Elmo: Recursive Virtual Payment Channels for Bitcoin

Aggelos Kiayias[1,2] and Orfeas Stefanos Thyfronitis Litos[1]

[1] University of Edinburgh
[2] IOHK
akiayias@inf.ed.ac.uk, o.thyfronitis@ed.ac.uk

Abstract. TODO: update for cooperative closing A dominant approach towards the solution of the scalability problem in blockchain systems has been the development of layer 2 protocols and specifically payment channel networks (PCNs) such as the Lightning Network (LN) over Bitcoin. Routing payments over LN requires the coordination of all path intermediaries in a multi-hop round trip that encumbers the layer 2 solution both in terms of responsiveness as well as privacy. The issue is resolved by "virtual channel" protocols that, capitalizing on a suitable setup operation, enable the two endpoints to engage as if they had a direct payment channel between them.

Apart from communication efficiency, virtual channel constructions have three natural desiderata. A virtual channel constructor is *recursive* if it can also be applied on pre-existing virtual channels, *variadic* if it can be applied on any number of pre-existing channels and *symmetric* if it encumbers in an egalitarian fashion all channel participants both in optimistic and pessimistic execution paths. We put forth the first Bitcoin-suitable recursive variadic virtual channel construction. Furthermore our virtual channel constructor is symmetric and offers optimal round complexity both in the optimistic and pessimistic execution paths. Our virtual channels can be implemented over Bitcoin assuming the `ANYPREVOUT` signature type, a feature that we prove necessary for any efficient protocol that has parties maintain a set of Bitcoin transactions in their local state. We express and prove the security of our construction in the universal composition setting.

1 Introduction

TODO: update for cooperative closing The popularity of blockchain protocols in recent years has stretched their performance exposing a number of scalability considerations. In particular, Bitcoin and related blockchain protocols exhibit very high latency (e.g. Bitcoin has a latency of 1h [1]) and a very low throughput (e.g., Bitcoin can handle at most 7 transactions per second [2]), both significant shortcomings that jeopardize wider use and adoption and are to a certain extent inherent [2]. To address these considerations a prominent approach is to optimistically handle transactions off-chain via a "Payment Channel Network"

(PCN) (see, e.g., [3] for a survey) and only use the underlying blockchain protocol as an arbiter in case of dispute.

The key primitive of PCN protocols is a payment (or more generally, state) channel. Two parties initiate the channel by locking some funds on-chain and subsequently exchange direct messages to update the state of the channel. The key feature is that state updates are not posted on-chain and hence they remain unencumbered by the performance limitations of the underlying blockchain protocol. Given this primitive, multiple overlapping payment channels can be combined and form the PCN.

Closing a channel is an operation that involves posting the state of the channel on-chain; it is essential that any party individually can close a channel as otherwise a malicious counterparty (i.e. the other channel participant) could prevent an honest party from accessing their funds. This functionality however raises an important design consideration: how to prevent malicious parties from posting old states of the channel. Addressing this issue can be done with some suitable use of transaction "timelocks", a feature that prevents a transaction or a specific script from being processed on-chain prior to a specific time (measured in block height). For instance, diminishing transaction timelocks facilitated the Duplex Micropayment Channels (DMC) [4] at the expense of bounding the overall lifetime of a channel. Using script timelocks, the Lightning Network (LN) [5] provided a better solution that enabled channels staying open for an arbitrary duration: the key idea was to duplicate the state of the channel between the two counterparties, say Alice and Bob, and facilitate a punishment mechanism that can be triggered by Bob whenever Alice posts an old state update and vice-versa. The script timelocking is essential to allow an honest counterparty some time to act.

Interconnecting state channels in LN enables any two parties to transmit funds to each other as long as they can find a route of payment channels that connects them. The downside of this mechanism is that it requires the direct involvement of all the parties along the path for each payment. Instead, "virtual payment channels", suggest the more attractive approach of putting a one-time initialization step to set up a virtual payment channel, which subsequently can be used for direct payments with complexity —in the optimistic case— independent of the length of the path. Initial constructions for virtual channels essentially capitalized on the extended functionality of Ethereum, e.g., Perun [6] and GSCN [7], while more recent work [8] brought them closer to Bitcoin-compatibility (by leveraging adaptor signatures [9]).

A virtual channel constructor can be thought of as an *operator* over the underlying primitive of a state channel. We can identify three natural desiderata for this operator.

- Recursive. A recursive virtual channel constructor can operate over channels that themselves could be the results of previous applications of the operator. This is important in the context of PCNs since it allows building virtual channels on top of pre-existing virtual channels.

- Variadic. A variadic virtual channel constructor can virtualize any number of input state channels directly, i.e. without leveraging recursion. This is important in the context of PCNs since it enables applying the operator to build virtual channels of arbitrary length, without the undue overhead of opening, managing and closing multiple virtual channels only to use the one at the "top" of the recursion.
- Symmetric. A symmetric virtual channel constructor offers setup and closing operations that are symmetric in terms of cost between the two "endpoints" or the "intermediaries" (but not a mix of both) for the optimistic and pessimistic execution paths. This is important in the context of PCNs since it ensures that no party is worse-off or better-off after an application of the operator in terms of accessing the basic functionality of the channel.

Endpoints are the two parties that share the virtual channel, intermediaries are the parties that take part in any of underlying channels.

We note that recursiveness, while identified already as an important design property (e.g., see [7]) it has not been achieved in the context of Bitcoin-compatible channels (it was achieved only for DCN-like fixed lifetime channels in [10] and left as an open question for LN-type channels in [8]). The reason behind this are the severe limitations imposed in the design by the scripting language of Bitcoin-compatible systems. With respect to the other two properties, observe that successive applications of a recursive *binary* virtual channel operator to make it variadic will break symmetry (since the sequence of operator applications will impact the participants' functions with respect to the resulting channel). This is of particular concern since all previous virtual channel constructors proposed are binary, cf. [7,8,10].

Our Contributions. We present the first Bitcoin-suitable recursive virtual channel constructor that is recursive and supports channels with an indefinite lifetime. In addition, our constructor, Elmo (named after St. Elmo's fire), is variadic and symmetric. In our constructor, both optimistic and pessimistic execution paths are optimal in terms of round complexity: issuing payments between the two endpoints requires just three messages of size independent of the length of the channel while closing the channel requires up to two on-chain transactions for any involved party (endpoint or intermediary) also independent of the channel's length. Our construction is also compatible with the current version of any blockchain that supports Turing-complete smart contracts, such as Ethereum [11].

We achieve the above by leveraging on a sophisticated virtual channel setup protocol which, on the one hand, enables endpoints to use an interface that is invariant between on-chain and off-chain (i.e. virtual) channels, while on the other, intermediaries can act following any arbitrary activation sequence when the channel is closed. The latter is achieved by making it feasible for anyone becoming an initiator towards closing the channel, while subsequent respondents, following the activation sequence, can choose the right action to successfully complete the closure process by posting a single transaction each.

3

We formally prove the security of the constructor protocol in the UC [12] setting; our ideal functionality is global, according to the definition of [13]. The construction relies on the ANYPREVOUT signature type, which does not sign the hash of the transaction it spends, therefore allowing for a single pre-signed transaction to spend any output with a suitable script. We further discuss the limitations of any constructor primitive that does not rely on ANYPREVOUT in Section ??. In particular in Theorem ??, we prove that any virtual channel constructor protocol that has participants store transactions in their local state and offers an efficient closing operation via $O(1)$ transactions will have an exponentially large state in the number of intermediaries, unless ANYPREVOUT is available.

Related work TODO: update for cooperative closing The first proposal for PCNs was due to [14] which only enabled unidirectional payment channels. As mentioned previously, DMCs [4] with their decrementing timelocks have the shortcoming of limited channel lifetime. This was ameliorated by LN [5] which has become the dominant paradigm for designing PCNs for Bitcoin-compatible systems. LN is currently implemented and operational for Bitcoin. It has also been adapted for Ethereum [11], where it is known as the Raiden Network [15].

A number of attacks have been identified against LN. The wormhole attack [16] against LN allows colluding parties in a multi-hop payment to steal the fees of the intermediaries between them and Flood & Loot [17] analyses the feasibility of an attack in which too many channels are forced to close in a short amount of time, reducing the blockchain liveness and enabling a malicious party to steal off-chain funds.

Payment routing [18,19,20] is another research area that aims to improve the network efficiency without sacrificing privacy. Actively rebalancing channels [21] can further increase network efficiency by preventing routes from becoming unavailable due to lack of well-balanced funds.

An alternantive payment channel construction that aspires to be the successor of Lightning is eltoo [22]. It has a conceptually simpler construction, smaller on-chain footprint and a more forgiving attitude towards submitting an old channel state than Lightning, but it needs the ANYPREVOUT sighash flag to be added to Bitcoin. Bolt ?? constructs privacy-preserving payment channels enabling both direct payments and payments with a single untrusted intermediary. Generalized Bitcoin-Compatible Channels [9] enable the creation of state channels on Bitcoin, extending channel functionality from simple payments to arbitrary Bitcoin scripts.

Sprites [23] leverages the scripting language of Ethereum to decrease the time collateral is locked up compared to Lightning. Perun [6] and GSCN [7] exploit the Turing-complete scripting language of Ethereum to provide virtual state channels, i.e. channels that can open without an on-chain transaction and that allow for arbitrary scripts to be executed off-chain. Similar features are provided by Celer [24]. Hydra [25] provides state channels for the Cardano [26] blockchain which combines a UTXO type of model with general purpose smart contract functionality that are also isomorphic, i.e. Hydra channels can accommodate any script that is compatible with the underlying blockchain.

BDW [27] shows how pairwise channels over Bitcoin can be funded with no on-chain transactions by allowing parties to form groups that can pool their funds together off-chain and then use those funds to open channels. ACMU [28] allows for multi-path atomic payments with reduced collateral, enabling new applications such as crowdfunding conditional on reaching a funding target.

TEE-based [29] solutions [30,31,32,20] improve the throughput and efficiency of PCNs by an order of magnitude or more, at the cost of having to trust TEEs. Brick [33] uses a partially trusted committee to extend PCNs to fully asynchronous networks.

Solutions alternative to PCNs include sidechains (e.g., [34,35,36]), non-custodial chains (e.g., [37,38,39,40]), and partially centralised payment networks that entirely avoid using a blockchain [41,42,43,44].

Last but not least, a number of works propose virtual channel constructions for Bitcoin. Lightweight Virtual Payment Channels [10] enables a virtual channel to be opened on top of two preexisting channels and uses a technique similar to DMC. Let "simple channels" be those built directly on-chain, i.e. channels that are not virtual. Bitcoin-Compatible Virtual Channels [8] also enables virtual channels on top of two preexisting simple channels and offers two protocols, the first of which guarantees that the channel will stay off-chain for an agreed period, while the second allows the single intermediary to turn the virtual into a simple channel. We remark that the above strategy has the shortcoming that even if it is made recursive (a direction left open in [8]) after k applications of the constructor the virtual channel participant will have to publish on-chain k transactions in order to close the channel if all intermediaries actively monitor the blockchain.

Furthermore, Donner [45] is the first work to achieve variadic virtual channels without the need for recursion nor features that are not yet available in Bitcoin. This is achieved by having the funder use funds that are external to the "base channels" (i.e. the channels that the virtual channel is based on), so a party that has all its coins in channels cannot fund a Donner channel; additionally, we conjecture that using external coins precludes variadic virtual channel designs that are not encumbered with limited lifetime. Furthermore, due to its incentive structure, all of the base channels are forced to close if the virtual channel is closed. Donner also relies on placeholder outputs which, due to the minimum coins they need to carry to exceed Bitcoin's "dust limit", may skew the incentives of rational players and adds to the opportunity cost of maintaining a channel. Furthermore, its design complicates future iterations that lift its current restriction that only one of the two channel parties can fund the virtual channel. Donner is more efficient than the present work in terms of storage, computation and communication complexity, and boasts a simpler design, but has less room for optimisations.

We refer the reader to Table 1 for a comparison of the features and limitations of virtual channel protocols, including the one put forth in the current work.

Table 1. Features & requirements comparison of virtual channel protocols

	Unlimited lifetime	Recursive	Variadic	Script requirements
LVPC [10]	✗	◐[a]	✗	Bitcoin
BCVC [8]	✓	✗	✗	Bitcoin
Perun [6]	✓	✗	✗	Ethereum
GSCN [7]	✓	✓	✗	Ethereum
Donner [45]	✗	✗	✓	Bitcoin
this work	✓	✓	✓	Bitcoin + `ANYPREVOUT`

[a] lacks security analysis

2 High Level Explanation

TODO: update for cooperative closing Conceptually, Elmo is split into three main actions: channel opening, payments and closing. A channel (P_1, P_n) between parties P_1 and P_n may be opened directly on-chain, in which case the two parties follow an opening procedure similar to that of LN, or it can be opened on top of a path of preexisting channels (P_1, P_2), (P_2, P_3), ..., (P_{n-2}, P_{n-1}), (P_{n-1}, P_n). In the latter case all parties P_i on the path follow our novel protocol, setting aside funds in their channels as collateral for the new virtual channel that is being opened. Once all intermediaries are done, P_1 and P_n finally create (and keep off-chain) their initial "commitment" transaction, following a logic similar to Lightning and thus their channel is open.

A payment over an established channel follows a procedure heavily inspired by LN, but without the use of HTLCs. To be completed, a payment needs three messages to be exchanged by the two parties.

Finally, the closing procedure of a channel C can be completed unilaterally and consists of signing and publishing a number of transactions on-chain. As we will discuss later, the exact transactions that a party will publish vary depending on the actions of the parties controlling the channels that form the "base" of C and the channels that are based on C. Our protocol can be augmented with a more efficient optimistic collaborative closing procedure, which however is left as future work.

Note that a virtual channel is built on top of two or more so-called "base" channels, which, due to the recursive property, may themselves be simple or virtual. The parties that control the base channels are called "base parties". The fact that more than two base channels can be used by a single virtual channel is ensured by the variadic property.

In more detail, to open a channel (c.f. Figure 29) the two counterparties (a.k.a. "endpoints") first create new keypairs and exchange the resulting public keys (2 messages), then prepare the underlying base channels if the new channel is virtual ($12 \cdot (n-1)$ total messages, i.e. 6 outgoing messages per endpoint and 12 outgoing messages per intermediary, for $n-2$ intermediaries), next they exchange signatures for their respective initial commitment transactions (2 messages) and lastly, if the channel is to be opened directly on-chain, the "funder" signs and

publishes the "funding" transaction to the ledger. As we alluded to earlier, a channel with its funding transaction on-chain is called "simple". A channel is either simple or virtual, not both. We here note that like LN, only one of the two parties, the funder, provides coins for a new channel. This limitation simplifies the execution model and the analysis, but can be lifted at the cost of additional protocol complexity.

Let us now introduce some notation used in figures with transactions. Reflecting the UTXO model, each transaction is represented by a cylindrical, named node with one incoming edge per input and one outgoing edge per output. Each output can be connected with at most one input of another transaction; cycles are not allowed. Above an input or an output edge we note the number of coins it carries. In some figures the coins are omitted. Below an input we place the data carried and below an output its spending conditions (a.k.a script). For a connected input-output pair, we omit the data carried by the input. σ_K is a signature on the transaction by sk_K; in all cases, signatures are carried by inputs. An output marked with pk_K needs a signature by sk_K to be spent. $m/\{pk_1, \ldots, pk_n\}$ is an m-of-n multisig ($m \leq n$) that needs signatures from m distinct keys among $sk_1, \ldots, sk_n$. If k is a spending condition, then $k + t$ is the same spending condition but with a relative timelock of t. Spending conditions or data can be combined with logical "AND" ($\wedge$) and "OR" ($\vee$), so an output $a \vee b$ can be spent either by matching the condition a or the condition b, and an input $\sigma_a \wedge \sigma_b$ carries signatures from sk_a and sk_b.

Note that all signatures for all multisig outputs use the `ANYPREVOUT` hash type.

2.1 Simple Channels

In a similar vein to earlier UTXO-based PCN proposals, having an open channel essentially means having very specific keys, transactions and signatures at hand, as well as checking the ledger periodically and being ready to take action if misbehaviour is detected. Let us first consider a simple channel that has been established between *Alice* and *Bob* where the former owns c_A and the latter c_B coins. There are three sets of transactions at play: A "funding" transaction that is put on-chain, off-chain "commitment" transactions that spend the funding output on channel closure and off-chain "revocation" transactions that spend commitment outputs in case of misbehaviour (c.f. Figure 1).
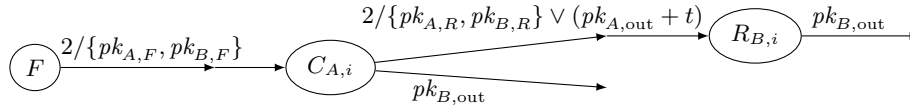

Fig. 1. Funding, commitment and revocation transactions

In particular, there is a single on-chain funding transaction that spends $c_A + c_B$ coins (originally belonging to the funder), with a single output that is encumbered with a $2/\{pk_{A,F}, pk_{B,F}\}$ multisig and carries $c_A + c_B$ coins.

Next, there are two commitment transactions, one for each party, each of which can spend the funding tx and produce two outputs with c_A and c_B coins each. The two txs differ in the outputs' spending conditions: The c_A output in *Alice*'s commitment tx can be spent either by *Alice* after it has been on-chain for a pre-agreed period (i.e. it is encumbered with a "timelock"), or by a "revocation" transaction (discussed below) via a 2-of-2 multisig between the counterparties, whereas the c_B output can be spent only by *Bob* without a timelock. *Bob*'s commitment tx is symmetric: the c_A output can be spent only by *Alice* without timelock and the c_B output can be spent either by *Bob* after the timelock expiration or by a revocation tx. When a new pair of commitment txs are created (either during channel opening or on each update) *Alice* signs *Bob*'s commitment tx and sends him the signature (and vice-versa), therefore *Alice* can later unilaterally sign and publish her commitment tx but not *Bob*'s (and vice-versa).

Last, there are $2m$ revocation transactions, where m is the total number of updates of the channel. The jth revocation tx held by an endpoint spends the output carrying the counterparty's funds in the counterparty's jth commitment tx. It has a single output spendable immediately by the aforementioned endpoint. Each endpoint stores m revocation txs, one for each superseded commitment tx. This creates a disincentive for an endpoint to cheat by using any other commitment transaction than its most recent one to close the channel: the timelock on the commitment output permits its counterparty to use the corresponding revocation transaction and thus claim the cheater's funds. Endpoints do not have a revocation tx for the last commitment transaction, therefore these can be safely published. For a channel update to be completed, the endpoints must exchange the signatures for the revocation txs that spend the commitment txs that just became obsolete.

Observe that the above logic is essentially a simplification of LN.

2.2   Virtual Channels

In order to gain intuition on how virtual channels function, consider $n-1$ simple channels established between n honest parties as before. P_1 (the funder) and P_n want to open a virtual channel over these base channels. Before opening the virtual, each base channel is entirely independent, having different unique keys, separate on-chain funding outputs, a possibly different balance and number of updates. After the n parties follow our novel virtual channel opening protocol, they will all hold off-chain a number of new, "virtual" transactions that spend their respective funding transactions and can themselves be spent by new commitment transactions in a manner that ensures fair funds allocation for all honest parties.

In particular, apart from the transactions of simple channels, each of the two endpoints also has an "initiator" transaction that spends the funding output of

its only base channel and produces two outputs: one new funding output for the base channel and one "virtual" output (c.f. Figures 2, 49). If the initiator transaction ends up on-chain, the latter output carries coins that will directly or indirectly fund the funding output of the virtual channel. This virtual funding output can in turn be spent by a commitment transaction that is negotiated and updated with direct communication between the two endpoints in exactly the same manner as the payments of simple channels.
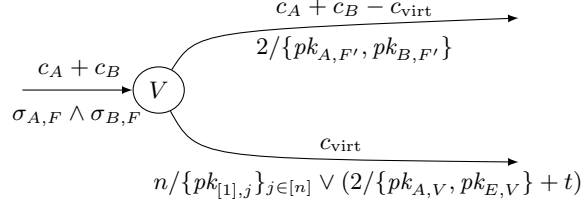
$$c_A + c_B - c_{\text{virt}}$$
$$2/\{pk_{A,F'}, pk_{B,F'}\}$$
$$c_A + c_B$$
$$\sigma_{A,F} \wedge \sigma_{B,F}$$
$$V$$
$$c_{\text{virt}}$$
$$n/\{pk_{[1],j}\}_{j\in[n]} \vee (2/\{pk_{A,V}, pk_{E,V}\} + t)$$

Fig. 2. $A - E$ virtual channel: A's initiator transaction. Spends the funding output of the $A - B$ channel. Can be used if B has not published a virtual transaction yet.

Intermediaries on the other hand store three sets of virtual transactions (Figure 48): "initiator" (Figure 3), "extend-interval" (Figure 4) and "merge-intervals" (Figure 5). Each intermediary has one initiator tx, which spends the party's two funding outputs and produces four: one funding output for each base channel, one output that directly pays the intermediary coins equal to the total value in the virtual channel, and one "virtual output", which carries coins that can potentially fund the virtual channel. If both funding outputs are still unspent, publishing its initiator tx is the only way for an intermediary to close either of its channels.

Furthermore, each intermediary has $O(n)$ extend-interval transactions. Being an intermediary, the party is involved in two base channels, each having its own funding output. In case exactly one of these two outputs has been spent honestly and the other is still unspent, publishing an extend-interval transaction is the only way for the party to close the base channel corresponding to the unspent output and take its funds. Such a transaction consumes two outputs: the only available funding output and a suitable virtual output, as discussed below. An extend-interval tx has three outputs: A funding output replacing the one just spent, one output that directly pays the intermediary coins equal to the total value of the virtual channel, and one virtual output.

Last, each intermediary has $O(n^2)$ merge-intervals transactions. If both base channels' funding outputs of the party have been spent honestly, publishing a merge-intervals transaction is the only way for the party to close either base channel. Such a transaction consumes two suitable virtual outputs, as discussed below. It has two outputs: One that directly pays the intermediary coins equal to the total value of the virtual channel, and one virtual output.
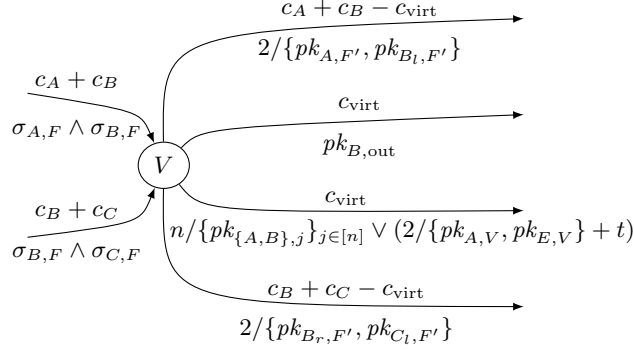
9

Fig. 3. $A - E$ virtual channel: B's initiator transaction. Spends the funding outputs of the $A - B$ and $B - C$ channels. Can be used if neither A nor C have published a virtual transaction yet.
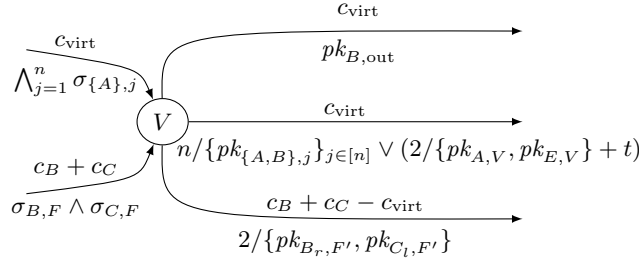

Fig. 4. $A-E$ virtual channel: One of B's extend interval transactions. σ is the signature. Spends the virtual output of A's initiator transaction and the funding output of the B-C channel. Can be used if A has already published its initiator transaction and C has not published a virtual transaction yet.

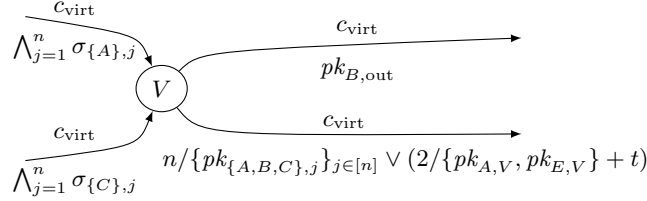**Fig. 5.** A - E virtual channel: One of B's merge intervals transactions. Spends the virtual outputs of A's and C's virtual transactions. Can be used if both A and C have already published their initiator transactions. Notice that the interval of C's virtual output only contains C, which can only happen if C has published its initiator and not any other of its virtual transactions.

To understand why this multitude of virtual transactions is needed, we now zoom out from the individual party and discuss the dynamic of the system as a whole. The first party P_i that wishes to close a base channel observes that its funding output(s) remain(s) unspent and publishes its initiator transaction. First, this allows P_i to use its commitment transaction to close the base channel. Second, in case P_i is an intermediary, it directly regains the coins it has locked for the virtual channel. Third, it produces a virtual output that can only be consumed by P_{i-1} and P_{i+1}, the parties adjacent to P_i (if any) with specific extend-interval transactions. The virtual output of this extend-interval transaction can in turn be spent by specific extend-interval transactions of P_{i-2} or P_{i+2} that have not published a virtual transaction yet (if any) and so on for the next neighbours. The idea is that each party only needs to publish a single virtual transaction to "collapse" the virtual layer and each virtual output uniquely defines the continuous interval of parties that have already published a virtual transaction and only allows parties at the edges of this interval to extend it. This prevents malicious parties from indefinitely replacing a virtual output with a new one. As the name suggests, merge-intervals transactions are published by parties that are adjacent to two parties that have already published their virtual transactions and in effect joins the two intervals into one.

Each virtual output can also be used as the funding output for the virtual channel after a timelock, to protect from unresponsive parties blocking the virtual channel indefinitely. This in turn means that if an intermediary observes either of its funding outputs being spent, it has to publish its suitable virtual transaction before the timelock expires to avoid losing funds. What is more, all virtual outputs need the signature of all parties to be spent before the timelock (i.e. they have an n-of-n multisig) in order to prevent colluding parties from faking the intervals progression. To ensure that parties have an opportunity to react, the timelock of a virtual output is the maximum of the required timelocks of the intermediaries that can spend it. Let p be a global constant representing the maximum number of blocks a party is allowed to stay offline between activations without becoming negligent (the latter term is explained in detail later), and s the maximum number of blocks needed for an honest transaction to enter the blockchain after being published, as in Proposition 1 of Section 11. The required timelock of a party is $p + s$ if its channel is simple, or $p + \sum_{j=2}^{n-1}(s - 1 + t_j)$ if the channel is virtual, where t_j is the required timelock of the base channel of the jth intermediary's channel. The only exception are virtual outputs that correspond to an interval that includes all parties, which can only be used as funding outputs for the virtual channel as its interval cannot be further extended, therefore the two separate spending methods and the associated timelock are dropped.

Many extend-interval and merge-intervals transactions have to be able to spend different outputs, depending on the order other base parties publish their virtual transactions. For example, P_3's extend-interval tx that extends the interval $\{P_1, P_2\}$ to $\{P_1, P_2, P_3\}$ must be able to spend both the virtual output of P_2's initiator transaction and P_2's extend-interval transaction which has spent

P_1's initiator transaction. The same issue is faced by commitment transactions of a virtual channel, as any virtual output can potentially be used as the funding ouput for the channel. In order for the received signatures for virtual and commitment txs to be valid for multiple previous outputs, the previously proposed ANYPREVOUT sighash flag [46] is needed to be added to Bitcoin. We show in Theorem ?? that variadic recursive virtual channels with $O(1)$ on-chain and subexponential number of off-chain transactions for each party cannot be constructed in Bitcoin without this flag. We hope this work provides additional motivation for this flag to be included in the future.

Note also that the newly established virtual channel can itself act as a base for further virtual channels, as its funding output can be unilaterally put on-chain in a pre-agreed maximum number of blocks. This in turn means that, as we discussed above, a further virtual channel must take the delay of its virtual base channels into account to determine the timelocks needed for its own virtual outputs.

As for the actual protocol needed to establish a virtual channel, 6 rounds of communication are needed, each starting from the funder and hop by hop reaching the fundee and back (c.f. Figure 25). The first communicates parties' identities, their funding keys and their neighbours' channel balances, the second creates new commitment transactions, the third communicates keys for virtual transactions (a.k.a. virtual keys), all parties' coins and desired timelocks, the fourth and the fifth communicate signatures for the virtual transactions (signatures for virtual outputs and funding outputs respectively) and the sixth shares revocation signatures for the old channel states.

In order to better grasp the manner in which the construction described achieves its intended goals, let us now turn to an example. Consider an established virtual channel on top of 4 preexisting simple base channels. Let A, B, C, D and E be the relevant parties, which control the (A, B), (B, C), (C, D) and (D, E) base channels, along with the (A, E) virtual channel. After carrying out some payments, A decides to close the virtual channel. It therefore publishes its initiator transaction, thus consuming the funding output of (A, B) and producing (among others) a virtual output with the interval $\{A\}$. B notices this before the timelock of the virtual output expires and publishes its extend-interval transaction that consumes the aforementioned virtual output and the funding output of (B, C), producing a virtual output with the interval $\{A, B\}$. C in turn publishes the corresponding extend-interval transaction, consuming the virtual output of B and the funding output of (C, D) while producing a virtual output with the interval $\{A, B, C\}$. Finally D publishes the last extend-interval transaction, thus producing an interval with all players. Instead of a virtual output, it produces the funding output for the virtual channel (A, E). Now A can spend this funding output with its latest commitment transaction. The entire process is depicted schematically in Figure 6. Note that if any of B, C or D does not act within the timelock prescribed in their consumed virtual output, then A or E can spend the virtual output with their latest commitment transaction, thus eventually A can close its virtual channel in all cases.
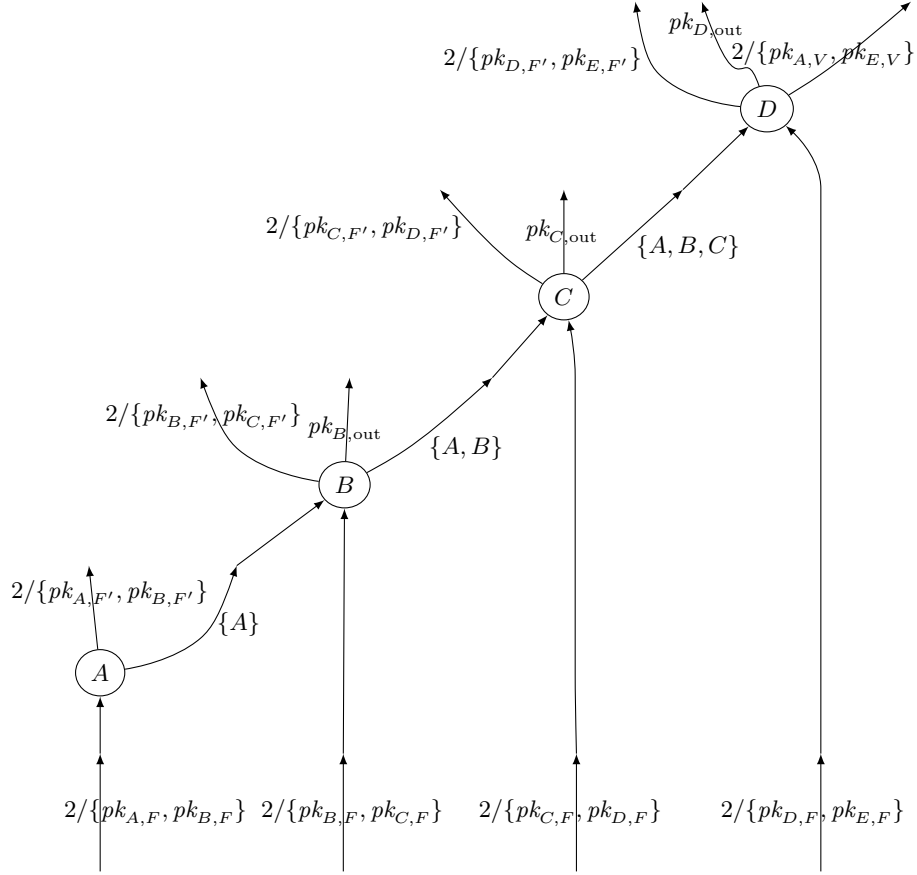
Fig. 6. 4 simple channels supporting a virtual. A initiates the closing procedure and no party is negligent. Virtual outputs are marked with their interval.

3 Preliminaries

3.1 Universal Composition Framework

In this work we embrace the Universal Composition (UC) framework [12] to model parties, network interactions, adversarial influence and corruptions, as well as formalise and prove security.

UC closely follows and expands upon the paradigm of simulation-based security [47]. For a particular real world protocol, the main goal of UC is allow us to provide a simple "interface", the ideal world functionality, that describes what the protocol achieves in an ideal way. The functionality takes the inputs of all protocol parties and knows which parties are corrupted, therefore it normally can achieve the intention of the protocol in a much more straightforward manner. At a high level, once we have the protocol and the functionality defined, our goal is to prove that no probabilistic polynomial-time (PPT) Interactive Turing Machine (ITM) can distinguish whether it is interacting with the real world protocol or the ideal world functionality. If this is true we then say that the protocol UC-realises the functionality.

The principal contribution of UC is the following: Once a functionality that corresponds to a particular protocol is found, any other higher level protocol that internally uses the former protocol can instead use the functionality. This allows cryptographic proofs to compose and obviates the need for re-proving the security of every underlying primitive in every new application that uses it, therefore vastly improving the efficiency and scalability of the effort of cryptographic proofs.

An Interactive Turing Instance (ITI) is a single instantiation of an ITM. In UC, a number of ITIs execute and send messages to each other. At each moment only one ITI is executing (has the "execution token") and when it sends a message to another ITI, it transfers the execution token to the receiver. Messages can be sent either locally (inputs, outputs) or over the network. There is no notion of time built in UC – the only requirement is that the total number of execution steps each ITI takes throughout the experiment is polynomial in the security parameter.

