Elmo: Recursive Virtual Payment Channels for Bitcoin

Anonymised Submission

Abstract

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 off-chain setup operation, enable the two endpoints to engage as if they had a direct payment channel between them. Once the channel is unneeded, it can be optimistically closed in an off-chain fashion.

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 equitable fashion all channel participants both in collaborative and non-collaborative 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 for payments, optimistic closing and unilateral closing. We express and prove the security of our construction in the universal composition setting, using a novel induction-based proof technique of independent interest. As an additional contribution, we implement a flexible simulation framework for on- and off-chain payments and compare the efficiency of Elmo with previous virtual channel constructors.

ACM Reference Format:
Anonymised Submission. 2025. Elmo: Recursive Virtual Payment Channels for Bitcoin. In . ACM, New York, NY, USA, 57 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

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 [51]) and a very low throughput (e.g., Bitcoin can handle at most 7 transactions per second [19]), both significant shortcomings that jeopardize wider use and adoption and are to a certain extent inherent [19]. To address these considerations a prominent approach is to optimistically handle payments via a *Payment Channel Network* (PCN) (see, e.g., [34] for a survey). Payments over a PCN happen *off-chain*, i.e., without adding any transactions to the underlying blockchain. They only use the blockchain as an arbiter in case of disputes.

The key primitive of PCN protocols is a payment channel. Two parties (the *endpoints*) 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, making them a natural choice for parties that interact often. Multiple overlapping payment channels can be combined and form a PCN.

Closing a non-virtual channel is an operation that involves posting the channel state on-chain. Closing should be efficient, i.e., needing $O(1)$ on-chain transactions, independent of how many payments have occured off-chain. It is also essential that any party can unilaterally 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) [23] at the expense of bounding the overall lifetime of a channel. Using script timelocks, the Lightning Network (LN) [55] provided a better solution that enabled channels to stay 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 channels in LN enables endpoints to transmit funds to each other as long as there is a route of payment channels that connects them. The downside of this mechanism is that it requires the active involvement of all parties along the path for each payment. Instead, *virtual payment channels* suggest the more attractive approach of an one-time off-chain initialization step to set up a virtual payment channel over the preexisting channels, which subsequently can be used for direct payments with complexity —in the optimistic case— independent of the path length. When the virtual channel has exhausted its usefulness, it can be closed off-chain if the involved parties cooperate. Initial constructions for virtual channels capitalized on the extended functionality of Ethereum, e.g., Perun [27] and GSCN [29], while more recent work [2] brought them closer to Bitcoin-compatibility (by leveraging adaptor signatures [1]).

We call the parties of the underlying channels *intermediaries*. A virtual channel constructor can be thought of as an *operator*

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

over the underlying payment channel primitive. We identify three natural desiderata for it.

- Recursive. A recursive virtual channel constructor can operate over channels that themselves could be the result of previous applications of the operator. This allows building virtual channels on top of pre-existing virtual channels, allowing the channel structure to evolve dynamically.
- Variadic. A variadic virtual channel constructor can virtualize any number of input payment channels directly, i.e., without leveraging recursion, contrary to a *binary* constructor. 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 computation, network and storage cost between the two endpoints or the intermediaries (but not necessarily a mix of both) for the optimistic and pessimistic execution paths. Furthermore, payments by the two endpoints are encumbered with the same delay. Importantly, this ensures that no party is worse-off or better-off after an application of the operator in terms of accessing the basic channel functionality.

We note that recursiveness, while identified already as an important design property [29], has been achieved for only for DMC-like fixed lifetime channels [36], but was left as an open question for indefinite-lifetime, LN-type channels [2]). This is because of the severe limitations imposed 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 connect distant endpoints 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 most previous virtual channel constructors proposed are binary [2, 29, 36].

