eltoo: A Simple Layer2 Protocol for Bitcoin

Christian Decker
Blockstream
decker@blockstream.com

Rusty Russell
Blockstream
rusty@blockstream.com

Olaoluwa Osuntokun
Lightning Labs
roasbeef@lightning.engineering

Abstract

Bitcoin, and other blockchain based systems, are inherently limited in their scalability. On-chain payments must be verified and stored by every node in the network, meaning that the node with the least resources limits the overall throughput of the system as a whole. Layer 2, also called off-chain protocols, are often seen as the solution to these scalability issues: by renegotiating a shared state among a limited set of participants, and not broadcasting the vast majority of state updates to the blockchain, the load on the network is reduced. Central to all Layer 2 protocols is the issue of guaranteeing that an old state may not be committed once it has been replaced. In this work we present eltoo, a simple, yet powerful replacement mechanims for Layer 2 protocols. It introduces the idea of state numbers, an on-chain enforceable variant of sequence numbers that were already present in the original implementation, but that were not enforceable.

1 Introduction

Bitcoin is the first and most successful cryptocurrency in the world. At the time of writing Bitcoin had a market cap of over 150 Billion USD, almost half of the market cap of all cryptocurrencies combined. However this success comes at a cost: increased demand for Bitcoin and emerging applications using Bitcoin not only as a speculative tool, but as a currency, have driven up the transaction volume. Over time Bitcoin's inherent scalability issues have become more and more prominent, and may severely limit its ability to function as a currency.

At its core Bitcoin relies on a replicated state machine, the *blockchain*, to order operations, *transactions*, on its global state, the *UTXO set*, i.e., the association of bitcoins to an owner. Transactions are verified and replayed by each participant, called a *node*, in the network. This limits the throughput of the network as a whole to the lowest throughput of any one node in the network. Increasing the load beyond that throughput may result in nodes unable to handle the load being pushed out of the network. It is therefore safe to say that, with the current architecture, Bitcoin will be unable to scale with increased demand, without losing its trustless nature and without excluding some participants.

Recently a number of *Layer 2* protocols have been proposed [3, 6, 7] to address the scalability issues that Bitcoin, and other blockchain based systems, are facing. The key insight of Layer 2 solutions is that not every transaction has to be applied globally. Instead it is possible to locally negotiate multiple operations among a smaller set of participants and only apply aggregates of these transactions to the global state.

Layer 2 protocols are a form of *smart contracts* between a fixed set of participants, that negotiate contract state changes locally, only involving the blockchain in case of dispute or final contract settlement. These protocols commonly consist of a *setup phase*, a *negotiation phase* and a *settlement phase*. The setup phase involves moving some funds into an address controlled by all participants, such that the participants have to agree on how to distribute the funds later. The negotiation phase is the core of the protocols and consists of repeated adjustments on the distribution of funds to participants. Finally, the settlement phase simply enforces the agreed upon distribution on the blockchain.

It is paramount that all participants have the means to enforce the latest agreed upon state at any point during the execution of the smart contract. In Bitcoin smart contracts this is achieved by representing the latest state in the form of a *settlement transaction* that aggregates any intermediate change. During the negotiation phase the participants repeatedly negotiate a new settlement transaction, invalidating any previous settlement transaction.

The general concept of renegotiating a settlement transaction among a set of participants was already part of the Nakamoto implementation of Bitcoin in the form of `nSequence` numbers in the transactions. Miners were supposed to replace transactions with a conflicting transaction if it had a higher sequence number. However, there was no way for this replacement to be enforced, and miners would simply prefer transactions that paid the most transaction fees. So if one party preferred an old state in which its share of the funds was larger than its final share, it could bribe miners into

confirming the outdated state instead.

It is this lack of enforceability of replacements that ultimately proved difficult to solve. With the nSequence number based replacements users had to trust miners to only include the latest version, and, in case multiple states were published concurrently miners could end up picking an old state, because they simply didn't see the replacing ones. The replacement strategy is the major contribution of both Duplex Micropayment Channels [3], with its invalidation tree, and Lightning [7], with its punishment based replacement.

In this paper we present *eltoo*, a novel protocol to update a negotiated contract state, that invalidates any previous state, making them unenforceable. eltoo introduces the concept of state numbers, similar to sequence numbers, but enforceable: by allowing a later state to respend any of the previous states we defer the on-chain settlement until the last settlement transaction is confirmed. In order to enable a later state to respend any of the previous states we introduce the concept of *floating transactions*, i.e., transactions that can be bound to any previous transaction with matching scripts.

We first present an idealized on-chain version to introduce the basic concept, and then lift this on-chain protocol off of the blockchain. The on-chain protocol introduces the basic idea of overriding a previous settlement, but still confirms all states on the blockchain. Finally, the off-chain protocol introduces floating transactions and with it the ability to skip intermediate states.

2 Bitcoin Basics

At its core Bitcoin is a decentralized replicated state machine, and like any replicated state machine, it has two fundamental primitives: a shared *state* that is managed by the replicas and *operations* that are applied to the state and modify it over time. The shared state of Bitcoin is the so called set of unspent transaction outputs, the *UTXO set*. An output is a tuple of a value, denominated in bitcoins, and a spending condition that determines the owner of the value. The sole type of operation on the shared state in Bitcoin are *transactions*. Transactions spend some output, i.e., remove them from the shared state, by claiming them as *inputs*, and then redistribute the value to new owners, creating new outputs specifying new spending conditions.