The first ITI to be activated is the environment $\mathcal{E}$. This can be an instance of any PPT ITM. This ITI encompasses everything that happens around the protocol under scrutiny, including the players that send instructions to the protocol. It also is the ITI that tries to distinguish whether it is in the real or the ideal world. Put otherwise, it plays the role of the distinguisher.

After activating and executing some code, $\mathcal{E}$ may input a message to any party. If this execution is in the real world, then each party is an ITI running the protocol Π. Otherwise if the execution takes place in the ideal world, then each party is a dummy that simply relays messages to the functionality $\mathcal{F}$. An activated real world party then follows its code, which may instruct it to parse its input and send a message to another party via the network.

In UC the network is fully controlled by the so called adversary $\mathcal{A}$, which may be any PPT ITI. Once activated by any network message, this machine can read

the message contents and act adaptively, freely communicate with $\mathcal{E}$ bidirectionally, choose to deliver the message right away, delay its delivery arbitrarily long, even corrupt it or drop it entirely. Crucially, it can also choose to corrupt any protocol party (in other words, UC allows adaptive corruptions). Once a party is corrupted, its internal state, inputs, outputs and execution comes under the full control of $\mathcal{A}$ for the rest of the execution. Corruptions take place covertly, so other parties do not necessarily learn which parties are corrupt. Furthermore, a corrupted party cannot become honest again.

The fact that $\mathcal{A}$ controls the network in the real world is modelled by providing direct communication channels between $\mathcal{A}$ and every other machine. This however poses an issue for the ideal world, as $\mathcal{F}$ is a single party that replaces all real world parties, so the interface has to be adapted accordingly. Furthermore, if $\mathcal{F}$ is to be as simple as possible, simulating internally all real world parties is not the way forward. This however may prove necessary in order to faithfully simulate the messages that the adversary expects to see in the real world. To solve these issues an ideal world adversary, also known as simulator $\mathcal{S}$, is introduced. This party can communicate freely with $\mathcal{F}$ and completely engulfs the real world $\mathcal{A}$. It can therefore internally simulate real world parties and generate suitable messages so that $\mathcal{A}$ remains oblivious to the fact that this is the ideal world. Normally messages between $\mathcal{A}$ and $\mathcal{E}$ are just relayed by $\mathcal{S}$, without modification or special handling.

From the point of view of the functionality, $\mathcal{S}$ is untrusted, therefore any information that $\mathcal{F}$ leaks to $\mathcal{S}$ has to be carefully monitored by the designer. Ideally it has to be as little as possible so that $\mathcal{S}$ does not learn more than what is needed to simulate the real world. This facilitates modelling privacy.

At any point during one of its activations, $\mathcal{E}$ may return a binary value (either 0 or 1). The entire execution then halts. Informally, we say that Π UC-realises $\mathcal{F}$, or equivalently that the ideal and the real worlds are indistinguishable, if $\forall$ PPT $\mathcal{A}, \exists$ PPT $\mathcal{S} : \forall$ PPT $\mathcal{E}$, the distance of the distributions over the machines' random tapes of the outputs of $\mathcal{E}$ in the two worlds is negligibly small. Note the order of quantifiers: $\mathcal{S}$ depends on $\mathcal{A}$, but not on $\mathcal{E}$.

3.2 $\mathcal{G}_{\text{Ledger}}$ Functionality

In this work we choose to model the Bitcoin ledger with the $\mathcal{G}_{\text{Ledger}}$ functionality as defined in [48] and further refined in [49]. $\mathcal{G}_{\text{Ledger}}$ formalizes an ideal data structure that is distributed and append-only, akin to a blockchain. Participants can read from $\mathcal{G}_{\text{Ledger}}$, which returns an ordered list of transactions. Additionally a party can submit a new transaction which, if valid, will eventually be added to the ledger when the adversary decides, but necessarily within a predefined time window. This property is named liveness. Once a transaction becomes part of the ledger, it then becomes visible to all parties at the discretion of the adversary, but necessarily within another predefined time window, and it cannot be reordered or removed. This is named persistence.

Moreover, $\mathcal{G}_{\text{Ledger}}$ needs the $\mathcal{G}_{\text{CLOCK}}$ functionality [50], which models the notion of time. Any $\mathcal{G}_{\text{CLOCK}}$ participant can request to read the current time (which

is initially 0) and inform $\mathcal{G}_{\text{CLOCK}}$ that her round is over. $\mathcal{G}_{\text{CLOCK}}$ increments the time by one once all parties have declared the end of their round. We further note that both $\mathcal{G}_{\text{Ledger}}$ and $\mathcal{G}_{\text{CLOCK}}$ are global functionalities [51] and therefore can be accessed directly by the environment.

3.3 Modelling time

The protocol and functionality defined in this work do not use $\mathcal{G}_{\text{CLOCK}}$ directly. Indeed, the only notion of time in our work is provided by the blockchain height, as reported by $\mathcal{G}_{\text{Ledger}}$. We therefore omit it in the statement of our lemmas and theorems for simplicity of notation; it should normally appear as a hybrid together with $\mathcal{G}_{\text{Ledger}}$.

Our protocol is fully asynchronous, i.e., the adversary can delay any network message arbitrarily long. The protocol is robust against such delays, as an honest party can unilaterally prevent loss of funds even if some of its incoming and outgoing network messages are dropped by $\mathcal{A}$, as long as the party has input-output communication with the ledger. We also note that, following the conventions of single-threaded UC execution model, the duration of local computation is not taken into account in any way (as long as it does not exceed its polynomial bound).

4 Model & Construction

4.1 Model

In this section we will examine the architecture and the details of our model, along with possible attacks and their mitigations. We follow the UCGS framework [13] to formulate the protocol and its security. We list the ideal-world global functionality $\mathcal{G}_{\text{Chan}}$ in Section 9 (Figures 7-11) and a simulator $\mathcal{S}$ (Figures 20-21), along with a real-world protocol Π_{Chan} (Figures 22-61) that UC-realizes $\mathcal{G}_{\text{Chan}}$ (Theorem 2). We give a self-contained description in this section, while pointing to figures in Sections 9 and 10, in case the reader is interested in a pseudocode style specification.

As in previous formulations, (e.g., [52]), the role of $\mathcal{E}$ corresponds to two distinct actors in a real world implementation. On the one hand $\mathcal{E}$ passes inputs that correspond to the desires of human users (e.g. open a channel, pay, close), on the other hand $\mathcal{E}$ is responsible with periodically waking up parties to check the ledger and act upon any detected counterparty misbehaviour, similar to an always-on "daemon" of real-life software that periodically nudges the implementation to perform these checks.

Since it is possible that $\mathcal{E}$ fails to wake up a party often enough, Π_{Chan} explicitly checks whether it has become "negligent" every time it is activated and all security guarantees are conditioned on the party not being negligent. A party is deemed negligent if more than p blocks have been added to $\mathcal{G}_{\text{Ledger}}$ between any consecutive pair of activations. The need for explicit negligence

checking stems from the fact that party activation is entirely controlled by $\mathcal{E}$ and no synchrony limitations are imposed (e.g. via the use of $\mathcal{G}_{\text{CLOCK}}$), therefore it can happen that an otherwise honest party is not activated in time to prevent a malicious counterparty from successfully using an old commitment transaction. If a party is marked as negligent, no balance security guarantees are given (c.f. Lemma 1). Note that in realistic software the aforementioned daemon is local and trustworthy, therefore it would never allow Π_{Chan} to become negligent, as long as the machine is powered on and in good order.

4.2 Ideal world functionality $\mathcal{G}_{\text{Chan}}$

Our ideal world functionality $\mathcal{G}_{\text{Chan}}$ represents a single channel, either simple or virtual. It acts as a relay between $\mathcal{A}$ and $\mathcal{E}$, leaking all messages. This simplifies the functionality and facilitates the indistinguishability argument by having $\mathcal{S}$ simply running internally the real world protocols of the channel parties Π_{Chan} with no modifications. Furthermore, the communication of parties with $\mathcal{G}_{\text{Ledger}}$ is handled by $\mathcal{G}_{\text{Chan}}$: when a simulated honest party in $\mathcal{S}$ needs to send a message to $\mathcal{G}_{\text{Ledger}}$, $\mathcal{S}$ instructs $\mathcal{G}_{\text{Chan}}$ to send this message to $\mathcal{G}_{\text{Ledger}}$ on this party's behalf. $\mathcal{G}_{\text{Chan}}$ internally maintains two state machines, one per channel party (c.f. Figures 13, 14, 15, 17, 16, 18, 19) that keep track of whether the parties are corrupted or negligent, whether the channel has opened, whether a payment is underway, which ITIs are to be considered *kindred* parties (as they correspond to other channels owned by the same human user, discussed below) and whether the channel is currently closing collaboratively or has already closed. The single security check performed is *whether the on-chain coins are at least equal to the expected balance once the channel closes*. If this check fails, $\mathcal{G}_{\text{Chan}}$ halts. Since the protocol Π_{Chan} (which realises $\mathcal{G}_{\text{Chan}}$, c.f. Theorems 1 and 2) never halts, this ideal world check corresponds to the security guarantee offered by Π_{Chan}. Note that this check is not performed for negligent parties, as $\mathcal{S}$ notifies $\mathcal{G}_{\text{Chan}}$ if a party becomes negligent and the latter omits the check. Thus indistinguishability between the real and the ideal world is not violated in case of negligence.

Observe that a human user may participate in various channels, therefore it corresponds to more than one ITMs. This is the case for example for the funder of a virtual channel and the corresponding party of the first base channel. Such parties are called *kindred*. They communicate locally (i.e. via inputs and outputs, without using the adversarially controlled network) and balance guarantees concern their aggregate coins. Formally this communication is modelled by having a virtual channel using its base channels as global subroutines, as defined in [13].

If we were using plain UC, the above would constitute a violation of the subroutine respecting property that functionalities have to fulfill. We leverage the concept of global functionalities put forth in [13] to circumvent the issue. More specifically, we say that a simple channel functionality is of "level" 1, which is written as $\mathcal{G}_{\text{Chan}}^1$. Inductively, a virtual channel functionality that is based on channels of any "level" up to and including $n-1$ has a "level" n, which write as $\mathcal{G}_{\text{Chan}}^n$. Then $\mathcal{G}_{\text{Chan}}^n$ is $(\mathcal{G}_{\text{Ledger}}, \mathcal{G}_{\text{Chan}}^1, \dots, \mathcal{G}_{\text{Chan}}^{n-1})$-subroutine respecting, according to the definition of [13]. The same structure is used in the real world

between protocols. This technique ensures that the necessary conditions for the validity of the functionality and the protocol are met and that the realisability proof can go through, as we will see in Section 5 in more detail.

Note that we could instead contain all the channels in a single, monolithic functionality (following the approach of [52]) and we believe that we could still carry out the security proof. Nevertheless, having the functionality correspond to a single channel has no drawbacks, as all desired security guarantees are provided by our modular architecture, and instead brings two benefits. Firstly, the functionality is easier to intuitively grasp, as it handles less tasks. Having a simple and intuitive functionality aids in its reusability and is an informal goal of the simulation-based paradigm. Secondly, this approach permits our functionality to be global, as defined in [13]. Furthermore, the ideal functionality defined in [9] is unsuitable for our case, as it requires direct access to the ledger, which is not the case for a $\mathcal{G}_{\text{Chan}}$ corresponding to a virtual channel.

4.3 Real world protocol Π_{Chan}

Our real world protocol Π_{Chan}, ran by party P, consists of two subprotocols: the Lightning-inspired part, dubbed LN (Figures 22-41) and the novel virtual layer subprotocol, named VIRT (Figures 46-61). A simple channel that is not the base of any virtual channel leverages only LN, whereas a simple channel that is the base of at least one virtual channel does leverage both LN and VIRT. A virtual channel uses both LN and VIRT.

LN subprotocol The LN subprotocol has two variations depending on whether P is the channel funder (*Alice*) or the fundee (*Bob*). It performs a number of tasks: Initialisation takes a single step for fundees and two steps for funders. LN first receives a public key $pk_{P,\text{out}}$ from $\mathcal{E}$. This is the public key that should eventually own all P's coins after the channel is closed. LN also initialises its internal variables. If P is a funder, LN waits for a second activation to generate a keypair and then waits for $\mathcal{E}$ to endow it with some coins, which will be subsequently used to open the channel (Figure 22).

After initialisation, the funder *Alice* is ready to open the channel. Once $\mathcal{E}$ gives to *Alice* the identity of *Bob*, the initial channel balance c and, (looking forward to Subsubsection 4.3) in case it is a virtual channel, the identities of the base channel owners (Figure 29), *Alice* generates and sends *Bob* her funding and revocation public keys ($pk_{A,F}$, $pk_{A,R}$, used for the funding and revocation outputs respectively) along with c, $pk_{A,\text{out}}$, and the base channel identities (if any). Given that *Bob* has been initialised, it generates funding and revocation keys and replies to *Alice* with $pk_{B,F}$, $pk_{B,R}$, and $pk_{B,\text{out}}$ (Figure 24).

The next step prepares the base channels (Figure 25) if needed. If our channel is a simple one, then *Alice* simply generates the funding tx. If it is a virtual and assuming all base parties (running LN) cooperate, a chain of messages from *Alice* to *Bob* and back via all base parties is initiated (Figures 31 and 32). These messages let each successive neighbour know the identities of all the base parties. Furthermore each party instantiates a new "host" party that runs VIRT.

It also generates new funding keys and communicates them, along with its "out" key $pk_{P,\text{out}}$ and its leftward and rightward balances. If this circuit of messages completes, *Alice* delegates the creation of the new virtual layer transactions to its new VIRT host, which will be discussed later in detail. If the virtual layer is successful, each base party is informed by its host accordingly, intermediaries return to the OPEN state (i.e., they have completed their part and are in standby, ready to accept instructions for, e.g., new payments) and *Alice* and *Bob* continue the opening procedure. In particular, *Alice* and *Bob* exchange signatures on the initial commitment transactions, therefore ensuring that the funding output can be spent (Figure 26). After that, in case the channel is simple the funding transaction is put on-chain (Figure 27) and finally $\mathcal{E}$ is informed of the successful channel opening.

There are two facts that should be noted: Firstly, in case the opened channel is virtual, each intermediary necessarily partakes in two channels. However each protocol instance only represents a party in a single channel, therefore each intermediary is in practice realised by two kindred Π_{Chan} instances that communicate locally, called "siblings". Secondly, our protocol is not designed to gracefully recover if other parties do not send an expected message at any point in the opening or payment procedure. Such anti-Denial-of-Service measures would greatly complicate the protocol and are left as a task for a real world implementation. It should however be stressed that an honest party with an open channel that has fallen victim to such an attack can still unilaterally close the channel, therefore no coins are lost in any case.

Once the channel is open, *Alice* and *Bob* can carry out an unlimited number of payments in either direction, only needing to exchange 3 direct network messages with each other per payment, therefore avoiding the slow and costly on-chain validation. The payment procedure is identical for simple and virtual channels and crucially it does not implicate the intermediaries (and therefore *Alice* and *Bob* do not incur any delays such an interaction with intermediaries would introduce). For a payment to be carried out, the payee is first notified by $\mathcal{E}$ (Figure 36) and subsequently the payer is instructed by $\mathcal{E}$ to commence the payment (Figure 35).

TODO: if this limitation is removed, change paragraph. If the channel is virtual, each party also checks that its upcoming balance is lower than the balance of its sibling's counterparty and that the upcoming balance of the counterparty is higher than the balance of its own sibling, otherwise it rejects the payment. This is to mitigate a "griefing" attack (i.e. one that does not lead to financial gain) where a malicious counterparty uses an old commitment transaction to spend the base funding output, therefore blocking the honest party from using its initiator virtual transaction. This check ensures that the coins gained by the punishment are sufficient to cover the losses from the blocked initiator transaction. If the attack takes place, other local channels based directly or indirectly on it are informed and are moved to a failed state. Note that this does not bring a risk of losing any of the total coins of all local channels. We conjecture that

this balance constraint can be lifted if the current Lightning-inspired payment method is replaced with an eltoo-inspired one [22].

Subsequently each of the two parties builds the new commitment transaction of its counterparty, signs it and sends over the signature, then the revocation transactions for the previously valid commitment transactions are generated, signed and the signatures are exchanged. To reduce the number of messages, the payee sends the two signatures in one message. This does not put it at risk of losing funds, since the new commitment transaction (for which it has already received a signature and therefore can spend) gives it more funds than the previous one.

Π_{Chan} also checks the chain for outdated commitment transactions by the counterparty and publishes the corresponding revocation transaction in case one is found (Figure 38). It also keeps track of whether the party is activated often enough and marks it as negligent otherwise (Figure 22). In particular, at the beginning of every activation while the channel is open, LN checks if the party has been activated within the last p blocks (where p is an implementation-dependent global constant) by reading from $\mathcal{G}_{\text{Ledger}}$ and comparing the current block height with that of the last activation.

TODO: Add paragraph explaining steps for cooperative closure

When either party is instructed by $\mathcal{E}$ to unilaterally close the channel (Figure 40), it first asks its host to unilaterally close (details on the exact steps are discussed later) and once that is done, the ledger is checked for any transaction spending the funding output. In case the latest remote commitment tx is on-chain, then the channel is already closed and no further action is necessary. If an old committment transaction is on-chain, the corresponding revocation transaction is used for punishment. If the funding output is still unspent, the party attempts to publish the latest commitment transaction after waiting for any relevant timelock to expire. Until the funding output is irrevocably spent, the party still has to periodically check the blockchain and again be ready to use a revocation transaction if an old commitment transaction spends the funding output after all (Figure 38).

VIRT subprotocol This subprotocol acts as a mediator between the base channels and the Lightning-based logic. Put otherwise, its responsibility is putting on-chain the funding output of the channel when needed. When first initialised by a machine that executes the LN subprotocol (Figure 46), it learns and stores the identities, keys, and balances of various relevant parties, along with the required timelock and other useful data regarding the base channels. It then generates a number of keys as needed for the rest of the base preparation. If the initialiser is also the channel funder, then the VIRT machine initiates 4 "circuits" of messages. Each circuit consists of one message from the funder P_1 to its neighbour P_2, one message from each intermediary P_i to the "next" neighbour P_{i+1}, one message from the fundee P_n to its neighbour P_{n-1} and one more message from each intermediary P_i to the "previous" neighbour P_{i-1}, for a total of $2 \cdot (n-1)$ messages per circuit.

The first circuit (Figure 47) communicates all "out", virtual, revocation and funding keys (both old and new), all balances and all timelocks among all parties. In the second circuit (Figure 54) every party receives and verifies all signatures for all inputs of its virtual transactions that spend a virtual output. It also produces and sends its own such signatures to the other parties. Each party generates and circulates $S = \sum_{i=2}^{n-2}(n-3+\chi_{i=2}+\chi_{i=n-1}+2(i-2+\chi_{i=2})(n-i-1+\chi_{i=n-1})) \in O(n^3)$ signatures (where χ_A is the characteristic function that equals 1 if A is true and 0 else), which is derived by calculating the total number of virtual outputs of all parties' virtual transactions – we remind that each virtual output can be spent by a n-of-n multisig. On a related note, the number of virtual transactions stored by each party is 1 for the two endpoints (Figure 49) and $n-2+\chi_{i=2}+\chi_{i=n-1}+(i-2+\chi_{i=2})(n-i-1+\chi_{i=n-1}) \in O(n^2)$ for the i-th intermediary (Figure 48). The latter is derived by counting the number of extend-interval and merge-intervals transactions held by the intermediary, which are equal to the number of distinct intervals that the party can extend and the number of distinct pairs of intervals that the party can merge respectively, plus 1 for the unique initiator transaction of the party. The third circuit concerns sharing signatures for the funding outputs (Figure 55). Each party signs all transactions that spend a funding output relevant to the party, i.e. the initiator transaction and some of the extend-interval transactions of its neighbours. The two endpoints send 2 signatures each when $n = 3$ and $n-2$ signatures each when $n > 3$, whereas each intermediary sends $2+\chi_{i+1<n}(n-2+\chi_{i=n-2})+\chi_{i-1>1}(n-2+\chi_{i=3}) \in O(n)$ signatures each. The last circuit of messages (Figure 56) carries the revocations of the previous states of all base channels. After this, base parties can only use the newly created virtual transactions to spend their funding outputs. In this step each party exchanges a single signature with each of its neighbours.

When VIRT is instructed to unilaterally close by party R (Figure 58), it first notifies its VIRT host (if any) and waits for it to unilaterally close. After that, it signs and publishes the unique valid virtual transaction. It then repeatedly checks the chain to see if the transaction is included (Figure 59). If it is included, the virtual layer is closed and VIRT informs (i.e. outputs (CLOSED) to) R. The instruction to close has to be received potentially many times, because a number of virtual transactions (the ones that spend the same output) are mutually exclusive and therefore if another base party publishes an incompatible virtual transaction contemporaneously and that remote transaction enters the chain, then our VIRT party has to try again with another, compatible virtual transaction. TODO: The next lifts the balance limitation, so remove it if it isn't fixed after all. It however may be also relevant for the collaborative closing. Check this again If any neighbour has spent the common funding output with an old, revoked state, VIRT creates a new virtual output (e.g. the one that would be produced by the malicious neighbour's intiator transaction) using the coins from the punishment of the revoked state. VIRT can then use its own virtual transaction

as if its neighbour were honest. The `ANYPREVOUT` flag ensures that this newly generated output can be spent by VIRT's virtual transaction.

5 Security

The first step to formally arguing about the security of Elmo is to clearly delineate the exact security guarantees it provides. To that end, we first prove two similar claims regarding the conservation of funds in the real and ideal world, Lemmas 4 and 5 respectively. Informally, the first establishes that if an honest, non-negligent party was implicated in a channel on which a number of payments took place and has now been unilaterally closed, then the party will have at least the expected funds on-chain.

Lemma 1 (Real world balance security (informal)). *Consider a real world execution with $P \in \{Alice, Bob\}$ honest LN ITI. Assume that all of the following are true:*

- *P is not negligent,*
- *P successfully opened the channel, with initial balance c ($c = 0$ if P is a fundee)*
- *P is the host of n channels, where the i-th channel is funded with f_i coins,*
- *P has successfully sent m payments, where the i-th payment involved d_i coins,*
- *P has successfully received l payments, where the i-th payment involved c_i coins.*

If P unilaterally closes its channel, eventually its view of $\mathcal{G}_{\text{Ledger}}$ will contain h outputs spendable only by P or another kindred party, each of value c_i, such that

$$\sum_{i=1}^{h} c_i \geq c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \ . \tag{1}$$

The formal statement, Lemma 4, is deferred to the Appendix.

TODO: maybe add another lemma for the case of cooperative closing

The second lemma states that for an ideal party in a similar situation, the balance that $\mathcal{G}_{\text{Chan}}$ has stored for it is at least equal to the expected funds.

Lemma 2 (Ideal world balance (informal)). *Consider an ideal world execution with functionality $\mathcal{G}_{\text{Chan}}$ and simulator $\mathcal{S}$. Let $P \in \{Alice, Bob\}$ one of the two parties of $\mathcal{G}_{\text{Chan}}$. Assume that all of the following are true:*

- *P is not corrupted or negligent, nor the any member of the transitive closure of its hosts has published a revocation transaction,*
- *P successfully opened the channel, with initial balance c ($c = 0$ if P is a fundee)*
- *P is the host of n channels, where the i-th channel is funded with f_i coins,*
- *P has successfully sent m payments, where the i-th payment involved d_i coins,*

– P has successfully received l payments, where the i-th payment involved c_i coins.

Let $balance_P$ be the balance that $\mathcal{G}_{\text{Chan}}$ stores internally for P. If the channel is closed (either unilaterally or cooperatively TODO: check that this is true and relevant), then the following holds with overwhelming probability on the security parameter:

$$balance_P = c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \tag{2}$$

The formal statement, Lemma 5, is deferred to the Appendix.

In both cases the expected funds are [initial balance - funds for supported virtuals - outbound payments + inbound payments]. Note that the funds for supported virtuals only refer to those funds used by the funder of the virtual channel, not the rest of the base parties.

Both proofs follow the various possible execution paths, keeping track of the resulting balance in each case and coming to the conclusion that balance is secure in all cases, except if signatures are forged.

It is important to note that in fact Π_{Chan} provides a stronger guarantee, namely that an honest, non-negligent party with an open channel can unilaterally close it and obtain the expected funds on-chain within a known number of blocks, given that $\mathcal{E}$ sends the necessary "daemon" messages. This stronger guarantee is sufficient to make this construction reliable enough for real-world applications. However a corresponding ideal world functionality with such guarantees would have to be aware of the specific transactions and signatures, therefore it would be essentially as complicated as the protocol, thus violating the spirit of the simulation-based security paradigm.

Subsequently we prove Lemma 3, which informally states that if an ideal party and all its kindred parties are honest, then $\mathcal{G}_{\text{Chan}}$ does not halt with overwhelming probability.

Lemma 3 (No halt). *In an ideal execution with $\mathcal{G}_{\text{Chan}}$ and $\mathcal{S}$, if the kindred parties of the honest parties of $\mathcal{G}_{\text{Chan}}$ are themselves honest, then the functionality halts with negligible probability in the security parameter (i.e. l. 21 of Fig. 11 is executed negligibly often).*

This is proven by first arguing that if the conditions of Lemma 5 for the ideal world hold, then the conditions of Lemma 4 also hold for the equivalent real world execution, therefore in this case $\mathcal{G}_{\text{Chan}}$ does not halt. We then argue that also in case the conditions of Lemma 5 do not hold, $\mathcal{G}_{\text{Chan}}$ may never halt as well, therefore concluding the proof.

A salient observation regarding an instance s of Π_{Chan} is that, in order to open a virtual channel, it passes inputs to another Π_{Chan} instance s' that belongs to a different extended session. This means that s (and therefore Π_{Chan}) is not *subroutine respecting*, as defined in [12]. To address this issue, we first annotate Π_{Chan} with a numeric superscript, i.e. Π_{Chan}^n. Π_{Chan}^1 is always a simple (i.e. on-chain) channel. To achieve this, Π_{Chan} undergoes a modification under which it

ignores all (OPEN, x, hops $\neq$ "ledger", ...) messages. Likewise we define $\mathcal{G}^1_{\text{Chan}}$ as a version of $\mathcal{G}_{\text{Chan}}$ that ignores (OPEN, x, hops $\neq$ "ledger", ...) messages. As for the rest of the superscripts, $\forall n \in \mathbb{N}^*, \Pi^{n+1}_{\text{Chan}}$ is a virtual channel protocol Π_{Chan} of which the base channels have a maximum superscript n. It then holds that $\forall n \in \mathbb{N}^*, \Pi^n_{\text{Chan}}$ is $(\mathcal{G}_{\text{Ledger}}, \Pi^1_{\text{Chan}}, \ldots, \Pi^{n-1}_{\text{Chan}})$-subroutine respecting, as defined in [13]. Likewise, $\mathcal{G}^{n+1}_{\text{Chan}}$ is a virtual channel functionality $\mathcal{G}_{\text{Chan}}$ of which the base channels have a maximum superscript n. It then holds that $\forall n \in \mathbb{N}^*, \mathcal{G}^n_{\text{Chan}}$ is $(\mathcal{G}_{\text{Ledger}}, \mathcal{G}^1_{\text{Chan}}, \ldots, \mathcal{G}^{n-1}_{\text{Chan}})$-subroutine respecting.

We now formulate and prove Theorem 1, which states that Π^1_{Chan} UC-realises $\mathcal{G}^1_{\text{Chan}}$.

Theorem 1 (Simple Payment Channel Security). *The protocol Π^1_{Chan} UC-realises $\mathcal{G}^1_{\text{Chan}}$ in the presence of a global functionality $\mathcal{G}_{\text{Ledger}}$ and assuming the security of the underlying digital signature. Specifically,*

$$\forall \; PPT \; \mathcal{A}, \exists \; PPT \; \mathcal{S} : \forall \; PPT \; \mathcal{E} \; it \; is \; \text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi^1_{\text{Chan}}, \mathcal{A}, \mathcal{E}} \approx \text{EXEC}^{\mathcal{G}^1_{\text{Chan}}, \mathcal{G}_{\text{Ledger}}}_{\mathcal{S}, \mathcal{E}}$$

The corresponding proof is a simple application of Lemma 3, the fact that $\mathcal{G}_{\text{Chan}}$ is a simple relay and that $\mathcal{S}$ faithfully simulates Π_{Chan} internally.

Proof (Proof of Theorem 1). By inspection of Figures 7 and 20 we can deduce that for a particular $\mathcal{E}$, in the ideal world execution $\text{EXEC}^{\mathcal{G}^1_{\text{Chan}}, \mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\mathcal{A}}, \mathcal{E}}$, $\mathcal{S}_{\mathcal{A}}$ simulates internally the two Π^1_{Chan} parties exactly as they would execute in $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi^1_{\text{Chan}}, \mathcal{A}, \mathcal{E}}$, the real world execution, in case $\mathcal{G}^1_{\text{Chan}}$ does not halt. Indeed, $\mathcal{G}^1_{\text{Chan}}$ only halts with negligible probability according to Lemma 3, therefore the two executions are computationally indistinguishable.

Lastly we prove that $\forall$ integers $n \geq 2, \Pi^n_{\text{Chan}}$ UC-realises $\mathcal{G}^n_{\text{Chan}}$ in the presence of $\mathcal{G}^1_{\text{Chan}}, \ldots, \mathcal{G}^{n-1}_{\text{Chan}}$ (leveraging the relevant definition from [13]).

Theorem 2 (Recursive Virtual Payment Channel Security). $\forall n \in \mathbb{N}^* \setminus \{1\}$, *the protocol Π^n_{Chan} UC-realises $\mathcal{G}^n_{\text{Chan}}$ in the presence of $\mathcal{G}^1_{\text{Chan}}, \ldots, \mathcal{G}^{n-1}_{\text{Chan}}$ and $\mathcal{G}_{\text{Ledger}}$, assuming the security of the underlying digital signature. Specifically,*

$$\forall n \in \mathbb{N}^* \setminus \{1\}, \forall \; PPT \; \mathcal{A}, \exists \; PPT \; \mathcal{S} : \forall \; PPT \; \mathcal{E} \; it \; is$$

$$\text{EXEC}^{\mathcal{G}_{\text{Ledger}}, \mathcal{G}^1_{\text{Chan}}, \ldots, \mathcal{G}^{n-1}_{\text{Chan}}}_{\Pi^n_{\text{Chan}}, \mathcal{A}, \mathcal{E}} \approx \text{EXEC}^{\mathcal{G}^n_{\text{Chan}}, \mathcal{G}_{\text{Ledger}}}_{\mathcal{S}, \mathcal{E}}$$

Proof (Proof of Theorem 2). The proof is exactly the same as that of Theorem 1, replacing superscripts 1 for n.

Formal proofs for the three lemmas can be found in Section 12.

24

6 Efficiency Evaluation

We offer here a comparison of this work with LVPC [10], BCVC [8] and Donner [45] in terms of communication efficiency when opening and updating aggregated for all parties TODO: decide on previous. We also compare the maximum on-chain cost for an endpoint to unilaterally close its virtual channel. Furthermore, we compare the maximum on-chain cost for an intermediary to close its base channel. In order to only show the costs caused by supporting a virtual channel, we subtract the cost the intermediary would pay to close its channel if it was not supporting any virtual channel. Lastly, we compare the maximum total on-chain cost, aggregated over all parties. On-chain cost is measured in terms of size[3] and number of transactions. The comparison is performed both for channels of length 2 and for channels of length n. In the second comparison BCVC is not included, since only LVPC and Donner offer a way to open virtual channels of length greater than 2 (since LVPC is recursive and Donner is variadic). For a virtual channel between P_1 and P_n over $n-1$ base channels via LVPC, we consider the case in which the funder P_1 initially has a channel with P_2 and then opens one virtual channel with party P_i on top of its channel with party P_{i-1} for $i \in \{3, \ldots, n\}$. We choose this topology, as P_1 cannot assume that there exist any virtual channels between other parties (which could be used as shortcuts).

BCVC proposes 4 configurations, each with different efficiency characteristics: over Generalised Channels [9] (GC) or Lightning, and with or without validity (V or NV respectively); we include all of them here. We refer the reader to [8] for more details.

Since all constructions produce pairwise channels, updates (a.k.a. payments) always implicate exactly two parties (the payer and the payee TODO: change if who is funder/fundee also plays a role for any of the constructions) and their interaction is independent of the number of underlying channels. Therefore the entries in Table 2 also cover the general case of n underlying channels. Note that the update-related storage requirements of Table 2 take into account savings that may arise by deleting old unneeded data at the end of an update. TODO: change previous if for some reason we do something else

Some results disagree with those of the tables found in related work, due to differences in the counting method.

In Table 3 we compare the resources needed to open a new virtual channel both for each party individually and for all parties in total, in terms of communication rounds (calculated as [#incoming messages + #outgoing messages]/2, for each party separately) and amount of data sent and stored. The data is counted as the sum of the relevant channel identifiers (8 bytes each, as defined by the Lightning Network specification[4]), transaction output identifiers (36 bytes), se-

---

[3] Transaction size is calculated in so-called "virtual bytes", which map directly to on-chain fees and thus are preferred to raw bytes. We used the tool found in `https://jlopp.github.io/bitcoin-transaction-size-calculator/` to aid size calculation.

[4] `https://github.com/lightning/bolts/blob/master/07-routing-gossip.md#definition-of-short_channel_id`

Table 2. Efficiency comparison of virtual channel protocols with 3 parties

	Open		Update				comm.	Close	
	#txs	size	Payer		Payee		rounds	#txs	size
	TODO	TODO	sent	stored	sent	stored		TODO	TODO
LVPC	14	100000	173	0	173	0	2	10	100000
BCVC-GC-NV	7	2829	fill	695 check	fill	695 check	fill	7	2829
BCVC-GC-V	8	2803	fill	695 check	fill	695 check	fill	6	2108
BCVC-LN-NV	16	7704	fill	2824 check	fill	2824 check	fill	3	1800
BCVC-LN-V	14	5722	fill	1412 check	fill	1412 check	fill	4	1778
Donner	14	4721	fill	695 check	fill	695 check	fill	10	3438
Elmo	1000	100000	fill	fill	fill	fill	fill	1000	100000

cret keys (32 bytes each), public keys (33 bytes each, compressed form – these double as party identifiers), signatures (71 bytes each), coins (8 bytes each), times and timelocks (both 4 bytes each). UC-specific data is ignored. TODO: add "auxiliary data", or specifically what other kind of data is counted, if needed TODO: see if any optimisations are used by other papers, decide if it makes sense to ignore them and change the next phrase Optimisations are not considered.

