the parties in a channel change. One approach

17 Or a specific linear list, which can be viewed as a degenerate form of tree.
18 As a result, signature-value covenants cannot be recursive.

to reducing these costs is to create factories that amortize the on-chain transaction costs over multiple channels.

5.2 Multisig Factories

Factories implemented with multi-signatures were first introduced by Burchert, Decker and Wattenhofer **[BDW18]** and are reviewed briefly below. In order to create a set of channels, the set S of parties involved in those channels work together to fund, update, and settle the channels.

Creation

In order to fund the channels, a funding transaction with a multisig output signed by everyone in S is created with the output going to a tree of pre-signed transactions that lead to the channels themselves. In order to guarantee that each party can be sure that their channel(s) can eventually be put on-chain if necessary, the following rule must be followed:

Multisig Factory Rule: If P is a party to a channel C, then each link on the path from the funding transaction to C must require P's signature **[BDW18]**.

Given this rule, if P does not sign any transaction that competes with those along the path to C, then P is assured that no competing transaction can be put on-chain, and thus the transaction creating C can be put on-chain. In order to avoid the need for a refund transaction, all transactions leading to and including the channels must be fully-signed before the funding transaction is signed. An example of a tree-structured multisig factory is given in Figure 4.

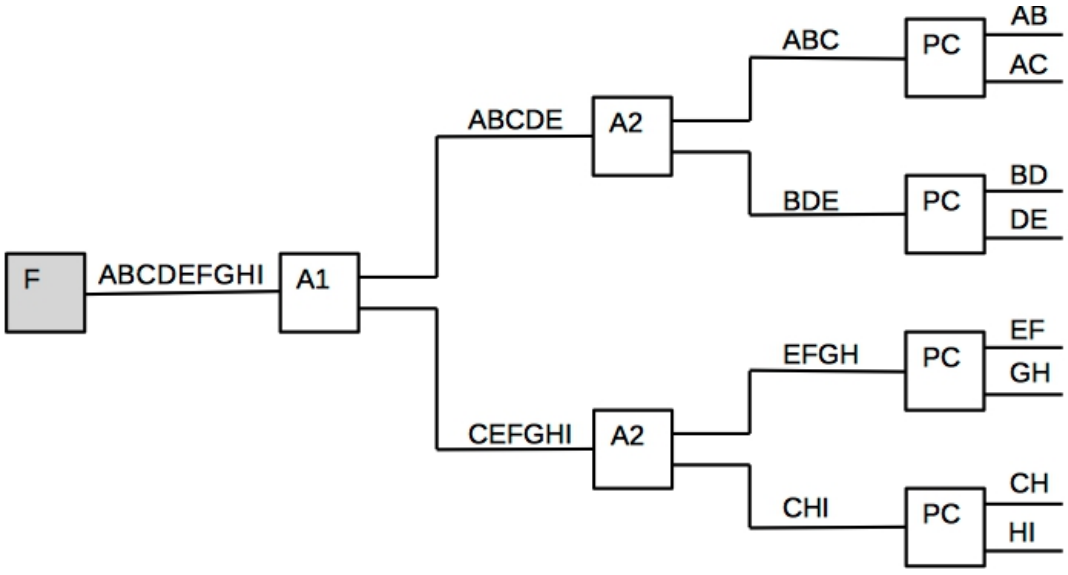

Figure 4: Tree-Structured Multisig Factory. The output of the funding transaction (F) leads to a first-level Allocation transaction (A1) and then to a pair of second-level Allocation transactions (A2) and to Payment Channel transactions (PC), each of which has a pair of channel outputs. The signatures required on each output are shown above the output.

Updates

If the parties in a factory want to create a new set of channels, they can create and sign new first-level and second-level Allocation transactions off-chain. However, in order to guarantee that these new Allocation transactions can be put on-chain if necessary, they also have to invalidate the previous Allocation transactions. This can be accomplished by using an invalidation tree to update the factory state, where the state is represented by the first-level Allocation transaction. Of course, all parties to the factory must sign the new first-level Allocation transaction, as that is required for any child of the Funding transaction.

Alternatively, if floating transactions were supported, it would be possible to use eltoo to update the factory from one first-level Allocation transaction to the next **[DRO18]**.

Settlement

If the parties want to cooperatively close a factory, they can all sign a close transaction that competes with the first-level Allocation transaction and that provides the final output amounts to each party.

Analysis

The number of on-chain first-level Allocation transactions required to support N factory states is given in Table 1 (Section 3.4) for both current Bitcoin (using invalidation trees) and Bitcoin with floating transactions (using eltoo). These results show that factories are promising and could produce a significant savings in on-chain transactions if implemented using a sufficiently-large set of parties S.

The real challenges in using multisig factories come from the following two limitations:

Multisig Factory Limitation 1: The parties in a multisig factory must solve a group coordination problem (see Section 2.2) in order to create or update the factory.

Multisig Factory Limitation 2: No new parties can be added to an existing multisig factory without putting a new funding transaction on-chain.

Both of these limitations are fundamental, as they follow directly from the Multisig Factory Rule. The actual impact of these limitations depends on the capabilities and motivations of the parties. Because of these limitations, multisig factories can help Bitcoin scale, but they cannot fully solve the Bitcoin scaling problem[19].

The problem with using signatures to secure off-chain transactions is that, while those who sign the transactions know they have not signed competing transactions, they cannot prove to others that they have not signed (and will not sign) competing transactions. The following subsection shows how covenants can be used to overcome these limitations.

19 A partial solution could be to use a tree-structured set of allocation transactions (as in Figure 4) and to allow updates of subtrees (using the invalidation tree or eltoo protocol) by the parties with channels in the subtree when not everyone in the factory cooperates.

5.3 Timeout-Trees

This subsection introduces a factory protocol, called *timeout-trees*, that relies on covenants. As was described in Section 4, either floating transactions or IIDs can be used to create the required covenants. One party in the timeout-tree protocol is designated as the *operator* and the remaining parties are designated as *users*. Users are divided into *funding users* and *non-funding users*. Each timeout-tree has a limited lifetime that is determined by its *factory_timeout* parameter.

Creation

The operator creates a tree-structured timeout-tree factory as shown in Figure 5.

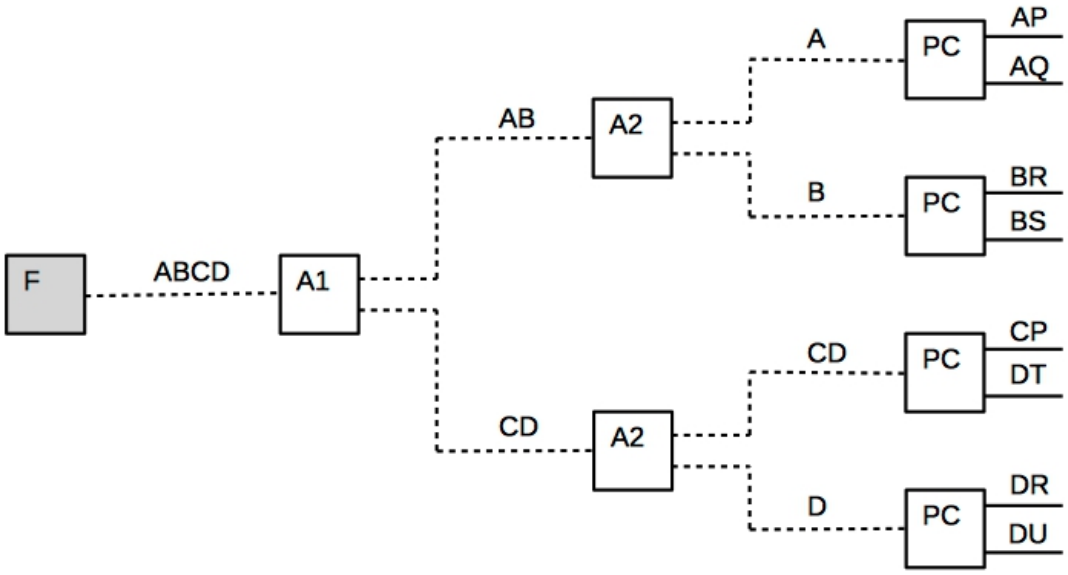

Figure 5: Timeout-Tree Factory. The output of the funding transaction (F) leads to a first-level Allocation transaction (A1) and then to a pair of second-level Allocation transactions (A2) and to Payment Channel transactions (PC), each of which has a pair of channel outputs. Outputs that are dashed carry covenants that restrict the form of the child transaction until the tree's timeout is reached, at which point the output can be spent in any way given the signatures of the party (or parties) associated with the output. Payment Channel outputs do not have to carry covenants, but they do require two-party multisigs as shown.

The overall structure of the factory is similar to that of the multisig factory shown in Figure 4. In Figure 5, the funding users (A, B, C and D) fund a factory that has eight channels, each of which is funded by one of the funding users and has a non-funding user (P, Q, R, S, T or U) as the other party in the channel. As in the multisig factory, each link on the path from the funding transaction to a given channel has an associated pubkey that includes the pubkey of the user funding the given channel.

However, unlike in the multisig factory, all of the links from the funding transaction to each payment channel transaction also carry covenants that constrain the child transactions. Each of these links has a two-case output script, where one case (which does not require waiting until the factory's timeout has been reached) places a covenant on the child transaction that enforces the tree structure shown in Figure 5, and the other case (which requires that the factory's timeout has been reached) allows the output to be spent by any child transaction that is signed with the link's pubkey. The amounts and output scripts associated with these links, and with the channel outputs, are as follows:

Funding or allocation transaction output amount: sum of $Value_C$ for all channels C reachable from this output

Funding or allocation transaction output script (referenced via IID or floating transaction):

OP_IF child transaction in covenant tree

> **<1 time window> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 time window from when parent transaction was put on-chain in order to allow the "else" case sufficient time
>
> **<signature creating an output-only covenant on child that creates the subtree with the desired set of channels>**
>
> **OP_CODESEPARATOR**
>
> **<standard pubkey with widely-known "private key"> OP_CHECKSIG** implement covenant

OP_ELSE factory timeout reached, so allow any child transaction that is signed by all users funding the channels that are reachable from this output

> **<factory_timeout> OP_CHECKLOCKTIMEVERIFY OP_DROP** delay until factory_timeout
>
> **OP_DUP <pubkey funding_set> OP_EQUALVERIFY OP_CHECKSIG** check signature for set of users funding the channels that are reachable from this output

OP_ENDIF

Payment channel C output amount: $Value_C$ for channel C implemented by this output

Payment channel C output script:

OP_IF channel C owned by funding user F_C and non-funding user N_C

> **<1 time window> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 time window from when parent transaction was put on-chain in order to allow the "else" case sufficient time
>
> **OP_DUP <pubkey F_C&N_C> OP_EQUALVERIFY OP_CHECKSIG** check signature for user that funded this channel (F_C) and for other party in channel (N_C)

OP_ELSE factory timeout reached, so allow any child transaction that is signed by the user funding this channel

<factory_timeout> OP_CHECKLOCKTIMEVERIFY OP_DROP delay until factory_timeout

OP_DUP <pubkey F_C> OP_EQUALVERIFY OP_CHECKSIG check signature for user that funded this channel (F_C)

OP_ENDIF

In order to create a timeout-tree factory, the operator receives messages from the funding users that specify the channels they would like created, including the amount of each channel and the multisig pubkey associated with the channel. Given these messages, the operator calculates the transactions that form a timeout-tree factory with the requested channels, including the transaction F that funds the factory. The operator then sends a message to each funding user that includes F and the transactions on the path to each of the funding user's channels. Because F has inputs that receive funds from the funding users, the operator must obtain a signature on F from each funding user before putting F on-chain. If the operator fails to obtain all of the required signatures promptly, the operator proposes a new timeout-tree factory with a new funding transaction and repeats this process until the proposed funding transaction can be put on-chain.

Updates

Timeout-trees do not support off-chain updates. In order to update a timeout-tree (for example to change the value of a channel or the parties owning the channel), it must be closed and a new timeout-tree created. In the best case, the closing of the old timeout-tree and the creation of the new timeout-tree consists of a single on-chain transaction. This single transaction, which funds the new tree, is put on-chain after the timeout of the old tree and it spends the old tree's funding transaction output.

Operation and Settlement

Once a timeout-tree is created by putting its funding transaction on-chain, each funding user sends a message to each of the non-funding users for whom they have created a channel. The message includes F and the set of (currently off-chain) transactions on the path from F to the non-funding user's channel. The two parties in this channel then use a channel protocol[20] to update the channel from its initial state in which all of the channel funds are owned by the funder. Note that the funder of the channel did not have to obtain a signed refund transaction for the channel, as the timeout on the channel provides the required refund guarantee.

At any time prior to the factory's timeout, a user can choose to put on-chain the transactions that lead from F to the user's channel, followed by the transactions that settle the channel at its current state.

Alternatively, a non-funding user can draw down their balance in their channel so that all of the channel's funds are owned by the funding user by the time the factory times out. In order to allow non-funding users to withdraw their funds, the non-funding users can maintain channels in multiple

20 Such as 2Stage or eltoo, depending on whether IIDs or floating transactions are supported in Bitcoin.

timeout-trees with staggered timeouts. A channel network (such as the Lightning Network) can be used to move funds away from channels that are nearing their timeout to channels that are far from their timeout, all without requiring any on-chain transactions.