Transactions are broadcast to all other participants in the network, and upon receiving a transaction a participant will validate the transaction. Validation includes verifying that the outputs being spent exist, that the newly

created outputs do not have higher value than the spent ones, and that the transaction is authorized to spend the outputs in the first place. If the transaction is valid, each participant applies it to the local view of the shared state, otherwise it is discarded. For consistency it is paramount that all participants agree on the same validity conditions and that validation of transactions is deterministic, otherwise the local view of replicas may diverge, resulting in an inconsistent state.

The spending conditions in the outputs are expressed using a simple scripting language, and they are fulfilled by a script in the inputs of the spending transaction. Most commonly the spending condition of an output simply asks for a valid signature of the spending transaction from a specific public key, such that only the owner, holding the matching private key, can authorize the transaction. In this common case the input script would then simply push the signature onto the stack and the output script, executed right after the input script would then verify the signature. Due to this case being so common, the spending condition is commonly referred to as `scriptPubKey`, and the input script is referred to as `scriptSig`. With the introduction of segregated witnesses the terms `witnessProgram` and `witness` were introduced, which serve the same purpose but are handled slightly differently. Throughout this paper we will use the terms input script to refer to `witnessProgram` and `scriptPubKey`, and output script to refer to the `witness` or `scriptSig`.

The scripting language however allows for a wide range of scripts, ranging from multisig transactions requiring multiple signers to authorize a transaction, to cryptographic puzzles that need to be solved in order to spend an output. Throughout this paper we will make use of a variety of scripts that combine cryptographic puzzles and signatures to authorize transactions.

As a final building block, Bitcoin makes use of a blockchain to prevent double-spends. Transactions are aggregated into blocks that are then broadcast and chained together to build a blockchain. The specifics of the blockchain are out of scope for this work, suffice it to say that the Bitcoin blockchain guarantees that eventually transactions are confirmed, and double-spends are resolved, by confirming one transaction and discarding all conflicting ones. The process of confirming transactions may take minutes to several hours to complete.

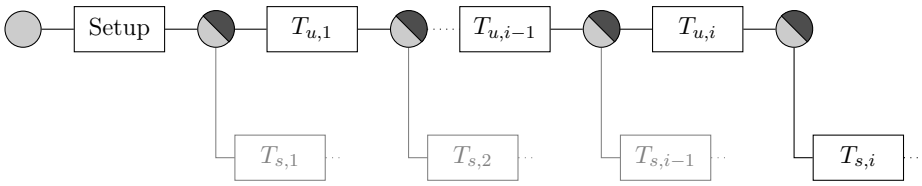

Figure 1: Overview of the on-chain update protocol. The setup transaction initiates the protocol. Each update transaction $T_{u,i}$ invalidates the previously negotiated settlement transaction $T_{s,i-1}$ (indicated by lighter color), until finally $T_{s,i}$ is not invalidated and settles the contract.

3 On-chain update protocol

Before lifting the protocol off the blockchain, making it a scalable off-chain protocol, we first introduce the basic functionality in an idealized on-chain update protocol. In the on-chain protocol all intermediate states are broadcast and confirmed on the blockchain. While not scalable the on-chain protocol introduces the intuition behind the off-chain protocol.

In the following we limit the description to the case of two endpoints attempting to update an agreed upon state, i.e., the balance of each party when settling. However, the protocol is trivial to generalize to any number of parties, and different semantics, as long as the resulting transactions adhere to the Bitcoin validity rules, e.g., size constraints.

The on-chain protocol follows the schema presented before, and has three phases:

- a *setup phase*, used to allocate some funds to a 2-of-2 multisig address;
- a *negotiation phase*, consisting of the participants creating a chain of update transactions that each reflect a state update;
- a *settlement phase*, during which updates are replayed on the blockchain, until no new updates are available;

In the on-chain protocol the negotiation phase and the settlement phase effectively overlap, as will be seen later.

3.1 Setup Phase

The setup phase is used to fund the contract by allocating some funds to a multisig address. Once the funding transaction is confirmed neither endpoint

can unilaterally move the funds, instead they will have to collaborate to spend the funds.

Both endpoints A and B generate and exchange two public-/private-keypairs:

- a *settlement keypair* (A_s and B_s) used to spend the funds to a settlement transaction and thus terminate the protocol;
- an *update keypair* (A_u and B_u) used to replace a previous update, continuing the protocol;

One endpoint, the *funder*, creates a *funding transaction* $T_{u,0}$ spending some of its coins, and creating a funding output o_0 that requires both endpoints to collaborate in order to spend the funds. This is enforced by requiring both endpoints to sign any spending transactions either with A_u and B_u, or with A_s and B_s. Figure 2 details the exact script that sets up the spending conditions for the shared output.

Before signing and broadcasting the funding transaction the funder requires the other endpoint to create an initial *settlement transaction* that returns the funds back to the funder. The initial settlement transaction spends the funding output, and creates a single output that returns all of the funds to the funder. This settlement transaction is then signed using the settlement key and returned to the funder. The funder verifies that the settlement transaction matches its expectations, i.e., it returns its funds and is signed by the other endpoint. Now the funder can broadcast the funding transaction, and wait for it to be confirmed effectively starting the contract. The funder also signs the initial settlement transaction, making it complete and returns it to the other endpoint. Both endpoints hold an identical set of settlement transactions, i.e., there is no asymmetry in what the endpoints know about. This is in stark contrast with the channel commitment invalidation procedure used today in Lightning, in which the settlement transactions are personalized to the endpoint and may not be shared without incurring a loss.