The primary motivation for recursive channels is adding flexibility in moving off-chain coins quickly, with minimal interaction, and at a low cost, even under consistently congested ledger conditions. Without recursiveness and when facing unresponsive channel parties, one would have to first close its channel on-chain in order to then use some of its coins with another party, which is as slow as any on-chain transaction and in case of high congestion prohibitively expensive. If a party needs to move only a fraction of the channel coins to a new channel, it still would have to close the entire original channel, even if the channel parties are collaborative. On the other hand, a recursive virtual channel permits using some of its coins with other parties by opening off-chain a new virtual channel on top, keeping the remaining coins in the original channel and without involving the parties of the latter. Importantly, users can decide to open a recursive virtual channel long after having established their underlying one. This flexibility can inspire confidence in virtual channels, prompting users to transfer more coins off-chain and reduce on-chain congestion.

Furthermore, Elmo channels have an indefinite lifetime, in contrast to [3]. This means that parties of Elmo channels do not need to periodically coordinate to extend the channel lifetime.

A scenario only possible with both the recursive and the variadic properties is as follows (Fig. 1): Initially Alice has a channel with Bob, Bob one with Charlie and Charlie one with Dave. Alice opens a virtual channel with Dave over the 3 channels – this needs the variadic property. After a while she realises she has to pay Eve a few times, who happens to have a channel with Dave. Alice interacts just with Dave and Eve to move half of her coins from her virtual channel with Dave to a new one with Eve – this needs the recursive property.

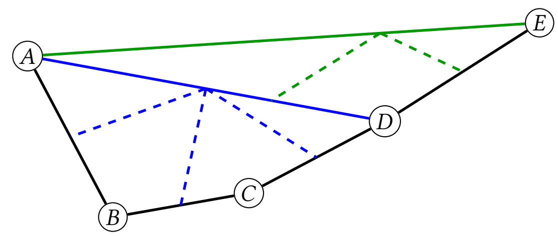

Figure 1: An example Elmo network with 5 nodes, 4 black *simple* (i.e., on-chain) channels, 1 blue virtual channel built only on simple ones, and 1 green virtual channel built on the blue virtual and a simple one. Each virtual channel is connected with its *base* channels with dashed lines of the same colour. The variadic and recursive properties of Elmo are showcased.

Our Contributions. Elmo (named after St. Elmo's fire) is the first Bitcoin-suitable recursive virtual channel constructor that supports channels of indefinite lifetime. In addition, our constructor is variadic and symmetric. Both optimistic and pessimistic execution paths are optimal in terms of round complexity: issuing payments between two endpoints requires just three messages of size independent of the channel length, closing a channel cooperatively needs at most three messages from each party while closing a channel unilaterally demands up to two on-chain transactions for any involved party (endpoint or intermediary) that can be submitted simultaneously, also independent of the channel length. We build Elmo on top of Bitcoin, as this means it can be adapted for any blockchain that supports Turing-complete smart contracts such as Ethereum [61]. The latter provides additional tools to increase Elmo efficiency. Furthermore, Elmo can inspire future blockchain designs that maintain minimal scripting capabilities while providing robust off-chain functionality.

We achieve the above by leveraging 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, parties can securely close the channel cooperatively off-chain, or instead close unilaterally on-chain, following an arbitrary activation sequence. The latter is achieved by enabling anyone to start closing the channel, while subsequent respondents, following the activation sequence, can choose the right action to complete the closure process by posting a single transaction each.

We formally prove the security of our protocol in the Universal Composition (UC) [16] setting (Appx. B); our functionality $\mathcal{G}_{\text{Chan}}$ (Appx. D.1) represents a single channel. It is a global functionality, as defined in [8] (cf. Sec. 5). We also guarantee that the real-world protocol Π_{Chan} has *balance security*, i.e., that honest parties can close the channel and get the expected coins on-chain (Lemma 5.1), similarly to other payment channel proofs [31]. Elmo requires the `ANYPREVOUT` signature type (candidate for inclusion in the next Bitcoin update[1]), which does not sign the hash of the transaction it spends, thus enabling a single pre-signed transaction to spend any output with a suitable script. We leverage `ANYPREVOUT` to avoid exponential storage. We further conjecture that without `ANYPREVOUT` no efficient off-chain virtual channel constructor over Bitcoin can be built. In particular, if any such protocol (i) offers an efficient closing operation (i.e., with $O(1)$ on-chain transactions), (ii) has parties locally store the channel state as transactions and signatures and (iii) does not require locking on-chain coins (unlike [3]), then each party needs exponential space in the number of intermediaries. Note that the second protocol requirement is natural, since, to our knowledge, all trustless layer 2 protocols over Bitcoin require all implicated protocol parties to actively sign off every state transition and locally store the relevant transactions and signatures of their counterparties to ensure they can unilaterally exit later.