Once a timeout-tree factory has reached its timeout, all of the factory's outputs that have not been spent are available to be spent by the users funding the output. These outputs can now be used to fund a new timeout-tree. Note that the "if" case of each output script is still in effect after the factory's timeout. As a result, if it is impossible to obtain all of the required signatures for a given on-chain output, it is possible to put the child transaction on-chain that meets the covenant specified in the output's "if" case. This process can be repeated as necessary until a descendant is put on-chain for which the required signatures can be obtained for funding a new timeout-tree. Because the "if" case of each output script includes a relative delay of one time window, the new timeout-tree can be put on-chain before the "if" case of the output script can be used to put the competing child transaction on-chain.

Payment Trees

When a user puts a channel on-chain before a timeout-tree factory times out, they put all of the transactions leading to the channel on-chain. As a result, they increase the number of on-chain outputs that must be included in the next timeout-tree, thus increasing the cost of creating that timeout-tree. It would be preferable if users had a way of signaling that they would like to remove their funds from all timeout-trees without increasing the costs for others. This can be accomplished with *payment trees*.

A payment tree is identical to a timeout-tree, except:

1. the pubkeys required for the "else" (timeout) case for the funding, allocation and payment channel outputs are those of the non-funding (rather than the funding) users, and
2. before the funding transaction for a payment tree is created, for each channel the funding user obtains the non-funding user's signature on a refund transaction that pays the entire channel to the funder.

Note that the second change above means that every party in the payment tree must provide a signature to participate (the funding parties' signatures are required for the funding transaction, and the non-funding parties' signatures are required for the refund transactions). However, this does not limit the scalability of payment trees as compared to timeout-trees, as the protocol does not require all of the parties to solve a group coordination problem. Specifically, the operator can propose the desired payment tree, the funders can attempt to get non-funders' signatures, and then the operator can update the payment tree by dropping any channels for which non-funders' signatures are missing[21].

If all of the channels in a payment tree use a channel network to move all of the channel's funds to the non-funding user before the payment tree times out, the payment tree gives the tree's funds to the non-

21 If IIDs are used, the dropping of channels that lack signatures cannot be allowed to change the IIDs of the remaining channels. This can be accomplished by replacing a dropped channel with a dummy output.

funding users in a tree that is off-chain (except for its root) and that allows the owners of each subtree to spend the subtree's funds as they wish.

The fact that users can obtain their funds in a payment tree without having to pay the fees to put their channel on-chain should motivate them to use the payment tree option. In addition, users could be encouraged to obtain payouts via payment trees (rather than timeout-trees) by offering them a slightly more attractive state in the new payout tree (for example by having a shorter timelock to get their funds or by having a slightly greater amount of funds when they are withdrawn).

Analysis

The timeout-tree factories are limited in comparison to the multisig factories presented above, as they cannot be updated entirely off-chain. As a result, timeout-trees complement, but do not replace, multisig factories. However, timeout-trees do appear to solve many of Bitcoin's scaling problems, including the efficient creation of channels for large numbers of casual users.

The requirement to obtain signatures from all funding parties is an example of the group coordination problem and is the main limit to the size of a multisig or timeout-tree factory. However, because the timeout-tree factory can include channels for non-funding users without requiring their signatures, it can support a far larger number of users and channels. In particular, the funding users of a timeout-tree factory could be dedicated users, each of whom charges a small fee from thousands of casual non-funding users for the creation of a channel that they can use. As a result, timeout-tree factories could achieve a dramatic increase in users and channels over multisig factories.

In addition, if the channels in a timeout-tree use the watchtower-free 2Stage protocol, users would not need to implement watchtower functionality. The only functionality required of non-funding users (in addition to the ability to operate a watchtower-free channel) is that they migrate their funds to channels in different timeout-trees before each such tree times out (or settle the channel on-chain if they are unable to do so).

6 Update-Forest Factories

6.1 Overview

As was shown above, the scalability of multisig and timeout-tree factories is impacted by the number of parties that can solve the group coordination problem of signing the factory's funding transaction. The update-forest protocol eliminates this group coordination problem by separating the funding of the factory from the updating of the state (and ownership) of the channels in the factory.

An update-forest factory goes through *eras*, where the update from one era to the next occurs at a previously-defined time (such as once per week or once per month). Like the timeout-tree protocol, the update-forest protocol uses an operator. The operator creates a sequence of update trees (called an update-forest), with update tree X controlling the era-X settlement for all of the channels in the factory.

Specifically, for each channel L in the factory, update tree X has a leaf U_{LX} with an output that places a covenant on the era-X settlement transaction S_{LX} for channel L.

The funding of the factory is done in a distributed manner, with each channel being an output of an on-chain transaction, or of an off-chain transaction that is a leaf of a covenant tree with an on-chain root. An arbitrary number of channels, funded from an arbitrary number of on-chain transactions or covenant tree leaves, can be included in a single update-forest factory. In order to bind the funding of a channel to the update-forest that controls the channel, the output that funds the channel places a covenant on any settlement transaction that spends the output. Crucially, this covenant identifies the update-forest to which it is bound via the IIDs of the leaves of the update-forest.

The overall structure of the update-forest protocol is shown in Figure 6, which shows the status of the protocol after the operator has committed to the updates for era X. This is followed by a specification of the output amounts and output scripts required by the protocol. The manner in which users communicate with the operator is then given, followed by a detailed description and analysis of the protocol. The correctness of the protocol is proven in Appendix C.

The update-forest protocol uses the following parameters:

- **max_eras:** the maximum number of eras supported by the protocol, and
- **era:** delay parameter giving the time between era updates.

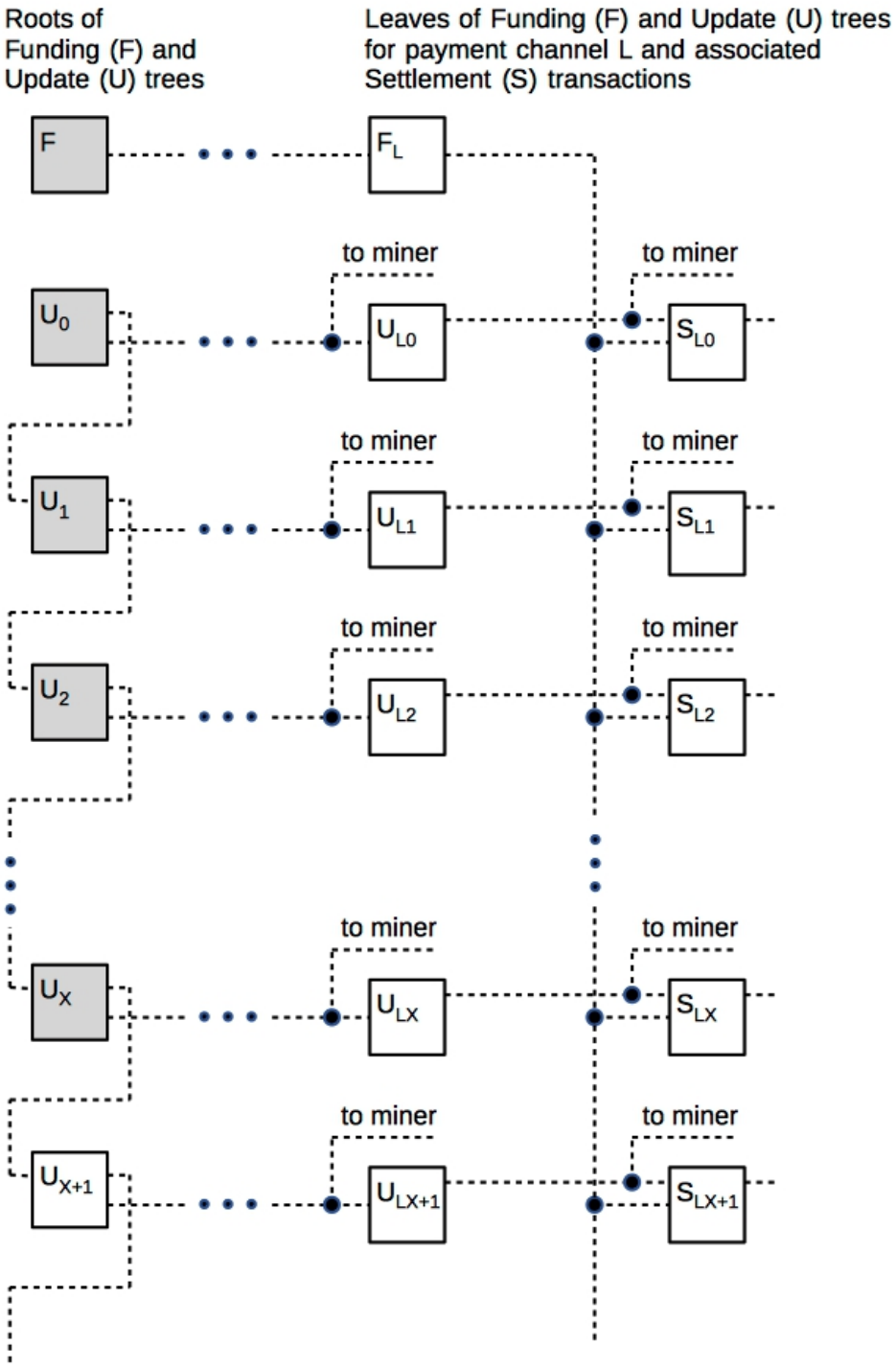

Figure 6: The Update-Forest Protocol. A representative funding covenant tree is rooted at F. Its Leaf F_L holds the value for channel L. The Update tree for era X is rooted at U_X. Its Leaf U_{LX} controls the update of channel L to era X by placing a covenant on channel L's era-X Settlement (S_{LX}) transaction. Each output in the era-X update tree has a timeout that allows it to be claimed by a miner before era X+1 begins, thus ensuring that era-X updates to channel L expire before era X+1 begins (unless channel L is settled in era X by putting S_{LX} on-chain).

6.2 Output Amounts and Output Scripts

Output amounts, output scripts and other fields for the transactions shown in Figure 6 are given below.

Funding transaction F output 0 amount: sum of $Value_L$ for all leaves L in the given funding tree

Funding transaction F output 0 script (referenced via IID):

<signature placing an output-only covenant on child that creates the desired funding tree with leaves F_L>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Funding transaction F_L output 0 amount: $Value_L$ which is the value of payment channel L

Funding transaction F_L output 0 script (referenced via IID):

[separate case for X, where $0 \le X \le max\_eras$][22]

> **<start + (X+1) eras>** **OP_CHECKLOCKTIMEVERIFY** **OP_DROP** delay X+1 eras from protocol start time
>
> **<signature placing an input-only covenant on S_{LX} that forces its input 0 to be U_{LX} output 0>**[23]

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Update transaction U_X nLocktime value: start + X eras

Update transaction U_X output 0 amount: minimum viable amount (output is for control only)

Update transaction U_X output 0 script (referenced via IID):

<start + (X+1) eras> **OP_CHECKLOCKTIMEVERIFY** **OP_DROP** delay X+1 eras from protocol start time

OP_DUP **<pubkey Operator>** **OP_EQUALVERIFY** **OP_CHECKSIGVERIFY** check operator's signature

<signature placing a single-output covenant on U_{X+1} forcing its output 0 to be as shown>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

22 These separate cases can be implemented efficiently via separate Taproot leaf scripts.

23 Of course U_{LX} is referenced via its IID. For example, for any $X > 0$, U_{LX} is a descendent of U_X with the path of descent depending on L, and U_X is U_0.0...0 (with X ".0" terms).

Update transaction U_X output 1 amount: minimum viable amount (output is for control only)

Update transaction U_X output 1 script (referenced via IID):

OP_IF child transaction in update tree

> **<1 time window> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 time window from when parent transaction was put on-chain in order to allow the "else" case sufficient time
>
> **<signature placing an output-only covenant on child that creates the desired update tree with leaves U_{LX} as desired>**
>
> **OP_CODESEPARATOR**
>
> **<standard pubkey with widely-known "private key"> OP_CHECKSIG** implement covenant

OP_ELSE timeout paid to miner mining block containing child transaction

> **<start + (X+2) eras - 2 time windows> OP_CHECKLOCKTIMEVERIFY OP_DROP** delay X+2 eras minus 2 time windows from protocol start time
>
> **OP_TRUE** allow empty input stack to satisfy output script, so any miner can create a transaction spending this output if it is at least X+2 eras minus 2 time windows after the protocol start time

OP_ENDIF

Update transaction U_{LX} output 0 amount: minimum viable amount (output is for control only)

Update transaction U_{LX} output 0 script (referenced via IID):

OP_IF settlement transaction S_{LX}

> **<1 time window> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 time window from when parent transaction was put on-chain in order to allow the "else" case sufficient time
>
> **<signature placing an input-output covenant on S_{LX} forcing its input 1 to be F_L output 0 and its outputs to be as shown>**
>
> **OP_CODESEPARATOR**
>
> **<standard pubkey with widely-known "private key"> OP_CHECKSIG** implement covenant

OP_ELSE timeout paid to miner mining block containing child transaction

> **<start + (X+2) eras - 2 time windows> OP_CHECKLOCKTIMEVERIFY OP_DROP** delay X+2 eras minus 2 time windows from protocol start time
>
> **OP_TRUE** allow empty input stack to satisfy output script, so any miner can create a transaction spending this output if it is at least X+2 eras minus 2 time windows after the protocol start time

OP_ENDIF

Settlement transaction S_{LX} nLocktime value: start + (X+1) eras

Settlement transaction S_{LX} output 0 amount: $Value_L$ which is the value of payment channel L

Settlement transaction S_{LX} output 0 script (referenced via IID):

OP_DUP <pubkey $Owner_{LX}$> OP_EQUALVERIFY OP_CHECKSIGVERIFY check era-X owner's signature

<signature placing covenant on child as specified by U_{LX} for payment channel L in era X>[24]

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

6.3 Communication Between Users and the Operator

Because the ownership of a channel can change during the course of the protocol, the operator needs to have a means for verifying the authenticity of requests to update the channel. For each channel L and era X, there is a unique owner $Owner_{LX}$ who holds the private key for the public key "pubkey $Owner_{LX}$"[25]. In order to transition channel L from era X to era X+1, $Owner_{LX}$ sends a message to the operator that is signed with "pubkey $Owner_{LX}$" which specifies:

1. the era-(X+1) public key "pubkey $Owner_{LX+1}$", and
2. the desired era-(X+1) U_{LX+1} transaction.