Figure 2 shows the output script that is used by the funding output as well as all successive update transaction outputs. It may be further optimized by moving some operations, but we omitted the optimization for the sake of clarity. The *if* branch is used to attach a settlement transaction while the *else* branch is used to attach future update transactions. Notice that the settlement branch is encumbered with an `OP_CHECKSEQUENCEVERIFY` (abbreviated as `OP_CSV`). The `OP_CSV` opcode is preceded by a numberic argument, which defines the number of blocks that the output being spent has to be

```
OP_IF
  10 OP_CSV
  2 A_s B_s 2 OP_CHECKMULTISIGVERIFY
OP_ELSE
  2 A_u B_u 2 OP_CHECKMULTISIGVERIFY
OP_ENDIF
```

Figure 2: The output script used by the on-chain update transactions.

```
<sig A_u>
<sig B_u>
OP_FALSE
```

(a) Update transaction input script

```
<sig A_s>
<sig B_s>
OP_TRUE
```

(b) Settlement transaction input script

Figure 3: Input scripts used when spending either for an update or a settlement

confirmed before the script is executed. If the required number of blocks is not reached the opcode will raise an error and the verification fails.

The `OP_CSV` in the *if*-branch creates a timeout during which only the *else* branch is valid, giving precedence to update transactions, and only allowing settlement transactions after the timeout expires.

Figure 3 shows the input scripts for the update transaction and the settlement transaction matching the conditions set up in the output script listed in Figure 2. They only differ in the branch of the *if*-statement that is being selected, and which keys provided the signatures. The use of different keypairs prevents an attacker from simply swapping out the branch selection and reusing the same signatures for the other branch.

The settlement is renegotiated with each update, changing the allocations of funds to each participant. While the initial settlement transaction had a single output returning funds to the funder, updates may now create new settlement transactions with any number of outputs, with varying allocations to any one of them. In addition to simple outputs directly owned by the participants, it is also possible to add more complex outputs, to the settlement transaction, such as HTLCs [4] for multi-hop payments or payments conditioned on the release of a secret.

3.2 Negotiation and Settlement phase

In the on-chain protocol the negotiation phase and the settlement phase overlap. Since the funding transaction was broadcast during the setup phase, the endpoints have some time, i.e., until the `OP_CSV` timeout expires, before the initial settlement transaction becomes valid.

Should they want to invalidate the settlement transaction and replace it with a new version, they collaborate to create an *update transaction*. The update transaction spends the contract's funding transaction output or the previous update transaction output and creates a new output, with the same script as the previous transaction. Update transactions are signed by A_u and B_u, and therefore immediately valid unlike the `OP_CSV` encumbered settlement script branch. The update transaction effectively doublespends the settlement transaction before it becomes valid.

As with the funding transaction, before signing and broadcasting the new update transaction, the two endpoints negotiate a new settlement transaction that spends the newly created contract output.

For example assuming that the outputs of the current settlement transaction are $\text{balance}_{A,i-1} = 5$ and $\text{balance}_{B,i-1} = 5$ as singlesig outputs owned by A and B respectively. If endpoint A wants to transfer 1 bitcoins to B then they create an unsigned update transaction $T_{u,i}$ that spends the previous output o_{i-1} from $T_{u,i-1}$ and creates a new output o_i with the same script as o_{i-1}. Then they create a settlement transaction $T_{s,i}$ that has two outputs, assigning $\text{balance}_{A,i} = 4$ to A and $\text{balance}_{B,i} = 6$ to B. Once the settlement transaction is created the endpoints exchange signatures using their settlement keys A_s and B_s for the settlement transaction. After verifying the validity of the settlement transaction they exchange signatures for $T_{u,i}$ using keys A_u and B_u and broadcast it to the network.

The old settlement transaction $T_{s,i-1}$ can be safely discarded since was doublespent by the update transaction $T_{u,i}$, and $T_{s,i-1}$ cannot be used at a later point in time.

This process is repeated multiple times, moving funds back and forth between the endpoints, or with more complex outputs such as HTLCs. Eventually there is no new update, and the endpoints decide to settle. In this case they simply wait for the `OP_CSV` timeout to expire and use the final settlement transaction, i.e., the last agreed upon settlement transaction, that has not been doublespent by an update, to move the funds to the respective endpoints, thus terminating the contract.

Alternatively they could collaborate and create a settlement transaction that uses the unencumbered branch of the script, i.e., by signing with the

update keypairs instead of the settlement keypairs. This would avoid the waiting period during the settlement.

Notice that choosing the correct timeout for the settlement branch is a trade-off. It must be chosen high enough to guarantee that any subsequent update is confirmed before the settlement transaction becomes valid. On the other hand this timeout is also the time participants have to wait before funds are returned to their sole control should the other participant stop cooperating.

4 Lifting the protocol off the chain

While the protocol in Section 3 is correct, and allows a multiparty contract to be updated any number of times, it does nothing to address the scalability issue: it still requires every single update to be broadcast to the blockchain. In this section we describe how the simple on-chain protocol of updating by doublespending settlement transactions can be lifted off the blockchain to build a scalable off-chain protocol.