In LVPC, every intermediary, apart from the first one, acts both as a fundee in a new virtual channel with the funder and as an intermediary in the funder's virtual channel with the party after said intermediary. In Table 3 the intermediary columns contain the total cost of any intermediary that is not the first one, therefore the first intermediary (the party after the funder) incurs [intermediary's costs - fundee's costs] for all three measured quantities.

Table 3. Open efficiency comparison of virtual channel protocols with n parties

	Open										
	Funder			Fundee			Intermediary			Total	
	comm. rounds	size		comm. rounds	size		comm. rounds	size		size	
		sent	stored		sent	stored		sent	stored	sent	stored
LVPC	$8(n-2)$	$1381(n-2)$	$3005(n-2)$	7	1254	2936	16	2989	6385	$4370n - 8740$	$9390n - 18780$
Donner	$n!$	$1000n$	$n!$	$1000n$	$n!$	$1000n$	$n!$	$1000n$	$n!$	$n!$	$1000n$
Elmo	$n!$	$1000n$	$n!$	$1000n$	$n!$	$1000n$	$n!$	$1000n$	$n!$	$n!$	$1000n$

The on-chain number of transactions to close a virtual channel in the case of LVPC is calculated as follows: One "split" transaction is needed for each base channel ($n-1$ in total), plus one "merge" transaction per virtual channel ($n-2$ in total), plus a single "refund" transaction for the virtual channel, for a total of $2n-2$ transactions.

We note that the linear size factor when the intermediary closes can, in the case of our work, be eliminated with the aid of Schnorr signatures (recently made available on Bitcoin). This is so because the n signatures that are needed to spend each virtual output can be shrunk down to a single aggregate signature without compromising security. For the same reason, Schnorr signatures help

Table 4. On-chain worst-case closing efficiency comparison of virtual channel protocols with n parties

		Close						
	Intermediary		Funder		Fundee		Total	
	#txs	size	#txs	size	#txs	size	#txs	size
LVPC	3	626.25	2	383	2	359	$2n-2$	$434.75n - 510.5$
Donner	1	204.5	4	$704 + 43n$	1	136.5	$2n$	$458n - 26$
Elmo	1	$263.75 + 26.75n$	2	371	2	371	n	$26.75n^2 + 210.25n$ -156.5

eliminate the quadratic term and reduce the linear term of the total on-chain cost. The same cannot be said for the linear factor in Donner, since there is currently no way to optimise away the n outputs of the funder's transaction $\mathtt{tx}^{\mathrm{vc}}$.

7 Discussion and Future work

A number of features can be added to our protocol for additional efficiency, usability and flexibility. First of all, a new subprotocol for cooperatively closing a virtual channel can be created. In the optimistic case, a virtual channel would then be closed with no on-chain transactions and its base channels would become independent once again. To achieve this goal, cooperation is needed between all base parties of the virtual channel and possibly parties implicated in other virtual channels that use the same base channels.

In our current construction, each time a particular channel C acts as a base channel for a new virtual channel, one more "virtualisation layer" is added. When one of its owners wants to close C, it has to put on-chain as many transactions as there are virtualisation layers. Also the timelocks associated with closing a virtual channel increase with the number of virtualisation layers of its base channels. Both these issues can be alleviated by extending the opening subprotocol with the ability to cooperatively open multiple virtual channels in the same layer, either simultaneously or as an amendment to an existing virtualisation layer.

Due to the possibility of the griefing attack discussed in Subsubsection 4.3, the range of balances a virtual channel can support is limited by the balances of neighbouring channels. We believe that this limitation can be lifted if instead of using a Lightning-based construction for the payment layer, we instead replace it with an eltoo-based [22] construction. Since in eltoo a maliciously published old state can be simply re-spent by the honest latest state, the griefing attack is completely avoided. What is more, our protocol shares with eltoo the need for the ANYPREVOUT sighash flag, therefore no additional requirements from the Bitcoin protocol would be added by this change. Lastly, due to the separation of intermediate layers with the payment layer in our pseudocode implementation as found in Section 10 (i.e. the distinction between the LN and the VIRT protocols),

this change should in principle not need extensive changes in all parts of the protocol.

As it currently stands, the timelocks calculated for the virtual channels are based on p (Figure 22) and s (Figure 26), which are global constants that are immutable and common to all parties. The parameter s stems from the liveness guarantees of Bitcoin, as discussed in Proposition 1 and therefore cannot be tweaked. However, p represents the maximum time (in blocks) between two activations of a non-negligent party, so in principle it is possible for the parties to explicitly negotiate this value when opening a new channel and even renegotiate it after the channel has been opened if need be. We leave this usability-augmenting protocol feature as future work.

As we mentioned earlier, our protocol is not designed to "gracefully" recover from a situation in which halfway through a subprotocol, one of the counterparties starts misbehaving. Currently the only solution is to unilaterally close the channel. This however means that DoS attacks (that still do not lead to financial losses) are possible. A practical implementation of our protocol would need to expand the available actions and states to be able to transparently and gracefully recover from such problems, avoiding closing the channel where possible, especially when the problem stems from network issues and not from malicious behaviour.

Furthermore, any deployment of the protocol has to explicitly handle the issue of transaction fees. These include miner fees for on-chain transactions and intermediary fees for the parties that own base channels and facilitate opening virtual channels. These fees should take into account the fact that each intermediary has quadratic storage requirements, whereas endpoints only need constant storage, creating an opportunity for amplification attacks. Our protocol is compatible with any such fee parameterization and we leave for future work the incentive analyses that can determine concrete values for such intermediary fees.

In order to increase readability and to keep focus on the salient points of the construction, our protocol does not exploit a number of possible optimisations. These include a number of techniques employed in Lightning that drastically reduce storage requirements, along with a variety of possible improvements to our novel virtual subprotocol. Most notably, the Taproot [53] update that is planned for Bitcoin will allow for a drastic reduction in the size of transactions, as in the optimistic case only the hash of the Script has to be added to the blockchain and the n signatures needed to spend a virtual output can be replaced with their aggregate, resulting in constant size storage. As this work is mainly a proof of feasibility, we leave these optimisations as future work.

Additionally, our protocol does not feature one-off multi-hop payments like those possible in Lightning. This however is a useful feature in case two parties know that they will only transact once, as opening a virtual channel needs substantially more network communication than performing an one-off multi-hop payment. It would be therefore fruitful to also enable the multi-hop payment technique used in Lightning and allow human users to choose which method to use in each case.

Moreover, the result of Theorem **??** excludes a large class of variadic recursive protocols that do not make use of ANYPREVOUT from achieving practical performance, but it does not preclude the existence of such protocols. Specifically, there may be some as of yet unknown protocol technique that allows parties to generate only the transactions that they need to put on-chain during the closing procedure, from a master secret-key that has been received when opening. This would permit parties to circumvent the need for exchanging and storing an exponential number of signatures and transactions even without ANYPREVOUT; we note that the theorem is not invalidated: there are still exponentially many signatures that are required to be accessible. It is just that there is a way to compress the information needed to generate them in the state of each party. The existence of such state compression techniques is left as an interesting future direction.

Last but not least, the current analysis gives no privacy guarantees for the protocol, as it does not employ onion packets [54] like Lightning. Furthermore, $\mathcal{G}_{\mathrm{Chan}}$ leaks all messages to the ideal adversary therefore theoretically no privacy is offered at all. Nevertheless, onion packets can be incorporated in the current construction and intuitively our construction leaks less data than Lightning for the same multi-hop payments, as intermediaries in our case do not learn the new balance after every payment, contrary to Lightning. Therefore a future extension can improve the privacy of the construction and formally demonstrate exact privacy guarantees.

8   Conclusion

In this work we presented Recursive Virtual Payment Channels for Bitcoin, a construction which enables the establishment and optimistic teardown of pairwise payment channels without the need for posting on-chain transactions. Such a channel can be opened over a path of consecutive base channels of arbitrary length, i.e., the virtual channel constructor is variadic.

The base channels themselves can be virtual, therefore the novel recursive nature of the construction. A key performance characteristic of our construction is that it has optimal round complexity for on-chain channel closing: a single transaction is required by any participant to turn the virtual channel into a simple one and one more transaction is needed to close it, be it an end-point or an intermediary.

We formally described the protocol in the UC setting, provided a corresponding ideal functionality and simulator and finally proved the indistinguishability of the protocol and functionality, along with the balance security properties that ensure no loss of funds for honest, non-negligent parties. This is achieved through the use of the ANYPREVOUT sighash flag, which is a proposed feature for Bitcoin, also required by the eltoo improvement to lightning [22].

9 Functionality & Simulator

Functionality $\mathcal{G}_{\text{Chan}}$ – general message handling rules

- On receiving input (msg) by $\mathcal{E}$ addressed to $P \in \{Alice, Bob\}$, handle it according to the corresponding rule in Fig. 8, 9, 10, 11 or 12 (if any) and subsequently send (RELAY, msg, P, $\mathcal{E}$, input) to $\mathcal{A}$.
- On receiving (msg) by party R addressed to $P \in \{Alice, Bob\}$ by means of mode $\in \{\text{output}, \text{network}\}$, handle it according to the corresponding rule in Fig. 8, 9, 10, 11 or 12 (if any) and subsequently send (RELAY, msg, P, $\mathcal{E}$, mode) to $\mathcal{A}$. // all messages are relayed to $\mathcal{A}$
- On receiving (RELAY, msg, P, R, mode) by $\mathcal{A}$ (mode $\in \{\text{input}, \text{output}, \text{network}\}$, $P \in \{Alice, Bob\}$), relay msg to R as P by means of mode. // $\mathcal{A}$ fully controls outgoing messages by $\mathcal{G}_{\text{Chan}}$
- On receiving (INFO, msg) by $\mathcal{A}$, handle (msg) according to the corresponding rule in Fig. 8, 9, 10, 11 or 12 (if any). After handling the message or after an "ensure" fails, send (HANDLED, msg) to $\mathcal{A}$. // (INFO, msg) messages by $\mathcal{S}$ always return control to $\mathcal{S}$ without any side-effect to any other ITI, except if $\mathcal{G}_{\text{Chan}}$ halts
- $\mathcal{G}_{\text{Chan}}$ keeps track of two state machines, one for each of *Alice*, *Bob*. If there are more than one suitable rules for a particular message, or if a rule matches the message for both parties, then both rule versions are executed. // the two rules act on different state machines, so the order of execution does not matter

---

Fig. 7.

Note that in UCGS [13], just like in UC, every message to an ITI may arrive via one of three channels: input, output and network. In the session of interest, input messages come from the environment $\mathcal{E}$ in the real world, whereas in the ideal world each input message comes from the corresponding dummy party, which forwards it as received by $\mathcal{E}$. Outputs may be received from any subroutine (local or global). This means that the "sender field" of inputs and outputs cannot be tampered with by $\mathcal{E}$ or $\mathcal{A}$. Network messages only come from $\mathcal{A}$; they may have been sent from any machine but are relayed (and possibly delayed, reordered, modified or even dropped) by $\mathcal{A}$. Therefore, in contrast to inputs and outputs, network messages may have a tampered "sender field".

Functionality $\mathcal{G}_{\text{Chan}}$ – open state machine, $P \in \{Alice, Bob\}$

1: On first activation: // before handing the message
2:      $pk_P \leftarrow \bot$; $\text{balance}_P \leftarrow 0$; $State_P \leftarrow \text{UNINIT}$
3:      $\text{enabler}_P \leftarrow \bot$ // if we are a virtual channel, the ITI of P's base channel
4:      $\text{host}_P \leftarrow \bot$ // if we are a virtual channel, the ITI of the common host of this channel and P's base channel

---

5: On (BECAME CORRUPTED OR NEGLIGENT, P) by $\mathcal{A}$ or on output (ENABLER USED REVOCATION) TODO: if balance limit fixed, remove by $\texttt{host}_P$ when in any state:
6:     $State_P \leftarrow$ IGNORED

7: On (INIT, pk) by P when $State_P =$ UNINIT:
8:     $pk_P \leftarrow pk$
9:     $State_P \leftarrow$ INIT

10: On (OPEN, x, "$\texttt{ledger}$", ...) by $Alice$ when $State_A =$ INIT:
11:     store x
12:     $State_A \leftarrow$ TENTATIVE BASE OPEN

13: On (BASE OPEN) by $\mathcal{A}$ when $State_A =$ TENTATIVE BASE OPEN:
14:     $\texttt{balance}_A \leftarrow x$
15:     $State_A \leftarrow$ OPEN

16: On (BASE OPEN) by $\mathcal{A}$ when $State_B =$ INIT:
17:     $State_B \leftarrow$ OPEN

18: On (OPEN, x, $\texttt{hops} \neq$ "$\texttt{ledger}$", ...) by $Alice$ when $State_A =$ INIT:
19:     store x
20:     $\texttt{enabler}_A \leftarrow \texttt{hops}[0].\texttt{left}$
21:     add $\texttt{enabler}_A$ to $Alice$'s kindred parties
22:     $State_A \leftarrow$ PENDING VIRTUAL OPEN

23: On output (FUNDED, $\texttt{host}$, ...) to $Alice$ by $\texttt{enabler}_A$ when $State_A =$ PENDING VIRTUAL OPEN:
24:     $\texttt{host}_A \leftarrow \texttt{host}[0].\texttt{left}$
25:     $State_A \leftarrow$ TENTATIVE VIRTUAL OPEN

26: On output (FUNDED, $\texttt{host}$, ...) to Bob by ITI $R \in \{\mathcal{G}_{\text{Chan}}, \text{LN}\}$ when $State_B =$ INIT:
27:     $\texttt{enabler}_B \leftarrow R$
28:     add $\texttt{enabler}_B$ to Bob's kindred parties
29:     $\texttt{host}_B \leftarrow \texttt{host}$
30:     $State_B \leftarrow$ TENTATIVE VIRTUAL OPEN

31: On (VIRTUAL OPEN) by $\mathcal{A}$ when $State_P =$ TENTATIVE VIRTUAL OPEN:
32:     **if** $P = Alice$ **then** $\texttt{balance}_P \leftarrow x$
33:     $State_P \leftarrow$ OPEN

Fig. 8. State machine in Fig. 13, 14, 15 and 19

Functionality $\mathcal{G}_{\text{Chan}}$ – payment state machine, $P \in \{Alice, Bob\}$

1: On (PAY, x) by P when $State_P$ = OPEN: // P pays $\bar{P}$
2:     store x
3:     $State_P \leftarrow$ TENTATIVE PAY

4: On (PAY) by $\mathcal{A}$ when $State_P$ = TENTATIVE PAY: // P pays $\bar{P}$
5:     $State_P \leftarrow$ (SYNC PAY, x)

6: On (GET PAID, y) by P when $State_P$ = OPEN: // $\bar{P}$ pays P
7:     store y
8:     $State_P \leftarrow$ TENTATIVE GET PAID

9: On (PAY) by $\mathcal{A}$ when $State_P$ = TENTATIVE GET PAID: // $\bar{P}$ pays P
10:     $State_P \leftarrow$ (SYNC GET PAID, x)

11: When $State_P$ = (SYNC PAY, x):
12:     **if** $State_{\bar{P}} \in \{$IGNORED, (SYNC GET PAID, x)$\}$ **then**
13:         balance$_P \leftarrow$ balance$_P - x$
14:         // if $\bar{P}$ honest, this state transition happens simultaneously with l. 21
15:         $State_P \leftarrow$ OPEN
16:     **end if**

17: When $State_P$ = (SYNC GET PAID, x):
18:     **if** $State_{\bar{P}} \in \{$IGNORED, (SYNC PAY, x)$\}$ **then**
19:         balance$_P \leftarrow$ balance$_P + x$
20:         // if $\bar{P}$ honest, this state transition happens simultaneously with l. 15
21:         $State_P \leftarrow$ OPEN
22:     **end if**

Fig. 9. State machine in Fig. 16

Functionality $\mathcal{G}_{\text{Chan}}$ – funding state machine, $P \in \{Alice, Bob\}$

1: On input (FUND ME, x, . . .) by ITI $R \in \{\mathcal{G}_{\text{Chan}}, \text{LN}\}$ when $State_P$ = OPEN:
2:     store x
3:     add R to P's kindred parties
4:     $State_P \leftarrow$ PENDING FUND

5: When $State_P$ = PENDING FUND:

32

6:    **if** we intercept the command "define new VIRT ITI host" by $\mathcal{A}$, routed through P **then**
7:        store host
8:        $State_P \leftarrow$ TENTATIVE FUND
9:        continue executing $\mathcal{A}$'s command
10:    **end if**

11: On (FUND) by $\mathcal{A}$ when $State_P =$ TENTATIVE FUND:
12:    $State_P \leftarrow$ SYNC FUND

13: When $State_P =$ OPEN:
14:    **if** we intercept the command "define new VIRT ITI host" by $\mathcal{A}$, routed through P **then**
15:        store host
16:        $State_P \leftarrow$ TENTATIVE HELP FUND
17:        continue executing $\mathcal{A}$'s command
18:    **end if**
19:    **if** we receive a RELAY message with $\mathtt{msg} = (\text{INIT}, \ldots, \mathtt{fundee})$ addressed from P by $\mathcal{A}$ **then**
20:        add fundee to P's kindred parties
21:        continue executing $\mathcal{A}$'s command
22:    **end if**

23: On (FUND) by $\mathcal{A}$ when $State_P =$ TENTATIVE HELP FUND:
24:    $State_P \leftarrow$ SYNC HELP FUND

25: When $State_P =$ SYNC FUND:
26:    **if** $State_{\bar{P}} \in \{\text{IGNORED}, \text{SYNC HELP FUND}\}$ **then**
27:        $\mathtt{balance}_P \leftarrow \mathtt{balance}_P - x$
28:        $\mathtt{host}_P \leftarrow \mathtt{host}$
29:        // if $\bar{P}$ honest, this state transition happens simultaneously with l. 36
30:        $State_P \leftarrow$ OPEN
31:    **end if**

32: When $State_P =$ SYNC HELP FUND:
33:    **if** $State_{\bar{P}} \in \{\text{IGNORED}, \text{SYNC FUND}\}$ **then**
34:        $\mathtt{host}_P \leftarrow \mathtt{host}$
35:        // if $\bar{P}$ honest, this state transition happens simultaneously with l. 30
36:        $State_P \leftarrow$ OPEN
37:    **end if**

Fig. 10. State machine in Fig. 17

Functionality $\mathcal{G}_{\text{Chan}}$ – close state machine, $P \in \{Alice, Bob\}$

1: On (FORCECLOSE) by P when $State_P = \text{OPEN}$:
2:      $State_P \leftarrow \text{CLOSING}$

3: On input (BALANCE) by R addressed to P where R is kindred with P:
4:      **if** $State_P \notin \{$UNINIT, INIT, PENDING VIRTUAL OPEN, TENTATIVE VIRTUAL OPEN, TENTATIVE BASE OPEN, IGNORED, CLOSED$\}$ **then**
5:         reply (MY BALANCE, balance_P, pk_P, $\text{balance}_{\bar{P}}$, $pk_{\bar{P}}$)
6:      **else**
7:         reply (MY BALANCE, 0, pk_P, 0, $pk_{\bar{P}}$)
8:      **end if**

9: On (FORCECLOSE, P) by $\mathcal{A}$ when $State_P \notin \{$UNINIT, INIT, PENDING VIRTUAL OPEN, TENTATIVE VIRTUAL OPEN, TENTATIVE BASE OPEN, IGNORED$\}$:
10:      input (READ) to $\mathcal{G}_{\text{Ledger}}$ as P and assign ouput to Σ
11:      $\text{coins} \leftarrow$ sum of values of outputs exclusively spendable or spent by pk_P in Σ
12:      $\text{balance} \leftarrow \text{balance}_P$
13:      **for all** P's kindred parties R **do**
14:         input (BALANCE) to R as P and extract balance_R, pk_R from response
15:         $\text{balance} \leftarrow \text{balance} + \text{balance}_R$
16:         $\text{coins} \leftarrow \text{coins} +$ sum of values of outputs exclusively spendable or spent by pk_R in Σ
17:      **end for**
18:      **if** $\text{coins} \geq \text{balance}$ **then**
19:         $State_P \leftarrow \text{CLOSED}$
20:      **else** // balance security is broken
21:         halt
22:      **end if**

Fig. 11.

Functionality $\mathcal{G}_{\text{Chan}}$ – cooperative close state machine, $P \in \{Alice, Bob\}$

1: TODO: fill in, or, if removed, remove all refs to {code:functionality:coop-close}. Possible approach: there will be an alternative message from $\mathcal{A}$, e.g. (COOPCLOSE, P), which will not check on-chain, but tell the host, e.g. via (COOPCLOSED, x) to demand our money. Of course the handling of COOPCLOSED must be added as well. Maybe the whole thing will fit in the previous box.
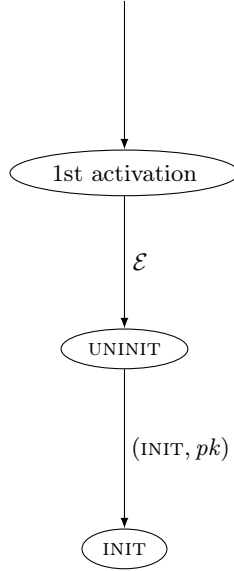

Fig. 12.

Fig. 13. $\mathcal{G}_{\text{Chan}}$ state machine up to INIT (both parties)

Simulator $\mathcal{S}$ – general message handling rules

– On receiving (RELAY, $\texttt{in\_msg}$, P, R, $\texttt{in\_mode}$) by $\mathcal{G}_{\text{Chan}}$ ($\texttt{in\_mode} \in \{$input, output, network$\}$, $P \in \{Alice, Bob\}$), handle ($\texttt{in\_msg}$) with the simulated party P as if it was received from R by means of $\texttt{in\_mode}$. In case simulated P does not exist yet, initialise it as an LN ITI. If there is a resulting message $\texttt{out\_msg}$ that is to be sent by simulated P to R' by means of $\texttt{out\_mode} \in \{$input, output, network$\}$, send (RELAY, $\texttt{out\_msg}$, P, R', $\texttt{out\_mode}$) to $\mathcal{G}_{\text{Chan}}$.
– On receiving by $\mathcal{G}_{\text{Chan}}$ a message to be sent by P to R via the network, carry on with this action (i.e. send this message via the internal $\mathcal{A}$).
– Relay any other incoming message to the internal $\mathcal{A}$ unmodified.
– On receiving a message ($\texttt{msg}$) by the internal $\mathcal{A}$, if it is addressed to one of the parties that correspond to $\mathcal{G}_{\text{Chan}}$, handle the message internally with the corresponding simulated party. Otherwise relay the message to its intended recipient unmodified. // Other recipients are $\mathcal{E}$, $\mathcal{G}_{\text{Ledger}}$ or parties unrelated to $\mathcal{G}_{\text{Chan}}$

Given that $\mathcal{G}_{\text{Chan}}$ relays all messages and that we simulate the real-world machines that correspond to $\mathcal{G}_{\text{Chan}}$, the simulation is perfectly indistinguishable from the real world.
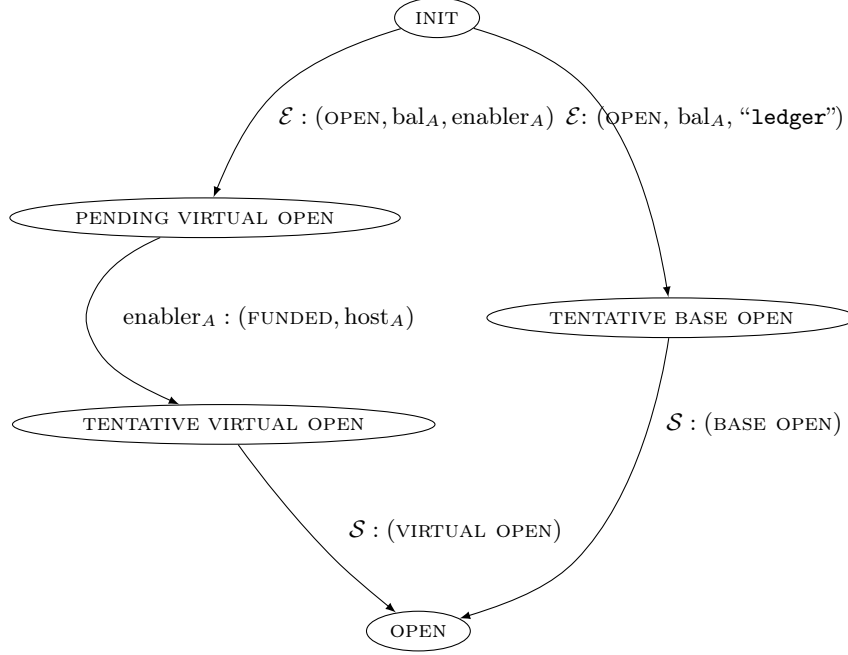
Fig. 14. $\mathcal{G}_{\mathrm{Chan}}$ state machine from INIT up to OPEN (funder)
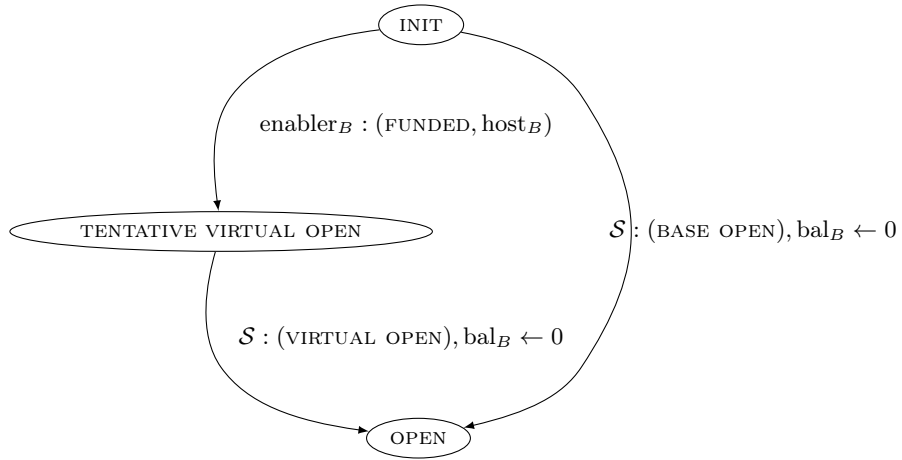
Fig. 15. $\mathcal{G}_{\mathrm{Chan}}$ state machine from INIT up to OPEN (fundee)

Fig. 16. $\mathcal{G}_{\mathrm{Chan}}$ state machine for payments (both parties)

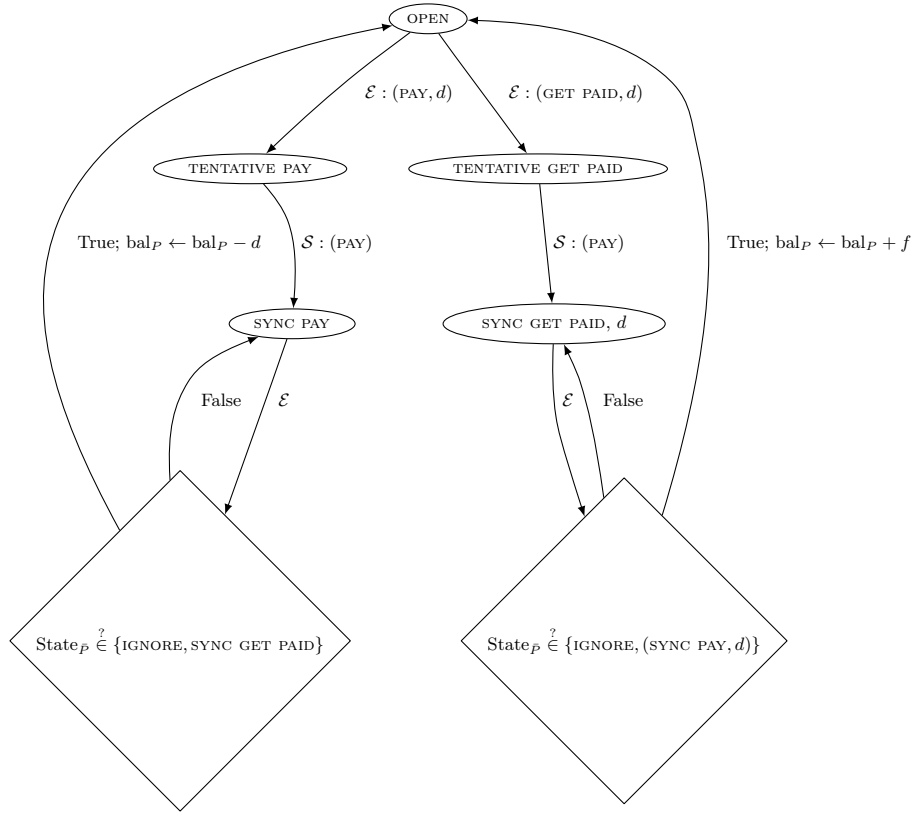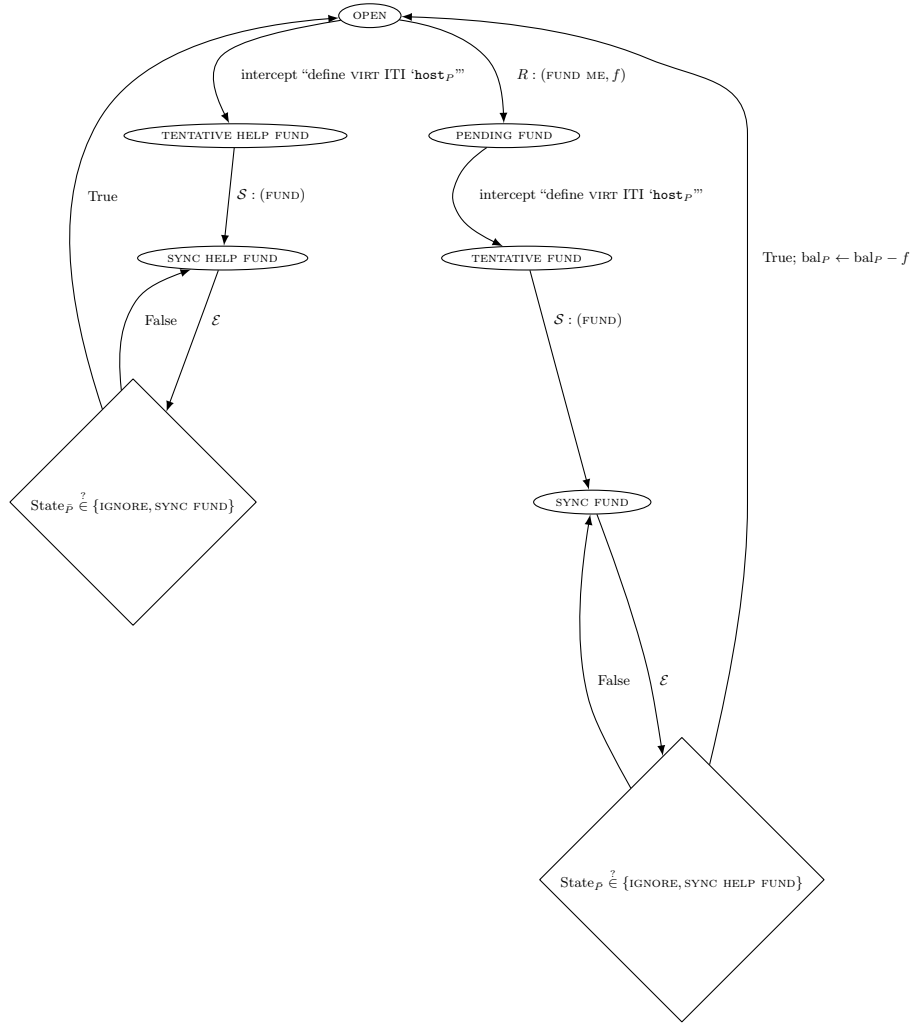**Fig. 17.** $\mathcal{G}_{\mathrm{Chan}}$ state machine for funding new virtuals (both parties)
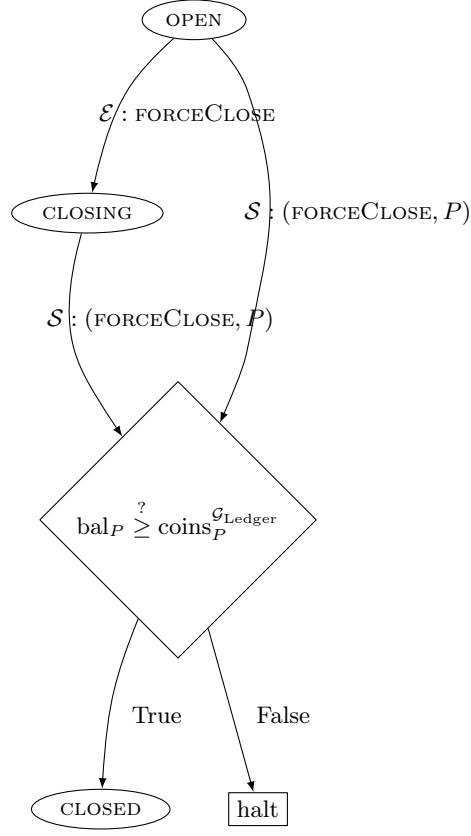
Fig. 18. $\mathcal{G}_{\text{Chan}}$ state machine for channel closure (both parties)
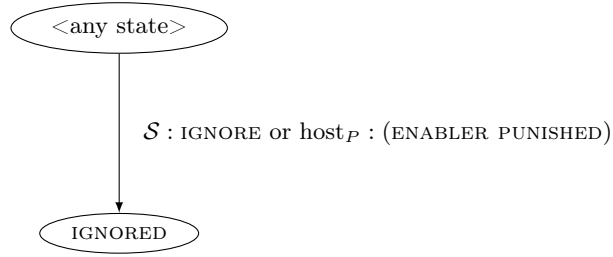

Fig. 19. $\mathcal{G}_{\text{Chan}}$ state machine for corruption, negligence or punishment of the counterparty of a lower layer (both parties)

Fig. 20.