6.4 Creation

The funding transaction F is signed and put on-chain as described in Section 5.3, except the era-0 update root transaction U_0 which gives the initial state of each channel must be put on-chain before any parties sign their funding transaction F[26].

6.5 Updates

In order to update channel L from era X to era X+1, $Owner_{LX}$ first calculates the desired era-(X+1) settlement S_{LX+1} transaction, and from this calculates the desired update transaction U_{LX+1}. Next, $Owner_{LX}$ sends the operator a signed message with both items listed in Section 6.3.

The operator then works backwards from the desired leaf transactions U_{LX+1} to calculate the covenant on output 1 of update root transaction U_{X+1} and puts U_{X+1} on-chain as soon as its nLocktime value has

24 The covenant created by this line and the remaining lines of the output script are as specified by U_{LX} and may be omitted by U_{LX} if the owner in era X does not want such a covenant.

25 This public key is a Schnorr public key **[WNR20]**, so it could represent a single user's public key, or a combination of two or more users' public keys.

26 For simplicity, the description given here assumes each channel is initialized by the era-0 update tree. In reality, a channel could be initialized with an era-X update tree for any value of X.

been reached. The operator then sends a message to $Owner_{LX}$ containing the series of transactions leading from update root U_{X+1} to update leaf U_{LX+1} so the owner can verify that the update is correct.

$Owner_{LX}$ then validates this series of transactions and that U_{LX+1} is as desired. If so, $Owner_{LX}$ sends a message to $Owner_{LX+1}$ containing the transactions leading from update root U_X to settlement S_{LX} and from update root U_{X+1} to settlement S_{LX+1}. Once $Owner_{LX+1}$ receives and validates the contents of this message, $Owner_{LX+1}$ waits until X+2 eras after the protocol start time, at which point either S_{LX} is on-chain or the path leading from U_X to S_{LX} cannot be put on-chain because a necessary output along that path has been claimed by a miner[27]. At this point, if S_{LX} is not on-chain, $Owner_{LX+1}$ can guarantee settlement as specified by S_{LX+1}, so channel L has logically transitioned to era X+1.

6.6 Settlement

If the operator does not promptly put U_{X+1} on-chain and provide $Owner_{LX}$ with the series of transactions leading from U_{X+1} to U_{LX+1}, or if U_{LX+1} is not as desired, $Owner_{LX}$ puts the series of transactions leading from U_X to S_{LX} on-chain as quickly as possible.

6.7 Setting the Era Parameter

The update-forest protocol uses one delay parameter, namely *era*. For correctness, the era parameter must be large enough to guarantee that S_{LX} can be put on-chain, using the settlement procedure specified in Section 6.6, by (start + (X+2) eras - 2 time windows) at the latest. Therefore, the era parameter must include time for 1) detecting that the era-(X+1) update for payment channel L is late and/or incorrect, and 2) putting the series of transactions leading from U_X to S_{LX} on-chain. Because U_{X+1}, which defines the era-(X+1) update for channel L, cannot be put on-chain until (start + (X+1) eras), it follows that 1 era minus 2 time windows is at least the time required for items 1) and 2) above.

In setting the era parameter, it should be noted that the operator (or a pool of colluding operators) could create a late or incorrect update for all leaves in the update tree (or the set of update trees controlled by the colluding operators), thus creating a mass exit **[GMRMG]** and a much-larger-than-typical amount of congestion for the blockchain. As a result, the era parameter needs to be defined very conservatively.

6.8 Analysis

Let N denote the number of channels supported and let M denote the maximum number of eras supported. The operator puts M update transactions on-chain in order to update all N channels M times, with each update assigning a new owner and a new covenant to each channel. Settling a single channel in some era X requires O(X + log N) on-chain transactions, while settling all N channels in the same era X requires O(X + N) on-chain transactions. Each update of a channel to a new era requires communication, verification and storage of O(log N) off-chain transactions. Finally, note that the update-forest protocol does not require watchtowers.

27 Given the assumption that the fees in the update tree are sufficient to motivate the miners to claim them, and that 1 time window is sufficient time in which to do so.

7 Challenge-and-Response Factories

7.1 Overview

In the update-forest protocol, each era must be long enough to allow an owner of a channel in that era to determine if the proposed update to the next era is correct (and prompt), and if it is not, to settle the channel in the current era before the next era begins. This places a lower bound on the length of an era, and thus on the rate at which channel ownership can be changed. While such a bound is likely to be suitable for many users, others may need faster updates. The challenge-and-response protocol provides such fast updates. It also introduces techniques for combining IIDs and signatures that could be of interest in other settings.

As in the update-forest protocol, the operator creates a sequence of update trees, one for each era. In order to support fast updates, the challenge-and-response protocol allows multiple update trees to be viable at the same time. As a result, in order to pass ownership of a channel L from era X to era X+1, the era-X owner needs a way to invalidate the era-X update to L without waiting for it to timeout. This is accomplished by adding a pair of Challenge and Response transactions for each channel and era.

The output of an era-X Challenge transaction C_{LX} can be spent by either an era-(X+1) Response transaction R_{LX+1} (if it is signed by the owner from era X, thus proving that the era-X owner has given up ownership of the channel) or by an era-X Settlement transaction S_{LX}. Thus, the era-X owner can invalidate the era-X update by signing the era-(X+1) Response transaction.

There is a delay in putting S_{LX} on-chain which gives the era-(X+1) owner time to put R_{LX+1} on-chain (if R_{LX+1} has been signed by the era-X owner). Therefore, the ability to put S_{LX} on-chain shows the lack of an appropriately-signed R_{LX+1}, thus demonstrating that X is the final era for channel L.

The overall structure of the challenge-and-response protocol is given in Figure 7, which shows the status of the protocol after the operator has committed to the updates for era X. This is followed by a specification of the output amounts, output scripts and other fields required by the protocol. The manner in which users communicate with the operator is then given, followed by a detailed description and analysis of the protocol. The correctness of the protocol is proven in Appendix C.

The challenge-and-response protocol uses the following parameters:

- **bond:** amount of funds required to put a challenge transaction on-chain,
- **max_eras:** maximum number of eras supported by the protocol
- **era:** delay parameter giving the time between era updates,
- **response_delay:** delay parameter specifying the time provided for a response to a challenge, and
- **timeout_delay:** delay parameter specifying the time after which unclaimed bonds go to miners.

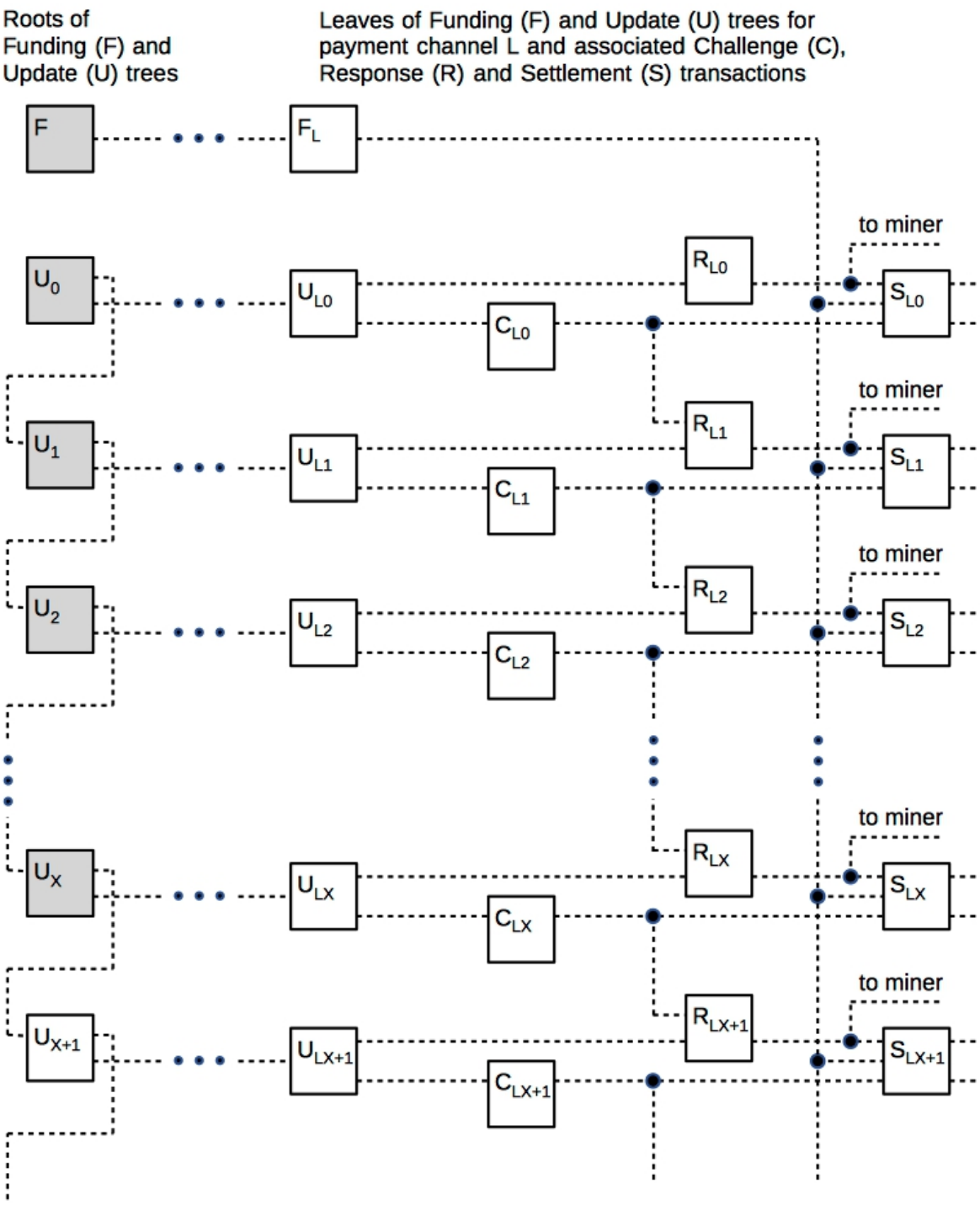

Figure 7: The Challenge-and-Response Protocol. A representative funding covenant tree is rooted at F. Its Leaf F_L holds the value for channel L. The Update tree for era X is rooted at U_X. Its Leaf U_{LX} controls the update of channel L to era X. In order to put the era-X Settlement (S_{LX}) transaction on-chain and pay out channel L's funds as specified by U_{LX}, it must be shown that there is no era-(X+1) Response transaction R_{LX+1} that can be put on-chain, thus demonstrating that the era-X owner has not given up ownership by signing R_{LX+1}. If no such R_{LX+1} exists, X must be the final era for channel L.

7.2 Output Amounts and Output Scripts

Output amounts, output scripts and other fields for the transactions shown in Figure 7 are given below.

Funding transaction F output 0 amount: sum of $Value_L$ for all leaves L in the given funding tree

Funding transaction F output 0 script (referenced via IID):

<signature placing an output-only covenant on child that creates the desired funding tree with leaves F_L>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Funding transaction F_L output 0 amount: $Value_L$ which is the value of payment channel L

Funding transaction F_L output 0 script (referenced via IID):

[separate case for X, where $0 \le X \le max\_eras$][28]

> **<start + (X+1) eras + 2 response_delays>** **OP_CHECKLOCKTIMEVERIFY** **OP_DROP** delay X+1 eras and 2 response_delays from protocol start time
>
> **<signature placing an input-only covenant on S_{LX} that forces its input 0 to be R_{LX} output 0 and its input 2 to be C_{LX} output 0>**[29]

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Update transaction U_X nLocktime value: start + X eras

Update transaction U_X output 0 amount: minimum viable amount (output is for control only)

Update transaction U_X output 0 script (referenced via IID):

<start + (X+1) eras> **OP_CHECKLOCKTIMEVERIFY** **OP_DROP** delay X+1 eras from protocol start time

OP_DUP **<pubkey Operator>** **OP_EQUALVERIFY** **OP_CHECKSIGVERIFY** check operator's signature

<signature placing a single-output covenant on U_{X+1} forcing its output 0 to be as shown>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

---

28 As in the update-forest protocol, these separate cases can be implemented efficiently via separate Taproot leaf scripts.
29 As in the update-forest protocol, R_{LX} and C_{LX} are referenced via their IIDs.