This is achieved by allowing update transactions to bind to any of the previous update transactions. Rebinding to any prior update effectively skips the intermediate update transactions, thus greatly reducing the on-chain transactions.

4.1 Floating Transactions

The key insight is that the intermediate update transactions do not have to be committed to the blockchain at all. After all they were used as entry-points to the intermediate settlements and respent by the following update transactions when updating. The latest set of update transaction $T_{u,i}$ and settlement transaction $T_{s,i}$ represent the entire state of the contract, and intermediate transactions are not needed. If it is possible to have later transaction attach to the output of any of the preceding update outputs, it is possible to skip intermediate transactions and still enforce the latest agreed upon state. We call transactions that can be attached to any output of one of its predecessors a *floating transaction*.

4.1.1 `SIGHASH_NOINPUT`

Bitcoin transactions commit to the outputs they are spending by including a reference, i.e., a previous output transaction hash and an index, in the transaction. It is usually not possible to change the reference without

```
OP_IF
  10 OP_CSV
  2 A_{s,i} B_{s,i} 2 OP_CHECKMULTISIGVERIFY
OP_ELSE
  <S_i + 1> OP_CHECKLOCKTIMEVERIFY
  2 A_u B_u 2 OP_CHECKMULTISIGVERIFY
OP_ENDIF
```

Figure 4: The output script used by the update transactions. The spending transaction's locktime is compared to the state number $S_i + 1$ (matching the spent transaciton locktime) in the script, before proceeding to signature verification.

invalidating signatures that authorize the transaction, since the signature commits to the reference. However, signatures in Bitcoin transactions can be parameterized with the *sighash-flag* that specifies which parts of the transaction are committed to in the signature. By introducing a new sighash flag, SIGHASH_NOINPUT, it is possible to selectively mark a transaction as a floating transaction.

SIGHASH_NOINPUT instructs the signature creation and the signature verification code to blank the previous output field of the input that is being signed. By doing so the signature no longer commits to any specific previous output, and the transaction can be rewritten to reference a different transaction output. The process of rewriting the transaction to reference different outputs is called *binding*. Notice that binding allows us to spend any output, without invalidating the signatures, as long as the output scripts and the input scripts match. We have therefore removed the tight coupling between the spending transaction and the outputs it is spending, and have replaced it with a weak coupling through the scripts.

In the eltoo protocol, the update transactions $T_{u,i}$, and the settlement transactions $T_{s,i}$ are signed making use of SIGHASH_NOINPUT. Since the outputs of the update transactions are all compatible, any update transaction can be rebound to spend any of the other update transaction's outputs. When attempting to bind to a different previous update transaction the participant can simply replace the previous output transaction hash in the input with the transaction hash that it is to be bound to.

Settlement transactions are also floating transactions, since rebinding the update transaction that created the output the settlement transaction spends, changes its hash. Like before we reintroduce the coupling from settlement transaction to the specific update output through the scripts.

4.1.2 Ordering of updates

Using the SIGHASH_NOINPUT flag for update transaction adds a lot of flexibility, however they are now too flexible. To illustrate why imagine an update transaction $T_{u,i}$ and a later update transaction $T_{u,j}$ with $j > i$. Without further restrictions it would be possible to use $T_{u,i}$ to spend the output created by $T_{u,j}$, i.e., replace a later state with an earlier one.

This was not a problem in the on-chain protocol since the transactions were not floating and could only be spent in the correct order, but by lifting the protocol off-chain and introducing SIGHASH_NOINPUT, we have lost the update ordering. To re-establish the ordering we introduce the concept of *state numbers*, similar to the original intent for sequence numbers.[1]

Generally speaking, by using SIGHASH_NOINPUT we have removed any commitment to the state we are replacing. We therefore have to selectively re-introduce some of the previous transaction's details into the validation.

The output script of the update transaction has the format given in Figure 4. The key difference with respect to the on-chain update output script is that the off-chain update output script includes a new operation, OP_CHECKLOCKTIMEVERIFY (abbreviated as OP_CLTV) and it commits to the next state number $S_i + 1$, i.e., the earliest state number that may replace this update $T_{u,i}$.

The output script therefore specifies that only an update transaction with a higher state number than the current state number may bind to this output. The current state number is stored in the nLocktime field of the transaction, while the next higher state number is stored in the output script.

We are now using the existing nLocktime field in a transaction to store the spending transaction's state number. This is necessary since the signature does not cover the signature field (or the witness field in segregated witnesses transactions) and hence an attacker could simply change the state number in the spending transaction without invalidating the signatures. Furthermore the spending transaction's state number cannot be committed to in the output script since at the time of its creation we do not yet know which one will be the last settlement transaction, nor its state number. On the other hand the minimum state number that may be bound to the update output is know at the time of creation of the update transaction, and can therefore simply be pushed onto the stack without incurring these problems.

With this mechanism we have repurposed the nLocktime field in a trans-

[1] Notice that we could use nSequence for this purpose, but due to the interplay with OP_CSV we opted to use nLocktime.

action to be able to have signatures commit to an arbitrary state number. However, the `nLocktime` field is already being used to invalidate transactions until some time. We need to therefore be careful about which values we assign to this field in order to avoid interference between the two mechanisms. Fortunately there is a vast range locktime values that are in the past: any number above 0.500 billion is interpreted as a UNIX timestamp, and with a current timestamp of ≈1.5 billion, that leaves about 1 billion numbers that are interpreted as being in the past. If a number in this range is used as a locktime value, then the existing locktime mechanism will consider them valid, hence they can be used for our purpose.