Related work. The first proposal for PCNs [60] only enabled unidirectional payment channels. As mentioned previously, DMCs [23] with their decrementing timelocks have the shortcoming of limited channel lifetime. This was ameliorated by LN [55] which has become the dominant paradigm for designing Bitcoin-compatible PCNs. LN is currently implemented and operational for Bitcoin. It has also been adapted for Ethereum, named Raiden Network. Compared to Elmo, LN is more lightweight in terms of storage and communication when setting up, but suffers from increased latency and communication for payments, as intermediaries have to actively participate in multi-hop payments. Its privacy also suffers, as intermediaries learn the exact time and value of each payment.

An alternantive payment channel system for Bitcoin that aspires to succeed LN is eltoo [21]. It is conceptually simpler, has smaller on-chain footprint and a more forgiving attitude towards submitting an old channel state than LN (the old state is superseded without punishment), but it needs `ANYPREVOUT`. Because eltoo and LN function similarly, the previous comparison of Elmo with LN applies to eltoo as well. On a related note, the payment logic of Elmo could also be designed based on the eltoo mechanism instead of the currently used LN.

Perun [27], Thunderdome [6], and GSCN [29] exploit the Turing-complete scripting language of Ethereum to build virtual state channels. GSCN uses a per-channel functionality and a recursive argument similar to that of our UC-security analysis. Their security argument is however flawed, as they incorrectly argue that every level is subroutine respecting with respect to the same environment and subroutines. We believe that, given the versatile Ethereum scripting, GSCN could be straightforwardly extended to support variadic channels. Similar features are provided by Celer [24]. Hydra [17] provides state channels for Cardano [18].

Closer to Elmo, several works propose virtual channel constructions for Bitcoin. LVPC [36] enables a virtual channel to be opened on top of two preexisting channels using a technique similar to DMC, unfortunately inheriting the fixed lifetime limitation. Let *simple channels* be those built directly on-chain, i.e., channels that are not virtual. Bitcoin-Compatible Virtual Channels [2] 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. This strategy has the shortcoming that even if it is made recursive (a direction left open in [2]) 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.

Donner [3] (released originally concurrently with the first technical report of Elmo) achieves variadic, but not recursive virtual channels. This is done by having the funder lock as collateral twice the amount of the desired channel funds: once on-chain with funds that are external to the *base channels* (i.e., the channels that the virtual channel is based on) and once off-chain within its base channel. Thus the funder's collateral is double that of LVPC and Elmo. The collateral for all other parties is the same across LVPC, Donner, and Elmo. Additionally, a Donner channel needs active periodic collaboration of the endpoints and all base channel parties to refresh its lifetime, therefore a Donner channel does not have a truly unlimited lifetime. We conjecture that using external coins precludes variadic virtual channels with unlimited lifetime. This design choice further means that Donner is not symmetric. Donner also uses placeholder outputs which, due to the minimum coins they need to exceed Bitcoin's *dust limit*, may skew the incentives of rational players and add to the channel opportunity cost. The aforementioned incentives together with its lack of recursiveness mean that if a party with coins in a Donner channel decides to use them with another party, it first has to close its channel either off-chain, which needs cooperation of all intermediaries, or else on-chain, with all the delays and fees this entails. Further, its design complicates future iterations that lift its current restriction that only one of the two channel parties can fund the virtual channel. On the positive side, Donner is more efficient than Elmo in terms of storage, computation and communication complexity, and boasts a simpler design. Their work also introduces the *Domino attack*, which we address in Section 7.