Update transaction U_X output 1 amount: minimum viable amount (output is for control only)

Update transaction U_X output 1 script (referenced via IID):

<signature placing an output-only covenant on child that creates the desired update tree with leaves U_{LX} as desired>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Update transaction U_{LX} output 0 amount: minimum viable amount (output is for control only)

Update transaction U_{LX} output 0 script (referenced via IID):

<signature placing an input-output covenant on R_{LX} forcing its input 0 to be C_{LX-1} output 0 and its outputs to be as shown>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Update transaction U_{LX} output 1 amount: minimum viable amount (output is for control only)

Update transaction U_{LX} output 1 script (referenced via IID):

<signature placing an output-only covenant on C_{LX} forcing its output to be as shown>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> **OP_CHECKSIG** implement covenant

Response transaction R_{LX} output 0 amount: 1 bond

Response transaction R_{LX} output 0 script (referenced via IID):

OP_IF settlement transaction S_{LX}

> **<signature placing an input-output covenant on S_{LX} forcing its input 1 to be F_L output 0, its input 2 to be C_{LX} output 0 and its output to be as shown>**
>
> **OP_CODESEPARATOR**
>
> **<standard pubkey with widely-known "private key">** **OP_CHECKSIG** implement covenant

OP_ELSE timeout paid to miner mining block containing child transaction

> **<1 timeout_delay>** **OP_CHECKSEQUENCEVERIFY** **OP_DROP** wait 1 timeout_delay
>
> **<TRUE>** allow empty input stack to satisfy output script, so any miner can create a transaction spending this output if it has been on-chain for 1 timeout_delay

OP_ENDIF

Challenge transaction C_{LX} output 0 amount: 1 bond

Challenge transaction C_{LX} output 0 script (referenced via IID):

OP_IF response transaction R_{LX+1}

> **OP_DUP <pubkey Owner$_{LX}$> OP_EQUALVERIFY OP_CHECKSIGVERIFY** check era-X owner's signature
>
> **<signature placing an input-only covenant on R_{LX+1} forcing its input 1 to be U_{LX+1} output 0>**

OP_ELSE settlement transaction S_{LX}

> **<1 era + 1 response_delay> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 era + 1 response_delay
>
> **<signature placing an input-only covenant on S_{LX} forcing its input 0 to be R_{LX} output 0 and its input 1 to be F_L output 0>**

OP_ENDIF

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

Settlement transaction S_{LX} nLocktime value: start + (X+1) eras + 2 response_delays

Settlement transaction S_{LX} input 2 nSequence value: 1 era + 1 response_delay

Settlement transaction S_{LX} output 0 amount: $Value_L$ which is the value of channel L

Settlement transaction S_{LX} output 0 script (referenced via IID):

OP_DUP <pubkey Owner$_{LX}$> OP_EQUALVERIFY OP_CHECKSIGVERIFY check era-X owner's signature

<signature creating covenant on child as specified by U_{LX} for payment channel L in era X>[30]

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

Settlement transaction S_{LX} output 1 amount: 2 bonds

Settlement transaction S_{LX} output 1 script (referenced via IID):

OP_DUP <pubkey Owner$_{LX}$> OP_EQUALVERIFY OP_CHECKSIG check era-X owner's signature

30 The covenant created by this line and the remaining lines of the output script are as specified by U_{LX} and may be omitted by U_{LX} if the owner in era X does not want such a covenant.

7.3 Communication Between Users and the Operator

The protocol for communication between users and the operator matches that of the update-forest protocol given in Section 6.3.

7.4 Creation

Funding and creation are the same as is given for the update-forest protocol in Section 6.4.

7.5 Updates

In order to update channel L from era X to era X+1, $Owner_{LX}$ first calculates the desired era-(X+1) settlement S_{LX+1}, response R_{LX+1} and challenge C_{LX+1} transactions, and from these calculates the desired update transaction U_{LX+1}. Next, $Owner_{LX}$ sends the operator a signed message with both items listed in Section 6.3.

The operator then works backwards from the desired leaf transactions U_{LX+1} to calculate the covenant on output 1 of update root transaction U_{X+1} and puts U_{X+1} on-chain as soon as its nLocktime value has been reached. The operator then sends a message to $Owner_{LX}$ containing the series of transactions leading from update root U_{X+1} to update leaf U_{LX+1} so the owner can verify that the update is correct.

$Owner_{LX}$ then validates this series of transactions and that U_{LX+1} is as desired. If so, $Owner_{LX}$ sends a message to $Owner_{LX+1}$ containing the transactions leading from update root U_W to settlement S_{LW}, including the $Owner_{LW-1}$ signature for R_{LW}, for all W, $0 \leq W \leq X+1$[31]. Once $Owner_{LX+1}$ receives and validates the contents of this message, $Owner_{LX+1}$ can guarantee settlement as specified by S_{LX+1}, so payment channel L has logically transitioned to era X+1.

7.6 Settlement

If the operator does not promptly put U_{X+1} on-chain and provide $Owner_{LX}$ with the series of transactions leading from update root U_{X+1} to update leaf U_{LX+1}, or if U_{LX+1} is not as desired by $Owner_{LX}$, $Owner_{LX}$ must settle the channel as quickly as possible. To do this, $Owner_{LX}$ does the following:

1. puts the series of transactions leading from update root U_{X-1} to challenge C_{LX-1} and from update root U_X to challenge C_{LX} on-chain,
2. uses the signature obtained from $Owner_{LX-1}$ to put response R_{LX} on-chain, and
3. waits until the nLocktime value of settlement S_{LX} has been reached and then puts settlement S_{LX} on-chain.

Throughout this process, $Owner_{LX}$ does not provide to any other party the signed transaction R_{LX+1} (or more generally, any signed transaction other than S_{LX} that spends the output of C_{LX}).

31 With the exception that R_{L0} does not need a signature from the previous owner, as there is no previous owner.

In addition to settling the channel when the operator provides a late or incorrect update, the owner must prevent any party from settling the channel in an earlier era. The owner does this by promptly responding to any challenge transactions from earlier eras[32]. Specifically, if any party puts challenge C_{LW} on-chain for any W < X, $Owner_{LX}$ must put response R_{LW+1} on-chain as soon as possible. $Owner_{LX}$ does this by first putting the series of transactions leading from update root U_{W+1} to update leaf U_{LW+1} on-chain, and then using the signature from $Owner_{LW}$ to put response R_{LW+1} on-chain (before any party can put the competing transaction S_{LW} on-chain).

7.7 Setting the Delay Parameters

The challenge-and-response protocol uses 3 delay parameters, namely *era*, *response_delay*, and *timeout_delay*. For correctness, the era parameter must be large enough to guarantee that any party can put a given transaction on-chain within a single era (and thus an era is at least one time window long). A larger value will allow the protocol to operate for a longer time, at the expense of less frequent channel state updates.

The response_delay parameter must be large enough to guarantee that transactions C_{LX} and R_{LX} can be put on-chain, using the settlement procedure specified in Section 7.6, by (start + X eras + 1 response_delay) at the latest. Therefore, response_delay must include time for 1) detecting that the era X+1 update for payment channel L is late and/or incorrect[33], 2) putting the series of transactions leading from update root U_{X-1} through challenge C_{LX-1} and from update root U_X through challenge C_{LX} on-chain, and 3) putting R_{LX} on-chain.

Finally, the timeout_delay parameter should be at least (2 eras + 2 response_delays).

7.8 Analysis

Let N denote the number of channels supported and let M denote the maximum number of eras supported. The operator puts M update transactions on-chain in order to update all N payment channels M times, with each update assigning a new owner and a new covenant to each channel. Settling a single channel in some era X requires O(X + log N) on-chain transactions, while settling all N channels in the same era X requires O(X + N) on-chain transactions. Finally, note that the challenge-and-response protocol does require watchtowers, as it requires a timely response to old challenge C_{LW} transactions.

While the challenge-and-response protocol is quite efficient in terms of on-chain transactions, each update of a channel to a new era X requires communication, verification and storage of O(X*(log N)) off-chain transactions.

If a party puts a challenge transaction C_{LW} on-chain for some prior era W, the current $Owner_{LX}$ of the channel has to put O(log N) transactions on-chain in order to prevent settlement of the channel in era

32 To be precise, the "owner" in the current era X may actually consist of multiple parties using a single Schnorr public key. In this case, any one of the owning parties can ensure that the channel is not settled in an earlier era, and a party that has an advantageous settlement payout in the current era X is motivated to do so.

33 Or for detecting that challenge C_{LW} was put on-chain for some earlier era W < X, as described in Section 7.6.

W. However, the party putting C_{LW} on-chain cannot benefit (even if that party is $Owner_{LW}$), so no self-interested party will do so. Furthermore, even if a malicious party puts C_{LW} on-chain, they must pay 1 bond to put C_{LW} on-chain. $Owner_{LX}$ will then put R_{LW+1} on-chain, thus spending the malicious party's bond. When the output of R_{LW+1} times out, the bond will be claimed by a miner.

On the other hand, when $Owner_{LX}$ pays 1 bond to put each of C_{LX-1} and C_{LX} on-chain in order to settle the channel in era X, $Owner_{LX}$ recovers both bonds by putting S_{LX} on-chain. As a result, the bond amount can be set quite high in order to penalize malicious parties without costing honest users anything other than a temporary expense that is fully reimbursed.

8 Factory Optimizations and Extensions

This section presents optimizations and extensions of the update-forest and challenge-and-response protocols.

8.1 Timeouts

In the challenge-and-response protocol, output 1 of the update root transaction U_X will never be spent if no party attempts to settle a channel in era X. As a result, it would make sense to add a conservative timeout condition to this output that allows it to be spent by the operator (or a miner) in order to reduce the size of the UTXO set. Similar timeouts could be added to the non-root nodes in the update trees. In addition to reducing the UTXO set, timeouts that spend these outputs improve the challenge-and-response protocol by eliminating the possibility of putting challenge transactions on-chain for eras that are one time window older than the timeout value. As a result, transactions for timed out eras would not have to be sent from one owner to the next as part of the channel update protocol.

8.2 Late Determination of Fees

Because the funding tree, update tree, and challenge transactions each have a single input and are defined by output-only or single-output covenants, it is possible to add another input to them immediately before putting them on-chain, thus providing for the late determination of fees as described in Section 4.4.

Response transaction R_{LX} has two inputs and is defined by covenants over the inputs, so it is impossible to add a new input to R_{LX} with the desired fees. However, it is possible to increase the value of one of the inputs (say input 0, which spends from C_{LX-1}) by a large fixed amount that equals the maximum fee that can be paid. Then, the actual fee paid can be adjusted by adding a new output that refunds that portion of the maximum fee that is not needed. Specifically, this new output should be R_{LX} output 0 (and the previously-defined output 0 should now be output 1), the covenant on R_{LX} defined by U_{LX} output 0 should be an input-single-output covenant, and the signature from $Owner_{LX-1}$ should only cover the transaction inputs. A similar approach can be used to allow a late determination of fees for S_{LX} by adding a new output 1 (for refunding unneeded fees), shifting the existing output 1 to output 2,

changing the covenant on S_{LX} defined by R_{LX} to be an input-single-output covenant, and adding a requirement for a signature from $Owner_{LX}$ to the output script of R_{LX}.

8.3 Redundant Operators

Because the successful operation of both protocols depends on the operator, users could choose to contract with multiple operators in a manner that allows a minority of the operators to be non-responsive or malicious without affecting the factory. For example, a factory could use three operators, any two of whom must provide a correct and timely update to a new era for that update to be effective. This can be accomplished by modifying the input-only covenant placed by the funding leaf transaction F_L on the settlement transaction S_{LX} so that there are three possible cases, each of which consists of receiving the required input from a different pair of operators. In addition, the covenants placed by each the three operators' U_{LX} transactions on S_{LX} must be modified to support two cases, each of which specifies the reception of an input from one of the other operators.

If at least two of the operators define correct and timely update transactions U_{LX}, it will be possible to use the update transactions from those operators to put the desired settlement transaction S_{LX} on-chain. Because the correct U_{LX} transactions place input-output covenants on S_{LX}, the output portions of those covenants must agree in order for the covenants to be met. Finally, even if a malicious operator defines an update transaction U_{LX} that does not place a covenant on S_{LX} (or places a covenant on S_{LX} that is too weak), thus allowing the update transaction from the malicious operator to provide input to S_{LX}, the input from the honest operator still guarantees that S_{LX} is correct.

8.4 Keeping a Channel Off-Chain

The max_eras parameter defines the maximum number of eras for which an update-forest or challenge-and-response factory can assign new owners and new covenants to a channel. Thus, after max_eras eras, the final channel owner could choose to put the channel on-chain. However, that is not required, even if a new owner and covenant for the channel is desired[34].

Recall that in each era, the factory places a covenant on each channel (if such a covenant is desired by the owner of the channel). Therefore, the owner of a given channel L in a given factory can have a covenant placed on the channel in the final era defined by that factory, where the covenant specifies how the channel can be updated by a new operator in a new factory (that is, the covenant placed on channel L matches the output 0 script defined for transaction F_L in the description of the update-forest or challenge-and-response protocol). In this way, a new operator and factory can update the given channel for another max_eras eras.

Of course, this process can be iterated an arbitrary number of times. This process is valuable, because it allows the number of calculations that are required to create a factory (which scales with the max_eras parameter) to be independent of the number of eras for which the channel can be updated off-chain.

34 Provided the final update tree in the update-forest protocol is modified to eliminate the "else" case that allows miners to claim outputs after they timeout.

Furthermore, a new operator can be selected for the channel dynamically, just before a given factory reaches its max_eras limit. The only cost of using this process is that each additional factory adds one settlement transaction and O(log N) update, challenge and response transactions that must be put on-chain to eventually settle the channel on-chain.

In fact, most of these additional settlement, update, challenge and response transactions can be eliminated by making a small modification to the original F_L output script. Rather than have that output directly fund the settlement transactions for the original factory, have it fund the settlement transactions of an update-forest *meta-factory*, each of which defines a *meta-era*, with each meta-era placing a covenant that assigns a given factory (and operator) to the channel[35]. By setting each meta-era to equal max_eras eras, a sequence of $S \leq$ max_meta_eras factories can be used to update the ownership and covenant for the channel while adding only O(log N) transactions (rather than O(S*(log N)) transactions) to the cost of settling the channel on-chain after the O(S*max_eras) update root transactions have been put on-chain.