Notice that this repurposing is only needed in order to maintain backward compatibility with the currently deployed version of Bitcoin. Other cryptocurrencies could introduce a new numeric field, that signatures commit to, that is specifically used for this purpose, potentially extending the number of valid updates.

4.1.3 Attaching settlement transactions to update transactions

In Section 4.1.1 we mentioned that settlement transactions are also signed using `SIGHASH_NOINPUT`. This is necessary since rebinding the update transaction also changes its hash, potentially breaking the reference from the settlement transaction to its corresponding update transaction. Unlike the case in which we want to enable an update transaction to bind to any previous transaction, we need limit the settlement transaction to be bindable solely to the matching update transaction.

In order to achieve the limited binding for settlement transaction a new set of public keys $A_{s,i}$ and $B_{s,i}$ is derived that is specific to each state number. The key-pair derivation can easily be done with hierarchical deterministic key derivation as used by many existing Bitcoin wallets. This ensures that a settlement transaction can only be bound to the matching update transaction. It does not increase the storage requirements for the participant since it'll only need to remember the key-pair matching the latest state.

Figure 5 shows an example execution of the off-chain protocol in which a number of updates invalidate the corresponding settlement transactions. An update transaction $T_{u,i}$ can be attached to any of the previous shared update outputs, hence the intermediate transactions can be skipped when settling on-chain. This is symbolized by the lighter color in the figure. In an ideal instance the setup output is directly spent by the final update which in turn enables the final settlement.

While nothing prevents an intermediate update $T_{u,i}$ from being broadcast

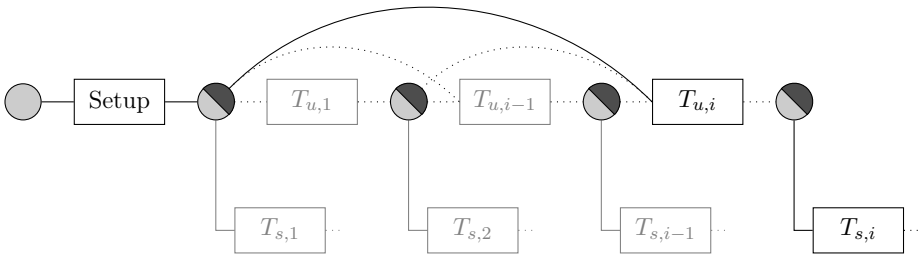

Figure 5: Overview of the off-chain protocol.

and confirmed in the blockchain, it can immediately be respent by any of the later update transactions $T_{u,j}$ with $j > i$. This is true for any of the intermediate states and terminates when there were no more agreed upon, i.e., fully signed, update transactions, when the final update transaction is broadcast.

4.1.4 Compatibility with P2SH and P2WSH transactions

In 2012 BIP 16 [1] introduced the concept of pay-to-script-hash which was used to defer revealing the spending conditions until the time of the output is being spent, hence reducing the state nodes need to maintain and improving extensibility. P2SH only commits to the hash of the script that sets up the spending conditions instead of listing the full script. A P2SH output script has the following format `OP_HASH160 [20-byte-hash-value] OP_EQUAL` and will verify that the script provided by the spending transaction in serialized form matches what we expect. With the introduction of segregated witnesses the P2SH scheme was extended with a witness version P2WSH, in which the script that the output commits to is no longer revealed in the input, but rather is part of the witness.

P2SH adds yet another level of indirection, since the spender needs to reconstruct the output script that was committed to in the previous transaction. Since the output scripts in the update transactions are all identical, except for the state number and the settlement keys which are derived using the state number, the spender needs to recover the state number.

The spender now has to recreate the P2SH or P2WSH script, which includes the state number, order to provide it by pushing it onto the stack during the verification. Since the only difference between the various scripts is the state number and the public keys derived from the seed and the sequence number this requires to look up `nLocktime` the transaction that is

being spent, i.e., the spender is being bound to. This lookup is already required in order to know what the spending transaction should be bound to, hence this does not add any more complexity.

4.1.5 Adding transaction fees to updates

One final issue is that all the transactions so far do not have any fees attached. Fees are important in order have miners confirm transactions in a timely manner. This is paramount for security, since if an intermediate update transaction is confirmed, but following update transactions are delayed, it could happen that the `OP_CSV` timeout expires and the invalidated settlement transaction is now valid as well, resulting in a race between the update and the invalidated settlement.

Fees are allocated by not assigning some of the input value to an output, creating unassigned value that can be claimed by the miner confirming the transaction. The update transactions cannot leave a part of the value unassigned since they'd be gradually reducing the funds in the channel with each update. Furthermore the unassigned funds would have to be sufficient to cover the cost for each update being confirmed individually in the worst case, severely limiting the lifetime of the channel.

The solution is to allow update transactions to add further funds on the fly, without invalidating the existing signatures. Since the update transactions have a single input and a single output, they can be signed using the *sighash-single* flag, which only ensures that the output matching the input is present, but leaving the transaction open for further modifications.

Should a participant wish to confirm an update transaction, and thus initiate the settlement phase, they can alter the update transaction by adding a new input, spending some of their funds, and add a new output. The new output returns the funds added, minus a miner fee, to the participant.