We refer the reader to Appx. C for further related work, including a technical issue in Donner and its resolution. Table 1 contains a comparison of the features and limitations of virtual channel protocols, including the current work.

2 Protocol Description

Conceptually, Elmo is split into four actions: channel opening, payments, cooperative closing and unilateral closing. Parties P_1 and P_n may open a channel (P_1, P_n) between them directly on-chain, in which case they follow an opening procedure like that of LN; such a channel is called *simple* and is explained in more detail below. Otherwise they can open it on top of a path of preexisting *base* channels $(P_1, P_2), (P_2, P_3), \dots, (P_{n-1}, P_n)$, in which case (P_1, P_n) is

[1] https://anyprevout.xyz/

Table 1: Features & requirements comparison of virtual channel protocols

	Unlimited lifetime	Recursive	Variadic	Symmetric	Script requirements
LVPC [36]	✗	◐[a]	✗	✓	Bitcoin
BCVC [2]	✓	✗	✗	✓	Bitcoin
Perun [27]	✓	✗	✗	✓	Ethereum
GSCN [29]	✓	✓	✗	✓	Ethereum
Donner [3]	✗	✗	✓	✗	Bitcoin
this work	✓	✓	✓	✓	Bitcoin + ANYPREVOUT

[a]lacks security analysis

a *virtual* channel, also explained later. A channel is either simple or virtual, not both. Since Elmo is recursive, each base channel may itself be simple or virtual. To open a virtual channel, all parties on the path set aside funds in their channels as collateral; they do this by creating so-called *virtual* transactions (txs) that essentially tie the spending of two adjacent base channels into a single atomic action. Once intermediaries are done, they have created a special *funding* output off-chain with the sum of P_1 and P_n's channel balance. P_1 and P_n finally create the channel, applying the logic of simple channels on top of the funding output: their channel is now open. LN demands that the funding output is on-chain, but we lift this requirement. We instead guarantee that either endpoint can put the funding output on-chain unilaterally. A payment over an established channel (described later for simple channels) follows a procedure inspired by LN.

Parties $P_1, \dots, P_n$ can optimistically close a virtual channel completely off-chain. At a high level, parties controlling the base channels *revoke* their *virtual* txs and the related *commitment* txs, i.e., they cannot use them anymore. This effectively "peels" one *virtualisation layer*. In the process, they redistribute coins so that intermediaries "break even", while P_1 and P_n get their rightful coins (as per the last virtual channel state) in their base channels ((P_1, P_2) and (P_{n-1}, P_n) respectively).

Finally, the unilateral closing procedure of a virtual channel (P_1, P_n) (needed if cooperation fails) consists of either P_1 or P_n signing and publishing a number of txs on-chain. In the simplest case, P_1 publishes her virtual tx. This prompts P_2 to publish her virtual tx as well and so on up to P_{n-1}, at which point the funding output of (P_1, P_n) is automatically on-chain and closing can proceed as for simple channels. If instead any intermediary stays inactive, then a timelock expires and causes a suitable output to become the funding output for (P_1, P_n), at the expense of the inactive party. As discussed later, the funding output employs ANYPREVOUT so that the channel needs only a single commitment tx per endpoint, avoiding an exponential state blowup in the recursion depth and making off-chain payments efficient.

In more detail, during the channel opening procedure (cf. Fig. 40) the two counterparties (i) create new keypairs and exchange the resulting public keys, then (ii) if the channel is virtual, prepare the underlying base channels, next (iii) they exchange signatures for their initial commitment txs and lastly, (iv) if the channel is simple, the *funder* (who is the only party that provides coins for a new channel, like in LN) signs and publishes the *funding* tx to the ledger.