8.5 Extensions

While the update-forest and challenge-and-response factory protocols have generally been presented as supporting the initiation and updating of channels, they are actually much more general. Each factory leaf L has a state that consists of:

- its value, $Value_L$,
- its current owner, $Owner_{LX}$, and
- its current covenant (as defined by S_{LX} output 0).

The value is fixed throughout the life of the factory. However, the owner and covenant can be updated in each era of the protocol. Because the owner is represented by a pubkey that could be owned by 1, 2, or an arbitrary number of parties, and because the covenant is arbitrary (and optional), each factory leaf L in era X could represent:

- a 2-party channel (implemented via the 2Stage protocol),
- an n-party multisig factory (implemented via a Burchert, Decker and Wattenhofer channel factory **[BDW18]** or a binary-representation-tree, as described in Appendix B),
- an operator-managed n-party timeout-tree factory,
- an operator-managed n-party update-forest or challenge-and-response factory (creating a meta-factory, as described in Section 8.4), or
- a single-owner output (with or without a covenant).

35 The protocol for communication between the users and the operator of the update-forest meta-factory must be modified to allow multiple changes of ownership within a single meta-era. Alternatively, if the sequence of factories created by the meta-factory is fixed in advance, the meta-factory could create a fixed update-forest without requiring an operator.

9 Channel Networks

9.1 Channel Network Background

The Lightning Network implements a channel network in which parties that do not share a channel can exchange bitcoin by setting up a path of channels, with successive channels in the path including a common party **[BWHTLC] [DW15] [GMRMG] [MMSKM] [PD16]**. Routing a value V along the path is accomplished by putting V bitcoin[36] into a timelocked contract within each channel, and tying those contracts together with a common secret S **[BWHTLC] [MMSKM] [Poe17] [Poe18]**[37].

In the following, assume L is a channel in the path through the channel network and Alice and Bob are parties in channel L, where Alice is also a party in the previous channel L-1 (or is the original sender) and Bob is also a party in the next channel L+1 (or is the final recipient). Also assume that Alice and Bob create a timelocked contract for sending V bitcoin from Alice to Bob in channel L. The contract has an expiration time such that if the secret S is revealed before expiration, V is paid to Bob, and receiving this payment on-chain forces the disclosure of S to Alice, thus allowing Alice to be reimbursed in channel L-1 (or, if Alice is the original sender, the secret S acts as a receipt that Bob has received the payment). On the other hand, if S is not revealed before expiration, V is returned to Alice.

Payment channel networks typically require that each party in a channel be able to settle the channel on-chain at any time. However, the watchtower-free version of 2Stage channels and the update-forest and challenge-and-response factories limit how quickly channels can be settled on-chain. The changes that enable channel networks to be implemented efficiently with those protocols are given next and are proven correct in Appendix C.

9.2 Off-Chain Channel Network Protocol

Assume that Alice and Bob implement a timelocked contract for transferring value V from Alice to Bob in an off-chain 2Stage channel L that is created by an update-forest or challenge-and-response factory. For simplicity, it will be assumed that the entire value V of channel L is put in the timelocked contract[38] and prior to implementing the timelocked contract, L's settlement transaction pays V to Alice. If L's settlement transaction is put on-chain, the transaction that spends the value V output of the settlement transaction will be called the *payout transaction*.

Each channel L sets its 2 delay parameters:

- off_chain_L = the maximun delay required to use the off-chain 2Stage protocol to update the state of channel L if both parties cooperate, and
- on_chain_L = the maximum delay required to put the payout transaction for L on-chain,

36 Or slightly more than V bitcoin, if the parties implementing the channel network charge fees for their service.
37 Such as the pre-image of a hash function or the discrete log of a point.
38 The case where there is additional value in the channel is handled easily by creating a covenant tree rooted at L's settlement transaction that has a leaf output with value V and other leaf outputs have the remaining value.

and its 3 timeout parameters:

- *early_timeout$_L$*: the latest time by which either 1) S is revealed to Alice, or 2) Bob closes their timelocked contract, assuming all parties in L and successive channels in the path cooperate,
- *mid_timeout$_L$*: the latest time by which either 1) S is revealed to Alice, or 2) Bob closes their timelocked contract, assuming both Alice and Bob cooperate, and
- *late_timeout$_L$*: the latest time by which Alice and Bob resolve their timelocked contract on-chain if Alice and/or Bob does not cooperate,

such that:

- early_timeout$_L$ < mid_timeout$_L$,
- mid_timeout$_L$ + off_chain$_L$ + on_chain$_L$ + 1 time window < late_timeout$_L$,

and if there exists a previous channel L-1 in the chain (that is, if Alice is not the original sender), then

- early_timeout$_{L-1}$ > early_timeout$_L$, and
- mid_timeout$_{L-1}$ > late_timeout$_L$.

Alice and Bob implement the timelocked contract as follows:

1. Alice and Bob use the 2Stage protocol to update L's settlement transaction to pay V to:
    a) Bob if the corresponding nSequence field of the payout transaction > 1 time window, where putting the payout transaction on-chain requires knowledge of S and reveals S to Alice, or
    b) Alice if the nLocktime of the payout transaction > (late_timeout$_L$ - 1 time window).
2. If Bob reveals S to Alice before early_timeout$_L$:
    a) Alice and Bob use 2Stage to update L's settlement transaction to pay V to Bob by (early_timeout$_L$ + off_chain$_L$), and
    b) Alice reveals S to the other party in channel L-1 before early_timeout$_{L-1}$ (unless Alice is the original sender).
3. If Bob closes the timelocked contract for channel L before early_timeout$_L$:
    a) Alice and Bob use 2Stage to update L's settlement transaction to pay V to Alice by (early_timeout$_L$ + off_chain$_L$), and
    b) Alice closes the timelocked contract for channel L-1 before early_timeout$_{L-1}$.
4. If Bob reveals S to Alice after early_timeout$_L$ but before mid_timeout$_L$:
    a) Alice and Bob use 2Stage to update L's settlement transaction to pay V to Bob by (mid_timeout$_L$ + off_chain$_L$), and

b) Alice reveals S to the other party in channel L-1 before $mid\_timeout_{L-1}$ (unless Alice is the original sender).

5. If Bob closes the timelocked contract for channel L after $early\_timeout_L$ but before $mid\_timeout_L$:

   a) Alice and Bob use 2Stage to update L's settlement transaction to pay V to Alice by ($mid\_timeout_L$ + off_chain_L), and

   b) Alice closes the timelocked contract for channel L-1 before $mid\_timeout_{L-1}$ (unless Alice is the original sender).

In addition, Alice and Bob follow these timelocked contract protocol rules:

Rule 1: Step 1 for channel L is only performed after Step 1 for channel L-1 is completed.

Rule 2: If the settlement transaction created in Step 1 is ever on-chain after ($late\_timeout_L$ - 1 time window), Alice attempts to claim the payout as soon as possible.

Rule 3: At any point after Step 1 is completed and before ($mid\_timeout_L$ + off_chain_L), if Alice fails to follow the protocol and if Bob knows S, then Bob puts the settlement transaction created in Step 1 on-chain and obtains the payout from that transaction[39].

Rule 4: If Bob ever receives the payout from the settlement transaction created in Step 1, this forces the disclosure of S to Alice, and Alice immediately discloses S to the other party in channel L-1.

Rule 5: If Bob is the final recipient, Bob either reveals S to Alice by $early\_timeout_L$ or closes the timelocked contract for L by $early\_timeout_L$.

9.3 Analysis

Because neither party can obtain extra funds by failing to follow the protocol, both parties are incentivized to follow the protocol. It is therefore reasonable to believe that in the normal case, all parties and operators for a path of channels will execute the protocol correctly. In this case, the timelocked contract for channel L will either pay out or be closed before ($early\_timeout_L$ + off_chain_L), in which case the amount of time that the funds in the channel are subject to the timelocked contract is independent of the time to enforce that contract on-chain. If a party in a later channel in the path is non-responsive or malicious, the timelocked contract for channel L can still be resolved without putting any transactions on-chain (but the funds will stay locked in the contract for longer).

39 This is possible because ($mid\_timeout_L$ + off_chain_L + on_chain_L + 1 time window) < $late\_timeout_L$, so Bob is able to put the payout transaction on-chain by ($late\_timeout_L$ - 1 time window), which is before Alice can put her competing payout transaction on-chain, given that the nLocktime of that payout transaction is $late\_timeout_L$ - 1 time window.

10 Related Work

The work presented here builds on a wide range of previously published research. At the channel level, the 2Stage protocol builds on the eltoo protocol **[DRO18]** and the BIP 118 proposal **[BIP118]**. In particular, 2Stage borrows eltoo's idea of using nLocktime values in the past to encode and enforce state ordering.

The idea of IIDs is most directly related to a brainstorm by McElrath regarding the ability to use a hash of a parent transaction's inputs, rather than a hash of the entire parent transaction, to identify the transaction when spending its outputs **[McE19]**. IIDs are a natural evolution of that idea, but differ in that 1) IIDs are based on a hash of only a portion of the transaction's inputs (namely the OutPoint of input 0) rather than all of the inputs, thus creating greater flexibility and supporting late addition of fees, 2) the use of IIDs is signaled by the output script in the parent transaction, thus avoiding a potential doubling of the number of UTXOs that needs to be tracked, and 3) IIDs are implemented via a softfork, rather than relying on a change to the definition of the OutPoint field, which would require a hardfork. In addition, the current paper takes the idea further by developing protocols, such as the update-forest and challenge-and-response protocols, that utilize the underlying idea.

The payment trees introduced in Section 5.3 are similar to the tree-based congestion controlled transactions based on OP_CHECKTEMPLATEVERIFY **[RUB21]**, but are designed to keep transactions off-chain rather than delaying them until transaction fees are lower.

The binary-representation-tree channels presented in Appendix B are an adaptation of the invalidation tree channels presented by Decker and Wattenhofer **[DW15]** to a slightly different model.

The idea of covenants was presented by Mösel, Eyal and Sirer **[MES16]** and has been investigated by many other researchers **[McE19] [Rub19] [SHMB20] [Som20] [Tow19]**. Finally, the idea of using an output script that includes a signature value to create a covenant was documented by Towns **[Tow19]** who credited Lau with having described the idea earlier.

11 Conclusions

The Lightning Network is operational and shows that a Layer 2 protocol can scale Bitcoin dramatically while providing nearly-instant bitcoin transfers. The proposed eltoo protocol greatly simplifies the Lightning Network's implementation of channels and improves them by supporting the late determination of fees and eliminating penalty transactions. However, no two-party channel protocol can fully solve Bitcoin's scaling problem, as an on-chain transaction is required to create a channel or to change the parties owning a channel. eltoo supports factories, thus providing for some amortization of the on-chain costs. However, the fact that a non-responsive party blocks the creation of (or updates to) an eltoo factory limits the number of parties than can be supported in practice. In addition, eltoo requires watchtowers and relies on a change to Bitcoin that supports floating transactions.

This paper has shown that a more restrictive change to Bitcoin, namely the use of Inherited IDs (IIDs), is sufficient to create simple and powerful channels and factories. Specifically, the 2Stage channel protocol supports late determination of fees, eliminates watchtowers and penalty transactions, and improves upon eltoo's worst-case performance. 2Stage channels can be implemented with timeout-tree factories to provide efficient watchtower-free channels to a large number of casual users, thus greatly expanding Bitcoin's reach and usability. In addition, IIDs support factory protocols that scale to an unbounded number of parties and channels, and that allow all of the channels in the factory to be bought and sold by anyone (including parties not originally in the factory) with a single on-chain transaction. As a result, one on-chain transaction could replace thousands, or even millions, of Lightning or eltoo on-chain transactions. These highly-scalable factory protocols make critical use of IIDs and do not appear to be possible with floating transactions. Furthermore, because IIDs support covenants, they enable new capabilities such as vaults that allow users to store their bitcoin more securely. Finally, IIDs are safer than floating transactions, as they eliminate the risk that a signed transaction could be used multiple times.

The verification of a transaction that uses IIDs requires slightly more storage and/or computation than the verification of a non-IID transaction (as is described in Appendix A). However, IIDs are compatible with Bitcoin's existing UTXO model, and the slightly increased cost of verifying IID transactions is trivial compared to the scalability (and usability) advantages that they provide. Given these very significant advantages, this paper argues that IIDs, rather than floating transactions, should be added to the Bitcoin protocol.

Acknowledgments

Thanks to Ruben Somsen and Jeremy Rubin for their helpful comments.

Also, thanks to Bob McElrath for his original brainstorm **[McE19]** that led to the creation of the IID concept.

References

Ant17 Andreas Antonopoulos. Mastering Bitcoin, 2nd. ed. 2017. See https://isidore.co/calibre#panel=book_details&book_id=6316.

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.

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

BS Bitcoin Scripts. See https://en.bitcoin.it/wiki/Script#Crypto.

BWHF Bitcoin Wiki: Hardfork. See https://en.bitcoin.it/wiki/Hardfork.

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

BWPC Bitcoin Wiki: Payment Channels. See https://en.bitcoin.it/wiki/Payment_channels.

BWPDTV Bitcoin Wiki: Protocol Documentation - Transaction Verification. See https://en.bitcoin.it/wiki/Protocol_documentation#Transaction_Verification.

BWSF Bitcoin Wiki: Softfork. See https://en.bitcoin.it/wiki/Softfork.

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

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.

LLW15 Eric Lombrozo, Johnson Lau and Pieter Wuille. BIP 141: Segregated Witness. 21 December 2015. See https://en.bitcoin.it/wiki/BIP_0141.