Simulator $\mathcal{S}$ – notifications to $\mathcal{G}_{\text{Chan}}$

- "P" refers one of the parties that correspond to $\mathcal{G}_{\text{Chan}}$.
- When an action in this Figure interrupts an ITI simulation, continue simulating from the interruption location once action is over/$\mathcal{G}_{\text{Chan}}$ hands control back.

1: On (CORRUPT) by $\mathcal{A}$, addressed to P:
2:     // After executing this code and getting control back from $\mathcal{G}_{\text{Chan}}$ (which always happens, c.f. Fig. 7), deliver (CORRUPT) to simulated P (c.f. Fig. 20).
3:     send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$

4: When simulated P sets variable `negligent` to True (Fig. 22, l. 7/Fig. 23, l. 26):
5:     send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$

6: When simulated honest *Alice* receives (OPEN, x, `hops`, ...) by $\mathcal{E}$:
7:     store `hops` // will be used to inform $\mathcal{G}_{\text{Chan}}$ once the channel is open

8: When simulated honest *Bob* receives (OPEN, x, `hops`, ...) by *Alice*:
9:     **if** *Alice* is corrupted **then** store `hops` // if *Alice* is honest, we already have `hops`. If *Alice* became corrupted after receiving (OPEN, ...), overwrite `hops`

10: When the last of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties moves to the OPEN *State* for the first time (Fig. 26, l. 19/Fig. 28, l. 16/Fig. 29, l. 18):
11:     **if** `hops` = "`ledger`" **then**
12:         send (INFO, BASE OPEN) to $\mathcal{G}_{\text{Chan}}$
13:     **else**
14:         send (INFO, VIRTUAL OPEN) to $\mathcal{G}_{\text{Chan}}$
15:     **end if**

16: When (both $\mathcal{G}_{\text{Chan}}$'s simulated parties are honest and complete sending and receiving a payment (Fig. 34, ll. 6 and 21 respectively), or (when only one party is honest and (completes either receiving or sending a payment)): // also send this message if both parties are honest when Fig. 34, l. 6 is executed by one party, but its counterparty is corrupted before executing Fig. 34, l. 21
17:     send (INFO, PAY) to $\mathcal{G}_{\text{Chan}}$

18: When honest P executes Fig. 31, l. 20 or (when honest P executes Fig. 31, l. 18 and $\bar{P}$ is corrupted): // in the first case if $\bar{P}$ is honest, it has already moved to the new host, (Fig 56, ll. 7, 23): lifting to next layer is done
19:     send (INFO, FUND) to $\mathcal{G}_{\text{Chan}}$

20: When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P moves to the CLOSED state (Fig. 38, l. 8 or l. 11):

21:     send (INFO, FORCECLOSE, P) to $\mathcal{G}_{\text{Chan}}$

22: TODO: Add message to $\mathcal{G}_{\text{Chan}}$ in case of cooperative close, e.g. P moving to the COOPCLOSED state

Fig. 21.

10 Protocol

Process LN − init

1: // When not specified, input comes from and output goes to $\mathcal{E}$.
2: // The ITI knows whether it is *Alice* (funder) or *Bob* (fundee). The activated party is P and the counterparty is $\bar{P}$.
3: On every activation, before handling the message:
4:     **if** last_poll $\neq \perp \wedge State \neq$ CLOSED **then** // channel is open
5:         input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
6:         **if** last_poll $+ p < |\Sigma|$ **then** // p is a global parameter
7:             negligent $\leftarrow$ True
8:         **end if**
9:     **end if**

10: On (INIT, $pk_{P,\text{out}}$):
11:     ensure $State = \perp$
12:     $State \leftarrow$ INIT
13:     store $pk_{P,\text{out}}$
14:     $(c_A, c_B, \text{locked}_A, \text{locked}_B) \leftarrow (0, 0, 0, 0)$
15:     $(\text{paid\_out}, \text{paid\_in}) \leftarrow (\emptyset, \emptyset)$
16:     negligent $\leftarrow$ False
17:     last_poll $\leftarrow \perp$
18:     output (INIT OK)

19: On (TOP UP):
20:     ensure $P = Alice$ // activated party is the funder
21:     ensure $State =$ INIT
22:     $(sk_{P,\text{chain}}, pk_{P,\text{chain}}) \leftarrow$ KEYGEN()
23:     input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
24:     output (TOP UP TO, $pk_{P,\text{chain}}$)

25:      **while** $\neg\exists\text{tx} \in \Sigma, c_{P,\text{chain}} : (c_{P,\text{chain}}, pk_{P,\text{chain}}) \in \text{tx.outputs}$ **do**

26:         // while waiting, all other messages by P are ignored

27:         wait for input (CHECK TOP UP)

28:         input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ

29:      **end while**

30:      $State \leftarrow$ TOPPED UP

31:      output (TOP UP OK, $c_{P,\text{chain}}$)

32: On (BALANCE):

33:      ensure $State^P \in \{\text{OPEN}, \text{CLOSED}\}$

34:      output (BALANCE, $c_A, pk_{A,\text{out}}, c_B, pk_{B,\text{out}}, \texttt{locked}_A, \texttt{locked}_B$)

Fig. 22.

Process LN – methods used by VIRT

1: REVOKEPREVIOUS():

2:      ensure $State \in$ WAITING FOR (OUTBOUND) REVOCATION

3:      $R_{\bar{P},i} \leftarrow$ TX $\{$input: $C_{P,i}.\text{outputs}.P$, output: $(C_{P,i}.\text{outputs}.P.\text{value}, pk_{\bar{P},\text{out}})\}$

4:         $\text{sig}_{A,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$

5:      **if** $State =$ WAITING FOR REVOCATION **then**

6:         $State \leftarrow$ WAITING FOR INBOUND REVOCATION

7:      **else** // $State =$ WAITING FOR OUTBOUND REVOCATION

8:         $i \leftarrow i + 1$

9:         $State \leftarrow$ WAITING FOR HOSTS READY

10:      **end if**

11:      $\texttt{host}_P \leftarrow \texttt{host}'_P$ // forget old host, use new host instead

12:      $\texttt{layer} \leftarrow \texttt{layer} + 1$

13:      **return** $\text{sig}_{P,R,i}$

14: PROCESSREMOTEREVOCATION($\text{sig}_{\bar{P},R,i}$):

15:      ensure $State =$ WAITING FOR (INBOUND) REVOCATION

16:      $R_{P,i} \leftarrow$ TX $\{$input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(C_{\bar{P},i}.\text{outputs}.\bar{P}.\text{value}, pk_{P,\text{out}})\}$

17:      ensure VERIFY($R_{P,i}, \text{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}$) = True

18:      **if** $State =$ WAITING FOR REVOCATION **then**

19:         $State \leftarrow$ WAITING FOR OUTBOUND REVOCATION

20:      **else** // $State =$ WAITING FOR INBOUND REVOCATION

21:         $i \leftarrow i + 1$

22:         $State \leftarrow$ WAITING FOR HOSTS READY

23:      **end if**

24:      **return** (OK)

```
25: NEGLIGENT():
26:     negligent ← True
27:     return (OK)
```

Fig. 23.

Process LN.EXCHANGEOPENKEYS()

1: $(sk_{A,F}, pk_{A,F}), (sk_{A,R,1}, pk_{A,R,1}), (sk_{A,R,2}, pk_{A,R,2}) \leftarrow \text{KEYGEN}()^3$
2: $State \leftarrow$ WAITING FOR OPENING KEYS
3: send (OPEN, c, hops, $pk_{A,F}$, $pk_{A,R,1}$, $pk_{A,R,2}$, $pk_{A,\text{out}}$) to fundee
4: // colored code is run by honest fundee. Validation is implicit
5: ensure we run the code of *Bob*
6: ensure $State = $ INIT
7: store $pk_{A,F}$, $pk_{A,R,1}$, $pk_{A,R,2}$, $pk_{A,\text{out}}$
8: $(sk_{B,F}, pk_{B,F}), (sk_{B,R,1}, pk_{B,R,1}), (sk_{B,R,2}, pk_{B,R,2}) \leftarrow \text{KEYGEN}()^3$
9: **if** hops $=$ "ledger" **then** // opening base channel
10:     layer $\leftarrow 0$
11:     $t_P \leftarrow s + p$ // s is the upper bound of η from Lemma 7.19 of [48]
12:     $State \leftarrow$ WAITING FOR COMM SIG
13: **else** // opening virtual channel
14:     $State \leftarrow$ WAITING FOR CHECK KEYS
15: **end if**
16: reply (ACCEPT CHANNEL, $pk_{B,F}$, $pk_{B,R,1}$, $pk_{B,R,2}$, $pk_{B,\text{out}}$)
17: ensure $State = $ WAITING FOR OPENING KEYS
18: store $pk_{B,F}$, $pk_{B,R,1}$, $pk_{B,R,2}$, $pk_{B,\text{out}}$
19: $State \leftarrow$ OPENING KEYS OK

Fig. 24.

Process LN.PREPAREBASE()

1: **if** hops $=$ "ledger" **then** // opening base channel
2:     $F \leftarrow$ TX {input: $(c, pk_{A,\text{chain}})$, output: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$}
3:     host$_P \leftarrow$ "ledger"
4:     layer $\leftarrow 0$
5:     $t_P \leftarrow s + p$
6: **else** // opening virtual channel
7:     input (FUND ME, *Bob*, hops, c, $pk_{A,F}$, $pk_{B,F}$) to hops[0].left and expect output (FUNDED, host$_P$, funder_layer, t_P) // ignore any other message
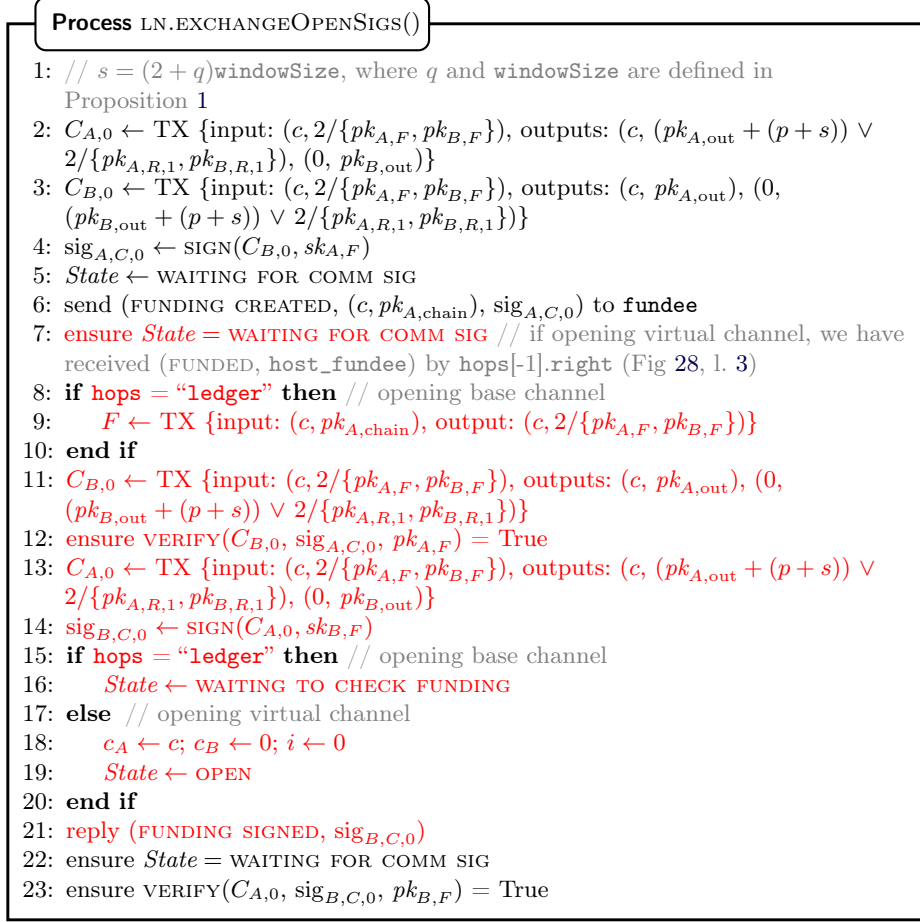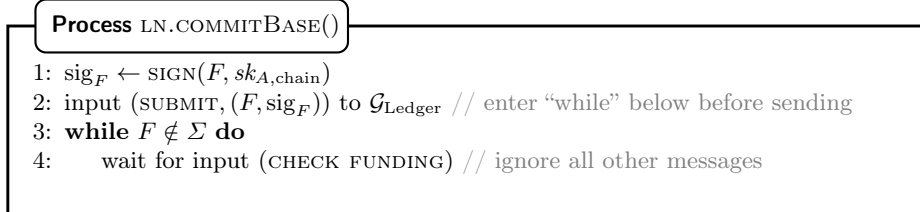8:   layer $\leftarrow$ funder_layer
9: **end if**

Fig. 25.

Process LN.EXCHANGEOPENSIGS()

1: // $s = (2 + q)$`windowSize`, where q and `windowSize` are defined in Proposition 1
2: $C_{A,0} \leftarrow$ TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, (pk_{A,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\}), (0, pk_{B,\text{out}})$}
3: $C_{B,0} \leftarrow$ TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, pk_{A,\text{out}}), (0, (pk_{B,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$}
4: $\text{sig}_{A,C,0} \leftarrow \text{SIGN}(C_{B,0}, sk_{A,F})$
5: $State \leftarrow$ WAITING FOR COMM SIG
6: send (FUNDING CREATED, $(c, pk_{A,\text{chain}}), \text{sig}_{A,C,0}$) to `fundee`
7: ensure $State =$ WAITING FOR COMM SIG // if opening virtual channel, we have received (FUNDED, `host_fundee`) by hops[-1].right (Fig 28, l. 3)
8: **if** `hops` $=$ "`ledger`" **then** // opening base channel
9:      $F \leftarrow$ TX {input: $(c, pk_{A,\text{chain}})$, output: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$}
10: **end if**
11: $C_{B,0} \leftarrow$ TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, pk_{A,\text{out}}), (0, (pk_{B,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$}
12: ensure VERIFY($C_{B,0}, \text{sig}_{A,C,0}, pk_{A,F}$) $=$ True
13: $C_{A,0} \leftarrow$ TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, (pk_{A,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\}), (0, pk_{B,\text{out}})$}
14: $\text{sig}_{B,C,0} \leftarrow \text{SIGN}(C_{A,0}, sk_{B,F})$
15: **if** `hops` $=$ "`ledger`" **then** // opening base channel
16:      $State \leftarrow$ WAITING TO CHECK FUNDING
17: **else** // opening virtual channel
18:      $c_A \leftarrow c; c_B \leftarrow 0; i \leftarrow 0$
19:      $State \leftarrow$ OPEN
20: **end if**
21: reply (FUNDING SIGNED, $\text{sig}_{B,C,0}$)
22: ensure $State =$ WAITING FOR COMM SIG
23: ensure VERIFY($C_{A,0}, \text{sig}_{B,C,0}, pk_{B,F}$) $=$ True

Fig. 26.

Process LN.COMMITBASE()

1: $\text{sig}_F \leftarrow \text{SIGN}(F, sk_{A,\text{chain}})$
2: input (SUBMIT, (F, sig_F)) to $\mathcal{G}_{\text{Ledger}}$ // enter "while" below before sending
3: **while** $F \notin \Sigma$ **do**
4:      wait for input (CHECK FUNDING) // ignore all other messages

```
5:      input (READ) to 𝒢_Ledger and assign output to Σ
6: end while
```

Fig. 27.

Process LN – external open messages for *Bob*

```
1: On output (FUNDED, host_P, funder_layer, t_P) by hops[-1].right:
2:      ensure State = WAITING FOR FUNDED
3:      store host_P  // we will talk directly to host_P
4:      layer ← funder_layer
5:      State ← WAITING FOR COMM SIG
6:      reply (FUND ACK)

7: On output (CHECK KEYS, (pk_1, pk_2)) by hops[-1].right:
8:      ensure State = WAITING FOR CHECK KEYS
9:      ensure pk_1 = pk_{A,F} ∧ pk_2 = pk_{B,F}
10:     State ← WAITING FOR FUDNED
11:     reply (KEYS OK)

12: On input (CHECK FUNDING):
13:     ensure State = WAITING TO CHECK FUNDING
14:     input (READ) to 𝒢_Ledger and assign output to Σ
15:     if F ∈ Σ then
16:         State ← OPEN
17:         reply (OPEN OK)
18:     end if
```

Fig. 28.

Process LN – On (OPEN, c, hops, fundee):

```
1: // fundee is Bob
2: ensure we run the code of Alice // activated party is the funder
3: if hops = "ledger" then // opening base channel
4:      ensure State = TOPPED UP
5:      ensure c = c_{A,chain}
6: else  // opening virtual channel
7:      ensure len(hops) ≥ 2 // cannot open a virtual over 1 channel
8: end if
9: LN.EXCHANGEOPENKEYS()
10: LN.PREPAREBASE()
```

11: LN.EXCHANGEOPENSIGS()
12: **if** hops = "ledger" **then**
13:     LN.COMMITBASE()
14: **end if**
15: input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
16: last_poll ← $|\Sigma|$
17: $c_A \leftarrow c$; $c_B \leftarrow 0$; $i \leftarrow 0$
18: $State \leftarrow$ OPEN
19: output (OPEN OK, c, fundee, hops)

Fig. 29.

Process LN.UPDATEFORVIRTUAL()

1: $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ with $pk'_{P,F}$, $pk'_{\bar{P},F}$, $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{P,F}$, $pk_{\bar{P},F}$, $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, reducing the input and P's output by c_{virt}
2: $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1})$ // kept by $\bar{P}$
3: $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \text{KEYGEN}()$
4: send (UPDATE FORWARD, $\text{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to $\bar{P}$
5: // P refers to payer and $\bar{P}$ to payee both in local and remote code
6: $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ with $pk'_{P,F}$, $pk'_{\bar{P},F}$, $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{P,F}$, $pk_{\bar{P},F}$, $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, reducing the input and P's output by c_{virt}
7: ensure VERIFY($C_{\bar{P},i+1}$, $\text{sig}_{P,C,i+1}$, $pk'_{P,F}$) = True
8: $C_{P,i+1} \leftarrow C_{P,i}$ with $pk'_{\bar{P},F}$, $pk'_{P,F}$, $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{\bar{P},F}$, $pk_{P,F}$, $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, reducing the input and P's output by c_{virt}
9: $\text{sig}_{\bar{P},C,i+1} \leftarrow \text{SIGN}(C_{P,i+1}, sk'_{\bar{P},F})$ // kept by P
10: $(sk_{\bar{P},R,i+2}, pk_{\bar{P},R,i+2}) \leftarrow \text{KEYGEN}()$
11: reply (UPDATE BACK, $\text{sig}_{\bar{P},C,i+1}$, $pk_{\bar{P},R,i+2}$)
12: $C_{P,i+1} \leftarrow C_{P,i}$ with $pk'_{\bar{P},F}$, $pk'_{P,F}$, $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{\bar{P},F}$, $pk_{P,F}$, $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, reducing the input and P's output by c_{virt}
13: ensure VERIFY($C_{P,i+1}$, $\text{sig}_{\bar{P},C,i+1}$, $pk'_{\bar{P},F}$) = True

Fig. 30.

Process LN – virtualise start and end

1: On input (FUND ME, fundee, hops, c_{virt}, $pk_{A,V}$, $pk_{B,V}$) by funder:
2:     ensure $State =$ OPEN

3:      ensure $c_P - \texttt{locked}_P \geq c_{\text{virt}}$

4:      $State \leftarrow$ VIRTUALISING

5:      $(sk'_{P,F}, pk'_{P,F}) \leftarrow$ KEYGEN()

6:      define new VIRT ITI $\texttt{host}'_P$

7:      send (VIRTUALISING, $\texttt{host}'_P$, $pk'_{P,F}$, hops, fundee, c_{virt}, 2, len(hops)) to $\bar{P}$
and expect reply (VIRTUALISING ACK, $\texttt{host}'_{\bar{P}}$, $pk'_{\bar{P},F}$)

8:      ensure $pk'_{\bar{P},F}$ is different from $pk_{\bar{P},F}$ and all older $\bar{P}$'s funding public keys

9:      LN.UPDATEFORVIRTUAL()

10:     $State \leftarrow$ WAITING FOR REVOCATION

11:     input (HOST ME, funder, fundee, $\texttt{host}'_{\bar{P}}$, $\texttt{host}_P$, c_P, $c_{\bar{P}}$, c_{virt}, $pk_{A,V}$, $pk_{B,V}$,
$(sk'_{P,F}, pk'_{P,F})$, $(sk_{P,F}, pk_{P,F})$, $pk_{\bar{P},F}$, $pk'_{\bar{P},F}$, $pk_{P,\text{out}}$, len(hops)) to $\texttt{host}'_P$

12: On output (HOSTS READY, t_P) by $\texttt{host}_P$: // $\texttt{host}_P$ is the new host, renamed
in Fig. 23, l. 12

13:     ensure $State =$ WAITING FOR HOSTS READY

14:     $State \leftarrow$ OPEN

15:     move $sk_{P,F}$, $pk_{P,F}$, $pk_{\bar{P},F}$ to list of old funding keys

16:     $(sk_{P,F}, pk_{P,F}) \leftarrow (sk'_{P,F}, pk'_{P,F})$; $pk_{\bar{P},F} \leftarrow pk'_{\bar{P},F}$

17:     **if** len(hops) $= 1$ **then** // we are the last hop

18:        output (FUNDED, $\texttt{host}_P$, layer, t_P) to fundee and expect reply (FUND
ACK)

19:     **else if** we have received input FUND ME just before we moved to the
VIRTUALISING state **then** // we are the first hop

20:        $c_P \leftarrow c_P - c_{\text{virt}}$

21:        output (FUNDED, $\texttt{host}_P$, layer, t_P) to funder // do not expect reply
by funder

22:     **end if**

23:     reply (HOST ACK)

Fig. 31.

Process LN – virtualise hops

1: On (VIRTUALISING, $\texttt{host}'_{\bar{P}}$, $pk'_{\bar{P},F}$, hops, fundee, c_{virt}, i, n) by $\bar{P}$:

2:     ensure $State =$ OPEN

3:     ensure $c_{\bar{P}} - \texttt{locked}_{\bar{P}} \geq c$; $1 \leq i \leq n$

4:     ensure $pk'_{\bar{P},F}$ is different from $pk_{\bar{P},F}$ and all older $\bar{P}$'s funding public keys

5:     $State \leftarrow$ VIRTUALISING

6:     $\texttt{locked}_{\bar{P}} \leftarrow \texttt{locked}_{\bar{P}} + c$ // if $\bar{P}$ is hosting the funder, $\bar{P}$ will transfer c_{virt}
coins instead of locking them, but the end result is the same

7:     $(sk'_{P,F}, pk'_{P,F}) \leftarrow$ KEYGEN()

8:     **if** len(hops) > 1 **then** // we are not the last hop

9:        define new VIRT ITI $\texttt{host}'_P$

10:     input (VIRTUALISING, host'_P, $(sk'_{P,F}, pk'_{P,F})$, $pk'_{\bar{P},F}$, $pk_{P,\text{out}}$, hops[1:],
   fundee, c_{virt}, $c_{\bar{P}}$, c_P, i, n) to hops[1].left and expect reply (VIRTUALISING
   ACK, host_sibling, $pk_{\text{sib},\bar{P},F}$)
11:     input (INIT, host_P, $\text{host}'_{\bar{P}}$, host_sibling, $(sk'_{P,F}, pk'_{P,F})$, $pk'_{\bar{P},F}$,
   $pk_{\text{sib},\bar{P},F}$, $(sk_{P,F}, pk_{P,F})$, $pk_{\bar{P},F}$, $pk_{P,\text{out}}$, c_P, $c_{\bar{P}}$, c_{virt}, i, t_P, "left", n) to host'_P
   and expect reply (HOST INIT OK)
12:   **else** // we are the last hop
13:     input (INIT, host_P, $\text{host}'_{\bar{P}}$, fundee=fundee, $(sk'_{P,F}, pk'_{P,F})$, $pk'_{\bar{P},F}$,
   $(sk_{P,F}, pk_{P,F})$, $pk_{\bar{P},F}$, $pk_{P,\text{out}}$, c_P, $c_{\bar{P}}$, c_{virt}, t_P, i, "left", n) to new VIRT ITI
   host'_P and expect reply (HOST INIT OK)
14:   **end if**
15:   $State \leftarrow$ WAITING FOR REVOCATION
16:   send (VIRTUALISING ACK, host'_P, $pk'_{P,F}$) to $\bar{P}$

17: On input (VIRTUALISING, host_sibling, $(sk'_{P,F}, pk'_{P,F})$, $pk_{\text{sib},\bar{P},F}$, $pk_{\text{sib},\text{out}}$,
   hops, fundee, c_{virt}, $c_{\text{sib,rem}}$, c_{sib}, i, n) by sibling:
18:   ensure $State =$ OPEN
19:   ensure $c_P - \text{locked}_P \geq c$
20:   ensure $c_{\text{sib,rem}} \geq c_P \wedge c_{\bar{P}} \geq c_{\text{sib}}$ // avoid value loss by griefing attack: one
   counterparty closes with old version, the other stays idle forever
21:   $State \leftarrow$ VIRTUALISING
22:   $\text{locked}_P \leftarrow \text{locked}_P + c$
23:   define new VIRT ITI host'_P
24:   send (VIRTUALISING, host'_P, $pk'_{P,F}$, hops, fundee, c_{virt}, $i+1$, n) to
   hops[0].right and expect reply (VIRTUALISING ACK, $\text{host}'_{\bar{P}}$, $pk'_{\bar{P},F}$)
25:   ensure $pk'_{\bar{P},F}$ is different from $pk_{\bar{P},F}$ and all older $\bar{P}$'s funding public keys
26:   LN.UPDATEFORVIRTUAL()
27:   input (INIT, host_P, $\text{host}'_{\bar{P}}$, host_sibling, $(sk'_{P,F}, pk'_{P,F})$, $pk'_{\bar{P},F}$, $pk_{\text{sib},\bar{P},F}$,
   $(sk_{P,F}, pk_{P,F})$, $pk_{\bar{P},F}$, $pk_{\text{sib,out}}$, c_P, $c_{\bar{P}}$, c_{virt}, i, "right", n) to host'_P and expect
   reply (HOST INIT OK)
28:   $State \leftarrow$ WAITING FOR REVOCATION
29:   output (VIRTUALISING ACK, host'_P, $pk'_{\bar{P},F}$) to sibling

Fig. 32.

Process LN.SIGNATURESROUNDTRIP()

1: $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ with $pk_{P,R,i+1}$ and $pk_{\bar{P},R,i+1}$ instead of $pk_{P,R,i}$ and $pk_{\bar{P},R,i}$
   respectively, and x coins moved from P's to $\bar{P}$'s output
2: $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk_{P,F})$ // kept by $\bar{P}$
3: $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \text{KEYGEN}()$
4: $State \leftarrow$ WAITING FOR COMMITMENT SIGNED
5: send (PAY, x, $\text{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to $\bar{P}$
6: // P refers to payer and $\bar{P}$ to payee both in local and remote code

7: ensure $State = $ WAITING TO GET PAID $\wedge\ x = y$

8: $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ with $pk_{P,R,i+1}$ and $pk_{\bar{P},R,i+1}$ instead of $pk_{P,R,i}$ and $pk_{\bar{P},R,i}$ respectively, and x coins moved from P's to $\bar{P}$'s output

9: ensure VERIFY$(C_{\bar{P},i+1},\ \mathrm{sig}_{P,C,i+1},\ pk_{P,F}) = $ True

10: $C_{P,i+1} \leftarrow C_{P,i}$ with $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, and x coins moved from P's to $\bar{P}$'s output

11: $\mathrm{sig}_{\bar{P},C,i+1} \leftarrow $ SIGN$(C_{P,i+1},\ sk_{\bar{P},F})$ // kept by P

12: $R_{P,i} \leftarrow $ TX $\{$input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_{\bar{P}},\ pk_{P,\text{out}})\}$

13: $\mathrm{sig}_{\bar{P},R,i} \leftarrow $ SIGN$(R_{P,i},\ sk_{\bar{P},R,i})$

14: $(sk_{\bar{P},R,i+2},\ pk_{\bar{P},R,i+2}) \leftarrow $ KEYGEN$()$

15: $State \leftarrow $ WAITING FOR PAY REVOCATION

16: reply (COMMITMENT SIGNED, $\mathrm{sig}_{\bar{P},C,i+1}$, $\mathrm{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i+2}$)

17: ensure $State = $ WAITING FOR COMMITMENT SIGNED

18: $C_{P,i+1} \leftarrow C_{P,i}$ with $pk_{\bar{P},R,i+1}$ and $pk_{P,R,i+1}$ instead of $pk_{\bar{P},R,i}$ and $pk_{P,R,i}$ respectively, and x coins moved from P's to $\bar{P}$'s output

Fig. 33.

Process LN.REVOCATIONSTRIP()

1: ensure VERIFY$(C_{P,i+1},\ \mathrm{sig}_{\bar{P},C,i+1},\ pk_{\bar{P},F}) = $ True

2: $R_{P,i} \leftarrow $ TX $\{$input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_{\bar{P}},\ pk_{P,\text{out}})\}$

3: ensure VERIFY$(R_{P,i},\ \mathrm{sig}_{\bar{P},R,i},\ pk_{\bar{P},R,i}) = $ True

4: $R_{\bar{P},i} \leftarrow $ TX $\{$input: $C_{P,i}.\text{outputs}.P$, output: $(c_P,\ pk_{\bar{P},\text{out}})\}$

5: $\mathrm{sig}_{P,R,i} \leftarrow $ SIGN$(R_{\bar{P},i},\ sk_{P,R,i})$

6: add x to `paid_out`

7: $c_P \leftarrow c_P - x; c_{\bar{P}} \leftarrow c_{\bar{P}} + x; i \leftarrow i+1$

8: $State \leftarrow $ OPEN

9: **if** $\text{host}_P \neq$ "ledger" $\wedge$ we have a `host_sibling` **then** // we are intermediary channel

10:     input (NEW BALANCE, c_P, $c_{\bar{P}}$) to host_P

11:     relay message as input to `sibling` // run by VIRT

12:     relay message as output to `guest` // run by VIRT

13:     store new sibling balance and reply (NEW BALANCE OK)

14:     output (NEW BALANCE OK) to `sibling` // run by VIRT

15:     output (NEW BALANCE OK) to `guest` // run by VIRT

16: **end if**

17: send (REVOKE AND ACK, $\mathrm{sig}_{P,R,i}$) to $\bar{P}$

18: ensure $State = $ WAITING FOR PAY REVOCATION

19: $R_{\bar{P},i} \leftarrow $ TX $\{$input: $C_{P,i}.\text{outputs}.P$, output: $(c_P,\ pk_{\bar{P},\text{out}})\}$

20: ensure VERIFY$(R_{\bar{P},i},\ \mathrm{sig}_{P,R,i},\ pk_{P,R,i}) = $ True

21: add x to `paid_in`

22: $c_P \leftarrow c_P - x; c_{\bar{P}} \leftarrow c_{\bar{P}} + x; i \leftarrow i+1$

23: $State \leftarrow $ OPEN

```
24: if host_P ≠ "ledger" ∧ P̄ has a host_sibling then // we are intermediary
    channel
25:     input (NEW BALANCE, c_P̄, c_P) to host_P̄
26:     relay message as input to sibling // run by VIRT
27:     relay message as output to guest // run by VIRT
28:     store new sibling balance and reply (NEW BALANCE OK)
29:     output (NEW BALANCE OK) to sibling // run by VIRT
30:     output (NEW BALANCE OK) to guest // run by VIRT
31: end if
```

Fig. 34.

Process LN – On (PAY, x):

1: ensure $State = \text{OPEN} \land c_P \geq x$
2: if $\text{host}_P \neq$ "ledger" $\land P$ has a host_sibling then // we are intermediary channel
3:     ensure $c_{\text{sib,rem}} \geq c_P - x \land c_{\bar{P}} + x \geq c_{\text{sib}}$ // avoid value loss by griefing attack: one counterparty closes with old version, the other stays idle forever
4: end if
5: LN.SIGNATURESROUNDTRIP()
6: LN.REVOCATIONSTRIP()
7: // No output is given to the caller, this is intentional

Fig. 35.

Process LN – On (GET PAID, y):

1: ensure $State = \text{OPEN} \land c_{\bar{P}} \geq y$
2: if $\text{host}_P \neq$ "ledger" $\land P$ has a host_sibling then // we are intermediary channel
3:     ensure $c_P + y \leq c_{\text{sib,rem}} \land c_{\text{sib}} \leq c_{\bar{P}} - y$ // avoid value loss by griefing attack
4: end if
5: store y
6: $State \leftarrow \text{WAITING TO GET PAID}$

Fig. 36.

Process LN − On (CHECK FOR LATERAL CLOSE):

1: **if** host$_P \neq$ "ledger" **then**
2:     input (CHECK FOR LATERAL CLOSE) to host$_P$
3: **end if**

Fig. 37.