In order to build intuition, let us present examples of the lifecycles of a simple and a virtual channel. Consider 5 parties, $A, B, \dots, E$ and 4 channels, $(A, B), \dots, (D, E)$, that will act as the base of the virtual channel (A, E). We first follow the operations of the simple channel (A, B) and then those of (A, E). We simplify some parts of the protocol to aid comprehension — see Appx. A for more details.

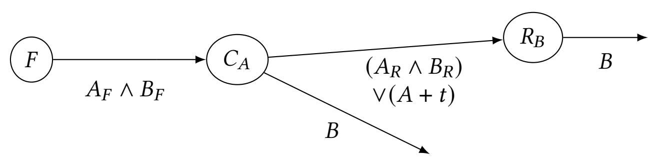

Figure 2: Left to right: funding (F), A's commitment (C_A) and B's revocation (R_B) txs. The symmetric commitment and revocation txs of B and A respectively are not shown. A_F is A's funding key, A_R her revocation key, $A \wedge B$ needs a signature by both A and B. $A + t$ needs a signature by A after relative timelock t. The first C_A output is spendable either by both A_R and B_R, or by $A + t$, with the "or" denoted with $\vee$.

Simple channel. First A and B generate funding and revocation keypairs and exchange public keys (A_F, B_F, A_R, B_R in Fig. 2 – 2 messages). Each then locally generates the *funding* and the two *commitment* txs (F, C_A in Fig. 2). They sign C_A and exchange the signatures (2 messages). A stores her signed commitment tx off-chain and publishes F on-chain. Once F is finalised, the channel is open.

The funding tx F moves A's initial coins to a 2-of-2 multisig, i.e., an output that needs signatures from both A_F and B_F to be spent. There is one commitment tx per party, stored locally off-chain. The one held by A (C_A in Fig. 2) spends the funding tx and has one output for A (initially with all coins) and one for B (initially with 0 coins). A's output can be spent by either a multisig, or by A after a *relative timelock* of t (relative means that the countdown starts at the moment of publication). This is, as we will promptly see, so that B has time to *punish* A if she cheats. B's commitment tx C_B is symmetric.

When A pays c coins to B in the channel, A and B create two new commitment txs. They have the same outputs and scripts as their previous ones, save for the coins: A's outputs have c coins less, B's outputs have c coins more. They sign them and swap signatures. In order to ensure only one set of commitment txs is valid at a time, they then revoke their previous ones. To that end, they build and sign *revocation* txs for the previous commitment txs. B's revocation tx (R_B in Fig. 2) gives B all coins that belonged to A and vice versa. This disincentivises both parties from publishing an old commitment tx under the threat of losing all their channel coins. A payment needs 3 messages in total. Note that, after generating the new commitment txs and before revoking the previous ones, each party has two valid commitments for different channel versions. This is fine, as the payment only concludes when revocation completes. If one party publishes the new commitment tx before revocation, then the other will consider the payment successful and vice versa for the old commitment tx.

A or B can now unilaterally close (A, B) by simply publishing the latest commitment tx on-chain and waiting for the timelock to expire. Since the last commitment tx is not revoked, punishment is impossible. The mechanics of simple Elmo channels are essentially a simplification of LN.

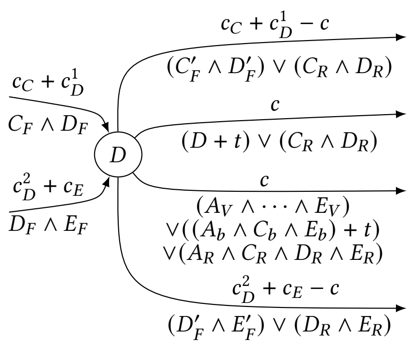

Figure 3: A–E virtual channel: D's initiator transaction. Spends the funding outputs of the C–D and D–E channels. D can use it if neither C nor E have published a virtual transaction yet. A_V: A's "virtual" key. A_b: A's "bridge" key.