LW16 Johnson Lau and Pieter Wuille. BIP 143: Transaction Signature Verification for Version 0 Witness Program. 3 January 2016. See https://en.bitcoin.it/wiki/BIP_0143.

Max18 Gregory Maxwell. Taproot: Privacy Preserving Switchable Scripting. 23 January 2018. See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-January/015614.html.

McE19 Bob McElrath. On-Chain Defense in Depth. See https://diyhpl.us/wiki/transcripts/2019-02-09-mcelrath-on-chain-defense-in-depth/.

MES16 Malte Möser, Ittay Eyal and Emin Gün Sirer. Bitcoin Covenants. In Financial Cryptography and Data Security 2016. See https://fc16.ifca.ai/bitcoin/papers/MES16.pdf.

MMSKM Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In Network and Distributed System Security Symposium. 2019. See https://eprint.iacr.org/2018/472.pdf.

Nak09 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2009. See http://bitcoin.org/bitcoin.pdf.

NRS20 Jonas Nick, Tim Ruffing and Yannick Seurin. MuSig2: Simple Two-Round Schnorr Multi-Signatures. in Real World Crypto 2021 (https://rwc.iacr.org/2021/program.php). See https://eprint.iacr.org/2020/1261.pdf.

Osu18 Laolu Osuntokun. Multi-Party Channels in the UTXO Model. In Scaling Bitcoin 2018. See https://tokyo2018.scalingbitcoin.org/files/Day1/multi-party-channels-sb-latest.pdf and https://tokyo2018.scalingbitcoin.org/presentations.

Poe17 Andrew Poelstra. Scriptless Scripts. March 4, 2017. See https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-03-mit-bitcoin-expo/slides.pdf.

Poe18 Andrew Poelstra. Scriptless Scripts, Adaptor Signatures and their Applications. In Scaling Bitcoin 2018 Workshops. See https://download.wpsoftware.net/bitcoin/2018-10-scaling-proposal/slides.pdf.

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.

Rub19 Jeremy Rubin. Bip Securethebag. In Scaling Bitcoin 2019. See https://telaviv2019.scalingbitcoin.org/files/bip-op-securethebag.pdf.

Rub21 Jeremy Rubin. UTXOS Scaling. See https://utxos.org/uses/scaling/.

Rus15 Rusty Russell. Reaching the Ground with Lightning (draft 0.2). November 20, 2015. See https://raw.githubusercontent.com/ElementsProject/lightning/master/doc/deployable-lightning.pdf.

SHMB20 Jacob Swambo, Spencer Hommel, Bob McElrath and Bryan Bishop. Bitcoin Covenants: Three Ways to Control the Future. See https://arxiv.org/abs/2006.16714.

Som20 Blind Merged Mining with Covenants. Ruben Somsen. 2020. See https://gist.github.com/RubenSomsen/5e4be6d18e5fa526b17d8b34906b16a5.

Tow19 AJ Towns. OP_SECURETHEBAG (supersedes OP_CHECKOUTPUTVERIFY). See https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-June/017036.html.

WNR20 Pieter Wuille, Jonas Nick and Tim Ruffing. BIP 340: Schnorr Signatures for secp256k1. 19 January 2020. See https://en.bitcoin.it/wiki/BIP_0340.

WNT20 Pieter Wuille, Jonas Nick and Anthony Towns. BIP 341: Taproot: SegWit Version 1 Spending Rules. 19 January 2020. See https://en.bitcoin.it/wiki/BIP_0341.

WNT20b Pieter Wuille, Jonas Nick and Anthony Towns. BIP 342: Validation of Taproot Scripts. 19 January 2020. See https://en.bitcoin.it/wiki/BIP_0342.

A Proposed Change to Support Inherited IDs

This appendix presents the details of the proposed change to Bitcoin to support Inherited IDs (IIDs). Currently, each input in a Bitcoin transaction includes a 36-byte structure, called an OutPoint, that references the parent transaction's output that is being spent **[BWPDTV]**. The OutPoint consists of the parent's 32-byte transaction ID and a 4-byte output index, specifying which of the parent's outputs is being spent.

As mentioned in Section 2.4, the simplest way to support IIDs would be to allow a transaction T with an output O to specify that the child spending output O must reference O with an OutPoint of the form (*IID(T), output index*), where:

- *IID(T)* is a hash of the OutPoint in input 0 of transaction T, and
- *output index* specifies which output from T is being spent.

However, this change would create a hardfork **[BWHF]**, as transactions that use the new OutPoint format would be valid under the new rules, but invalid under the original rules.

Instead, three options will be presented for supporting IIDs in Bitcoin. Any one of these options could be implemented as a softfork **[BWSF].** All three options leave the OutPoint structure unchanged, so that a child transaction that spends output O of transaction T still uses an OutPoint of the form (transaction ID of T, output index of O). However, all of the options modify the definition of signatures that use IIDs in order to simulate the effect of implementing the hardfork described above.

Definitions Common To All Three Options

An output is an *IID output* if it is a native SegWit output with version 2 and a 32-byte witness program **[LLW15]**, and is a *non-IID output* otherwise. A transaction is an *IID transaction* if it has at least one IID output, and is a *non-IID transaction* otherwise. The *first parent* of a (non-coinbase) transaction T is the transaction referenced by the 32-byte transaction ID in the OutPoint in T's input 0.

Each transaction T has a 32-byte *Inherited ID (IID)* defined as follows:

1. if T is a coinbase transaction, T's IID is equal to its transaction ID,
2. if T is a non-IID transaction, T's IID is equal to its transaction ID,
3. if T is an IID transaction that is not a coinbase transaction, T's IID is equal to hash_{IID}(F_IID || F_index)[40] where F is T's first parent, F_IID is F's IID, and F_index is the index field in the OutPoint in T's input 0 (that is, T's input 0 spends F's output F_index).

40 Where $\text{hash}_{\text{tag}}(x)$ equals SHA256(SHA256(tag) || SHA256(tag) || x), as defined in **[WNT20]**.

An IID output is identical to a Taproot output[41], except for the changes to signature validation and transaction verification defined below.

Signature Validation For All Three Options

Regardless of which option is selected, signature validation for an IID output[42] is identical to signature validation in a Taproot output **[WNR20] [WNT20] [WNT20b]**, except:

- each 32-byte transaction ID within an OutPoint that references an IID output is replaced with the IID (as defined above) of the same transaction in the signature message, and
- signature validation of an IID output commits to the portion of the output script of the IID output being spent that follows the last executed OP_CODESEPARATOR (as in the scriptCode of the input for a P2WSH witness program as defined in BIP143 **[LW16]**), but not to the tapleaf hash, the sha_scriptpubkeys, or the scriptPubKey, and if SIGHASH_SINGLE is used, sha_single_output is reordered to appear immediately before spend_type (**[WNT20] [WNT20b]**)[43].

Option 1: IIDs Are Calculated, Not Stored

If this option is selected, IIDs are not stored within transactions. Instead, IIDs are calculated based on the definitions given above.

Transaction Verification

The following change is made to the verification of a transaction T that is an IID transaction and is not a coinbase transaction:

- if the validation of T's input 0 does not include the execution of at least one Verify[44], OP_CHECKSIG, OP_CHECKSIGVERIFY or OP_CHECKSIGADD signature validation operation, verification fails[45].

41 That is, a SegWit output with version 1 and a 32-byte witness program **[WNT20].**

42 The following rules for signature evaluation only apply to IID outputs. As a result, if a transaction has an input X that spends an IID output and another input Y that spends a non-IID output, and if Y has a signature that commits to input X, that signature will commit to the transaction ID in input X (rather than the corresponding IID). This differs from the hardfork scheme presented in Section 2.4 and the body of the paper, but the difference is essential in order to allow a softfork.

43 This change is made because an OP_CODESEPARATOR must be used to prevent the signature from committing to the state number in the 2Stage protocol (as described in Section 3.3) and to prevent circularity in the creation of a covenant (as described in Section 4.2). The elimination of sha_scriptpubkeys is required to prevent circularity when a child transaction receives covenants covering all inputs that are placed by two or more parent transactions (as is the case with settlement transactions in the update-forest and challenge-and-response protocols). The reordering of sha_single_output allows the channel owner to provide the prefix of the hash preimage for the covenant-producing signature that includes all fields except the "Data about this input", thus allowing an operator to create covenants for multiple eras (using multiple update tree inputs) in the update-forest protocol when the channel owner is unchanged or non-responsive.

44 The Verify operation is performed as part of the Taproot key path spending signature validation (see **[WNT20]**).

45 This rule is included to support efficient SVP nodes, as described below.

Calculation and Verification of IIDs

While each transaction has an IID as defined above, the IID is not stored in any transaction or block, so it must be calculated[46].

A full node can calculate the IID of each transaction in blockchain order by using the above definition[47]. In particular, for each unspent transaction output (UTXO), the full node can add an IID field that is used in the calculation of the spending transaction's IID.

In contrast, a Simplified Payment Verification (SPV) node does not have the complete blockchain history and so cannot calculate IIDs for all transactions. Furthermore, the IID of a transaction T can depend on the IID of its first parent, which in turn can depend on the IID of its first parent, etc., thus creating a long dependency chain that is prohibitively expensive for an SPV node to traverse.

Instead, when an SPV node N that supports IIDs receives a tx (transaction) message for a transaction T from a peer that supports IIDs, the tx message includes T's IID. The SPV node N can use the following procedure to verify T's IID:

- If T is a coinbase or non-IID transaction, N verifies that T's IID equals its transaction ID.
- If T is an IID transaction that is not a coinbase transaction, N requests and receives T's first parent F, including F's IID, from a peer that supports IIDs. There are two cases:
    - If F is a coinbase or non-IID transaction, N verifies that T's IID equals hash_{IID}(F_TXID || F_index) where F_TXID is F's transaction ID and F_index is the index field in the OutPoint in T's input 0 (that is, T's input 0 spends F's output F_index).
    - If F is an IID transaction that is not a coinbase transaction, N validates T's input 0 witness program using the provided value of F's IID when performing signature validation[48]. If N's validation of T's input 0 witness program succeeds, N can conclude that it has the correct value of F's IID, in which case N verifies that T's IID equals hash_{IID}(F_IID || F_index) where F_IID is the provided value of F's IID and F_index is the index field in the OutPoint in T's input 0 (that is, T's input 0 spends F's output F_index).
- In either case, if N fails to verify T's IID, N repeats the above procedure by requesting T's IID (and F and F's IID, if required) until N obtains a value that verifies.

Option 2: OP_RETURN Provides IID

If this option is selected, whenever the IID differs from the transaction ID, the IID is stored within the transaction. In particular, the IID is recorded in an output script that includes an OP_RETURN.

---

46 Alternatively, the IID could be stored in the extensible commitment structure introduced in BIP 141 **[LLW15]**.

47 This process can be optimized by noting that prior to the introduction of version 2 SegWit outputs, each transaction's IID equaled its transaction ID due to rule 2 in the definition of a transaction's IID.

48 Recall that at least one such signature validation will be executed, given the above rule for transaction verification.

Transaction Verification

The following change is made to the verification of a transaction T that is a non-coinbase IID transaction:

- T must have a commitment to its IID. The commitment is recorded in an output script in T. It must be at least 38 bytes, with the first 6 bytes being 0x6a24aa21a9ee, that is[49]:
  - 1-byte -- OP_RETURN (0x6a)
  - 1-byte -- push the following 36 bytes (0x24)
  - 4-byte -- IID header (0xaa21a9ee)
  - 32-byte -- IID field (see below)
  - 39-th byte onwards: Optional data with no consensus meaning.
  - If there is more than one output script matching the pattern, the one with the highest output index is assumed to be the commitment.
- If T has a commitment to its IID, and if that commitment's IID field does not equal T's IID (as defined above), verification fails.

Note that a full node or an SPV node is able to perform this transaction verification given the transaction T being verified and given T's first parent transaction, assuming T's first parent transaction has already been verified.

Option 3: New Transaction Metadata Field Stores IIDs

If this option is selected, whenever the IID differs from the transaction ID, the IID is stored in a new metadata field within the transaction. BIP 141 **[LLW15]** introduced a new witness field that holds per-input witness data for a transaction. This option introduces a new metadata field that holds per-transaction metadata for a transaction, which consists of the transaction's IID whenever its IID differs from its transaction ID. BIP 141 also defined an extensible commitment structure that can be used to commit to new transaction data, including the proposed new metadata field.

With this option, nodes must verify the presence and correctness of the IID contained within each non-coinbase IID transaction. This verification process is analogous to the one given for Option 2.

Comparison Of The Three Options

Option 1 is the most efficient in terms of transaction space, as it does not increase transaction size in order to store IID information. However, Option 1 does require that nodes either calculate IIDs when needed, or (more practically) store IIDs of non-coinbase IID transactions that contain outputs in the UTXO set. As a result, Option 1 slightly increases node costs.

49 This commitment structure is modeled after the wtxid commitment structure defined in BIP 141 (see **[LLW15]**).

Option 2 adds 38 bytes to non-coinbase IID transactions, and is thus less efficient than Option 1. However, with Option 2 transaction size more accurately reflects the cost incurred by nodes that store IIDs. As a result, Option 2 is fairer as it forces the users of IIDs to pay for the costs of IIDs via increased transaction fees. In addition, it may be easier to implement Option 2 for SVP nodes.

Finally, Option 3 is semantically cleaner than Option 2, but it is more complex and requires more extensive changes to node software. It may be the best option if other transaction-level metadata is required due to a separate BIP.

Constructing the Taproot Output

In all three options, IID outputs are identical to Taproot outputs, except for the changes to signature validation and transaction verification given above. In constructing a Taproot output, one selects an "internal key" that is then "tweaked" with the root of a merkle tree representing the output scripts[50]. This internal key is quite general, as it can be created as a shared N-of-N pubkey for any output with a fixed set of N participants **[Max18]**.