Process LN − On (CHECK CHAIN FOR CLOSED):

1: ensure $State \notin \{\bot, \text{INIT}, \text{TOPPED UP}\}$ // channel open
2: // even virtual channels check $\mathcal{G}_{\text{Ledger}}$ directly. This is intentional
3: input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ
4: `last_poll` $\leftarrow |\Sigma|$
5: **if** $\exists 0 \leq j < i : C_{\bar{P},j} \in \Sigma$ **then** // counterparty has closed maliciously
6:     $State \leftarrow$ CLOSING
7:     LN.SUBMITANDCHECKREVOCATION(j)
8:     $State \leftarrow$ CLOSED
9:     output (CLOSED)
10: **else if** $C_{P,i} \in \Sigma \vee C_{\bar{P},i} \in \Sigma$ **then**
11:     $State \leftarrow$ CLOSED
12:     output (CLOSED)
13: **end if**

Fig. 38.

Process LN.SUBMITANDCHECKREVOCATION(j)

1: $\text{sig}_{P,R,j} \leftarrow \text{SIGN}(R_{P,j}, sk_{P,R,j})$
2: input (SUBMIT, $(R_{P,j}, \text{sig}_{P,R,j}, \text{sig}_{\bar{P},R,j})$) to $\mathcal{G}_{\text{Ledger}}$
3: **while** $\neg \exists R_{P,j} \in \Sigma$ **do**
4:     wait for input (CHECK REVOCATION) // ignore other messages
5:     input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
6: **end while**
7: $c_P \leftarrow c_P + c_{\bar{P}}$
8: **if** host$_P \neq$ "ledger" **then**
9:     input (USED REVOCATION) to host$_P$
10: **end if**

Fig. 39.

Process LN – On (FORCECLOSE):

1: ensure $State \notin \{\bot, \text{INIT}, \text{TOPPED UP}, \text{CLOSED}, \text{BASE PUNISHED}\}$ // channel open
2: **if** $\text{host}_P \neq \text{"ledger"}$ **then** // we have a virtual channel
3: $\quad State \leftarrow \text{HOST CLOSING}$
4: $\quad$ input (FORCECLOSE) to host_P and keep relaying any (CHECK IF CLOSING)
$\quad$ or (FORCECLOSE) input to host_P until receiving output (CLOSED) by host_P
5: $\quad \text{host}_P \leftarrow \text{"ledger"}$
6: **end if**
7: $State \leftarrow \text{CLOSING}$
8: input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
9: **if** $C_{\bar{P},i} \in \Sigma$ **then** // counterparty has closed honestly
10: $\quad$ no-op // do nothing
11: **else if** $\exists 0 \leq j < i : C_{\bar{P},j} \in \Sigma$ **then** // counterparty has closed maliciously
12: $\quad$ LN.SUBMITANDCHECKREVOCATION(j)
13: **else** // counterparty is idle
14: $\quad$ **while** $\nexists$ unspent **output** $\in \Sigma$ that $C_{P,i}$ can spend **do** // possibly due to
$\quad$ an active timelock
15: $\quad\quad$ wait for input (CHECK VIRTUAL) // ignore other messages
16: $\quad\quad$ input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
17: $\quad$ **end while**
18: $\quad \text{sig}'_{P,C,i} \leftarrow \text{SIGN}(C_{P,i}, sk_{P,F})$
19: $\quad$ input (SUBMIT, $(C_{P,i}, \text{sig}_{P,C,i}, \text{sig}'_{P,C,i})$) to $\mathcal{G}_{\text{Ledger}}$
20: **end if**

Fig. 40.

Process LN – On output (ENABLER USED REVOCATION) by host_P:

1: $State \leftarrow \text{BASE PUNISHED}$

Fig. 41.

Process LN – On (COOPCLOSE):

// any endpoint or intermediary can initiate virtual channel closing
1: ensure $\text{host}_P \neq \text{"ledger"}$
2: ensure $State = \text{OPEN}$
3: $State \leftarrow \text{COOP CLOSING}$
4: **if** we are not **fundee then**
5: $\quad$ `we_are_close_initiator` $\leftarrow$ True
6: $\quad$ send (COOP CLOSE) to **fundee**
7: **else** // we are fundee
8: $\quad$ `close_initiator` $\leftarrow P$

52

```
 9:      execute code of Fig. 44
10: end if
```

Fig. 42.

Process LN – On (coopClosed) by R:

```
1: ensure State ∈ {OPEN, COOP CLOSED}
2: ensure we_are_close_initiator = True
3: ensure that the last cooperatively closed channel in which we acted as a base
   had R as its fundee
4: we_are_close_initiator ← False
5: output (coopClosed)
```

Fig. 43.

Process LN – On (coop close) by R:

// also executed when we are instructed to close a channel cooperatively by $\mathcal{E}$– c.f. Fig. 42, l. 9

```
1: ensure we are fundee
2: ensure State = OPEN
3: State ← COOP CLOSING
4: close_initiator ← R
5: sig_bal ← ((c_P̄, c_P), SIGN((c_P̄, c_P), sk_{P,F}))
6: State ← WAITING TO REVOKE VIRT COMM
7: send (COOP CLOSE, sig_bal) to P̄
```

Fig. 44.

Process LN – On (coop close, $\texttt{sig\_bal}_{\bar{P}}$) by $\bar{P}$:

```
1: ensure we are funder
2: ensure State = OPEN
3: parse sig_bal_P̄ as ((c'_P, c'_P̄), sig_P̄)
4: ensure c_P = c'_P ∧ c_P̄ = c'_P̄ ∧ VERIFY((c'_P, c'_P̄), sig_P̄, pk_{P̄,F}) = True
5: sig_bal ← ((c_P, c_P̄), SIGN((c_P, c_P̄), sk_{P,F}), sig_P̄)
6: State ← COOP CLOSING
7: input (COOP CLOSE, sig_bal) to host_P
8: ensure State = OPEN // executed by host_P
```

9: $State \leftarrow$ COOP CLOSING
10: output (COOP CLOSE SIGN COMM, (c_1, c_2)) to guest
11: ensure $State =$ OPEN // executed by guest of host$_P$
12: $State \leftarrow$ COOP CLOSING
13: remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$, $pk'_{\bar{P},F}$
14: $C_{\bar{P},i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c_1 + c_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c_1, pk_{P,\text{out}})$, $(c_{\bar{P}} + c_2, (pk_{\bar{P},\text{out}} + (p+s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$}
15: $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk'_{P,F})$
16: $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \text{KEYGEN}()$
17: input (NEW COMM TX, $\text{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to host$_P$
18: rename received signature to $\text{sig}_{1,\text{right},C}$ // executed by host$_P$
19: rename received public key to $pk_{1,\text{right},R}$
20: send (COOP CLOSE, $\texttt{sig\_bal}$, $\text{sig}_{1,\text{right},C}$, $pk_{1,\text{right},R}$) to $\bar{P}$ and expect reply (COOP CLOSE BACK, ($\texttt{right\_comms\_revkeys}$, $\texttt{right\_revocations}$)
21: $R_{\text{loc}} \leftarrow$ TX {input: $(c_{\text{virt}}, 2/\{pk_{1,\text{rev}}, pk_{2,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{1,\text{out}})$}
22: extract $\text{sig}_{2,\text{right,rev}}$ from $\texttt{right\_revocations}$
23: ensure VERIFY($R_{\text{loc,right}}$, $\text{sig}_{2,\text{right,rev}}$, $pk_{2,\text{rev}}$) = True
24: extract $\text{sig}_{2,\text{right},R}$ from $\texttt{right\_revocations}$
25: extract $\text{sig}_{2,\text{right},C}$ from $\texttt{right\_comms\_revkeys}$
26: extract $pk_{2,R}$ from $\texttt{right\_comms\_revkeys}$
27: output (VERIFY REVOKE, $\text{sig}_{2,\text{right},C}$, $\text{sig}_{2,\text{right},R}$, $pk_{2,R}$, host$_P$) to guest
28: store $\text{sig}_{2,\text{right},C}$ as $\text{sig}_{\bar{P},C,i+1}$ // executed by guest of host$_P$
29: store $\text{sig}_{2,\text{right},R}$ as $\text{sig}_{\bar{P},R,i}$
30: store received public key as $pk_{\bar{P},R,i+2}$
31: $C_{P,i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c_1 + c_2)$, outputs: $(c_P + c_1, (pk_{P,\text{out}} + (p+s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$, $(c_{\bar{P}} + c_2, pk_{\bar{P},\text{out}})$}
32: ensure VERIFY($C_{P,i+1}$, $\text{sig}_{\bar{P},C,i+1}$, $pk'_{\bar{P},F}$) = True
33: $R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\text{out}})$}
34: ensure VERIFY($R_{P,i}$, $\text{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
35: input (VERIFIED) to host$_P$
36: extract $\text{sig}_{n,\text{left},R}$ from $\texttt{right\_revocations}$ // executed by host$_P$
37: output (VERIFY REVOCATION, $\text{sig}_{n,\text{left},R}$) to funder
38: $R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\text{out}})$}
39: ensure VERIFY($R_{P,i}$, $\text{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
40: $R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}.\text{outputs}.P$, output: $(c_P, pk_{\bar{P},\text{out}})$}
41: $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$
42: $State \leftarrow$ COOP CLOSED // in LN, only virtual channels can end up in this state
43: input (COOP CLOSE REVOCATION, $\text{sig}_{P,R,i}$) to host$_P$
44: output (COOP CLOSE REVOCATIONS, host$_P$) to guest // executed by host$_P$
45: $R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}.\text{outputs}.P$, output: $(c_P, pk_{\bar{P},\text{out}})$} // executed by guest of host$_P$
46: $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$
47: add $sk_{P,F}$, $pk_{P,F}$, $pk_{\bar{P},F}$ to list of old enable channel funding keys
48: add host$_P$ to list of old hosts
49: assign received host to host$_P$
50: $State \leftarrow$ OPEN

51: input (REVOCATION, $\text{sig}_{P,R,i}$) to last old host

52: rename received signature to $\text{sig}_{1,\text{right},R}$ // executed by host_P

53: $R_{\text{rem,right}} \leftarrow \text{TX} \{\text{input: } (c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{1,\text{rev}}, pk_{2,\text{rev}}, pk_{n,\text{rev}}\}), \text{output: } (c_{\text{virt}}, pk_{2,\text{out}})\}$

54: $\text{sig}_{1,\text{right,rev}} \leftarrow \text{SIGN}(R_{\text{rem,right}}, sk_{1,\text{rev}})$

55: **for all** $j \in \{2, \ldots, n\}$ **do**

56: $\quad R_{j,\text{left}} \leftarrow \text{TX} \{\text{input: } (c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{n,\text{rev}}\}), \text{output: } (c_{\text{virt}}, pk_{j,\text{out}})\}$

57: $\quad \text{sig}_{1,j,\text{left,rev}} \leftarrow \text{SIGN}(R_{j,\text{left}}, sk_{1,\text{rev}})$

58: **end for**

59: $State \leftarrow \text{COOP CLOSED}$

60: send (COOP CLOSE REVOCATIONS, $(\text{sig}_{1,\text{right},R}, \text{sig}_{1,\text{right,rev}}, (\text{sig}_{1,j,\text{left,rev}})_{j\in\{2,\ldots,n\}})$) to $\bar{P}$

Fig. 45.

Process VIRT

1: On every activation, before handling the message:

2: $\quad$ **if** $\texttt{last\_poll} \neq \bot$ **then** // virtual layer is ready

3: $\quad\quad$ input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ

4: $\quad\quad$ **if** $\texttt{last\_poll} + p < |\Sigma|$ **then**

5: $\quad\quad\quad$ **for** $P \in \{\texttt{guest}, \texttt{funder}, \texttt{fundee}\}$ **do** // at most 1 of funder, fundee is defined

6: $\quad\quad\quad\quad$ ensure $P.\text{NEGLIGENT}()$ returns (OK)

7: $\quad\quad\quad$ **end for**

8: $\quad\quad$ **end if**

9: $\quad$ **end if**

10: // guest is trusted to give sane inputs, therefore a state machine and input verification are redundant

11: On input (INIT, $\texttt{host}_P$, $\bar{P}$, sibling, fundee, $(sk_{\text{loc,fund,new}}, pk_{\text{loc,fund,new}})$, $pk_{\text{rem,fund,new}}$, $pk_{\text{sib,rem,fund,new}}$, $(sk_{\text{loc,fund,old}}, pk_{\text{loc,fund,old}})$, $pk_{\text{rem,fund,old}}$, $pk_{\text{loc,out}}$, c_P, $c_{\bar{P}}$, c_{virt}, t_P, i, side, n) by guest:

12: $\quad$ ensure $1 < i \leq n$ // host_funder ($i = 1$) is initialised with HOST ME

13: $\quad$ ensure side $\in \{\text{"left"}, \text{"right"}\}$

14: $\quad$ store message contents and guest // sibling, $pk_{\text{sib},\bar{P},F}$ are missing for endpoints, fundee is present only in last node

15: $\quad$ $(sk_{i,\text{fund,new}}, pk_{i,\text{fund,new}}) \leftarrow (sk_{\text{loc,fund,new}}, pk_{\text{loc,fund,new}})$

16: $\quad$ $pk_{\texttt{myRem},\text{fund,new}} \leftarrow pk_{\text{rem,fund,new}}$

17: $\quad$ **if** $i < n$ **then** // we are not last hop

18: $\quad\quad$ $pk_{\texttt{sibRem},\text{fund,new}} \leftarrow pk_{\text{sib,rem,fund,new}}$

19: $\quad$ **end if**

20: $\quad$ **if** side $= \text{"left"}$ **then**

21: $\quad\quad$ $\text{side}' \leftarrow \text{"right"}$; $\texttt{myRem} \leftarrow i - 1$; $\texttt{sibRem} \leftarrow i + 1$

22:     $pk_{i,\text{out}} \leftarrow pk_{\text{loc,out}}$
23:     $(sk_{i,j,k}, pk_{i,j,k})_{j \in \{2,\ldots,n-1\}, k \in [n] \setminus \{j\}} \leftarrow \text{KeyGen}()^{(n-2)(n-1)}$
24:     $(sk_{i,\text{rev}}, pk_{i,\text{rev}}) \leftarrow \text{KeyGen}()$
25:   **else** // side = "right"
26:     $\text{side}' \leftarrow$ "left"; $\text{myRem} \leftarrow i + 1$; $\text{sibRem} \leftarrow i - 1$
27:     // sibling will send keys in KEYS AND COINS FORWARD
28:   **end if**
29:   $(sk_{i,\text{side,fund,old}}, pk_{i,\text{side,fund,old}}) \leftarrow (sk_{\text{loc,fund,old}}, pk_{\text{loc,fund,old}})$
30:   $pk_{\text{myRem},\text{side}',\text{fund,old}} \leftarrow pk_{\text{rem,fund,old}}$
31:   $(c_{i,\text{side}}, c_{\text{myRem},\text{side}'}, t_{i,\text{side}}) \leftarrow (c_P, c_{\bar{P}}, t_P)$
32:   $\texttt{last\_poll} \leftarrow \perp$
33:   output (HOST INIT OK) to guest

34: On input (HOST ME, **funder**, **fundee**, $\bar{P}$, host_P, c_P, $c_{\bar{P}}$, c_{virt}, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $(sk_{1,\text{fund,new}}, pk_{1,\text{fund,new}})$, $(sk_{1,\text{right,fund,old}}, pk_{1,\text{right,fund,old}})$, $pk_{2,\text{left,fund,old}}$, $pk_{2,\text{left,fund,new}}$, $pk_{1,\text{out}}$, n) by **guest**:
35:   $\texttt{last\_poll} \leftarrow \perp$
36:   $i \leftarrow 1$
37:   $c_{1,\text{right}} \leftarrow c_P$; $c_{2,\text{left}} \leftarrow c_{\bar{P}}$
38:   $(sk_{1,j,k}, pk_{1,j,k})_{j \in \{2,\ldots,n-1\}, k \in [n] \setminus \{j\}} \leftarrow \text{KeyGen}()^{(n-2)(n-1)}$
39:   $(sk_{1,\text{rev}}, pk_{1,\text{loc,rev}}) \leftarrow \text{KeyGen}()$
40:   ensure VIRT.CIRCULATEKEYSCOINSTIMES() returns (OK)
41:   ensure VIRT.CIRCULATEVIRTUALSIGS() returns (OK)
42:   ensure VIRT.CIRCULATEFUNDINGSIGS() returns (OK)
43:   ensure VIRT.CIRCULATEREVOCATIONS() returns (OK)
44:   output (HOSTS READY, $p + \sum_{j=2}^{n-1} (s - 1 + t_j)$) to guest // p is every how
   many blocks we have to check the chain

Fig. 46.

Process VIRT.CIRCULATEKEYSCOINSTIMES(`left_data`):

1: **if** `left_data` is given as argument **then** // we are not host_funder
2:     parse `left_data` as $((pk_{j,\text{fund,new}})_{j \in [i-1]}, (pk_{j,\text{left,fund,old}})_{j \in \{2,\ldots,i-1\}},$
   $(pk_{j,\text{right,fund,old}})_{j \in [i-1]}, (pk_{j,\text{out}})_{j \in [i-1]}, (c_{j,\text{left}})_{j \in \{2,\ldots,i-1\}}, (c_{j,\text{right}})_{j \in [i-1]},$
   $(t_j)_{j \in [i-1]}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, (pk_{h,j,k})_{h \in [i-1], j \in \{2,\ldots,n-1\}, k \in [n] \setminus \{j\}},$
   $(pk_{h,\text{loc,rev}})_{h \in [i-1]}, (pk_{h,\text{rem,rev}})_{h \in [i-1]})$
3:     **if** we have a **sibling then** // we are not host_fundee
4:         input (KEYS AND COINS FORWARD, (`left_data`, $(sk_{i,\text{left,fund,old}},$
   $pk_{i,\text{left,fund,old}}), pk_{i,\text{out}}, c_{i,\text{left}}, t_{i,\text{left}}, (sk_{i,j,k}, pk_{i,j,k})_{j \in \{2,\ldots,n-1\}, k \in [n] \setminus \{j\}},$
   $(sk_{i,\text{rev}}, pk_{i,\text{rev}}))$ to **sibling**
5:         store input as `left_data` and parse it as $((pk_{j,\text{fund,new}})_{j \in [i-1]},$
   $(pk_{j,\text{left,fund,old}})_{j \in \{2,\ldots,i\}}, (pk_{j,\text{right,fund,old}})_{j \in [i-1]}, (pk_{j,\text{out}})_{j \in [i]}, (c_{j,\text{left}})_{j \in \{2,\ldots,i\}},$

56

$(c_{j,\text{right}})_{j\in[i-1]}$, $(t_j)_{j\in[i-1]}$, $sk_{i,\text{left,fund,old}}$, $t_{i,\text{left}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$,
$(pk_{h,j,k})_{h\in[i],j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $(sk_{i,j,k})_{j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $(pk_{h,\text{rev}})_{h\in[i]}$,
$sk_{i,\text{rev}}$

6:        $t_i \leftarrow \max\left(t_{i,\text{left}}, t_{i,\text{right}}\right)$

7:        replace $t_{i,\text{left}}$ in `left_data` with t_i

8:        remove $sk_{i,\text{left,fund,old}}$, $(sk_{i,j,k})_{j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $sk_{i,\text{loc,rev}}$ and $sk_{i,\text{rem,rev}}$ from `left_data`

9:        call VIRT.CIRCULATEKEYSCOINSTIMES(`left_data`) of $\bar{P}$ and assign returned value to `right_data`

10:        parse `right_data` as $((pk_{j,\text{fund,new}})_{j\in\{i+1,...,n\}}$,
$(pk_{j,\text{left,fund,old}})_{j\in\{i+1,...,n\}}$, $(pk_{j,\text{right,fund,old}})_{j\in\{i+1,...,n-1\}}$, $(pk_{j,\text{out}})_{j\in\{i+1,...,n\}}$,
$(c_{j,\text{left}})_{j\in\{i+1,...,n\}}$, $(c_{j,\text{right}})_{j\in\{i+1,...,n-1\}}$, $(t_j)_{j\in\{i+1,...,n\}}$,
$(pk_{h,j,k})_{h\in\{i+1,...,n\},j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $(pk_{h,\text{rev}})_{h\in\{i+1,...,n\}}$

11:        output (KEYS AND COINS BACK, `right_data`, $(sk_{i,\text{right,fund,old}}$,
$pk_{i,\text{right,fund,old}})$, $c_{i,\text{right}}$, t_i)

12:        store output as `right_data` and parse it as $((pk_{j,\text{fund,new}})_{j\in\{i+1,...,n\}}$,
$(pk_{j,\text{left,fund,old}})_{j\in\{i+1,...,n\}}$, $(pk_{j,\text{right,fund,old}})_{j\in\{i,...,n-1\}}$, $(pk_{j,\text{out}})_{j\in\{i+1,...,n\}}$,
$(c_{j,\text{left}})_{j\in\{i+1,...,n\}}$, $(c_{j,\text{right}})_{j\in\{i,...,n-1\}}$, $(t_j)_{j\in\{i,...,n\}}$,
$(pk_{h,j,k})_{h\in\{i+1,...,n\},j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $(pk_{h,\text{loc,rev}})_{h\in\{i+1,...,n\}}$,
$(pk_{h,\text{rem,rev}})_{h\in\{i+1,...,n\}}$, $sk_{i,\text{right,fund,old}})$

13:        remove $sk_{i,\text{right,fund,old}}$ from `right_data`

14:        **return** (`right_data`, $pk_{i,\text{fund,new}}$, $pk_{i,\text{left,fund,old}}$, $pk_{i,\text{out}}$, $c_{i,\text{left}}$)

15:    **else** // we are host_fundee

16:        output (CHECK KEYS, $(pk_{\text{left,virt}}, pk_{\text{right,virt}})$) to `fundee` and expect reply (KEYS OK)

17:        **return** ($pk_{n,\text{fund,new}}$, $pk_{n,\text{left,fund,old}}$, $pk_{n,\text{out}}$, $c_{n,\text{left}}$, t_n,
$(pk_{n,j,k})_{j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $pk_{n,\text{loc,rev}}$, $pk_{n,\text{rem,rev}}$)

18:    **end if**

19: **else** // we are host_funder

20:    call VIRT.CIRCULATEKEYSCOINSTIMES($pk_{1,\text{fund,new}}$, $pk_{1,\text{right,fund,old}}$, $pk_{1,\text{out}}$,
$c_{1,\text{right}}$, t_1, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $(pk_{1,j,k})_{j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$, $pk_{1,\text{loc,rev}}$,
$pk_{1,\text{rem,rev}}$) of $\bar{P}$ and assign returned value to `right_data`

21:    parse `right_data` as $((pk_{j,\text{fund,new}})_{j\in\{2,...,n\}}$, $(pk_{j,\text{left,fund,old}})_{j\in\{2,...,n\}}$,
$(pk_{j,\text{right,fund,old}})_{j\in\{2,...,n-1\}}$, $(pk_{j,\text{out}})_{j\in\{2,...,n\}}$, $(c_{j,\text{left}})_{j\in\{2,...,n\}}$,
$(c_{j,\text{right}})_{j\in\{2,...,n-1\}}$, $(t_j)_{j\in\{2,...,n\}}$, $(pk_{h,j,k})_{h\in\{2,...,n\},j\in\{2,...,n-1\},k\in[n]\setminus\{j\}}$,
$(pk_{h,\text{loc,rev}})_{h\in\{2,...,n\}}$, $(pk_{h,\text{rem,rev}})_{h\in\{2,...,n\}}$)

22:    **return** (OK)

23: **end if**

Fig. 47.

Process VIRT

1: GETMIDTXS(i, n, c_{virt}, $c_{\text{rem,left}}$, $c_{\text{loc,left}}$, $c_{\text{loc,right}}$, $c_{\text{rem,right}}$, $pk_{\text{rem,left,fund,old}}$,
$pk_{\text{loc,left,fund,old}}$, $pk_{\text{loc,right,fund,old}}$, $pk_{\text{rem,right,fund,old}}$, $pk_{\text{rem,left,fund,new}}$,

$pk_{\mathrm{loc,left,fund,new}}$, $pk_{\mathrm{loc,right,fund,new}}$, $pk_{\mathrm{rem,right,fund,new}}$, $pk_{\mathrm{left,virt}}$, $pk_{\mathrm{right,virt}}$, $pk_{\mathrm{loc,out}}$, $pk_{\mathrm{funder,rev}}$, $pk_{\mathrm{left,rev}}$, $pk_{\mathrm{loc,rev}}$, $pk_{\mathrm{right,rev}}$, $pk_{\mathrm{fundee,rev}}$, $(pk_{h,j,k})_{h \in [n], j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,j\}}$, $(pk_{h,2,1})_{h \in [n]}$, $(pk_{h,n-1,n})_{h \in [n]}$, $(t_j)_{j \in [n-1] \setminus \{1\}}$:

2:    ensure $1 < i < n$

3:    ensure $c_{\mathrm{rem,left}} \geq c_{\mathrm{virt}} \wedge c_{\mathrm{loc,left}} \geq c_{\mathrm{virt}}$ // left parties fund virtual channel

4:    ensure $c_{\mathrm{rem,left}} \geq c_{\mathrm{loc,right}} \wedge c_{\mathrm{rem,right}} \geq c_{\mathrm{loc,left}}$ // avoid griefing attack

5:    $c_{\mathrm{left}} \leftarrow c_{\mathrm{rem,left}} + c_{\mathrm{loc,left}}$; $c_{\mathrm{right}} \leftarrow c_{\mathrm{loc,right}} + c_{\mathrm{rem,right}}$

6:    `left_old_fund` $\leftarrow 2/\{pk_{\mathrm{rem,left,fund,old}}, pk_{\mathrm{loc,left,fund,old}}\}$

7:    `right_old_fund` $\leftarrow 2/\{pk_{\mathrm{loc,right,fund,old}}, pk_{\mathrm{rem,right,fund,old}}\}$

8:    `left_new_fund` $\leftarrow$
$(2/\{pk_{\mathrm{rem,left,fund,new}}, pk_{\mathrm{loc,left,fund,new}}\}) \vee 2/\{pk_{\mathrm{left,rev}}, pk_{\mathrm{loc,rev}}\}$

9:    `right_new_fund` $\leftarrow$
$(2/\{pk_{\mathrm{loc,right,fund,new}}, pk_{\mathrm{rem,right,fund,new}}\}) \vee 2/\{pk_{\mathrm{loc,rev}}, pk_{\mathrm{right,rev}}\}$

10:    `virt_fund` $\leftarrow 2/\{pk_{\mathrm{left,virt}}, pk_{\mathrm{right,virt}}\}$

11:    `revocation` $\leftarrow 4/\{pk_{\mathrm{funder,rev}}, pk_{\mathrm{loc,rev}}, pk_{\mathrm{right,rev}}, pk_{\mathrm{fundee,rev}}\}$

12:    `refund` $\leftarrow (pk_{\mathrm{loc,out}} + (p + s)) \vee 2/\{pk_{\mathrm{left,rev}}, pk_{\mathrm{loc,rev}}\}$

13:    **for all** $j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,j\}$ **do**

14:        $all_{j,k} \leftarrow n/\{pk_{1,j,k}, \ldots, pk_{n,j,k}\} \wedge$ `"k"`

15:    **end for**

16:    **if** $i = 2$ **then**

17:        $all_{2,1} \leftarrow n/\{pk_{1,2,1}, \ldots, pk_{n,2,1}\} \wedge$ `"1"`

18:    **end if**

19:    **if** $i = n - 1$ **then**

20:        $all_{n-1,n} \leftarrow n/\{pk_{1,n-1,n}, \ldots, pk_{n,n-1,n}\} \wedge$ `"n"`

21:    **end if**

22:    // After funding is complete, A_j has the signature of all other parties for all $all_{j,k}$ inputs, but other parties do not have A_j's signature for this input, therefore only A_j can publish it.

23:    // $\mathrm{TX}_{i,j,k} := i$-th move, j, k input interval start and end. j, k unneeded for $i = 1$, k unneeded for $i = 2$.

24:    $\mathrm{TX}_1 \leftarrow$ TX:

25:        inputs:

26:            $(c_{\mathrm{left}}, $ `left_old_fund`$)$,

27:            $(c_{\mathrm{right}}, $ `right_old_fund`$)$

28:        outputs:

29:            $(c_{\mathrm{left}} - c_{\mathrm{virt}}, $ `left_new_fund`$)$,

30:            $(c_{\mathrm{right}} - c_{\mathrm{virt}}, $ `right_new_fund`$)$,

31:            $(c_{\mathrm{virt}}, $ `refund`$)$,

32:            $(c_{\mathrm{virt}}, $

33:                (if $(i - 1 > 1)$ then $all_{i-1,i}$ else False)

34:                $\vee$ (if $(i + 1 < n)$ then $all_{i+1,i}$ else False)

35:                $\vee$ `revocation`

36:                $\vee$ (

37:                    if $(i - 1 = 1 \wedge i + 1 = n)$ then `virt_fund`

38:                    else if $(i - 1 > 1 \wedge i + 1 = n)$ then `virt_fund` $+ t_{i-1}$

39:                     else if $(i - 1 = 1 \land i + 1 < n)$ then $\mathtt{virt\_fund} + t_{i+1}$
40:                     else $/^*i - 1 > 1 \land i + 1 < n^*/$ $\mathtt{virt\_fund} + \max(t_{i-1}, t_{i+1})$
41:                 )
42:             )
43:     if $i = 2$ then
44:         $\mathrm{TX}_{2,1} \leftarrow \mathrm{TX}$:
45:             inputs:
46:                 $(c_{\mathrm{virt}}, all_{2,1})$,
47:                 $(c_{\mathrm{right}}, \mathtt{right\_old\_fund})$
48:             outputs:
49:                 $(c_{\mathrm{right}} - c_{\mathrm{virt}}, \mathtt{right\_new\_fund})$,
50:                 $(c_{\mathrm{virt}}, \mathtt{refund})$,
51:                 $(c_{\mathrm{virt}}$,
52:                     $\mathtt{revocation} \lor$
53:                     (if $(n > 3)$ then $(all_{3,2} \lor (\mathtt{virt\_fund} + t_3))$
54:                     else $\mathtt{virt\_fund})$
55:                 )
56:     end if
57:     if $i = n - 1$ then
58:         $\mathrm{TX}_{2,n} \leftarrow \mathrm{TX}$:
59:             inputs:
60:                 $(c_{\mathrm{left}}, \mathtt{left\_old\_fund})$,
61:                 $(c_{\mathrm{virt}}, all_{n-1,n})$
62:             outputs:
63:                 $(c_{\mathrm{left}} - c_{\mathrm{virt}}, \mathtt{left\_new\_fund})$,
64:                 $(c_{\mathrm{virt}}, \mathtt{refund})$,
65:                 $(c_{\mathrm{virt}}$,
66:                     $\mathtt{revocation} \lor$
67:                     (if $(n - 2 > 1)$ then $(all_{n-2,n-1} \lor (\mathtt{virt\_fund} + t_{n-2}))$
68:                     else $\mathtt{virt\_fund})$
69:                 )
70:     end if
71:     for all $k \in \{2, \ldots, i-1\}$ do // $i - 2$ txs
72:         $\mathrm{TX}_{2,k} \leftarrow \mathrm{TX}$:
73:             inputs:
74:                 $(c_{\mathrm{virt}}, all_{i,k})$,
75:                 $(c_{\mathrm{right}}, \mathtt{right\_old\_fund})$
76:             outputs:
77:                 $(c_{\mathrm{right}} - c_{\mathrm{virt}}, \mathtt{right\_new\_fund})$,
78:                 $(c_{\mathrm{virt}}, \mathtt{refund})$,
79:                 $(c_{\mathrm{virt}}$,
80:                     (if $(k - 1 > 1)$ then $all_{k-1,i}$ else False)
81:                     $\lor$ (if $(i + 1 < n)$ then $all_{i+1,k}$ else False)

```
82:                    ∨ revocation
83:                    ∨ (
84:                        if (k − 1 = 1 ∧ i + 1 = n) then virt_fund
85:                        else if (k − 1 > 1 ∧ i + 1 = n) then virt_fund + t_{k−1}
86:                        else if (k − 1 = 1 ∧ i + 1 < n) then virt_fund + t_{i+1}
87:                        else /*k−1>1∧i+1<n*/ virt_fund + max (t_{k−1}, t_{i+1})
88:                    )
89:                )
90:      end for
91:      for all k ∈ {i + 1, . . . , n − 1} do // n − i − 1 txs
92:          TX_{2,k} ← TX:
93:              inputs:
94:                  (c_left, left_old_fund)
95:                  (c_virt, all_{i,k}),
96:              outputs:
97:                  (c_left − c_virt, left_new_fund),
98:                  (c_virt, refund),
99:                  (c_virt,
100:                      (if (i − 1 > 1) then all_{i−1,k} else False)
101:                      ∨ (if (k + 1 < n) then all_{k+1,i} else False)
102:                      ∨ revocation
103:                      ∨ (
104:                          if (i − 1 = 1 ∧ k + 1 = n) then virt_fund
105:                          else if (i − 1 > 1 ∧ k + 1 = n) then virt_fund + t_{i−1}
106:                          else if (i − 1 = 1 ∧ k + 1 < n) then virt_fund + t_{k+1}
107:                          else /*i − 1 > 1 ∧ k + 1 < n*/
    virt_fund + max (t_{i−1}, t_{k+1})
108:                      )
109:                  )
110:      end for
111:      if i = 2 then m ← 1 else m ← 2
112:      if i = n − 1 then l ← n else l ← n − 1
113:      for all (k_1, k_2) ∈ {m, . . . , i − 1} × {i + 1, . . . , l} do // (i − m) · (l − i) txs
114:          TX_{3,k_1,k_2} ← TX:
115:              inputs:
116:                  (c_virt, all_{i,k_1}),
117:                  (c_virt, all_{i,k_2})
118:              outputs:
119:                  (c_virt, refund),
120:                  (c_virt,
121:                      (if (k_1 − 1 > 1) then all_{k_1−1,min (k_2,n−1)} else False)
122:                      ∨ (if (k_2 + 1 < n) then all_{k_2+1,max (k_1,2)} else False)
123:                      ∨ revocation
124:                      ∨ (
```

60

```
125:                        if (k₁ − 1 ≤ 1 ∧ k₂ + 1 ≥ n) then virt_fund
126:                        else if (k₁ − 1 > 1 ∧ k₂ + 1 ≥ n) then virt_fund + t_{k₁−1}
127:                        else if (k₁ − 1 ≤ 1 ∧ k₂ + 1 < n) then virt_fund + t_{k₂+1}
128:                        else /*k₁ − 1 > 1 ∧ k₂ + 1 < n*/
129:                            virt_fund + max (t_{k₁−1}, t_{k₂+1})
130:                    )
131:                )
132:        end for
133:        return (
134:            TX₁,
135:            (TX_{2,k})_{k∈{m,...,l}\{i}},
136:            (TX_{3,k₁,k₂})_{(k₁,k₂)∈{m,...,i−1}×{i+1,...,l}}
137:        )
```

Fig. 48.