The ability to dynamically add fees at the settlement time maximizes the flexibility of the settling party. If the update transaction represents the last agreed upon state it can use relatively low fees being certain that it will not be replaced. Should the selected fee be too low during times of network congestion, then either party can create a new version of the update transaction bumping the fee (Replace-by-fee [5]) ensuring a timely settlement.

Should an attacker attempt to settle an invalidated state, then the fees may be collected by a miner, and the other endpoint can enforce the latest state regardless, by adding fees to her update. This last case effectively punishes the attacker by allowing the transaction to be confirmed, and sub-

sequently replacing it, but without returning the fees on the intermediate update.

4.2 Settling a contract

In its current form the off-chain protocol has a limited lifetime since the shared output script has a settlement branch that becomes active after a timeout. This is implemented using the `OP_CHECKSEQUENCEVERIFY` opcode, which starts counting down the timeout as soon as the output is confirmed. The timeout was not a problem in the on-chain variant since the participant either created a new update transaction before the timeout ran out, or let the contract settle using the settlement transaction. In the off-chain variant however, the timeout would require broadcasting and confirming intermediate update transactions in order to extend the lifetime of the contract, or *refresh* the contract.

In order to avoid having to refresh the contract on-chain simply to keep the timeouts from expiring, we introduce an additional step in-between the setup phase and the settlement phase: the *trigger step*. The sole purpose of the trigger step is to defer the time at which the timeout starts. The output from the setup transaction is changed into a simple 2-of-2 multisig output, which is then spent by a *trigger transaction* that has a single output with the output script from Figure 4. Update and settlement transaction no longer spend the setup transaction's output, but rather they spend the trigger transaction's output. The trigger phase starts with the broadcast of the setup transaction and ends with the broadcast of the trigger transaction.

During the setup phase both endpoints ensure not only that they have a valid settlement transaction that returns the funds to the funder, but they also exchange signatures for the trigger transaction. This in turn enables either party to initiate the settlement phase by broadcasting the trigger transaction, the latest update transaction if any, and the latest settlement transaction.

5 Analysis

The eltoo renegotation protocol simplifies existing off-chain protocols, and enables new use-cases for off-chain protocols. In the following we will analyze the security assumptions as well as lay out some of the new enabled use-cases.

5.1 Safety

We define a state i, consisting of the tuple of update transaction $T_{u,i}$ and settlement transaction $S_{u,i}$, to be committed if the settlement transaction is confirmed in the blockchain. For simplification we consider any transaction to be confirmed if it appears in a block, i.e., we do not consider blockchain reorganizations.

We define an unsafe execution of the protocol as any execution in which a participant in the off-chain protocol, making use of the eltoo renegotiation protocol, is able to commit an old state to the blockchain. Consequently any execution in which only the final state is eventually committed is considered safe. This matches the above definition of confirmation since any confirmation of a settlement transaction that is not the final settlement is considered sufficient to fail the protocol, even in the presence of reorganizations.

Notice that the setup of the contract is considered safe. It is easy to see that, if the first update transaction and settlement transaction is signed before the setup transaction is signed, then funds never are locked in without the ability to settle again.

We consider the scenario with two participants in the protocol, one of which is an attacker and the other one, the victim, behaves correctly. It is the goal of the attacker to commit an old state, that maximized its payout. For this purpose the attacker may store an arbitrary number of intermediate update transactions, while the victim only stores the latest set of update and settlement transactions.

At any point in time the attacker may broadcast an old update transaction $T_{u,i}$, in the hope of also confirming $T_{s,i}$. $T_{s,i}$ however will have to wait for the `OP_CSV` timeout in the update's output script to expire. This gives the victim the opportunity of broadcasting the final update transaction as a reaction. The victim can either witness $T_{u,i}$ being broadcast or by seeing it confirmed in the blockchain. The reaction consists of creating two versions of the latest $T_{u,j}$ transaction with $j > i$:

- $T'_{u,j}$ bound to the setup output, effectively doublespending $T_{u,i}$;
- $T''_{u,j}$ bound to the output of $T_{u,i}$, which doublespends $T_{s,i}$;

Generally speaking, no matter which update transaction the attacker broadcasts, the victim can doublespend both the update transaction itself, or, in the case the update transaction succeeds, it can doublespend the settlement. The eventual success of the doublespend is guaranteed by the `OP_CSV`

timeout, which ensures that the doublespend is prioritized over the attacker's settlement transaction.

The safety of the protocol therefore depends on two key assumptions:

- The victim can detect an attack in time to react to it, either by actively participating in the network, or by outsourcing the reaction to a third party;
- The later update transaction can be confirmed in the specified time to doublespend the outdated update;

Both of these depend on the `OP_CSV` timeout duration, so if a user is offline for a prolonged period it may chose a higher timeout. Higher timeouts however also mean longer waiting time to retrieve its own funds in case the other participant stops cooperating. The timeout can be collaboratively chosen by the participants in order to optimize the safety and liveness of the protocol, depending on the specific capabilities of the participants.

Notice that the settlement phase is little more than an update without the need for a timeout, and therefore the same safety analysis applies.

5.2 Extending the protocol to more parties

As mentioned above the storage requirements for participants consist of the latest tuple of update and settlement transaction. This is because they can be rebound to any of the intermediate update transactions in case it gets broadcast. This is in stark contract to the Lightning Network, where the reaction to a previous state being published needs to tailored to that specific state.