However, much of the power of IIDs is that they allow the creation of outputs for extremely large values of N and the creation of outputs for which the set of participants is not fixed (as in the update-forest and challenge-and-response protocols). As a result, when using IID outputs with very large or dynamic sets of participants, the internal key may need to be something other than a shared N-of-N pubkey. Fortunately, in such a case a pubkey with an unknown private key can be used, and it can be selected in a manner that does not reveal that the private key is unknown[51].

B Bounds On Payment Channels

This appendix proves nearly matching lower and upper bounds on the worst-case time requirements for a payment channel with an unbounded number of parties (whether or not those parties must solve a group coordination problem). The payment channel is assumed to have the following three properties.

Property 1: The value in the channel can be transferred and distributed arbitrarily between the parties from one channel state to the next[52].

Property 2: The protocol uses a fixed trigger transaction that can remain off-chain indefinitely.

Property 3: After the trigger transaction is put on-chain, the protocol uses time windows which are long enough such that at least one protocol transaction is put on-chain during each time window (assuming at least one party attempts to put such a transaction on-chain during the time window).

50 See "Constructing and spending Taproot outputs" in **[WNT20]**.
51 See the "Initial steps" portion of "Constructing and spending Taproot outputs" in **[WNT20]**.
52 Most, but not all, payment channel protocols have this property. Most notably, the simple micropayment channels introduced by Hearn and Spilman (https://en.bitcoin.it/wiki/Contracts and https://en.bitcoin.it/wiki/Payment_channels) are unidirectional and thus do not have this property.

Lower Bound

Claim: Any payment channel protocol for an unbounded number of parties that satisfies the three properties above and implements N channel states requires at least log N time windows from when the trigger transaction is put on-chain (in the worst case).

Proof: Let L denote the worst-case number of time windows required by such a payment channel protocol after the trigger transaction is put on-chain and assume for the sake of contradiction that $L \leq (\log N)-1$. Because we are analyzing the worst-case number of time windows required, we will assume throughout that at most one payment channel transaction is put on-chain during each time window.

Assume there are $N \geq 2$ parties and that the payment channel has a fixed amount of funds. Assume further that in state X, $1 \leq X \leq N$, party X receives all of the channel funds and every other party receives nothing. Therefore, immediately after all protocol messages for state X have been exchanged and before the trigger transaction has been put on-chain, party X must be able to put enough transactions on-chain (regardless of what transactions other parties put on-chain) to settle in state X (and thus obtain all of the channel's funds) within L time windows of the trigger transaction.

In particular, after all protocol messages for state 1 have been exchanged, party 1 must be able to put some sequence S_1 of transactions on-chain within L time windows of the trigger transaction in order to create on-chain settlement transactions that pay all of the channel funds to party 1, even if no other party puts any transactions on-chain.

Similarly, after all protocol messages for state 2 have been exchanged, party 2 must be able to put enough transactions on-chain that create settlement transactions that pay all of the channel funds to party 2, regardless of the transactions that other parties put on-chain. In particular, assume only parties 1 and 2 put transactions on-chain and that whenever party 1 attempts to put a transaction from S_1 on-chain during the same time window in which party 2 attempts to put a transaction on-chain, party 1 wins, until some time window T_2 when party 2 first succeeds in putting a transaction on-chain. Note that S_1 does not attempt to put any transaction on-chain during time window T_2 (because party 1 wins when competing with party 2) and assume party 1 stops putting transactions on-chain after T_2. Let S_2 denote the sequence of transactions from S_1 put on-chain by party 1 until time window T_2, followed by the sequence of transactions put on-chain by party 2 that results in paying the channel funds to party 2.

In general, for any X, $1 \leq X \leq N$, let S_X denote the sequence of transactions that are put on-chain by parties 1 through X, where parties 1 through X-1 succeed in putting the sequence of transactions S_{X-1} on-chain until some time window T_X when party X first puts a transaction on-chain while S_{X-1} does not attempt to put any transaction on-chain during the same time window, after which only party X puts transactions on-chain, resulting in paying all of the channel funds to party X.

For any sequence of transactions S put on-chain during the L time windows after the trigger transaction is put on-chain, let fingerprint(S) be the binary number with L bits such that each bit B, $0 \leq B \leq L-1$,

equals 1 (and thus has value 2^B) if and only if S puts a transaction on-chain in time window L-B after the trigger transaction is put on-chain. Note that for any S, fingerprint(S) < $2^L \leq N/2$.

Note that for each X, $2 \leq X \leq N$, and for each $W < T_X$, S_{X-1} and S_X are identical during time window W. Therefore, bit L-W of fingerprint(S_X) and bit L-W of fingerprint(S_{X-1}) are identical. Also, note that during time window T_X, S_X puts a transaction on-chain and S_{X-1} does not, so bit L-T_X of fingerprint(S_X) = 1 and bit L-T_X of fingerprint(S_{X-1}) = 0. As a result, for each X, $2 \leq X \leq N$, fingerprint(S_X) > fingerprint(S_{X-1}). Finally, note that fingerprint(S_1) ≥ 0, so fingerprint(S_N) ≥ N-1 < N/2, which is a contradiction. □

Upper Bound

The lower bound proof given above naturally leads to a payment channel protocol for an arbitrary number of parties that satisfies Properties 1 through 3 and nearly matches the lower bound. That payment channel protocol, which we call the *binary-representation-tree protocol*, is described next[53]. It does not use IIDs, and thus does not require any changes to Bitcoin.

The trigger transaction has a single multisig output that is routed to a settlement transaction after passing through 0 to log N update transactions, where N (which is a power of 2) is the number of channel states supported. Update transactions are put on-chain in time windows 1 through log N, while settlement transactions are put on-chain in time window (log N)+1. Settlement transaction X, $0 \leq X \leq N-1$, is routed through an update transaction that is put on-chain in time window (log N)-B for each B such that the binary representation of X has a 1 in bit position B. An example is shown in Figure 8.

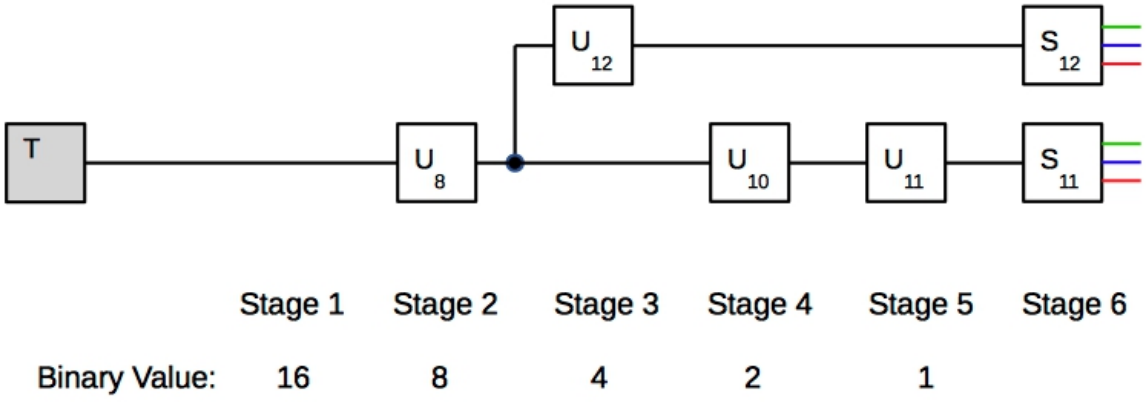

Figure 8: Binary-Representation-Tree Channel. Execution of the (log N)+1 time window binary-representation-tree protocol with N = 32, showing the transition from state 11 (binary 01011) to state 12 (binary 01100). For clarity, update and settlement transactions for earlier states are not shown.

53 The invalidation tree protocol created by Decker and Wattenhofer **[DW15]** can easily be extended to support an arbitrary number of parties (limited only by their ability to solve the group coordination problem) and has parameters n = maximum number of time windows and d = maximum number of transactions put on-chain. Setting n = d = log N yields a protocol that supports at least N states and requires only log N time windows. However, that protocol does not meet Property 3 above, as it assumes that up to d transactions can be put on-chain in a single time window.

Each settlement transaction X is the child of update transaction X, except settlement transaction 0 which is the child of the trigger transaction. Each update transaction X is the child of the update transaction obtained by zeroing-out the least-significant 1-valued bit of the binary representation of X, if one exists, or of the trigger transaction otherwise. Each update transaction X has an nSequence value that prevents it from being put on-chain before stage (log N)-B, where B is the position of the least-significant 1 in the binary representation of X.

In order to update the channel from state X to state X+1, every party first signs the new settlement transaction X+1 and then signs the new update transaction X+1. Let B be the most-significant bit position in which the binary representations of X and X+1 differ. Because X+1 > X, it follows that X+1 has a 1 in bit position B, while X has a 0 in bit position B. It also follows that for every bit position A that is less-significant than bit-position B, X+1 has a 0 in bit position A. Finally, it follows that X has a 1 in bit position B-1. Therefore, update transaction X+1 invalidates settlement transaction X when it is put on-chain in stage (log N)-B. For example, update transaction 12 invalidates settlement transaction 11 when update transaction 12 is put on-chain in stage (log N)-B, where B = 2 is the most-significant bit position in which the binary representations of 11 and 12 differ (see Figure 8).

While this payment channel protocol is quite efficient and nearly matches the lower bound on the number of time windows required, in practice one may choose to slightly increase the number of time windows in order to decrease the number of transactions that need to be put on-chain. In particular, if log N is even, it is possible to support N channel states by implementing the protocol for 2N states (with (log N)+1 stages for update transactions), but only creating settlement and update transaction for those values of X, $0 \le X < 2N$, which have Hamming weight[54] at most (log N)/2. Because there are exactly N such values of X, this version of the protocol halves the worst-case number of update transactions that are put on-chain at the expense of adding a single time window.

C Proofs Of Correctness

C.1 Correctness of the Update-Forest Protocol

Claim: For any channel L and era X, where $1 \le X \le$ max_eras, if the creation protocol of Section 6.4 is followed, and for every era $W \le X$ the update protocol of Section 6.5 is followed, if channel L has logically transitioned to era X as defined in Section 6.5, if Owner_{LX} attempts to settle the channel using the protocol of Section 6.6, if the era parameter is set as required by Section 6.7, and if Owner_{LX} provides sufficient fees for all transactions[55], then Owner_{LX} will successfully put the settlement transaction S_{LX} on-chain[56].

54 The Hamming weight of an integer is the number of ones in its binary representation.

55 Techniques for supporting late determination of fees are presented in Section 8.2.

56 For ease of description, the possibility that some other party puts the desired transaction on-chain will be included in the phrase "Owner_{LX} puts the desired transaction on-chain", as which party puts the transaction on-chain does not matter.

Proof: Assume for the sake of contradiction that $Owner_{LX}$ fails to put S_{LX} on-chain, despite using the settlement protocol from Section 6.6 and providing sufficient fees for all transactions. Let T denote the first transaction in the series of transactions leading from U_X to S_{LX} that is not put on-chain. T must not be put on-chain because either (1) $Owner_{LX}$ does not know T, or (2) there exists a different transaction D that competes with T and is put on-chain before T. First, (1) above cannot happen because $Owner_{LX}$ received and verified T as part of the update from era X-1 to era X. Therefore, (2) must have happened. Recall from the definition of the era parameter that $Owner_{LX}$ has sufficient time to put the series of transactions leading from U_X through S_{LX} on-chain by (start + (X+2) eras - 2 time windows) at the latest. There are 3 cases:

Case 1: D is a timeout transaction put on-chain by a miner after (start + (X+2) eras - 2 time windows). In this case, it follows from the definition of the era parameter that $Owner_{LX}$ has sufficient time to put all of the required transactions on-chain before the miner put D on-chain, which is a contradiction.

Case 2: D = S_{LY} for some Y < X. In this case, note that S_{LY} was not on-chain when channel L logically transitioned to era X at (start + (X+2) eras), which implies U_{LY} was put on-chain after (start + (X+2) eras - 1 time window) (because U_{LY} output 0 was not spent by a miner), which is a contradiction (because of the CHECKSEQUENCEVERIFY in the U_{LY} output 0 script).

Case 3: D = S_{LY} for some Y > X. In this case, S_{LY} must have been put on-chain after (start + (Y+1) eras) which is at least (start + (X+2) eras). However, it was noted above that $Owner_{LX}$ has sufficient time to put S_{LX} on-chain by (start + (X+2) eras - 2 time windows) at the latest, and therefore will succeed in doing so, which is a contradiction.

Therefore, in any case there is a contradiction, which completes the proof. □

C.2 Correctness of the Challenge-and-Response Protocol

Claim: For any channel L and era X, where $1 \leq X \leq max\_eras$, if the creation protocol of Section 7.4 is followed, and for every era $W \leq X$ the update protocol of Section 7.5 is followed, if $Owner_{LX}$ attempts to settle the channel using the protocol of Section 7.6, if the delay parameters are set as required by Section 7.7, and if $Owner_{LX}$ provides sufficient fees for all transactions, $Owner_{LX}$ will successfully put the settlement transaction S_{LX} on-chain[57].