Process VIRT

1: // left and right refer to the two counterparties, with left being the one closer to the funder. Note difference with left/right meaning in VIRT.GETMIDTXS.

2: GETENDPOINTTX(i, n, c_{virt}, c_{left}, c_{right}, $pk_{\text{left,fund,old}}$, $pk_{\text{right,fund,old}}$, $pk_{\text{left,fund,new}}$, $pk_{\text{right,fund,new}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $pk_{\text{interm,rev}}$, $pk_{\text{endpoint,rev}}$, $(pk_{\text{all},j})_{j\in[n]}$, t):

3:     ensure $i \in \{1, n\}$
4:     ensure $c_{\text{left}} \geq c_{\text{virt}}$ // left party funds virtual channel
5:     $c_{\text{tot}} \leftarrow c_{\text{left}} + c_{\text{right}}$
6:     `old_fund` $\leftarrow 2/\{pk_{\text{left,fund,old}}, pk_{\text{right,fund,old}}\}$
7:     `new_fund` $\leftarrow (2/\{pk_{\text{left,fund,new}}, pk_{\text{right,fund,new}}\}) \lor 2/\{pk_{\text{left,rev}}, pk_{\text{right,rev}}\}$
8:     `virt_fund` $\leftarrow 2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$
9:     `revocation` $\leftarrow 2/\{pk_{\text{interm,rev}}, pk_{\text{endpoint,rev}}\}$
10:    **if** $i = 1$ **then** // funder's tx
11:        $all \leftarrow n/\{pk_{\text{all},1}, \ldots, pk_{\text{all},n}\} \land$ `"1"`
12:    **else** // fundee's tx
13:        $all \leftarrow n/\{pk_{\text{all},1}, \ldots, pk_{\text{all},n}\} \land$ `"n"`
14:    **end if**
15:    TX₁ $\leftarrow$ TX: // endpoints only have an "initiator" tx
16:        inputs:
17:            $(c_{\text{tot}}, $ `old_fund`$)$
18:        outputs:
19:            $(c_{\text{tot}} - c_{\text{virt}}, $ `new_fund`$)$,
20:            $(c_{\text{virt}}, all \lor $ `revocation` $\lor ($ `virt_fund` $+ t))$
21:    **return** TX₁

61
```

Fig. 49.

**Process** VIRT.SIBLINGSIGS()

1: parse input as $\text{sigs}_{\text{byLeft}}$
2: **if** $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
3: **if** $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
4: $(\text{TX}_{i,1}, (\text{TX}_{i,2,k})_{k \in \{m,\dots,l\} \setminus \{i\}}, (\text{TX}_{i,3,k_1,k_2})_{(k_1,k_2) \in \{m,\dots,i-1\}\{i+1,\dots,l\}}) \leftarrow$
   VIRT.GETMIDTXS$(i, n, c_{\text{virt}}, c_{i-1,\text{right}}, c_{i,\text{left}}, c_{i,\text{right}}, c_{i+1,\text{left}}, pk_{i-1,\text{right,fund,old}},$
   $pk_{i,\text{left,fund,old}}, pk_{i,\text{right,fund,old}}, pk_{i+1,\text{left,fund,old}}, pk_{i-1,\text{fund,new}}, pk_{i,\text{fund,new}},$
   $pk_{i,\text{fund,new}}, pk_{i+1,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{i,\text{out}}, pk_{1,\text{rev}}, pk_{i-1,\text{rev}},$
   $pk_{i,\text{rev}}, pk_{i+1,\text{rev}}, pk_{n,\text{rev}}, (pk_{h,j,k})_{h \in [n],j \in [n-1]\setminus\{1\},k \in [n-1]\setminus\{1,j\}}, (pk_{h,2,1})_{h \in [n]},$
   $(pk_{h,n-1,n})_{h \in [n]}, (t_h)_{h \in [n]\setminus\{1\}})$

5: // notation: $\text{sig}(\text{TX}, pk) := \text{sig}$ with ANYPREVOUT flag such that
   VERIFY$(\text{TX}, \text{sig}, pk) = \text{True}$
6: ensure that the following signatures are present in $\text{sigs}_{\text{byLeft}}$ and store them:
   – // $(l - m) \cdot (i - 1)$ signatures
7:     $\forall k \in \{m, \dots, l\} \setminus \{i\}, \forall j \in [i-1]:$
8:         $\text{sig}(\text{TX}_{i,2,k}, pk_{j,i,k})$
   – // $2 \cdot (i - m) \cdot (l - i) \cdot (i - 1)$ signatures
9:     $\forall k_1 \in \{m, \dots, i-1\}, \forall k_2 \in \{i+1, \dots, l\}, \forall j \in [i-1]:$
10:         $\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{j,i,k_1}), \text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{j,i,k_2})$
11: $\text{sigs}_{\text{toRight}} \leftarrow \text{sigs}_{\text{byLeft}}$

12: **for all** $j \in \{2, \dots, n-1\} \setminus \{i\}$ **do**
13:     **if** $j = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$
14:     **if** $j = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$
15:     $(\text{TX}_{j,1}, (\text{TX}_{j,2,k})_{k \in \{m',\dots,l'\} \setminus \{i\}}, (\text{TX}_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m',\dots,i-1\}\{i+1,\dots,l'\}}) \leftarrow$
       GETMIDTXS$(j, n, c_{\text{virt}}, c_{j-1,\text{right}}, c_{j,\text{left}}, c_{j,\text{right}}, c_{j+1,\text{left}}, pk_{j-1,\text{right,fund,old}},$
       $pk_{j,\text{left,fund,old}}, pk_{j,\text{right,fund,old}}, pk_{j+1,\text{left,fund,old}}, pk_{j-1,\text{fund,new}}, pk_{j,\text{fund,new}},$
       $pk_{j,\text{fund,new}}, pk_{j+1,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{j,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}},$
       $pk_{j,\text{rev}}, pk_{j+1,\text{rev}}, pk_{n,\text{rev}}, (pk_{k,p,s})_{k \in [n],p \in [n-1]\setminus\{1\},s \in [n-1]\setminus\{1,p\}}, (pk_{k,2,1})_{k \in [n]},$
       $(pk_{k,n-1,n})_{k \in [n]}, (t_k)_{k \in [n-1]\setminus\{1\}})$
16:     **if** $j < i$ **then** $\text{sigs} \leftarrow \text{sigs}_{\text{toLeft}}$ **else** $\text{sigs} \leftarrow \text{sigs}_{\text{toRight}}$
17:     **for all** $k \in \{m', \dots, l'\} \setminus \{j\}$ **do**
18:         add SIGN$(\text{TX}_{j,2,k}, sk_{i,j,k}, \text{ANYPREVOUT})$ to sigs
19:     **end for**
20:     **for all** $k_1 \in \{m', \dots, j-1\}, k_2 \in \{j+1, \dots, l'\}$ **do**
21:         add SIGN$(\text{TX}_{j,3,k_1,k_2}, sk_{i,j,k_1}, \text{ANYPREVOUT})$ to sigs
22:         add SIGN$(\text{TX}_{j,3,k_1,k_2}, sk_{i,j,k_2}, \text{ANYPREVOUT})$ to sigs
23:     **end for**
24: **end for**
25: **if** $i + 1 = n$ **then** // next hop is host_fundee

26: $\quad$ TX$_{n,1} \leftarrow$ VIRT.GETENDPOINTTX$(n, n, c_{\text{virt}}, c_{n-1,\text{right}}, c_{n,\text{left}},$
$pk_{n-1,\text{right,fund,old}}, pk_{n,\text{left,fund,old}}, pk_{n-1,\text{fund,new}}, pk_{n,\text{fund,new}}, pk_{\text{left,virt}},$
$pk_{\text{right,virt}}, pk_{n-1,\text{rev}}, pk_{n,\text{rev}}, (pk_{j,n-1,n})_{j\in[n]}, t_{n-1})$

27: **end if**

28: call $\bar{P}$.CIRCULATEVIRTUALSIGS$(\text{sigs}_{\text{toRight}})$ and assign returned value to $\text{sigs}_{\text{byRight}}$

29: ensure that the following signatures are present in $\text{sigs}_{\text{byRight}}$ and store them:
- $// (l-m) \cdot (n-i)$ signatures

30: $\quad \forall k \in \{m, \ldots, l\} \setminus \{i\}, \forall j \in \{i+1, \ldots, n\}:$

31: $\quad\quad \text{sig}(\text{TX}_{i,2,k}, pk_{j,i,k})$
- $// 2 \cdot (i-m) \cdot (l-i) \cdot (n-i)$ signatures

32: $\quad \forall k_1 \in \{m, \ldots, i-1\}, \forall k_2 \in \{i+1, \ldots, l\}, \forall j \in \{i+1, \ldots, n\}:$

33: $\quad\quad \text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{j,i,k_1}), \text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{j,i,k_2})$

34: output (VIRTUALSIGSBACK, $\text{sigs}_{\text{toLeft}}, \text{sigs}_{\text{byRight}})$

**Fig. 50.**

**Process** VIRT.INTERMEDIARYSIGS()

1: **if** $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$

2: **if** $i = n-1$ **then** $l \leftarrow n$ **else** $l \leftarrow n-1$

3: $(\text{TX}_{i,1}, (\text{TX}_{i,2,k})_{k\in\{m,\ldots,l\}\setminus\{i\}}, (\text{TX}_{i,3,k_1,k_2})_{(k_1,k_2)\in\{m,\ldots,i-1\}\{i+1,\ldots,l\}}) \leftarrow$
VIRT.GETMIDTXS$(i, n, c_{\text{virt}}, c_{i-1,\text{right}}, c_{i,\text{left}}, c_{i,\text{right}}, c_{i+1,\text{left}}, pk_{i-1,\text{right,fund,old}},$
$pk_{i,\text{left,fund,old}}, pk_{i,\text{right,fund,old}}, pk_{i+1,\text{left,fund,old}}, pk_{i-1,\text{fund,new}}, pk_{i,\text{fund,new}},$
$pk_{i,\text{fund,new}}, pk_{i+1,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{i,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}},$
$pk_{j,\text{rev}}, pk_{j+1,\text{rev}}, pk_{n,\text{rev}}, (pk_{h,j,k})_{h\in[n],j\in[n-1]\setminus\{1\},k\in[n-1]\setminus\{1,j\}}, (pk_{h,2,1})_{h\in[n]},$
$(pk_{h,n-1,n})_{h\in[n]}, (t_h)_{h\in[n-1]\setminus\{1\}})$

4: $//$ not verifying our signatures in $\text{sigs}_{\text{byLeft}}$, our (trusted) sibling will do that

5: input (VIRTUAL SIGS FORWARD, $\text{sigs}_{\text{byLeft}}$) to sibling

6: VIRT.SIBLINGSIGS()

7: $\text{sigs}_{\text{toLeft}} \leftarrow \text{sigs}_{\text{byRight}} + \text{sigs}_{\text{toLeft}}$

8: **if** $i = 2$ **then** $//$ previous hop is host_funder

9: $\quad$ TX$_{1,1} \leftarrow$ VIRT.GETENDPOINTTX$(1, n, c_{\text{virt}}, c_{1,\text{right}}, c_{2,\text{left}}, pk_{1,\text{right,fund,old}},$
$pk_{2,\text{left,fund,old}}, pk_{1,\text{fund,new}}, pk_{2,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{2,\text{rev}}, pk_{1,\text{rev}},$
$(pk_{j,2,1})_{j\in[n]}, t_2)$

10: **end if**

11: **return** $\text{sigs}_{\text{toLeft}}$

**Fig. 51.**

**Process** VIRT.HOSTFUNDEESIGS()

1: $\text{TX}_{n,1} \leftarrow$ VIRT.GETENDPOINTTX$(n, n, c_{\text{virt}}, c_{n-1,\text{right}}, c_{n,\text{left}},$
   $pk_{n-1,\text{right,fund,old}}, pk_{n,\text{right,fund,old}}, pk_{n-1,\text{fund,new}}, pk_{n,\text{fund,new}}, pk_{\text{left,virt}},$
   $pk_{\text{right,virt}}, pk_{n-1,\text{rev}}, pk_{n,\text{rev}}, (pk_{j,n-1,n})_{j\in[n]}, t_{n-1})$
2: **for all** $j \in [n-1] \setminus \{1\}$ **do**
3:     **if** $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
4:     **if** $j = n-1$ **then** $l \leftarrow n$ **else** $l \leftarrow n-1$
5:     $(\text{TX}_{j,1}, (\text{TX}_{j,2,k})_{k\in\{m,\dots,l\}\setminus\{j\}}, (\text{TX}_{i,3,k_1,k_2})_{(k_1,k_2)\in\{m,\dots,i-1\}\{i+1,\dots,l\}}) \leftarrow$
   VIRT.GETMIDTXS$(j, n, c_{\text{virt}}, c_{j-1,\text{right}}, c_{j,\text{left}}, c_{j,\text{right}}, c_{j+1,\text{left}},$
   $pk_{j-1,\text{right,fund,old}}, pk_{j,\text{left,fund,old}}, pk_{j,\text{right,fund,old}}, pk_{j+1,\text{left,fund,old}},$
   $pk_{j-1,\text{fund,new}}, pk_{j,\text{fund,new}}, pk_{j,\text{fund,new}}, pk_{j+1,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}},$
   $pk_{j,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{j+1,\text{rev}}, pk_{n,\text{rev}},$
   $(pk_{h,s,k})_{h\in[n],s\in[n-1]\setminus\{1\},k\in[n-1]\setminus\{1,s\}}, (pk_{h,2,1})_{h\in[n]}, (pk_{h,n-1,n})_{h\in[n]},$
   $(t_h)_{h\in[n-1]\setminus\{1\}})$
6:     $\text{sigs}_{\text{toLeft}} \leftarrow \emptyset$
7:     **for all** $k \in \{m,\dots,l\} \setminus \{j\}$ **do**
8:         add SIGN$(\text{TX}_{j,2,k}, sk_{n,j,k}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toLeft}}$
9:     **end for**
10:     **for all** $k_1 \in \{m,\dots,j-1\}, k_2 \in \{j+1,\dots,l\}$ **do**
11:         add SIGN$(\text{TX}_{j,3,k_1,k_2}, sk_{n,j,k_1}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toLeft}}$
12:         add SIGN$(\text{TX}_{j,3,k_1,k_2}, sk_{n,j,k_2}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toLeft}}$
13:     **end for**
14: **end for**
15: **return** $\text{sigs}_{\text{toLeft}}$

**Fig. 52.**

**Process** VIRT.HOSTFUNDERSIGS()

1: **for all** $j \in [n-1] \setminus \{1\}$ **do**
2:     **if** $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
3:     **if** $j = n-1$ **then** $l \leftarrow n$ **else** $l \leftarrow n-1$
4:     $(\text{TX}_{j,1}, (\text{TX}_{j,2,k})_{k\in\{m,\dots,l\}\setminus\{j\}}, (\text{TX}_{i,3,k_1,k_2})_{(k_1,k_2)\in\{m,\dots,i-1\}\{i+1,\dots,l\}}) \leftarrow$
   VIRT.GETMIDTXS$(j, n, c_{\text{virt}}, c_{j-1,\text{right}}, c_{j,\text{left}}, c_{j,\text{right}}, c_{j+1,\text{left}},$
   $pk_{j-1,\text{right,fund,old}}, pk_{j,\text{left,fund,old}}, pk_{j,\text{right,fund,old}}, pk_{j+1,\text{left,fund,old}},$
   $pk_{j-1,\text{fund,new}}, pk_{j,\text{fund,new}}, pk_{j,\text{fund,new}}, pk_{j+1,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}},$
   $pk_{j,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{j+1,\text{rev}}, pk_{n,\text{rev}},$
   $(pk_{h,s,k})_{h\in[n],s\in[n-1]\setminus\{1\},k\in[n-1]\setminus\{1,s\}}, (pk_{h,2,1})_{h\in[n]}, (pk_{h,n-1,n})_{h\in[n]},$
   $(t_h)_{h\in[n-1]\setminus\{1\}})$
5:     $\text{sigs}_{\text{toRight}} \leftarrow \emptyset$
6:     **for all** $k \in \{m,\dots,l\} \setminus \{j\}$ **do**
7:         add SIGN$(\text{TX}_{j,2,k}, sk_{1,j,k}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toRight}}$
8:     **end for**
9:     **for all** $k_1 \in \{m,\dots,j-1\}, k_2 \in \{j+1,\dots,l\}$ **do**
10:         add SIGN$(\text{TX}_{j,3,k_1,k_2}, sk_{1,j,k_1}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toRight}}$

11:         add $\text{SIGN}(\text{TX}_{j,3,k_1,k_2}, sk_{1,j,k_2}, \texttt{ANYPREVOUT})$ to $\text{sigs}_{\text{toRight}}$
12:     **end for**
13: **end for**
14: call $\text{VIRT.CIRCULATEVIRTUALSIGS}(\text{sigs}_{\text{toRight}})$ of $\bar{P}$ and assign output to $\text{sigs}_{\text{byRight}}$
15: $\text{TX}_{1,1} \leftarrow \text{VIRT.GETENDPOINTTX}(1, n, c_{\text{virt}}, c_{1,\text{right}}, c_{2,\text{left}}, pk_{1,\text{right,fund,old}}, pk_{2,\text{left,fund,old}}, pk_{1,\text{fund,new}}, pk_{2,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{2,\text{rev}}, pk_{1,\text{rev}}, (pk_{j,2,1})_{j \in [n]}, t_2)$
16: **return** $(\text{OK})$

**Fig. 53.**

---

**Process** $\text{VIRT.CIRCULATEVIRTUALSIGS}(\text{sigs}_{\text{byLeft}})$

1: **if** $1 < i < n$ **then** // we are not `host_funder` nor `host_fundee`
2:     **return** $\text{VIRT.INTERMEDIARYSIGS}()$
3: **else if** $i = 1$ **then** // we are `host_funder`
4:     **return** $\text{VIRT.HOSTFUNDERSIGS}()$
5: **else if** $i = n$ **then** // we are `host_fundee`
6:     **return** $\text{VIRT.HOSTFUNDEESIGS}()$
7: **end if** // it is always $1 \le i \le n$ – c.f. Fig. 46, l. 12 and l. 37

**Fig. 54.**

---

**Process** $\text{VIRT.CIRCULATEFUNDINGSIGS}(\text{sigs}_{\text{byLeft}})$

1: **if** $1 < i < n$ **then** // we are not endpoint
2:     **if** $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
3:     **if** $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
4:     ensure that the following signatures are present in $\text{sigs}_{\text{byLeft}}$ and store them:
  – // 1 signature
5:        $\text{sig}(\text{TX}_{i,1}, pk_{i-1,\text{right,fund,old}})$
  – // $n - 3 + \chi_{i=2} + \chi_{i=n-1}$ signatures
6:        $\forall k \in \{m, \ldots, l\} \setminus \{i\}$
7:          $\text{sig}(\text{TX}_{i,2,k}, pk_{i-1,\text{right,fund,old}})$
8:     input $(\text{VIRTUAL BASE SIG FORWARD}, \text{sigs}_{\text{byLeft}})$ to `sibling`
9:     extract and store $\text{sig}(\text{TX}_{i,1}, pk_{i-1,\text{right,fund,old}})$ and $\forall k \in \{m, \ldots, l\} \setminus \{i\}$ $\text{sig}(\text{TX}_{i,2,k}, pk_{i-1,\text{right,fund,old}})$ from $\text{sigs}_{\text{byLeft}}$ // same signatures as `sibling`
10:     $\text{sigs}_{\text{toRight}} \leftarrow \{\text{SIGN}(\text{TX}_{i+1,1}, sk_{i,\text{right,fund,old}}, \texttt{ANYPREVOUT})\}$
11:     **if** $i + 1 < n$ **then**
12:        **if** $i + 1 = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$

13:  **for all** $k \in \{2, \ldots, l'\}$ **do**
14:      add SIGN($\mathrm{TX}_{i+1,2,k}$, $sk_{i,\mathrm{right,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toRight}}$
15:  **end for**
16:  **else** $// \ i + 1 = n$
17:      add SIGN($\mathrm{TX}_{n,1}$, $sk_{i,\mathrm{right,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toRight}}$
18:  **end if**
19:  call VIRT.CIRCULATEFUNDINGSIGS(sigs$_{\mathrm{toRight}}$) of $\bar{P}$ and assign returned values to sigs$_{\mathrm{byRight}}$
20:  ensure that the following signatures are present in sigs$_{\mathrm{byRight}}$ and store them:
  $-$ $// \ 1$ signature
21:      $\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i+1,\mathrm{left,fund,old}})$
  $-$ $// \ n - 3 + \chi_{i=2} + \chi_{i=n-1}$ signatures
22:      $\forall k \in \{m, \ldots, l\} \setminus \{i\}$
23:          $\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i+1,\mathrm{right,fund,old}})$
24:  output (VIRTUAL BASE SIG BACK, sigs$_{\mathrm{byRight}}$)
25:  extract and store $\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i+1,\mathrm{right,fund,old}})$ and $\forall k \in \{m, \ldots, l\} \setminus \{i\}$ $\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i+1,\mathrm{right,fund,old}})$ from sigs$_{\mathrm{byRight}}$ $//$ same signatures as `sibling`
26:  sig$_{\mathrm{toLeft}} \leftarrow \{$SIGN($\mathrm{TX}_{i-1,1}$, $sk_{i,\mathrm{left,fund,old}}$, ANYPREVOUT)$\}$
27:  **if** $i - 1 > 1$ **then**
28:      **if** $i - 1 = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$
29:      **for all** $k \in \{m', \ldots, n - 1\}$ **do**
30:          add SIGN($\mathrm{TX}_{i-1,2,k}$, $sk_{i,\mathrm{left,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toLeft}}$
31:      **end for**
32:  **else** $// \ i - 1 = 1$
33:      add SIGN($\mathrm{TX}_{1,1}$, $sk_{i,\mathrm{left,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toLeft}}$
34:  **end if**
35:  **return** sigs$_{\mathrm{toLeft}}$
36: **else if** $i = 1$ **then** $//$ we are `host_funder`
37:  sigs$_{\mathrm{toRight}} \leftarrow \{$SIGN($\mathrm{TX}_{2,1}$, $sk_{1,\mathrm{right,fund,old}}$, ANYPREVOUT)$\}$
38:  **if** $2 = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$
39:  **for all** $k \in \{3, \ldots, l'\}$ **do**
40:      add SIGN($\mathrm{TX}_{2,2,k}$, $sk_{1,\mathrm{right,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toRight}}$
41:  **end for**
42:  call VIRT.CIRCULATEFUNDINGSIGS(sigs$_{\mathrm{toRight}}$) of $\bar{P}$ and assign returned value to sigs$_{\mathrm{byRight}}$
43:  ensure that $\mathrm{sig}(\mathrm{TX}_{1,1}, pk_{2,\mathrm{left,fund,old}})$ is present in sigs$_{\mathrm{byRight}}$ and store it
44:  **return** (OK)
45: **else if** $i = n$ **then** $//$ we are `host_fundee`
46:  ensure $\mathrm{sig}(\mathrm{TX}_{n,1}, pk_{n-1,\mathrm{right,fund,old}})$ is present in sigs$_{\mathrm{byLeft}}$ and store it
47:  sigs$_{\mathrm{toLeft}} \leftarrow \{$SIGN($\mathrm{TX}_{n-1,1}$, $sk_{n,\mathrm{left,fund,old}}$, ANYPREVOUT)$\}$
48:  **if** $n - 1 = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$
49:  **for all** $k \in \{m', \ldots, n - 2\}$ **do**
50:      add SIGN($\mathrm{TX}_{n-1,2,k}$, $sk_{n,\mathrm{left,fund,old}}$, ANYPREVOUT) to sigs$_{\mathrm{toLeft}}$
51:  **end for**
52:  **return** sigs$_{\mathrm{toLeft}}$
53: **end if** $//$ it is always $1 \le i \le n$ – c.f. Fig. 46, l. 12 and l. 37

**Fig. 55.**

**Process** VIRT.CIRCULATEREVOCATIONS(`revoc_by_prev`)

1: **if** `revoc_by_prev` is given as argument **then** // we are not `host_funder`
2:     ensure **guest**.PROCESSREMOTEREVOCATION(`revoc_by_prev`) returns (OK)
3: **else** // we are `host_funder`
4:     `revoc_for_next` ← **guest**.REVOKEPREVIOUS()
5:     input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to $\Sigma$
6:     `last_poll` ← $|\Sigma|$
7:     call VIRT.CIRCULATEREVOCATIONS(`revoc_for_next`) of $\bar{P}$ and assign returned value to `revoc_by_next`
8:     ensure **guest**.PROCESSREMOTEREVOCATION(`revoc_by_next`) returns (OK)
    // If the "ensure" fails, the opening process freezes, this is intentional. The channel can still close via (FORCECLOSE)
9:     **return** (OK)
10: **end if**
11: **if** we have a **sibling** **then** // we are not `host_fundee` nor `host_funder`
12:     input (VIRTUAL REVOCATION FORWARD) to **sibling**
13:     `revoc_for_next` ← **guest**.REVOKEPREVIOUS()
14:     input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to $\Sigma$
15:     `last_poll` ← $|\Sigma|$
16:     call VIRT.CIRCULATEREVOCATIONS(`revoc_for_next`) of $\bar{P}$ and assign output to `revoc_by_next`
17:     ensure **guest**.PROCESSREMOTEREVOCATION(`revoc_by_next`) returns (OK)
18:     output (HOSTS READY, $t_i$) to **guest** and expect reply (HOST ACK)
19:     output (VIRTUAL REVOCATION BACK)
20: **end if**
21: `revoc_for_prev` ← **guest**.REVOKEPREVIOUS()
22: **if** $1 < i < n$ **then** // we are intermediary
23:     output (HOSTS READY, $t_i$) to **guest** and expect reply (HOST ACK) // $p$ is every how many blocks we have to check the chain
24: **else** // we are `host_fundee`, case of `host_funder` covered earlier
25:     output (HOSTS READY, $p + \sum\limits_{j=2}^{n-1}(s - 1 + t_j)$) to **guest** and expect reply (HOST ACK)
26: **end if**
27: **return** `revoc_for_prev`

**Fig. 56.**

**Process** VIRT − poll

1: On input (CHECK FOR LATERAL CLOSE) by $R \in \{\texttt{guest}, \texttt{funder}, \texttt{fundee}\}$:
2:     input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to $\Sigma$
3:     $k_1 \leftarrow 0$
4:     **if** $\text{TX}_{i-1,1}$ is defined and $\text{TX}_{i-1,1} \in \Sigma$ **then**
5:         $k_1 \leftarrow i - 1$
6:     **end if**
7:     **for all** $k \in [i-2]$ **do**
8:         **if** $\text{TX}_{i-1,2,k}$ is defined and $\text{TX}_{i-1,2,k} \in \Sigma$ **then**
9:             $k_1 \leftarrow k$
10:         **end if**
11:     **end for**
12:     $k_2 \leftarrow 0$
13:     **if** $\text{TX}_{i+1,1}$ is defined and $\text{TX}_{i+1,1} \in \Sigma$ **then**
14:         $k_2 \leftarrow i + 1$
15:     **end if**
16:     **for all** $k \in \{i+2, \ldots, n\}$ **do**
17:         **if** $\text{TX}_{i+1,2,k}$ is defined and $\text{TX}_{i+1,2,k} \in \Sigma$ **then**
18:             $k_2 \leftarrow k$
19:         **end if**
20:     **end for**
21:     $\texttt{last\_poll} \leftarrow |\Sigma|$
22:     **if** $k_1 > 0 \vee k_2 > 0$ **then** // at least one neighbour has published its TX
23:         ignore all messages except for (CHECK IF CLOSING) by $R$
24:         $State \leftarrow \text{CLOSING}$
25:         $\text{sigs} \leftarrow \emptyset$
26:     **end if**
27:     **if** $k_1 > 0 \wedge k_2 > 0$ **then** // both neighbours have published their TXs
28:         add $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$ to sigs
29:         add $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$ to sigs
30:         add $\text{SIGN}(\text{TX}_{i,3,k_1,k_2}, sk_{i,i,k_1}, \texttt{ANYPREVOUT})$ to sigs
31:         add $\text{SIGN}(\text{TX}_{i,3,k_1,k_2}, sk_{i,i,k_2}, \texttt{ANYPREVOUT})$ to sigs
32:         input (SUBMIT, $\text{TX}_{i,3,k_1,k_2}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
33:     **else if** $k_1 > 0$ **then** // only left neighbour has published its TX
34:         add $(\text{sig}(\text{TX}_{i,2,k_1}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$ to sigs
35:         add $\text{SIGN}(\text{TX}_{i,2,k_1}, sk_{i,i,k_1}, \texttt{ANYPREVOUT})$ to sigs
36:         add $\text{SIGN}(\text{TX}_{i,2,k_1}, sk_{i,\text{left,fund,old}}, \texttt{ANYPREVOUT})$ to sigs
37:         input (SUBMIT, $\text{TX}_{i,2,k_1}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
38:     **else if** $k_2 > 0$ **then** // only right neighbour has published its TX
39:         add $(\text{sig}(\text{TX}_{i,2,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$ to sigs
40:         add $\text{SIGN}(\text{TX}_{i,2,k_2}, sk_{i,i,k_2}, \texttt{ANYPREVOUT})$ to sigs
41:         add $\text{SIGN}(\text{TX}_{i,2,k_2}, sk_{i,\text{right,fund,old}}, \texttt{ANYPREVOUT})$ to sigs
42:         input (SUBMIT, $\text{TX}_{i,2,k_2}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
43:     **end if**

**Fig. 57.**

**Process** VIRT – On input (FORCECLOSE) by $R$:

1: // At most one of funder, fundee is defined
2: ensure $R \in \{\text{guest}, \text{funder}, \text{fundee}\}$
3: **if** $State = \text{CLOSED}$ **then** output (CLOSED) to $R$
4: **if** $State = \text{GUEST PUNISHED}$ **then** output (GUEST PUNISHED) to $R$
5: ensure $State \in \{\text{OPEN}, \text{CLOSING}\}$
6: **if** $\text{host}_P \neq \mathcal{G}_{\text{Ledger}}$ **then** // $\text{host}_P$ is a VIRT
7:      ignore all messages except for output (CLOSED) by $\text{host}_P$. Also relay to $\text{host}_P$ any (CHECK IF CLOSING) or (FORCECLOSE) input received
8:      input (FORCECLOSE) to $\text{host}_P$
9: **end if**
10: // if we have a $\text{host}_P$, continue from here on output (CLOSED) by it
11: send (READ) to $\mathcal{G}_{\text{Ledger}}$ as $R$ and assign reply to $\Sigma$
12: **if** $i \in \{1, n\} \land (\text{TX}_{(i-1)+\frac{2}{n-1}(n-i),1} \in \Sigma \lor \exists k \in [n] : \text{TX}_{(i-1)+\frac{2}{n-1}(n-i),2,k} \in \Sigma)$
     **then** // we are an endpoint and our counterparty has closed – 1st subscript of TX is 2 if $i = 1$ and $n - 1$ if $i = n$
13:      ignore all messages except for (CHECK IF CLOSING) and (FORCECLOSE) by $R$
14:      $State \leftarrow \text{CLOSING}$
15:      give up execution token // control goes to $\mathcal{E}$
16: **end if**
17: let $\texttt{tx}$ be the unique TX among $\text{TX}_{i,1}, (\text{TX}_{i,2,k})_{k \in [n]}, (\text{TX}_{i,3,k_1,k_2})_{k_1, k_2 \in [n]}$ that can be appended to $\Sigma$ in a valid way // ignore invalid subscript combinations
18: let $\texttt{sigs}$ be the set of stored signatures that sign $\texttt{tx}$
19: add $\text{SIGN}(\texttt{tx}, sk_{i,\text{left,fund,old}}, \texttt{ANYPREVOUT}), \text{SIGN}(\texttt{tx}, sk_{i,\text{right,fund,old}}, \texttt{ANYPREVOUT}), (\text{SIGN}(\texttt{tx}, sk_{i,j,k}, \texttt{ANYPREVOUT}))_{j,k \in [n]}$ to $\texttt{sigs}$ // ignore invalid signatures
20: ignore all messages except for (CHECK IF CLOSING) by $R$
21: $State \leftarrow \text{CLOSING}$
22: send (SUBMIT, $\texttt{tx}$, $\texttt{sigs}$) to $\mathcal{G}_{\text{Ledger}}$

**Fig. 58.**

**Process** VIRT – On input (CHECK IF CLOSING) by $R$:

1: ensure $State = \text{CLOSING}$
2: ensure $R \in \{\text{guest}, \text{funder}, \text{fundee}\}$
3: send (READ) to $\mathcal{G}_{\text{Ledger}}$ as $R$ and assign reply to $\Sigma$
4: **if** $i = 1$ **then** // we are host_funder

5:     ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\mathrm{virt}}$ coins and a $2/\{pk_{1,\mathrm{fund,new}}, pk_{2,\mathrm{fund,new}}\}$ spending method with expired/non-existent timelock in $\Sigma$ // new base funding output

6:     ensure that there exists an output with $c_{\mathrm{virt}}$ coins and a $2/\{pk_{\mathrm{left,virt}}, pk_{\mathrm{right,virt}}\}$ spending method with expired/non-existent timelock in $\Sigma$ // virtual funding output

7: **else if** $i = n$ **then** // we are host_fundee

8:     ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\mathrm{virt}}$ coins and a $2/\{pk_{n-1,\mathrm{fund,new}}, pk_{n,\mathrm{fund,new}}\}$ spending method with expired/non-existent timelock in $\Sigma$ // new base funding output

9:     ensure that there exists an output with $c_{\mathrm{virt}}$ coins and a $2/\{pk_{\mathrm{left,virt}}, pk_{\mathrm{right,virt}}\}$ spending method with expired/non-existent timelock in $\Sigma$ // virtual funding output

10: **else** // we are intermediary

11:     **if** side $=$ "left" **then** $j \leftarrow i - 1$ **else** $j \leftarrow i + 1$ // side is defined for all intermediaries – c.f. Fig. 46, l. 11

12:     ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\mathrm{virt}}$ coins and a $2/\{pk_{i,\mathrm{fund,new}}, pk_{j,\mathrm{fund,new}}\}$ spending method with expired/non-existent timelock and an output with $c_{\mathrm{virt}}$ coins and a $pk_{i,\mathrm{out}}$ spending method with expired/non-existent timelock in $\Sigma$

13: **end if**

14: $State \leftarrow$ CLOSED

15: output (CLOSED) to $R$

**Fig. 59.**

**Process** VIRT – On (COOP CLOSE, `sig_bal`, `left_comms_revkeys`) by $\bar{P}$:

// we are left intermediary or host of fundee

1: ensure $State =$ OPEN

2: $State \leftarrow$ COOP CLOSING

3: parse `sig_bal` as $(c_1, c_2), \mathrm{sig}_1, \mathrm{sig}_2$

4: ensure $c_{\mathrm{virt}} = c_1 + c_2$

5: ensure VERIFY$((c_1, c_2), \mathrm{sig}_1, pk_{\mathrm{left,virt}}) =$ True

6: ensure VERIFY$((c_1, c_2), \mathrm{sig}_2, pk_{\mathrm{right,virt}}) =$ True

7: $State \leftarrow$ COOP CLOSING

8: extract $\mathrm{sig}_{i-1,\mathrm{right},C}$, $pk_{i-1,\mathrm{right},R}$ from `left_comms_revkeys`

9: output (CHECK COOP CLOSE, $(c_1, c_2)$, $\mathrm{sig}_{i-1,\mathrm{right},C}$, $pk_{i-1,\mathrm{right},R}$) to guest

10: ensure $State =$ OPEN // executed by guest

11: $State \leftarrow$ COOP CLOSING

12: store received signature as $\mathrm{sig}_{\bar{P},C,i+1}$ // in guests, $i$ is the current state number

13: store received revocation key as $pk_{\bar{P},R,i+2}$

14: remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$ and $pk'_{\bar{P},F}$

70

15: $C_{P,i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c_1 + c_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c_2, (pk_{P,\text{out}} + (p + s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\}), (c_{\bar{P}} + c_1, pk_{\bar{P},\text{out}})\}$

16: ensure VERIFY$(C_{P,i+1}, \text{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = $ True

17: input (COOP CLOSE CHECK OK) to host$_P$

18: **if** $i < n$ **then** // we are intermediary

19:     input (COOP CLOSE, left_comms_keys) to sibling

20:     ensure $State = $ OPEN // executed by sibling

21:     $State \leftarrow$ COOP CLOSING

22:     output (COOP CLOSE SIGN COMM, $(c_1, c_2)$) to guest

23:     ensure $State = $ OPEN // executed by guest of sibling

24:     $State \leftarrow$ COOP CLOSING

25:     remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$ and $pk'_{\bar{P},F}$

26:     $C_{\bar{P},i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c_1 + c_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c_1, pk_{P,\text{out}}), (c_{\bar{P}} + c_2, (pk_{\bar{P},\text{out}} + (p + s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})\}$

27:     $\text{sig}_{P,C,i+1} \leftarrow$ SIGN$(C_{\bar{P},i+1}, sk'_{P,F})$

28:     $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow$ KEYGEN()

29:     input (NEW COMM TX, $\text{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to host$_P$

30:     rename received signature to $\text{sig}_{i,\text{right},C}$ // executed by sibling

31:     rename received public key to $pk_{i,\text{right},R}$ // in hosts, $i$ is our hop number

32:     send (COOP CLOSE, sig_bal, (left_comms_keys, $\text{sig}_{i,\text{right},C}$, $pk_{i,\text{right},R}$) to $\bar{P}$ and expect reply (COOP CLOSE BACK, (right_comms_revkeys, right_revocations))

33:     $R_{\text{loc,right}} \leftarrow$ TX {input: $(c_{\text{virt}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i,\text{out}})\}$

34:     extract $\text{sig}_{i+1,\text{right,rev}}$ from right_revocations

35:     ensure VERIFY$(R_{\text{loc,right}}, \text{sig}_{i+1,\text{right,rev}}, pk_{i+1,\text{rev}}) = $ True

36:     extract $\text{sig}_{i+1,\text{left},C}$ from right_comms_revkeys

37:     extract $\text{sig}_{i+1,\text{left},R}$ from right_revocations

38:     extract $pk_{i+1,\text{left},R}$ from right_comms_revkeys

39:     output (VERIFY COMM REV, $\text{sig}_{i+1,\text{left},C}$, $\text{sig}_{i+1,\text{left},R}$, $pk_{i+1,\text{left},R}$) to guest

40:     store received public key as $pk_{\bar{P},R,i+2}$ // executed by guest of sibling

41:     store $\text{sig}_{i+1,\text{left},C}$ as $\text{sig}_{\bar{P},C,i+1}$, $pk_{\bar{P},R,i+2}$

42:     $C_{P,i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c_1 + c_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c_1, (pk_{P,\text{out}} + (p + s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\}), (c_{\bar{P}} + c_2, pk_{\bar{P},\text{out}})\}$

43:     ensure VERIFY$(C_{P,i+1}, \text{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = $ True

44:     store $\text{sig}_{i+1,\text{left},R}$ as $\text{sig}_{\bar{P},R,i}$

45:     $R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_P + c_{\bar{P}}, pk_{P,\text{out}})\}$

46:     ensure VERIFY$(R_{P,i}, \text{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = $ True

47:     input (COMM REV VERIFIED) to host$_P$

48:     output (COOP CLOSE BACK, right_comms_revkeys, right_revocations) to sibling // executed by sibling

49:     $R_{\text{loc,left}} \leftarrow$ TX {input: $(c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{i-1,\text{rev}}, pk_{i,\text{rev}}, pk_{n,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i,\text{out}})\}$ // the input corresponds to the revocation path of the virtual output of all virtual txs owned by $\bar{P}$

50:     extract $\text{sig}_{n,i,\text{left,rev}}$ from right_revocations

51:     ensure VERIFY$(R_{\text{loc,left}}, \text{sig}_{n,\text{left,rev}}, pk_{n,\text{rev}}) = $ True

52: **else** // $i = n$, we are host of fundee

53:     output (REVOKE) to **fundee**

54:     $R_{\bar{P},i} \leftarrow$ TX $\{$input: $C_{P,i}$.outputs.$P$, output: $(c_P, pk_{\bar{P},\text{out}})\}$ // executed by **fundee**

55:     $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$

56:     input (REVOCATIONS, $\text{sig}_{P,R,i}$) to **host**$_P$

57:     rename received signature to $\text{sig}_{n,\text{right},R}$

58:     **for all** $j \in \{2, \ldots, n\}$ **do**

59:         $R_{j,\text{left}} \leftarrow$ TX $\{$input: $(c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{n,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{j,\text{out}})\}$

60:         $\text{sig}_{n,j,\text{left},\text{rev}} \leftarrow \text{SIGN}(R_{j,\text{left}}, sk_{n,\text{rev}})$

61:     **end for**

62: **end if**

63: output (NEW COMM REV) to **guest**

64: $C_{\bar{P},i+1} \leftarrow$ TX $\{$input: $(c_P + c_{\bar{P}} + c_1 + c_2, 2\{pk'_{\bar{P},F}, pk'_{P,F}\})$, outputs: $(c_{\bar{P}} + c_1, (pk_{\bar{P},\text{out}} + (p+s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\}), (c_P + c_2, pk_{P,\text{out}})\}$ // executed by **guest**

65: $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk'_{P,F})$

66: $R_{\bar{P},i} \leftarrow$ TX $\{$input: $C_{P,i}$.outputs.$P$, output: $(c_P + c_{\bar{P}}, pk_{\bar{P},\text{out}})\}$

67: $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$

68: $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \text{KEYGEN}()$

69: input (NEW COMM REV, $\text{sig}_{P,C,i+1}$, $\text{sig}_{P,R,i}$, $pk_{P,R,i+2}$) to **host**$_P$

70: rename $\text{sig}_{P,C,i+1}$ to $\text{sig}_{i,\text{left},C}$

71: rename $\text{sig}_{P,R,i}$ to $\text{sig}_{i,\text{left},R}$

72: rename received public key to $pk_{i,\text{left},R}$

73: $R_{\text{rem},\text{left}} \leftarrow$ TX $\{$input: $(c_{\text{virt}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i-1,\text{out}})\}$

74: $\text{sig}_{i,\text{left},\text{rev}} \leftarrow \text{SIGN}(R_{\text{rem},\text{left}}, sk_{i,\text{rev}})$

75: **if** $i < n$ **then** // we are intermediary

76:     $M \leftarrow$ (COOP CLOSE BACK, ((**right_comms_revkeys**, $\text{sig}_{i,\text{left},C}$, $pk_{i,\text{left},R}$), (**right_revocations**, $\text{sig}_{i,\text{left},\text{rev}}$, $\text{sig}_{i,\text{left},R}$)))

77: **else** // $i = n$, we are host of fundee

78:     $M \leftarrow$ (COOP CLOSE BACK, ($\text{sig}_{i,\text{left},C}$, $pk_{i,\text{left},R}$, $\text{sig}_{n,\text{left},R}$), ($\text{sig}_{n,\text{left},\text{rev}}$, $(\text{sig}_{n,j,\text{left},\text{rev}})_{j \in \{2,\ldots,n\}}$))

79: **end if**

80: send $M$ to $\bar{P}$ and expect reply (COOP CLOSE REVOCATIONS, **left_revocations**)

81: extract $\text{sig}_{i-1,\text{right},R}$, $\text{sig}_{1,i,\text{right},\text{rev}}$, $\text{sig}_{i-1,\text{right},\text{rev}}$ from **left_revocations**

82: ensure $\text{VERIFY}(R_{\text{loc},\text{left}}, \text{sig}_{1,\text{right},\text{rev}}, pk_{1,\text{rev}}) = \text{True}$

83: ensure $\text{VERIFY}(R_{\text{loc},\text{left}}, \text{sig}_{i-1,\text{right},\text{rev}}, pk_{i-1,\text{rev}}) = \text{True}$

84: output (VERIFY REV, $\text{sig}_{i-1,\text{right},R}$, **host**$_P$) to **guest**

85: store received signature as $\text{sig}_{barP,R,i}$ // executed by **guest**

86: $R_{P,i} \leftarrow$ TX $\{$input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: $(c_P + c_{\bar{P}}, pk_{P,\text{out}})\}$

87: ensure $\text{VERIFY}(R_{P,i}, \text{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = \text{True}$

88: add **host**$_P$ to list of old hosts

89: assign received host to **host**$_P$

90: $i \leftarrow i + 1$; $c_P \leftarrow c_P + c_2$; $c_{\bar{P}} \leftarrow c_{\bar{P}} + c_1$

91: add $sk_{P,F}, pk_{P,F}, pk_{\bar{P},F}$ to list of old enabler channel funding keys

92: $(sk_{P,F}, pk_{P,F}) \leftarrow (sk'_{P,F}, pk'_{P,F})$

93: $State \leftarrow \text{OPEN}$

94: input (REV VERIFIED) to last old host
95: $State \leftarrow$ COOP CLOSED
96: **if** $i < n$ **then** // we are intermediary
97:     send (COOP CLOSE REVOCATIONS, `left_revocations`) to `sibling`
98:     output (COOP CLOSE REVOCATIONS, `host`$_P$) to `guest` // executed by `sibling`
99:     $R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.$P$, output: $(c_P, pk_{\bar{P},\text{out}})$} // executed by `guest` of `sibling`
100:     $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$
101:     add `host`$_P$ to list of old hosts
102:     assign received host to `host`$_P$
103:     add $sk_{P,F}, pk_{P,F}, pk_{\bar{P},F}$ to list of old enabler channel funding keys
104:     $(sk_{P,F}, pk_{P,F}) \leftarrow (sk'_{P,F}, pk'_{P,F})$
105:     $State \leftarrow$ OPEN
106:     input (REVOCATION, $\text{sig}_{P,R,i}$) to last old host
107:     rename received signature to $\text{sig}_{i,\text{right},R}$ // executed by `sibling`
108:     $R_{\text{rem,right}} \leftarrow$ TX {input: $(c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{i,\text{rev}}, pk_{i+1,\text{rev}}, pk_{n,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i+1,\text{out}})$}
109:     $\text{sig}_{i,\text{right,rev}} \leftarrow \text{SIGN}(R_{\text{rem,right}}, sk_{i,\text{rev}})$
110:     $State \leftarrow$ COOP CLOSED
111:     send (COOP CLOSE REVOCATIONS, (`left_revocations`, $\text{sig}_{i,\text{right},R}$, $\text{sig}_{i,\text{right,rev}}$) to $\bar{P}$)
112: **else** // $i = n$, we are host of fundee
113:     extract $\text{sig}_{1,\text{right},R}$ from `left_revocations`
114:     output (VERIFY REVOCATION, $\text{sig}_{1,\text{right},R}$) to `fundee`
115:     store received signature as $\text{sig}_{\bar{P},R,i}$ // executed by `fundee`
116:     $R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\text{out}})$}
117:     ensure VERIFY($R_{P,i}$, $\text{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
118:     $State \leftarrow$ COOP CLOSED
119:     **if** `close_initiator` $= P$ **then** // $\mathcal{E}$ instructed us to close the channel
120:         execute code of Fig. 43
121:     **else** // $\mathcal{E}$ instructed another party to close the channel
122:         send (COOPCLOSED) to `close_initiator`
123:     **end if**
124: **end if**

**Fig. 60.**

---

**Process** VIRT – punishment handling

1: On input (USED REVOCATION) by `guest`: // (USED REVOCATION) by funder/fundee is ignored
2:     $State \leftarrow$ GUEST PUNISHED
3:     input (USED REVOCATION) to `host`$_P$, expect reply (USED REVOCATION OK)
4:     **if** `funder` or `fundee` is defined **then**