In Lightning the information stored is asymmetric, i.e., the information stored by one endpoint is different from the information stored by the other. In fact the security of Lightning hinges on the information being kept private since publishing it could result in the funds being claimed by the other endpoint. We refer to this information about previous states as being toxic.

With eltoo the information stored by the participants is symmetric, eliminating the toxic information, and greatly simplifying the protocol as a whole. The information being symmetric also enables extending the protocol to any number of participants, since there is no longer a combinatorial problem of how to react to a specific participant misbehaving.

The protocol can be generalized to any number of participants by simply gathering all the settlement and update public keys of the participants and listing them in the public key list. Due to the size constraints imposed on

the output scripts it is currently not possible to go beyond 7 participants. This results from each participant contributing 2 public keys, 33 bytes each, and the script size for P2SH scripts being limited to 520 bytes.

This limit is raised to 10'000 bytes for P2WSH scripts, allowing up to 150 participants, but producing very costly on-chain transactions. However, with the introduction of schnorr signatures, and aggregatable signature it is possible to extend this to any number of participants, and without incurring the on-chain cost, since all public keys and signatures are aggregated into a single public key and a single signature.

6 Related Work

The invalidation problem of superceded states is central to the all layer 2 protocols, and a number of proposals have been proposed. The idea of renegotiating transactions while they are still unconfirmed dates back to the original design Bitcoin by Nakamoto. This original design aimed to use sequence numbers in the transactions to allow replacing superseded transactions simply by incrementing the sequence number. Miners were supposed to replace any transaction in their memory pool by transactions with higher sequence numbers. However, this mechanism was flawed since a rational miner will always prefer transactions with a higher expected payout, even though they may have a lower sequence number. An attacker could incentivize miners to confirm a specific version by adding fees either publicly or by directly bribing the miners.

A first invalidation mechanism that was actually deployed was used by the simple micropayment channels by Hearn and Spilman [6]. The simple micropayment channel supports incremental transfer of value in only one direction, from a sender to a recipient. It uses partially signed transactions that can be completed only by the recipient, which will only ever enforce the latest state since it is the state that maximizes its payout. The unidirectional nature of the simple micropayment channels severely limit their utility as they can only be used for incremental payments and, once the funds in a channel are exhausted, the channel has to be settled on-chain, and a new one has to be set up.

The Lightning Network, proposed by Joseph Poon and Thaddeus Dryja [7] is a much more advanced off-chain protocol that enabled bidirectional movement of funds, and also used hashed timelock contracts (HTLCs) to enable multi-hop payments that are end-to-end secure. The central idea of Lightning is to invalidate an old state by punishing the participant publishing

it, and claiming all the funds in the channel. This however introduces an intrinsic asymmetry in the information tracked by each participant. The replaced states turn into toxic information as soon as they are replaced, and leaking that information may result in funds being stolen. The asymmetry also limits Lightning to two participants.

Duplex Micropayment Channels [3], a design created in parallel to the Lightning Network, also offer bidirectional movement of funds. They rely on decreasing timelocks, arranged in an invalidation tree, to replace earlier states. The major downsides of this design are the limited number of replacements, since the timelocks can only be counted down to the current time. The invalidation tree extended the range of timelocks, however this came at the cost of more on-chain transactions in the non-collaborative close case.

All of the previous protocols had one major issue: since the transactions need to be signed potentially hours or days before they were released into the network, the participants would have to estimate the future fees to be able to ensure a timely confirmation. This is particularly important for the Lightning Network and Duplex Micropayment Channels, since they rely on timelocks to allow a defrauded party to react. While the need to guarantee timely confirmation is also true for eltoo, the need to estimate future fees was completely removed. In eltoo the fees are added a posteriori at the time the transaction is published, and, should the fee turn out to be insufficient it can be amended simply by creating a new version of the transaction with higher fees a broadcasting it.

The ability to extend the protocol to a larger number of participants also means that it can be used for other protocols, such as the channel factories presented by Burchert et al. [2]. Prior to eltoo this used the Duplex Micropayment Channel construction, which resulted in a far larger number of transactions being published in the case of a non-cooperative settlement of the contract.

Finally, it is worth noting that the update mechanism presented in this paper is a drop-in replacement for the update mechanism used in the Lightning Network specification [8]. It can be deployed without invalidating the ongoing specification efforts by the specification authors or the implementations currently being deployed. This possible since the existing stack of transport, multi-hop and onion routing layers are orthogonal to the update mechanism used in the update layer.

7 Conclusion

In this work we have introduced eltoo, a simple, yet powerful, renegotiation and invalidation mechanism for off-chain protocols and smart contracts. Eltoo is much simpler to implement and easier to analyze than previous protocols, can be easily extended to any number of participants and has a very small footprint.

The modifications required to the Bitcoin protocol to support eltoo are minimal and can be seen in Appendix A, and thanks to the recent deployment of segwit can be deployed easily.

References

[1] Gavin Andresen. Pay to script hash. `https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki`. Online; accessed 06 November 2017.

[2] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable Funding of Bitcoin Micropayment Channel Networks. In *Symposium on Self-Stabilizing Systems*, November 2017.

[3] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, 2015.

[4] David A. Harding. Hashed timelock contracts. `https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts`. [Online; accessed March 2018].

[5] David A. Harding and Peter Todd. Opt-in full replace-by-fee signaling. `https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki`. Online; accessed 06 November 2017.

[6] Mike Hearn and Jeremy Spilman. Bitcoin contracts. `https://en.bitcoin.it/wiki/Contracts`. [Online; accessed May 2015].