Proof: Assume for the sake of contradiction that $Owner_{LX}$ fails to put S_{LX} on-chain, despite using the settlement protocol from Section 7.6 and providing sufficient fees for all transactions. There are 3 cases:

Case 1: $Owner_{LX}$ fails to complete Step 1 of the settlement protocol. In this case, $Owner_{LX}$ fails to put the series of transactions leading from update root U_{X-1} through challenge C_{LX-1} or from update root U_X through challenge C_{LX} on-chain. This failure cannot be due to insufficient fees (by assumption), so it must be because there is some transaction T in one of these series of transactions that $Owner_{LX}$ cannot

57 For ease of description, the possibility that some other party puts the desired transaction on-chain will be included in the phrase "$Owner_{LX}$ puts the desired transaction on-chain", as which party puts the transaction on-chain does not matter.

put on-chain because (1) $Owner_{LX}$ does not know T or (2) a different transaction D that competes with T is put on-chain before T. First, (1) above cannot happen because $Owner_{LX}$ received and verified T as part of the update from era W = X-1 to era W+1 = X. Next, (2) above cannot happen because every transaction in both of the given series of transactions is defined by an output-only covenant and any transaction that meets that covenant will suffice, as it has all of the required outputs.

Case 2: $Owner_{LX}$ fails to complete Step 2 of the settlement protocol. In this case, $Owner_{LX}$ fails to put response R_{LX} on-chain. This cannot be due to insufficient fees (by assumption) nor due to lack of knowledge of R_{LX} or of the required signature for R_{LX}, as $Owner_{LX}$ received both R_{LX} and the signature for R_{LX} during the update from era W = X-1 to era W+1 = X. Furthermore, both parents of R_{LX}, namely C_{LX-1} and U_{LX}, were put on-chain successfully (as was established in Case 1 above). Finally, the only transaction that competes with R_{LX} is S_{LX-1}. However, S_{LX-1} cannot be put on-chain until (1 era + 1 response_delay) after C_{LX-1} is put on-chain (from the output script of C_{LX-1}), C_{LX-1} cannot be put on-chain until after U_{X-1} is put on-chain (because it is a descendant of U_{X-1}), and U_{X-1} cannot be put on-chain until (start + (X-1) eras) (due to its nLocktime field), so S_{LX-1} cannot be put on-chain until after (start + X eras + 1 response_delay). However, the response_delay parameter was set so that R_{LX} can be put on-chain by (start + X eras + 1 response_delay) at the latest. Therefore, R_{LX} can be put on-chain before S_{LX-1}, so there is no reason that $Owner_{LX}$ cannot put R_{LX} on-chain.

Case 3: $Owner_{LX}$ fails to complete Step 3 of the settlement protocol. In this case, $Owner_{LX}$ fails to put settlement S_{LX} on-chain. By the same reasoning as was used in the previous cases, this cannot be due to insufficient fees or lack of knowledge of S_{LX}. Therefore, it must be because a different transaction D that competes with S_{LX} is put on-chain before S_{LX}.

Note that S_{LX} spends outputs from 3 transactions, namely R_{LX}, F_L, and C_{LX}. It follows from the definition of the response_delay parameter that R_{LX} can be put on-chain by (start + X eras + 1 response_delay) at the latest, and the output script from R_{LX} does not put any additional delays on S_{LX}, so R_{LX} forces S_{LX} to wait until (start + X eras + 1 response_delay) at the latest. F_L forces S_{LX} to wait until (start + (X+1) eras + 2 response_delays) at the latest. Furthermore, it follows from the definition of the response_delay parameter that C_{LX} can be put on-chain by (start + X eras + 1 response_delay) at the latest, the output script from C_{LX} imposes a delay of (1 era + 1 response_delay) on S_{LX}, so C_{LX} forces S_{LX} to wait until (start + (X+1) eras + 2 response_delays) at the latest. Finally, note that the era parameter is set so that $Owner_{LX}$ can put S_{LX} on-chain within 1 era, so S_{LX} can be put on-chain by (start + (X+2) eras + 2 response_delays) at the latest.

There are 3 subcases:

> Subcase 3a: D is a child of R_{LX}. In this case, R_{LX} cannot be put on-chain until after U_X is put on-chain (because it is a descendant of U_X), U_X cannot be put on-chain until (start + X eras) (due to its nLocktime field), D cannot be put on-chain until 1 timeout_delay after R_{LX} is put on-chain (due to the "else" case of R_{LX}'s output script), and timeout_delay ≥ 2 eras + 2 response_delays,

so D cannot be put on-chain until after (start + (X+2) eras + 2 response_delays), which is after $Owner_{LX}$ is able to put S_{LX} on-chain.

Subcase 3b: D is a child of F_L. In this case, $D = S_{LY}$ for some $Y \neq X$. If $Y < X$, then challenge C_{LY} must have been put on-chain earlier than D, as C_{LY} is a parent of D. Therefore, $Owner_{LX}$ must have responded to C_{LY} being put on-chain by using the protocol defined in Section 7.6 to prevent $D = S_{LY}$ from being put on-chain. Recall that the response_delay parameter is large enough to allow 1) detecting that challenge C_{LY} was put on-chain, 2) putting the series of transactions leading from update root U_{Y+1} through update leaf U_{LY+1} on-chain, and 3) putting R_{LY+1} on-chain. Also, note that $D = S_{LY}$ must wait at least (1 era + 1 response_delay) after C_{LY} is put on-chain (due to its nSequence value). Therefore, $Owner_{LX}$ can put competing transaction R_{LY+1} on-chain at most 1 response_delay after C_{LY} was put on-chain, which is before any party can put $D = S_{LY}$ on-chain[58], so $D = S_{LY}$ is not put on-chain. On the other hand, if $Y > X$, then $Y \geq X+1$, so $D = S_{LY}$ cannot be put on-chain until after (start + (X+2) eras + 2 response_delays) due to F_L's output script, which is after S_{LX} can be put on-chain.

Subcase 3c: D is a child of C_{LX}. In this case, D satisfies the "if" case in C_{LX}'s output script, which implies that D has been signed by $Owner_{LX}$. However, that is impossible, as $Owner_{LX}$ has not signed any transaction other than S_{LX} that spends the output of C_{LX} (as specified in Section 7.6).

Therefore, in any case there is a contradiction, which completes the proof. □

C.3 Correctness of the Off-Chain Channel Network Protocol

Note that $early\_timeout_L < early\_timeout_{L-1}$ and $mid\_timeout_L < late\_timeout_L < mid\_timeout_{L-1}$, so Alice and Bob do have sufficient time to perform Steps 1 through 5 if they choose to do so[59]. The remaining issue is whether or not Alice can lose her funds if she follows the protocol, which is shown to be impossible by the following claim.

Claim: If Alice follows the above protocol in all channels in the path in which she is a party, and if Bob receives value V in channel L, then Alice receives value V in channel L-1 (or, Alice is the original sender and she obtains the secret S that acts as a receipt that Bob has received the payment).

Proof: Assume for the sake of contradiction that Alice follows the above protocol in all channels in the path in which she is a party and Bob receives V in L, but Alice does not receive V in L-1 (or Alice is the original sender, and Alice does not obtain the secret S). Because Alice follows the protocol in channel L, there are 2 ways in which Bob could receive V in channel L:

58 To be precise, ownership could transition from one era's owner to another era's owner while responding to such a challenge. In this case, the new era's owner needs to take over from the previous era's owner, and must verify that it can do so promptly enough before taking ownership.

59 To simplify the analysis, it is assumed that Alice can inform the other party in channel L-1 without delay. It would be straightforward to add a delay parameter for the time it takes for her to inform that party and to modify the timeout parameters to incorporate that delay parameter.

Case 1: Bob obtains V in a payout transaction that spends the output of the settlement transaction created in Step 1. In this case, note that Step 1 for channel L-1 must have been completed (from Rule 1). Also, note that Bob's payout transaction is not put on-chain until 1 time window after the settlement transaction (from the corresponding nSequence field in Bob's payout transaction), and that Alice would have claimed the output of the settlement transaction within 1 time window if the settlement transaction were put on-chain after ($late\_timeout_L$ - 1 time window) (from Rule 2). Therefore, Bob's payout transaction must be put on-chain by $late\_timeout_L$ and putting it on-chain reveals S to Alice (from Step 1), which implies Alice is not the original sender (as that would be a contradiction). As a result, Alice discloses S to the other party in channel L-1 by $late\_payout_L$ (from Rule 4). Because $late\_timeout_L < mid\_timeout_{L-1}$, it follows that Alice discloses S to the other party in channel L-1 by $mid\_timeout_{L-1}$.

Case 2: Bob obtains V from an update of L using 2Stage in Step 2 or Step 4. In this case, note that Step 1 for channel L-1 must have been completed (from Rule 1). Also, note that Bob reveals S to Alice before $mid\_timeout_L$, which implies Alice is not the original sender (as that would be a contradiction). Therefore, Alice reveals S to the other party in channel L-1 before $mid\_timeout_{L-1}$.

Therefore, in either case 1) Step 1 for channel L-1 has been completed, 2) Alice knows S, 3) Alice reveals S to the other party in channel L-1 before $mid\_timeout_{L-1}$, and 4) Alice is not the original sender. Therefore, Alice attempts to perform Step 4 with the other party in L-1 in order to update channel L-1 to pay V to her by ($mid\_timeout_{L-1} + off\_chain_{L-1}$). If the other party in L-1 cooperates and completes Step 4, Alice receives V in L-1, which is a contradiction. Therefore, sometime before ($mid\_timeout_{L-1} + off\_chain_{L-1}$) the other party in L-1 fails to follow the protocol. Recall that Step 1 for channel L-1 was completed and Alice knows S, which implies Alice put the settlement transaction created in Step 1 for channel L-1 on-chain and obtained the payout from that transaction (from Rule 3), which is a contradiction. □

D Vaults With Inherited IDs

A vault is a Bitcoin security mechanism that allows the owner of a Vault transaction output to associate two new pubkeys with the output, namely a hot pubkey and a clawback pubkey **[McE19]**. This appendix demonstrates how IIDs can be used to implement vaults.

The vault starts with an on-chain Vault transaction V that has an output that requires a signature with the vault private key. In addition, V places a covenant on the transaction that spends its output in order to allow the output to be "clawed back" in the case of theft, as described below.

In normal operation, Alice transfers a vault output to Bob using the vault private key and the hot private key as follows:

1. Alice puts an Unlock transaction U on-chain that spends the vault output and is signed with the vault private key,
2. Alice waits for a predefined vault_delay, and

3. Alice then puts a transaction S on-chain that spends the output of U, is signed with the hot private key, and has a single output controlled by Bob's pubkey.

However, if Alice's Bitcoin security is compromised and a hacker steals her vault private key and attempts to unlock the vault output, Alice can "clawback" the funds as follows:

1. the hacker puts an Unlock transaction U on-chain that spends the vault output and is signed with the vault private key,
2. Alice monitors the blockchain and sees the unauthorized transaction U that is attempting to steal her vault output, and
3. before the predefined vault_delay has passed, Alice puts a clawback transaction C on-chain that spends the output of U, is signed with the clawback private key, and has a single output controlled by a secure cold pubkey owned by Alice.

Therefore, as long as Alice's clawback private key and secure cold private key are not compromised, the hacker is not able to steal her funds. The design of a vault using IIDs is shown in Figure 9. This is followed by a specification of the output amounts and output scripts required by the vault protocol.

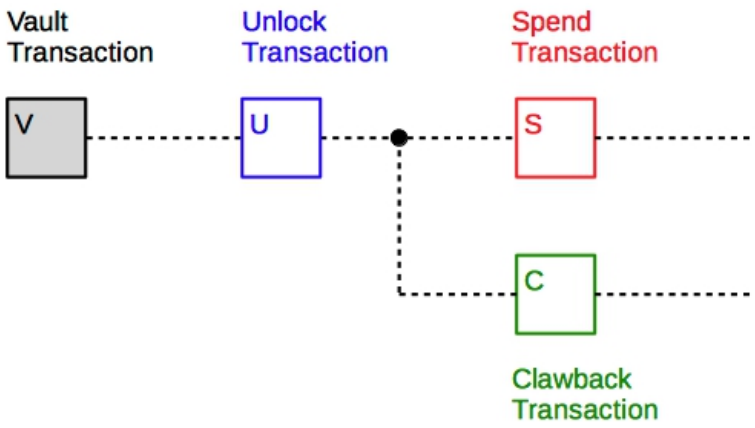

Figure 9: Vault. The on-chain Vault transaction V has a single output which is spent by the Unlock transaction U. V places a covenant on U's output forcing it to wait to be spent by a hot pubkey, but allowing it to be spent by a clawback pubkey without waiting. Colors indicate the signature needed to put the given transaction on-chain: Blue = vault pubkey, Red = hot pubkey, Green = clawback pubkey.

Vault transaction V output amount: vault amount A

Vault transaction V output script (referenced via IID):

OP_DUP <pubkey vault> OP_EQUALVERIFY OP_CHECKSIGVERIFY check vault owner's signature

<signature creating a single-output covenant[60] on U forcing its output to be as shown>

OP_CODESEPARATOR

<standard pubkey with widely-known "private key"> OP_CHECKSIG implement covenant

Unlock transaction U output amount: Vault output amount A

Unlock transaction U output script (referenced via IID):

OP_IF normal spend path using hot pubkey

> **<1 vault_delay> OP_CHECKSEQUENCEVERIFY OP_DROP** wait 1 vault_delay
>
> **OP_DUP <hot pubkey> OP_EQUALVERIFY OP_CHECKSIG** check hot pubkey owner's signature

OP_ELSE clawback path using clawback pubkey

> **OP_DUP <clawback pubkey> OP_EQUALVERIFY OP_CHECKSIG** check clawback pubkey owner's signature

OP_ENDIF

60 Note that the covenant only covers its single output. Therefore, it is possible to add other inputs providing fees and/or outputs returning excess fees, thus supporting the late determination of fees.