```
 5:        output (ENABLER USED REVOCATION) to it
 6:     else  // sibling is defined
 7:        output (ENABLER USED REVOCATION) to sibling
 8:     end if

 9: On input (ENABLER USED REVOCATION) by sibling:
10:     State ← GUEST PUNISHED
11:     output (ENABLER USED REVOCATION) to guest

12: On output (USED REVOCATION) by host_P:
13:     State ← GUEST PUNISHED
14:     if funder or fundee is defined then
15:        output (ENABLER USED REVOCATION) to it
16:     else  // sibling is defined
17:        output (ENABLER USED REVOCATION) to sibling
18:     end if
```

Fig. 61.

11 Liveness

Proposition 1. *Consider a synchronised honest party that submits a transaction* `tx` *to the ledger functionality [49] by the time the block indexed by h is added to* `state` *in its view. Then* `tx` *is guaranteed to be included in the block range* $[h + 1, h + s]$*, where* $s = (2 + q)\texttt{windowSize}$ *and* $q = \lceil(\texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2})/\texttt{minTime}_{\texttt{window}}\rceil$.

The proof can be found in [52].

12 Omitted Proofs

Lemma 4 (Real world balance security). *Consider a real world execution with* $P \in \{Alice, Bob\}$ *honest* LN *ITI and* $\bar{P}$ *the counterparty ITI. Assume that all of the following are true:*

- *the internal variable* `negligent` *of P has value "False",*
- *P has transitioned to the* OPEN *State for the first time after having received* (OPEN, c, . . .) *by either* $\mathcal{E}$ *or* $\bar{P}$,
- *P [has received* (FUND ME, f_i, . . .) *as input by another* LN *ITI while State was* OPEN *and subsequently P transitioned to* OPEN *State] n times,*

- P [has received (PAY, d_i) by $\mathcal{E}$ while State was OPEN and P subsequently transitioned to OPEN State] m times,
- P [has received $(\text{GET PAID}, e_i)$ by $\mathcal{E}$ while State was OPEN and P subsequently transitioned to OPEN State] l times.

Let $\phi = 1$ if $P = Alice$, or $\phi = 0$ if $P = Bob$. If P receives (FORCECLOSE) by $\mathcal{E}$ and, if $\text{host}_P \neq$ "ledger" the output of host_P is (CLOSED), then eventually the state obtained when P inputs (READ) to $\mathcal{G}_{\text{Ledger}}$ will contain h outputs each of value c_i and that has been spent or is exclusively spendable by $pk_{R,\text{out}}$ such that

$$\sum_{i=1}^{h} c_i \geq \phi \cdot c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \tag{3}$$

with overwhelming probability in the security parameter, where R is a local, kindred machine (i.e. either P, P's sibling, the party to which P sent FUND ME if such a message has been sent, or the sibling of one of the transitive closure of hosts of P).

Proof (Proof of Lemma 4). We first note that, as signature forgeries only happen with negligible probability and only a polynomial number of signatures are verified by honest parties throughout an execution, the event in which any forged signature passes the verification of an honest party or of $\mathcal{G}_{\text{Ledger}}$ happens only with negligible probability. We can therefore ignore this event throughout this proof and simply add a computationally negligible distance between $\mathcal{E}$'s outputs in the real and the ideal world at the end.

We also note that $pk_{P,\text{out}}$ has been provided by $\mathcal{E}$, therefore it can freely use coins spendable by this key. This is why we allow for any of the $pk_{P,\text{out}}$ outputs to have been spent.

Define the *history* of a channel as $H = (F, C)$, where each of F, C is a list of lists of integers. A party P which satisfies the Lemma conditions has a unique, unambiguously and recursively defined history: If the value hops in the (OPEN, c, hops, ...) message was equal to $\mathcal{G}_{\text{Ledger}}$, then F is the empty list, otherwise F is the concatenation of the F and C lists of the party that sent (FUNDED, ...) to P, as they were at the moment the latter message was sent. After initialised, F remains immutable. Observe that, if $\text{hops} \neq \mathcal{G}_{\text{Ledger}}$, both aforementioned messages must have been received before P transitions to the OPEN state.

The list C of party P is initialised to $[[g]]$ when P's *State* transitions for the first time to OPEN, where $g = c$ if $P = Alice$, or $g = 0$ if $P = Bob$; this represents the initial channel balance. The value x or $-x$ is appended to the last list in C when a payment is received (Fig. 34, l. 21) or sent (Fig. 34, l. 6) respectively by P. Moving on to the funding of new virtual channels, whenever P funds a new virtual channel (Fig. 31, l. 20), $[-c_{\text{virt}}]$ is appended to C and whenever P helps with the opening of a new virutal channel, but does not fund it (Fig. 31, l. 23), $[0]$ is appended to C. Therefore C consists of one list of integers for each sequence of inbound and outbound payments that have not been interrupted by a virtualisation step and a new list is added for every new virtual layer. We

also observe that a non-negligent party with history (F, C) satisfies the Lemma conditions and that the value of the right hand side of the inequality (3) is equal to $\sum_{s \in C} \sum_{x \in s} x$, as all inbound and outbound payment values and new channel funding values that appear in the Lemma conditions are recorded in C.

Let party P with a particular history. We will inductively prove that P satisfies the Lemma. The base case is when a channel is opened with hops $= \mathcal{G}_{\text{Ledger}}$ and is closed right away, therefore $H = ([], [g])$, where $g = c$ if $P = Alice$ and $g = 0$ if $P = Bob$. P can transition to the OPEN *State* for the first time only if all of the following have taken place:

- It has received (OPEN, c, ...) while in the INIT *State*. In case $P = Alice$, this message must have been received as input by $\mathcal{E}$ (Fig. 29, l. 1), or in case $P = Bob$, this message must have been received via the network by $\bar{P}$ (Fig. 24, l. 3).
- It has received $pk_{\bar{P}, F}$. In case $P = Bob$, $pk_{\bar{P}, F}$ must have been contained in the (OPEN, ...) message by $\bar{P}$ (Fig. 24, l. 3), otherwise if $P = Alice$ $pk_{\bar{P}, F}$ must have been contained in the (ACCEPT CHANNEL, ...) message by $\bar{P}$ (Fig. 24, l. 16).
- It internally holds a signature on the commitment transaction $C_{P,0}$ that is valid when verified with public key $pk_{\bar{P}, F}$ (Fig. 26, ll. 12 and 23).
- It has the transaction F in the $\mathcal{G}_{\text{Ledger}}$ state (Fig. 27, l. 3 or Fig. 28, l. 16).

We observe that P satisfies the Lemma conditions with $m = n = l = 0$. Before transitioning to the OPEN *State*, P has produced only one valid signature for the "funding" output $(c, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ of F with $sk_{P,F}$, namely for $C_{\bar{P},0}$ (Fig. 26, ll. 4 or 14), and sent it to $\bar{P}$ (Fig. 26, ll. 6 or 21), therefore the only two ways to spend $(c, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ are by either publishing $C_{P,0}$ or $C_{\bar{P},0}$. We observe that $C_{P,0}$ has a $(g, (pk_{P,\text{out}}+(t+s)) \vee 2/\{pk_{P,R}, pk_{\bar{P},R}\})$ output (Fig. 26, l. 2 or 3). The spending method $2/\{pk_{P,R}, pk_{\bar{P},R}\}$ cannot be used since P has not produced a signature for it with $sk_{P,R}$, therefore the alternative spending method, $pk_{P,\text{out}}+ (t + s)$, is the only one that will be spendable if $C_{P,0}$ is included in $\mathcal{G}_{\text{Ledger}}$, thus contributing g to the sum of outputs that contribute to inequality (3). Likewise, if $C_{\bar{P},0}$ is included in $\mathcal{G}_{\text{Ledger}}$, it will contribute at least one $(g, pk_{P,\text{out}})$ output to this inequality, as $C_{\bar{P},0}$ has a $(g, pk_{P,\text{out}})$ output (Fig. 26, l. 2 or 3). Additionally, if P receives (FORCECLOSE) by $\mathcal{E}$ while $H = ([], [g])$, it attempts to publish $C_{P,0}$ (Fig. 40, l. 19), and will either succeed or $C_{\bar{P},0}$ will be published instead. We therefore conclude that in every case $\mathcal{G}_{\text{Ledger}}$ will eventually have a state Σ that contains at least one $(g, pk_{P,\text{out}})$ output, therefore satisfying the Lemma consequence.

Let P with history $H = (F, C)$. The induction hypothesis is that the Lemma holds for P. Let c_P the sum in the right hand side of inequality (3). In order to perform the induction step, assume that P is in the OPEN state. We will prove all the following (the facts to be proven are shown with emphasis for clarity):

- If P receives (FUND ME, f, ...) by a (local, kindred) LN ITI R, subsequently transitions back to the OPEN state (therefore moving to history (F, C') where

$C' = C + [-f]$) and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by host$_P$ before any further change to its history, then *eventually P's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by $pk_{P,\text{out}}$) that are descendants of an output with spending method $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ such that $\sum_{i=1}^{h} c_i \geq \sum_{s \in C'} \sum_{x \in s} x$.* Furthermore, given that P moves to the OPEN state after the (FUND ME, ...) message, it also sends (FUNDED, ...) to R (Fig. 31, l. 21). If subsequently the state of R transitions to OPEN (therefore obtaining history (F_R, C_R) where $F_R = F + C$ and $C_R = [[f]]$), and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by host$_R$ (host$_R$ = host$_P$ – Fig. 28, l. 3) before any further change to its history, then *eventually R's $\mathcal{G}_{\text{Ledger}}$ state will contain k transaction outputs each of value c_i^R exclusively spendable or already spent by $pk_{R,\text{out}}$) that are descendants of an output with spending method $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ such that*

$$\sum_{i=1}^{k} c_i^R \geq \sum_{s \in C_R} \sum_{x \in s} x.$$

- If P receives (VIRTUALISING, ...) by $\bar{P}$, subsequently transitions back to OPEN (therefore moving to history (F, C') where $C' = C + [0]$) and finally receives FORCECLOSE by $\mathcal{E}$ and (CLOSED) by host$_P$ before any further change to its history, then *eventually P's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by $pk_{P,\text{out}}$) that are descendants of an output with spending method $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ such that $\sum_{i=1}^{h} c_i \geq \sum_{s \in C} \sum_{x \in s} x$.* Furthermore, given that P moves to the OPEN state after the (VIRTUALISING, ...) message and in case it sends (FUNDED, ...) to some party R (Fig. 31, l. 18), the latter party is the (local, kindred) fundee of a new virtual channel. If subsequently the state of R transitions to OPEN (therefore obtaining history (F_R, C_R) where $F_R = F + C$ and $C_R = [[0]]$), and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by host$_R$ (host$_R$ = host$_P$ – Fig. 28, l. 3) before any further change to its history, then *eventually R's $\mathcal{G}_{\text{Ledger}}$ state will contain an output with a $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ spending method.*
- If P receives (PAY, d) by $\mathcal{E}$, subsequently transitions back to OPEN (therefore moving to history (F, C') where C' is C with $-d$ appended to the last list of C) and finally receives FORCECLOSE by $\mathcal{E}$ and (CLOSED) by host$_P$ (the latter only if host$_P \neq \mathcal{G}_{\text{Ledger}}$ or equivalently $F \neq []$) before any further change to its history, then *eventually P's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by $pk_{P,\text{out}}$) that are descendants of an output with a $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ spending method such that $\sum_{i=1}^{h} c_i \geq \sum_{s \in C'} \sum_{x \in s} x$.*
- If P receives (GET PAID, e) by $\mathcal{E}$, subsequently transitions back to OPEN (therefore moving to history (F, C') where C' is C with e appended to the last list of C) and finally receives FORCECLOSE by $\mathcal{E}$ and (CLOSED) by host$_P$ (the latter only if host$_P \neq \mathcal{G}_{\text{Ledger}}$ or equivalently $F = []$) before any fur-

ther change to its history, then *eventually P's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by* $pk_{P,\text{out}}$) *that are descendants of an output with a $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ spending method such that* $\sum\limits_{i=1}^{h} c_i \geq \sum\limits_{s \in C'} \sum\limits_{x \in s} x$.

By the induction hypothesis, before the funding procedure started P could close the channel and end up with on-chain transaction outputs exclusively spendable or already spent by $pk_{P,\text{out}}$ with a sum value of c_P. When P is in the OPEN state and receives (FUND ME, f, ...), it can only move again to the OPEN state after doing the following state transitions: OPEN $\rightarrow$ VIRTUALISING $\rightarrow$ WAITING FOR REVOCATION $\rightarrow$ WAITING FOR INBOUND REVOCATION $\rightarrow$ WAITING FOR HOSTS READY $\rightarrow$ OPEN. During this sequence of events, a new host_P is defined (Fig. 31, l. 6), new commitment transactions are negotiated with $\bar{P}$ (Fig. 31, l. 9), control of the old funding output is handed over to host_P (Fig. 31, l. 11), host_P negotiates with its counterparty a new set of transactions and signatures that spend the aforementioned funding output and make available a new funding output with the keys $pk'_{P,F}, pk'_{\bar{P},F}$ as P instructed (Fig. 53 and 55) and the previous valid commitment transactions of both P and $\bar{P}$ are invalidated (Fig. 23, l. 1 and l. 14 respectively). We note that the use of the `ANYPREVOUT` flag in all signatures that correspond to transaction inputs that may spend various different transaction outputs ensures that this is possible, as it avoids tying each input to a specific, predefined output. When P receives (FORCECLOSE) by $\mathcal{E}$, it inputs (FORCECLOSE) to host_P (Fig. 40, l. 4). As per the Lemma conditions, host_P will output (CLOSED). This can happen only when $\mathcal{G}_{\text{Ledger}}$ contains a suitable output for both P's and R's channel (Fig. 59, and 5 ll. 6 respectively).

If the `host` of host_P is $\mathcal{G}_{\text{Ledger}}$, then the funding output $o_{1,2} = (c_P + c_{\bar{P}}, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ for the $P, \bar{P}$ channel is already on-chain. Regarding the case in which $\text{host}_P \neq \mathcal{G}_{\text{Ledger}}$, after the funding procedure is complete, the new host_P will have as its `host` the old host_P of P. If the (FORCECLOSE) sequence is initiated, the new host_P will follow the same steps that will be described below once the old host_P succeeds in closing the lower layer (Fig. 58, l. 6). The old host_P however will see no difference in its interface compared to what would happen if P had received (FORCECLOSE) before the funding procedure, therefore it will successfully close by the induction hypothesis. Thereafter the process is identical to the one when the old $\text{host}_P = \mathcal{G}_{\text{Ledger}}$.

Moving on, host_P is either able to publish its $\text{TX}_{1,1}$ (it has necessarily received a valid signature $\text{sig}(\text{TX}_{1,1}, pk_{\bar{P},F})$ (Fig. 55, l. 43) by its counterparty before it moved to the OPEN state for the first time), or the output $(c_P + c_{\bar{P}}, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ needed to spend $\text{TX}_{1,1}$ has already been spent. The only other transactions that can spend it are $\text{TX}_{2,1}$ and any of $(\text{TX}_{2,2,k})_{k>2}$, since these are the only transactions that spend the aforementioned output and that host_P has signed with $sk_{P,F}$ (Fig. 55, ll. 37-41). The output can be also spent by old, revoked commitment transactions, but in that case host_P would not have output (CLOSED); P would have instead detected this triggered by a (CHECK CHAIN FOR CLOSED) message by $\mathcal{E}$ (Fig. 38) and would have moved to the

CLOSED state on its own accord (lack of such a message by $\mathcal{E}$ would lead P to become `negligent`, something that cannot happen according to the Lemma conditions). Every transaction among $TX_{1,1}$, $TX_{2,1}$, $(TX_{2,2,k})_{k>2}$ has a $(c_P + c_{\bar{P}} - f, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$ output (Fig. 49, l. 19 and Fig. 48, ll. 29 and 97) which will end up in $\mathcal{G}_{\text{Ledger}}$ – call this output o_P. We will prove that at most $\sum_{i=2}^{n-1} (t_i + p + s - 1)$ blocks after (FORCECLOSE) is received by P, an output o_R with c_{virt} coins and a $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ spending condition without or with an expired timelock will be included in $\mathcal{G}_{\text{Ledger}}$. In case party $\bar{P}$ is idle, then $o_{1,2}$ is consumed by $TX_{1,1}$ and the timelock on its virtual output expires, therefore the required output o_R is on-chain. In case $\bar{P}$ is active, exactly one of $TX_{2,1}$, $(TX_{2,2,k})_{k>2}$ or $(TX_{2,3,1,k})_{k>2}$ is a descendant of $o_{1,2}$; if the transaction belongs to one of the two last transaction groups then necessarily $TX_{1,1}$ is on-chain in some block height h and given the timelock on the virtual output of $TX_{1,1}$, $\bar{P}$'s transaction can be at most at block height $h + t_2 + p + s - 1$. If $n = 3$ or $k = n - 1$, then $\bar{P}$'s unique transaction has the required output o_R (without a timelock). The rest of the cases are covered by the following sequence of events:

Closing sequence

1: `maxDel` $\leftarrow t_2 + p + s - 1$ // A_2 is active and the virtual output of $TX_{1,1}$ has a timelock of t_2
2: $i \leftarrow 3$
3: **loop**
4:     **if** A_i is idle **then**
5:         The timelock on the virtual output of the transaction published by A_{i-1} expires and therefore the required o_R is on-chain
6:     **else** // A_i publishes a transaction that is a descendant of $o_{1,2}$
7:         `maxDel` $\leftarrow$ `maxDel` $+ t_i + p + s - 1$
8:         The published transaction can be of the form $TX_{i,2,2}$ or $(TX_{i,3,2,k})_{k>i}$ as it spends the virtual output which is encumbered with a public key controlled by R and R has only signed these transactions
9:         **if** $i = n - 1$ or $k \geq n - 1$ **then** // The interval contains all intermediaries
10:             The virtual output of the transaction is not timelocked and has only a $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ spending method, therefore it is the required o_R
11:         **else** // At least one intermediary is not in the interval
12:             **if** the transaction is $TX_{i,3,2,k}$ **then** $i \leftarrow k$ **else** $i \leftarrow i + 1$
13:         **end if**
14:     **end if**
15: **end loop**
16: // `maxDel` $\leq \sum_{i=2}^{n-1} (t_i + p + s - 1)$

Fig. 62.

79

In every case o_P and o_R end up on-chain in at most s and $\sum_{i=2}^{n-1}(t_i + p + s - 1)$ blocks respectively from the moment (FORCECLOSE) is received. The output o_P an be spent either by $C_{P,i}$ or $C_{\bar{P},i}$. Both these transactions have a $(c_P - f, pk_{P,\text{out}})$ output. This output of $C_{P,i}$ is timelocked, but the alternative spending method cannot be used as P never signed a transaction that uses it (as it is reserved for revocation, which has not taken place yet in this virtualisation layer). We have now proven that if P completes the funding of a new channel then it can close its channel for a $(c_P - f, pk_{P,\text{out}})$ output that is a descendant of an output with spending method $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ and that lower bound of value holds for the duration of the funding procedure, i.e. we have proven the first claim of the first bullet.

We will now prove that the newly funded party R can close its channel securely. After R receives (FUNDED, host$_P$, . . .) by P and before moving to the OPEN state, it receives $\text{sig}_{\bar{R},C,0} = \text{sig}(C_{R,0}, pk_{\bar{R},F})$ and sends $\text{sig}_{R,C,0} = \text{sig}(C_{\bar{R},0}, pk_{R,F})$. Both these transactions spend o_R. As we showed before, if R receives (FORCECLOSE) by $\mathcal{E}$ then o_R eventually ends up on-chain. After receiving (CLOSED) from host$_P$, R attempts to add $C_{R,0}$ to $\mathcal{G}_{\text{Ledger}}$, which may only fail if $C_{\bar{R},0}$ ends up on-chain instead. Similar to the case of P, both these transactions have an $(f, pk_{R,\text{out}})$ output. This output of $C_{R,0}$ is timelocked, but the alternative spending method cannot be used as R never signed a transaction that uses it (as it is reserved for revocation, which has not taken place yet) so the timelock will expire and the desired spending method will be available. We have now proven that if R's channel is funded to completion (i.e. R moves to the OPEN state for the first time) then it can close its channel for a $(f, pk_{R,\text{out}})$ output that is a descendant of o_R. We have therefore proven the first bullet.

We now move on to the second bullet. In case P is the funder (i.e. $i = n$), then the same arguments as in the previous bullet hold here with "WAITING FOR INBOUND REVOCATION" replaced with "WAITING FOR OUTBOUND REVOCATION", $o_{1,2}$ with $o_{n-1,n}$, TX$_{1,1}$ with TX$_{n,1}$, TX$_{2,1}$ with TX$_{n-1,1}$, (TX$_{2,2,k}$)$_{k>2}$ with (TX$_{n-1,2,k}$)$_{k<n-1}$, (TX$_{2,3,1,k}$)$_{k>2}$ with (TX$_{n-1,3,n,k}$)$_{k<n-1}$, t_2 with t_{n-1}, TX$_{i,3,2,k}$ with TX$_{i,3,n-1,k}$, i is initialized to $n-2$ in l. 2 of Fig. 62, i is decremented instead of incremented in l. 12 of the same Figure and f is replaced with 0. This is so because these two cases are symmetric.

In case P is not the funder $(1 < i < n)$, then we only need to prove the first statement of the second bullet. By the induction hypothesis and since sibling is kindred, we know that both P's and sibling's funding outputs either are or can be eventually put on-chain and that P's funding output has at least $c_P = \sum_{s \in C} \sum_{x \in s} x$ coins. If P is on the "left" of its sibling (i.e. there is an untrusted party that sent the (VIRTUALISING, . . .) message to P which triggered the latter to move to the VIRTUALISING state and to send a (VIRTUALISING, . . .) message to its own sibling), the "left" funding output o_{left} (the one held with the untrusted party to the left) can be spent by one of TX$_{i,1}$, (TX$_{i,2,k}$)$_{k>i}$, TX$_{i-1,1}$,

80

or $(\text{TX}_{i-1,2,k})_{k<i-1}$, as these are the only transactions that P has signed with $sk_{P,F}$. All these transactions have a $(c_P + c_{\bar{P}} - f,\ 2/\{pk_{P,F'}, pk_{\bar{P},F'}\})$ output that can in turn be spent by either $C_{P,0}$ or $C_{\bar{P},0}$, both of which have an output of value c_P and a $pk_{P,\text{out}}$ spending method and no other spending method can be used (as P has not signed the "revocation" spending method of $C_{P,0}$).

In the case that P is to the right of its `sibling` (i.e. P receives by `sibling` the (VIRTUALISING, ...) message that causes P's transition to the VIRUTALIS-ING state), the "right" funding output o_{right} (the one held with the untrusted party to the right) can be spent by one of $\text{TX}_{i,1}$, $(\text{TX}_{i,2,k})_{k<i}$, $\text{TX}_{i+1,1}$, or $(\text{TX}_{i+1,2,k})_{k>i+1}$, as these are the only transactions that P has signed with $sk_{P,F}$. All these transactions have a $(c_P + c_{\bar{P}} - f,\ 2/\{pk_{P,F'}, pk_{\bar{P},F'}\})$ output that can in turn be spent by either $C_{P,0}$ or $C_{\bar{P},0}$, both of which have an output of value $c_P - f$ and a $pk_{P,\text{out}}$ spending method and no other spending method can be used (as P has not signed the "revocation" spending method of $C_{P,0}$). P can get the remaining f coins as follows: $\text{TX}_{i,1}$ and all of $(\text{TX}_{i,2,k})_{k<i}$ already have an $(f,\ pk_{P,\text{out}})$ output. If instead $\text{TX}_{i+1,1}$ or one of $(\text{TX}_{i+1,2,k_2})_{k_2>i+1}$ spends o_{right}, then P will publish $\text{TX}_{i,2,i+1}$ or $\text{TX}_{i,2,k_2}$ respectively if o_{left} is unspent, otherwise o_{left} is spent by one of $\text{TX}_{i-1,1}$ or $(\text{TX}_{i-1,2,k_1})_{k_1<i-1}$ in which case P will publish one of $\text{TX}_{i,3,k_1,i+1}$, $\text{TX}_{i,3,i-1,k_2}$, $\text{TX}_{i,3,i-1,i+1}$ or $\text{TX}_{i,3,k_1,k_2}$. In particular, $\text{TX}_{i,3,k_1,i+1}$ is published if $\text{TX}_{i-1,2,k_1}$ and $\text{TX}_{i+1,1}$ are on-chain, $\text{TX}_{i,3,i-1,k_2}$ is published if $\text{TX}_{i-1,1}$ and $\text{TX}_{i+1,2,k_2}$ are on-chain, $\text{TX}_{i,3,i-1,i+1}$ is published if $\text{TX}_{i-1,1}$ and $\text{TX}_{i+1,1}$ are on-chain, or $\text{TX}_{i,3,k_1,k_2}$ is published if $\text{TX}_{i-1,2,k_1}$ and $\text{TX}_{i+1,2,k_2}$ are on-chain. All these transactions include an $(f,\ pk_{P,\text{out}})$ output. We have therefore covered all cases and proven the second bullet.

Regarding now the third bullet, once again the induction hypothesis guarantees that before (PAY, d) was received, P could close the channel resulting in on-chain outputs exclusively spendable or already spent by $pk_{P,\text{out}}$ that are descendants of an output with a $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ spending method that have a sum value of $c_P = \sum_{s\in C}\sum_{x\in s} x$. (Note that $\sum_{s\in C'}\sum_{x\in s} x = d + \sum_{s\in C}\sum_{x\in s} x$.) When P receives (PAY, d) while in the OPEN state, it moves to the WAITING FOR COMMITMENT SIGNED state before returning to the OPEN state. It signs (Fig. 33, l. 2) the new commitment transaction $C_{\bar{P},i+1}$ in which the counterparty owns d more coins than before that moment (Fig. 33, l. 1), sends the signature to the counterparty (Fig. 33, l. 5) and expects valid signatures on its own updated commitment transaction (Fig. 34, l. 1) and the revocation transaction for the old commitment transaction of the counterparty (Fig. 34, l. 3). Note that if the counterparty does not respond or if it responds with missing/invalid signatures, either P can close the channel with the old commitment transaction $C_{P,i}$ exactly like before the update started (as it has not yet sent the signature for the old revocation transaction), or the counterparty will close the channel either with the new or with the old commitment transaction. In all cases in which validation fails and the channel closes, there is an output with a $pk_{P,\text{out}}$ spending method and no other useable spending method that carries at least $c_P - d$ coins. Only if the verification succeeds does P sign (Fig. 34, l. 5) and send (Fig. 34, l. 17) the counterparty's revocation transaction for P's previous commitment transaction.