[7] Joseph Poon and Tadge Dryja. Lightning network, 2015.

[8] Lightning Specification Team. Lightning network specifications. `https://github.com/lightningnetwork/lightning-rfc`. [Online; accessed April 2018].

A `SIGHASH_NOINPUT` BIP

```
BIP: xyz
Layer: Consensus (soft fork)
Title: SIGHASH_NOINPUT
Author: Christian Decker <decker.christian@gmail.com
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-xyz
Status: Draft
Type: Standards Track
Created: 2017-02-28
License: PD
```

A.1 Abstract

This BIP describes a new signature hash flag (`sighash`-flag) for segwit transactions. It removes any commitment to the output being spent from the signature verification mechanism. This enables dynamic binding of transactions to outputs, predicated solely on the compatibility of output scripts to input scripts.

A.2 Motivation

Off-chain protocols make use of transactions that are not yet broadcast to the Bitcoin network in order to renegotiate the final state that should be settled on-chain. In a number of cases it is desirable to react to a given transaction being seen on-chain with a predetermined reaction in the form of another transaction. Often the reaction is identical, no matter which transaction is seen on-chain, but the application still needs to create many identical transaction. This is because signatures in the input of a transaction uniquely commit to the hash of the transaction that created the output being spent.

This proposal introduces a new sighash flag that modifies the behavior of the transaction digest algorithm used in the signature creation and verification, to exclude the previous output commitment. By removing the commitment we enable dynamic rebinding of a signed transaction to outputs whose `witnessProgram` and value match the ones in the `witness` of the spending transaction.

The dynamic binding is opt-in and can further be restricted by using unique `witnessProgram` scripts that are specific to the application in-

stance, e.g., using public keys that are specific to the off-chain protocol instance.

A.3 Specification

`SIGHASH_NOINPUT` is a flag with value `0x40` appended to a signature to that the signature does not commit to any of the inputs, and therefore to the outputs being spent. The flag applies solely to the verification of that single signature.

The `SIGHASH_NOINPUT` flag is only active for segwit scripts with version 1 or higher. Should the flag be used in a non-segwit script or a segwit script of version 0, the current behavior is maintained and the script execution MUST abort with a failure.

The transaction digest algorithm from BIP 143 is used when verifying a `SIGHASH_NOINPUT` signature, with the following modifications:

```
2. hashPrevouts (32-byte hash) is 32 0x00 bytes
4. outpoint (32-byte hash + 4-byte little endian) is
   set to 36 0x00 bytes
5. scriptCode of the input is set to an empty script
   0x00
```

The `value` of the previous output remains part of the transaction digest and is therefore also committed to in the signature.

The `NOINPUT` flag MAY be combined with the `SINGLE` flag in which case the `hashOutputs` is modified as per BIP 143[2]: it only commits to the output with the matching index, if such output exists, and is a `uint256` `0x0000......0000` otherwise.

Being a change in the digest algorithm, the `NOINPUT` flag applies to all segwit signature verification opcodes, specifically it applies to:

- `OP_CHECKSIG`
- `OP_CHECKSIGVERIFY`
- `OP_CHECKMULTISIG`
- `OP_CHECKMULTISIGVERIFY`

[2]BIP143: Transaction Signature Verification for Version 0 Witness Program

A.4 Binding through scripts

Using `NOINPUT` the input containing the signature no longer references a specific output. Any participant can take a transaction and rewrite it by changing the hash reference to the previous output, without invalidating the signatures. This allows transactions to be bound to any output that matches the value committed to in the `witness` and whose `witnessProgram`, combined with the spending transaction's `witness` returns `true`.

Previously, all information in the transaction was committed in the signature itself, while now the relationship between the spending transaction and the output being spent is solely based on the compatibility of the `witnessProgram` and the `witness`.

This also means that particular care has to be taken in order to avoid unintentionally enabling this rebinding mechanism. `NOINPUT` MUST NOT be used, unless it is explicitly needed for the application, i.e., it MUST NOT be a default signing flag in a wallet implementation. Rebinding is only possible when the outputs the transaction may bind to all use the same public keys. Any public key that is used in a `NOINPUT` signature MUST only be used for outputs that the input may bind to, and they MUST NOT be used for transactions that the input may not bind to. For example an application SHOULD generate a new key-pair for the application instance using `NOINPUT` signatures and MUST not reuse them afterwards.

A.5 Deployment

The `NOINPUT` sighash flag is to be deployed during a regular segwit script update.

A.6 Backward compatibility

As a soft fork, older software will continue to operate without modification. Non-upgraded nodes, however, will not verify the validity of the new sighash flag and will consider the transaction valid by default. Being only applicable to segwit transaction, non-segwit nodes will see an anyone-can-spend script and will consider it valid.

A.7 Acknowledgments

The NOINPUT sighash flag was first proposed by Joseph Poon in February 2016[3], after being mentioned in the original Lightning paper[4]. A formal proposal was however deferred until after the activation of segwit. This proposal is a continuation of this discussion and attempts to formalize it in such a way that it can be included in the Bitcoin protocol. As such we'd like acknowledge Joseph Poon and Thaddeus Dryja as the original inventors of the NOINPUT sighash flag, and its uses in off-chain protocols.

[3] bitcoin-dev SIGHASH$_{\text{NOINPUT}}$ in Segregated Witness
[4] Lightning Network paper