Similarly to previous bullets, if $\text{host}_P \neq \mathcal{G}_{\text{Ledger}}$ the funding output can be put on-chain, otherwise the funding output is already on-chain. In both cases, since the closing procedure continues, one of $C_{P,i+1}$ $(C_{\bar{P},j})_{0 \leq j \leq i+1}$ will end up on-chain. If $C_{\bar{P},j}$ for some $j < i+1$ is on-chain, then P submits $R_{P,j}$ (we discussed how P obtained $R_{P,i}$ and the rest of the cases are covered by induction) and takes the entire value of the channel which is at least $c_P - d$. If $C_{\bar{P},i+1}$ is on-chain, it has a $(c_P - d,\ pk_{P,\text{out}})$ output. If $C_{P,i+1}$ is on-chain, it has an output of value $c_P - d$, a timelocked $pk_{P,\text{out}}$ spending method and a non-timelocked spending method that needs the signature made with $sk_{P,R}$ on $R_{\bar{P},i+1}$. P however has not generated that signature, therefore this spending method cannot be used and the timelock will expire, therefore in all cases outputs that descend from the funding output, can be spent exclusively by $pk_{P,\text{out}}$ and carry at least $c_P - d$ coins are put on-chain. We have proven the third bullet.

For the fourth and last bullet, again by the induction hypothesis, before (GET PAID, e) was received P could close the channel resulting in on-chain outputs exclusively spendable or already spent by $pk_{P,\text{out}}$ that are descendants of an output o_F with a $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ spending method and have a sum value of $c_P = \sum_{s \in C} \sum_{x \in s} x$. (Note that $e + \sum_{s \in C'} \sum_{x \in s} x = \sum_{s \in C} \sum_{x \in s} x$ and that o_F either is already on-chain or can be eventually put on-chain as we have argued in the previous bullets by the induction hypothesis.) When P receives (GET PAID, e) while in the OPEN state, if the balance of the counterparty is enough it moves to the WAITING TO GET PAID state (Fig. 36, l. 6). If subsequently it receives a valid signature for $C_{P,i+1}$ (Fig. 33, l. 9) which is a commitment transaction that can spend the o_F output and gives to P an additional e coins compared to $C_{P,i}$. Subsequently P's state transitions to WAITING FOR PAY REVOCATION and sends signatures for $C_{\bar{P},i+1}$ and $R_{\bar{P},i}$ to $\bar{P}$. If the o_F output is spent while P is in the latter state, it can be spent by one of $C_{P,i+1}$ or $(C_{\bar{P},j})_{0 \leq j \leq i+1}$. If it is spent by $C_{P,i+1}$ or $C_{\bar{P},i+1}$, then these two transactions have a $(c_P + e,\ pk_{P,\text{out}})$ output. (Note that the former is encumbered with a timelock, but the alternative spending method cannot be used as P has not signed $R_{\bar{P},i+1}$.) If it is spent by $C_{\bar{P},i}$ then a $(c_P,\ pk_{P,\text{out}})$ output becomes available instead, therefore P can still get the c_P coins that correspond to the previous state. If any of $(C_{\bar{P},j})_{0 \leq j < i}$ spends o_F then it makes available a $pk_{P,\text{out}}$ output with the coins that P had at state j and additionally P can publish $R_{P,j}$ that spends $\bar{P}$'s output of $C_{\bar{P},j}$ and obtain the entirety of $\bar{P}$'s coins at state j for a total of $c_P + c_{\bar{P}}$ coins. Therefore in every case P can claim at least c_P coins. In the case that P instead subsequently receives a valid signature to $R_{P,i}$ (Fig. 34, l. 20) it finally moves to the OPEN state once again. In this state the above analysis of what can happen when o_F holds similarly, with the difference that if $\bar{P}$ spends o_F with $C_{\bar{P},i}$ now P can publish $R_{P,i}$ which gives P the coins of $\bar{P}$. Therefore with this difference P is now guaranteed to gain at least $c_P + e$ coins upon channel closure. We have therefore proven the fourth bullet.

Lemma 5 (Ideal world balance). *Consider an ideal world execution with functionality $\mathcal{G}_{\text{Chan}}$ and simulator $\mathcal{S}$. Let $P \in \{Alice, Bob\}$ one of the two parties of $\mathcal{G}_{\text{Chan}}$. Assume that all of the following are true:*

- *$State_P \neq$ IGNORED,*
- *P has transitioned to the OPEN State at least once. Additionally, if $P = Alice$, it has received (OPEN, c, . . .) by $\mathcal{E}$ prior to transitioning to the OPEN State,*
- *P [has received (FUND ME, f_i, . . .) as input by another $\mathcal{G}_{\text{Chan}}/$LN ITI while $State_P$ was OPEN and P subsequently transitioned to OPEN State] n times,*
- *P [has received (PAY, d_i) by $\mathcal{E}$ while $State_P$ was OPEN and P subsequently transitioned to OPEN State] m times,*
- *P [has received (GET PAID, e_i) by $\mathcal{E}$ while $State_P$ was OPEN and P subsequently transitioned to OPEN State] l times.*

Let $\phi = 1$ if $P = Alice$, or $\phi = 0$ if $P = Bob$. If $\mathcal{G}_{\text{Chan}}$ receives (FORCECLOSE, P) by $\mathcal{S}$, then the following holds with overwhelming probability on the security parameter:

$$\texttt{balance}_P = \phi \cdot c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \tag{4}$$

Proof (Proof of Lemma 5). We will prove the Lemma by following the evolution of the $\texttt{balance}_P$ variable.

- When $\mathcal{G}_{\text{Chan}}$ is activated for the first time, it sets $\texttt{balance}_P \leftarrow 0$ (Fig. 8, l. 1).
- If $P = Alice$ and it receives (OPEN, c, . . .) by $\mathcal{E}$, it stores c (Fig. 8, l. 11). If later $State_P$ becomes OPEN, $\mathcal{G}_{\text{Chan}}$ sets $\texttt{balance}_P \leftarrow c$ (Fig. 8, ll. 14 or 32). In contrast, if $P = Bob$, it is $\texttt{balance}_P = 0$ until at least the first transition of $State_P$ to OPEN (Fig. 8).
- Every time that P receives input (FUND ME, f_i, . . .) by another party while $State_P = $ OPEN, P stores f_i (Fig. 10, l. 1). The next time $State_P$ transitions to OPEN (if such a transition happens), $\texttt{balance}_P$ is decremented by f_i (Fig. 10, l. 27). Therefore, if this cycle happens $n \geq 0$ times, $\texttt{balance}_P$ will be decremented by $\sum_{i=1}^{n} f_i$ in total.
- Every time P receives input (PAY, d_i) by $\mathcal{E}$ while $State_P = $ OPEN, d_i is stored (Fig. 9, l. 2). The next time $State_P$ transitions to OPEN (if such a transition happens), $\texttt{balance}_P$ is decremented by d_i (Fig. 9, l. 13). Therefore, if this cycle happens $m \geq 0$ times, $\texttt{balance}_P$ will be decremented by $\sum_{i=1}^{m} d_i$ in total.
- Every time P receives input (GET PAID, e_i) by $\mathcal{E}$ while $State_P = $ OPEN, e_i is stored (Fig. 9, l. 7). The next time $State_P$ transitions to OPEN (if such a transition happens) $\texttt{balance}_P$ is incremented by e_i (Fig. 9, l. 19). Therefore, if this cycle happens $l \geq 0$ times, $\texttt{balance}_P$ will be incremented by $\sum_{i=1}^{l} e_i$ in total.

On aggregate, after the above are completed and then $\mathcal{G}_{\text{Chan}}$ receives (FORCECLOSE, P) by $\mathcal{S}$, it is $\texttt{balance}_P = c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i$ if $P = Alice$, or else if $P = Bob$, $\texttt{balance}_P = - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i$.

Proof (Proof of Lemma 3). We prove the Lemma in two steps. We first show that if the conditions of Lemma 5 hold, then the conditions of Lemma 4 for the real world execution with protocol LN and the same $\mathcal{E}$ and $\mathcal{A}$ hold as well for the same m, n and l values.

For $State_P$ to become IGNORED, either $\mathcal{S}$ has to send (BECAME CORRUPTED OR NEGLIGENT, P) or $\texttt{host}_P$ must output (ENABLER USED REVOCATION) to $\mathcal{G}_{\text{Chan}}$ (Fig. 8, l. 5). The first case only happens when either P receives (CORRUPT) by $\mathcal{A}$ (Fig. 21, l. 1), which means that the simulated P is not honest anymore, or when P becomes $\texttt{negligent}$ (Fig. 21, l. 4), which means that the first condition of Lemma 4 is violated. In the second case, it is $\texttt{host}_P \neq \mathcal{G}_{\text{Ledger}}$ and the state of $\texttt{host}_P$ is GUEST PUNISHED (Fig. 61, ll. 1 or 12), so in case P receives (FORCECLOSE) by $\mathcal{E}$ the output of $\texttt{host}_P$ will be (GUEST PUNISHED) (Fig. 58, l. 4). In all cases, some condition of Lemma 4 is violated.

For $State_P$ to become OPEN at least once, the following sequence of events must take place (Fig. 8): If $P = Alice$, it must receive (INIT, pk) by $\mathcal{E}$ when $State_P =$ UNINIT, then either receive (OPEN, c, $\mathcal{G}_{\text{Ledger}}$, ...) by $\mathcal{E}$ and (BASE OPEN) by $\mathcal{S}$ or (OPEN, c, $\texttt{hops}$ ($\neq \mathcal{G}_{\text{Ledger}}$), ...) by $\mathcal{E}$, (FUNDED, HOST, ...) by $\texttt{hops}[0].\texttt{left}$ and (VIRTUAL OPEN) by $\mathcal{S}$. In either case, $\mathcal{S}$ only sends its message only if all its simulated honest parties move to the OPEN state (Fig. 21, l. 10), therefore if the second condition of Lemma 5 holds and $P = Alice$, then the second condition of Lemma 4 holds as well. The same line of reasoning can be used to deduce that if $P = Bob$, then $State_P$ will become OPEN for the first time only if all honest simulated parties move to the OPEN *state*, therefore once more the second condition of Lemma 5 holds only if the second condition of Lemma 4 holds as well. We also observe that, if both parties are honest, they will transition to the OPEN state simultaneously.

Regarding the third Lemma 5 condition, we assume (and will later show) that if both parties are honest and the state of one is OPEN, then the state of the other is also OPEN. Each time P receives input (FUND ME, f, ...) by $R \in \{\mathcal{G}_{\text{Chan}}, \text{LN}\}$, $State_P$ transitions to PENDING FUND, subsequently when a command to define a new VIRT ITI through P is intercepted by $\mathcal{G}_{\text{Chan}}$, $State_P$ transitions to TENTATIVE FUND and afterwards when $\mathcal{S}$ sends (FUND) to $\mathcal{G}_{\text{Chan}}$, $State_P$ transitions to SYNC FUND. In parallel, if $State_{\bar{P}} =$ IGNORED, then $State_P$ transitions directly back to OPEN. If on the other hand $State_{\bar{P}} =$ OPEN and $\mathcal{G}_{\text{Chan}}$ intercepts a similar VIRT ITI definition command through $\bar{P}$, $State_{\bar{P}}$ transitions to TENTATIVE HELP FUND. On receiving the aforementioned (FUND) message by $\mathcal{S}$ and given that $State_{\bar{P}} =$ TENTATIVE HELP FUND, $\mathcal{G}_{\text{Chan}}$ also sets $State_{\bar{P}}$ to SYNC HELP FUND. Then both $State_{\bar{P}}$ and $State_P$ transition simultaneously to OPEN (Fig. 10). This sequence of events may repeat any $n \geq 0$ times. We observe that throughout these steps, honest simulated P has received (FUND

ME, f, ...) and that $\mathcal{S}$ only sends (FUND) when all honest simulated parties have transitioned to the OPEN state (Fig. 21, l. 18 and Fig. 31, l. 12), so the third condition of Lemma 4 holds with the same n as that of Lemma 5.

Regarding the fourth Lemma 5 condition, we again assume that if both parties are honest and the state of one is OPEN, then the state of the other is also OPEN. Each time P receives input (PAY, d) by $\mathcal{E}$, $State_P$ tranisitions to TENTATIVE PAY and subsequently when $\mathcal{S}$ sends (PAY) to $\mathcal{G}_{\text{Chan}}$, $State_P$ transitions to (SYNC PAY, d). In parallel, if $State_{\bar{P}} = $ IGNORED, then $State_P$ transitions directly back to OPEN. If on the other hand $State_{\bar{P}} = $ OPEN and $\mathcal{G}_{\text{Chan}}$ receives (GET PAID, d) by $\mathcal{E}$ addressed to $\bar{P}$, $State_{\bar{P}}$ transitions to TENTATIVE GET PAID. On receiving the aforementioned (PAY) message by $\mathcal{S}$ and given that $State_{\bar{P}} = $ TENTATIVE GET PAID, $\mathcal{G}_{\text{Chan}}$ also sets $State_{\bar{P}}$ to SYNC GET PAID. Then both $State_P$ and $State_{\bar{P}}$ transition simultaneously to OPEN (Fig. 9). This sequence of events may repeat any $m \geq 0$ times. We observe that throughout these steps, honest simulated P has received (PAY, d) and that $\mathcal{S}$ only sends (PAY) when all honest simulated parties have completed sending or receiving the payment (Fig. 21, l. 16), so the fourth condition of Lemma 4 holds with the same m as that of Lemma 5. As far as the fifth condition of Lemma 5 goes, we observe that this case is symmetric to the one discussed for its fourth condition above if we swap P and $\bar{P}$, therefore we deduce that if Lemma 5 holds with some l, then Lemma 4 holds with the same l.

As promised, we here argue that if both parties are honest and one party moves to the OPEN state, then the other party will move to the OPEN state as well. We already saw that the first time one party moves to the OPEN state, it will happen simultaneously with the same transition for the other party. We also saw that, when a party transitions from the SYNC HELP FUND or the SYNC FUND state to the OPEN state, then the other party will also transition to the OPEN state simultaneously. Furthermore, we saw that if one party transitions from the SYNC PAY or the SYNC GET PAID state to the OPEN state, the other party will also transition to the OPEN state simultaneously. Lastly we notice that we have exhausted all manners in which a party can transition to the OPEN state, therefore we have proven that transitions of honest parties to the OPEN state happen simultaneously.

Now, given that $\mathcal{S}$ internally simulates faithfully both LN parties and that $\mathcal{G}_{\text{Chan}}$ relinquishes to $\mathcal{S}$ complete control of the external communication of the parties as long as it does not halt, we deduce that $\mathcal{S}$ replicates the behaviour of the aforementioned real world. By combining these facts with the consequences of the two Lemmas and the check that leads $\mathcal{G}_{\text{Chan}}$ to halt if it fails (Fig. 11, l. 18), we deduce that if the conditions of Lemma 5 hold for the honest parties of $\mathcal{G}_{\text{Chan}}$ and their kindred parties, then the functionality halts only with negligible probability.

In the second proof step, we show that if the conditions of Lemma 5 do not hold, then the check of Fig. 11, l. 18 never takes place. We first discuss the $State_P = $ IGNORED case. We observe that the IGNORED $State$ is a sink state, as there is no way to leave it once in. Additionally, for the balance check to happen,

$\mathcal{G}_{\text{Chan}}$ must receive (CLOSED, P) by $\mathcal{S}$ when $State_P \neq$ IGNORED (Fig. 11, l. 9). We deduce that, once $State_P =$ IGNORED, the balance check will not happen. Moving to the case where $State_P$ has never been OPEN, we observe that it is impossible to move to any of the states required by l. 9 of Fig. 11 without first having been in the OPEN state. Moreover if $P = Alice$, it is impossible to reach the OPEN state without receiving input (OPEN, c, ...) by $\mathcal{E}$. Lastly, as we have observed already, the three last conditions of Lemma 5 are always satisfied. We conclude that if the conditions to Lemma 5 do not hold, then the check of Fig. 11, l. 18 does not happen and therefore $\mathcal{G}_{\text{Chan}}$ does not halt.

On aggregate, $\mathcal{G}_{\text{Chan}}$ may only halt with negligible probability in the security parameter.

References

1. Nakamoto S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
2. Croman K., Decker C., Eyal I., Gencer A. E., Juels A., Kosba A., Miller A., Saxena P., Shi E., Sirer E. G., et al.: On scaling decentralized blockchains. In International Conference on Financial Cryptography and Data Security: pp. 106–125: Springer (2016)
3. Gudgeon L., Moreno-Sanchez P., Roos S., McCorry P., Gervais A.: SoK: Layer-Two Blockchain Protocols. In Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers: pp. 201–226: doi:10.1007/978-3-030-51280-4_12: URL https://doi.org/10.1007/978-3-030-51280-4_12 (2020)
4. Decker C., Wattenhofer R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In Symposium on Self-Stabilizing Systems: pp. 3–18: Springer (2015)
5. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf (2016)
6. Dziembowski S., Eckey L., Faust S., Malinowski D.: Perun: Virtual Payment Hubs over Cryptocurrencies. In 2019 2019 IEEE Symposium on Security and Privacy (SP): pp. 344–361: IEEE Computer Society, Los Alamitos, CA, USA: ISSN 2375–1207: doi:10.1109/SP.2019.00020: URL https://doi.ieeecomputersociety.org/10.1109/SP.2019.00020 (2019)
7. Dziembowski S., Faust S., Hostáková K.: General State Channel Networks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018: pp. 949–966: doi:10.1145/3243734.3243856: URL https://doi.org/10.1145/3243734.3243856 (2018)
8. Aumayr L., Ersoy O., Erwig A., Faust S., Hostáková K., Maffei M., Moreno-Sanchez P., Riahi S.: Bitcoin-Compatible Virtual Channels. In IEEE Symposium on Security and Privacy, Oakland, USA; 2021-05-23 - 2021-05-27: https://eprint.iacr.org/2020/554.pdf (2021)

9. Aumayr L., Ersoy O., Erwig A., Faust S., Hostakova K., Maffei M., Moreno-Sanchez P., Riahi S.: Generalized Bitcoin-Compatible Channels. Cryptology ePrint Archive, Report 2020/476: https://eprint.iacr.org/2020/476 (2020)

10. Jourenko M., Larangeira M., Tanaka K.: Lightweight Virtual Payment Channels. In S. Krenn, H. Shulman, S. Vaudenay (editors), Cryptology and Network Security: pp. 365–384: Springer International Publishing, Cham: ISBN 978-3-030-65411-5 (2020)

11. Wood G.: Ethereum: A secure decentralised generalised transaction ledger

12. Canetti R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA: pp. 136–145: doi:10.1109/SFCS.2001.959888: URL https://eprint.iacr.org/2000/067.pdf (2001)

13. Badertscher C., Canetti R., Hesse J., Tackmann B., Zikas V.: Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC. In Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part III: pp. 1–30: doi:10.1007/978-3-030-64381-2_1: URL https://doi.org/10.1007/978-3-030-64381-2_1 (2020)

14. Spilman J.: Anti dos for tx replacement. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html (2013)

15. Raiden Network. https://raiden.network/

16. Malavolta G., Moreno-Sanchez P., Schneidewind C., Kate A., Maffei M.: Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019 (2019)

17. Harris J., Zohar A.: Flood & Loot: A Systemic Attack on The Lightning Network. In Proceedings of the 2nd ACM Conference on Advances in Financial Technologies: AFT '20: pp. 202–213: Association for Computing Machinery, New York, NY, USA: ISBN 9781450381390: doi:10.1145/3419614.3423248: URL https://doi.org/10.1145/3419614.3423248 (2020)

18. Sivaraman V., Venkatakrishnan S. B., Alizadeh M., Fanti G. C., Viswanath P.: Routing Cryptocurrency with the Spider Network. CoRR: vol. abs/1809.05088: URL http://arxiv.org/abs/1809.05088 (2018)

19. Prihodko P., Zhigulin S., Sahno M., Ostrovskiy A., Osuntokun O.: Flare: An approach to routing in lightning network. White Paper (2016)

20. Lee J., Kim S., Park S., Moon S. M.: RouTEE: A Secure Payment Network Routing Hub using Trusted Execution Environments (2020)

21. Khalil R., Gervais A.: Revive: Rebalancing Off-Blockchain Payment Networks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017: pp. 439–453: doi:10.1145/3133956.3134033: URL https://doi.org/10.1145/3133956.3134033 (2017)

22. Decker C., Russell R., Osuntokun O.: eltoo: A Simple Layer2 Protocol for Bitcoin. https://blockstream.com/eltoo.pdf

23. Miller A., Bentov I., Kumaresan R., Cordi C., McCorry P.: Sprites and State Channels: Payment Networks that Go Faster than Lightning. ArXiv preprint arXiv:1702.05812 (2017)

24. Dong M., Liang Q., Li X., Liu J.: Celer Network: Bring Internet Scale to Every Blockchain (2018)

25. Chakravarty M. M. T., Coretti S., Fitzi M., Gazi P., Kant P., Kiayias A., Russell A.: Hydra: Fast Isomorphic State Channels. Cryptology ePrint Archive, Report 2020/299: https://eprint.iacr.org/2020/299 (2020)

26. Chakravarty M. M. T., Kireev R., MacKenzie K., McHale V., Müller J., Nemish A., Nester C., Peyton Jones M., Thompson S., Valentine R., Wadler P.: Functional blockchain contracts. https://iohk.io/en/research/library/papers/functional-blockchain-contracts/ (2019)

27. Burchert C., Decker C., Wattenhofer R.: Scalable funding of Bitcoin micropayment channel networks. In The Royal Society: doi:10.1098/rsos.180089 (2018)

28. Egger C., Moreno-Sanchez P., Maffei M.: Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security: CCS '19: pp. 801–815: Association for Computing Machinery, New York, NY, USA: ISBN 9781450367479: doi:10.1145/3319535.3345666: URL https://doi.org/10.1145/3319535.3345666 (2019)

29. Zhao L., Shuang H., Xu S., Huang W., Cui R., Bettadpur P., Lie D.: SoK: Hardware Security Support for Trustworthy Execution (2019)

30. Lind J., Eyal I., Pietzuch P. R., Sirer E. G.: Teechan: Payment Channels Using Trusted Execution Environments. CoRR: vol. abs/1612.07766: URL http://arxiv.org/abs/1612.07766 (2016)

31. Lind J., Naor O., Eyal I., Kelbert F., Sirer E. G., Pietzuch P.: Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In Proceedings of the 27th ACM Symposium on Operating Systems Principles: SOSP '19: pp. 63–79: Association for Computing Machinery, New York, NY, USA: ISBN 9781450368735: doi:10.1145/3341301.3359627: URL https://doi.org/10.1145/3341301.3359627 (2019)

32. Liao J., Zhang F., Sun W., Shi W.: Speedster: A TEE-assisted State Channel System (2021)

33. Avarikioti G., Kogias E. K., Wattenhofer R., Zindros D.: Brick: Asynchronous Payment Channels (2020)

34. Back A., Corallo M., Dashjr L., Friedenbach M., Maxwell G., Miller A., Poelstra A., Timón J., Wuille P.: Enabling blockchain innovations with pegged sidechains. URL http://web.archive.org/save/https://pdfs.semanticscholar.org/1b23/cd2050d5000c05e1da3c9997b308ad5b7903.pdf (2014)

35. Gaži P., Kiayias A., Zindros D.: Proof-of-Stake Sidechains. In 2019 2019 IEEE Symposium on Security and Privacy (SP): pp. 677–694: IEEE Computer Society, Los Alamitos, CA, USA: ISSN 2375-1207: doi:10.1109/SP.2019.00040: URL https://doi.ieeecomputersociety.org/10.1109/SP.2019.00040 (2019)

36. Kiayias A., Zindros D.: Proof-of-Work Sidechains. In Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers: pp. 21–34: doi:10.1007/978-3-030-43725-1_3: URL https://doi.org/10.1007/978-3-030-43725-1_3 (2019)

37. Poon J., Buterin V.: Plasma: Scalable Autonomous Smart Contracts. http://plasma.io/plasma.pdf

38. Konstantopoulos G.: Plasma Cash: Towards more efficient Plasma constructions (2019)

39. Dziembowski S., Fabiański G., Faust S., Riahi S.: Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma. Cryptology ePrint Archive, Report 2020/175: https://eprint.iacr.org/2020/175 (2020)

40. Adler J.: The Why's of Optimistic Rollup. https://medium.com/@adlerjohn/the-why-s-of-optimistic-rollup-7c6a22cbb61a (2019)
41. Armknecht F., Karame G. O., Mandal A., Youssef F., Zenner E.: Ripple: Overview and Outlook. In Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings: pp. 163–180: doi:10.1007/978-3-319-22846-4_10: URL https://doi.org/10.1007/978-3-319-22846-4_10 (2015)
42. Mazieres D.: The stellar consensus protocol: A federated model for internet-level consensus. Stellar Development Foundation (2015)
43. Malavolta G., Moreno-Sanchez P., Kate A., Maffei M.: SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks (2016)
44. Roos S., Moreno-Sanchez P., Kate A., Goldberg I.: Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018: URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-3_Roos_paper.pdf (2018)
45. Aumayr L., Moreno-Sanchez P., Kate A., Maffei M.: Donner: UTXO-Based Virtual Channels Across Multiple Hops. Cryptology ePrint Archive, Report 2021/855: https://eprint.iacr.org/2021/855 (2021)
46. Decker C., Towns A.: SIGHASH_ANYPREVOUT for Taproot Scripts. https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki
47. Lindell Y.: How to Simulate It - A Tutorial on the Simulation Proof Technique. In Tutorials on the Foundations of Cryptography: pp. 277–346: doi:10.1007/978-3-319-57048-8_6: URL https://doi.org/10.1007/978-3-319-57048-8_6 (2017)
48. Badertscher C., Maurer U., Tschudi D., Zikas V.: Bitcoin as a transaction ledger: A composable treatment. In Annual International Cryptology Conference: pp. 324–356: Springer (2017)
49. Badertscher C., Gaži P., Kiayias A., Russell A., Zikas V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security: pp. 913–930: ACM (2018)
50. Katz J., Maurer U., Tackmann B., Zikas V.: Universally Composable Synchronous Computation. In Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings: pp. 477–498: doi:10.1007/978-3-642-36594-2_27: URL https://doi.org/10.1007/978-3-642-36594-2_27 (2013)
51. Canetti R., Dodis Y., Pass R., Walfish S.: Universally Composable Security with Global Setup. In Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings: pp. 61–85: doi:10.1007/978-3-540-70936-7_4: URL https://doi.org/10.1007/978-3-540-70936-7_4 (2007)
52. Kiayias A., Litos O. S. T.: A Composable Security Treatment of the Lightning Network. In 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020: pp. 334–349: doi:10.1109/CSF49147.2020.00031: URL https://doi.org/10.1109/CSF49147.2020.00031 (2020)
53. Wuille P., Nick J., Towns A.: Taproot: SegWit version 1 spending rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki
54. Danezis G., Goldberg I.: Sphinx: A compact and provably secure mix format. In Security and Privacy, 2009 30th IEEE Symposium on: pp. 269–282: IEEE (2009)