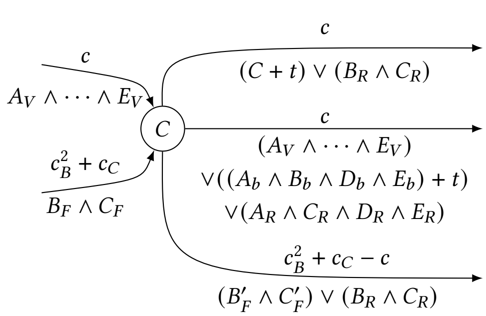

Figure 4: A–E virtual channel: One of C's extend-interval transactions. Spends the virtual output of D's initiator transaction and the funding output of the B–C channel. C can use it if D has already published its initiator transaction and B has not published a virtual transaction yet.

Virtual channel. Assume now that channels $(A, B), \ldots, (D, E)$, are open and the "left" party of each owns at least c coins in it. These channels can be either simple or virtual. In the latter case, the virtual channel (A, E) will leverage the recursive property. Thanks to the similarity of all layers, the description below is identical in both cases. In order for (A, E) to open using $(A, B), \ldots, (D, E)$ as base channels, initially with A having c coins, the 5 parties act as follows. First, they generate and exchange a number of new keys (i.e., all keys that appear in the outputs of Figs. 3–6). Then each base pair removes c coins from the "left" party in their base channel. The updated commitment txs use some of the new keys for the multisig in their input, since, as we will see, so-called *virtual* txs will stand between the funding and the commitment txs from now on. These virtual txs will form the *virtual layer* (Fig. 7).

Next, parties generate and sign these virtual txs and send the signatures among them. These txs sit at the core of Elmo. Intuitively, they force each intermediary to interact with both of its base channels instead of one at a time, ensuring that information relevant to the virtual channel "flows" along the base channels. There are 3 virtual tx types – looking ahead, we now explain them via example of their use during channel closing. The *initiator* virtual tx spends the two funding outputs of an intermediary (say D) and thus needs to be signed by C_F, D_F, and E_F. It produces 4 outputs: one new funding output for each of (C, D) and (D, E) (Fig. 3, top & bottom outputs), one output that refunds the collateral to D (2nd from top) and, crucially, a so-called *virtual* output (3rd from top). The latter can be spent by C with an *extend-interval* tx, the second type of virtual tx (Fig. 4), which needs signatures from all 5 parties (top input). This tx also spends the other, as-of-yet unspent, C's funding output, namely that of (B, C) (bottom input). It has 3 outputs: one refunding the collateral to C (top), another virtual output (middle), and a funding output that replaces the one just spent (bottom). In our example, A also uses its initiator tx, which is different since A is an endpoint (Fig. 5). It spends only the funding output of (A, B) and produces 2 outputs: a new funding output for (A, B) (top) and a virtual output (bottom) – this is the only virtual tx A needs. B can in turn spend A's and C's virtual outputs with a *merge-intervals* tx (Fig. 6), the last virtual tx type, which also needs signatures from the *virtual keys* $A_V \ldots E_V$ of all 5 parties. It has 2 outputs: one refunds B and the other produces a new virtual output. Now all base funding outputs are spent, all intermediaries are refunded and the virtual layer has "collapsed". Additionally, the virtual output of B's tx plays the role of the funding output of the virtual channel (A, E).

The virtual txs are designed around two axes: First, each intermediary can publish only one virtual tx, which refunds its collateral exactly once. We will see how this is enforced below. Second, if the chain of virtual txs is at any point broken by an inactive intermediary that does not publish its virtual tx, the virtual channel will still be funded correctly and the inactive party will lose its collateral. This is guaranteed because the unclaimed virtual output automatically turns into the funding output of the virtual channel after a timelock. See, e.g., 2nd spending method of the 3rd output of Fig. 3. Keys $A_b \ldots E_b$ can be spent by *bridge* txs, the output of which funds the (A, E) channel.

Bridge txs ensure that parties only need to maintain a single commitment tx. Any virtual output may end up funding the virtual channel, but the various virtual outputs do not have the same script, thus there cannot be a single commitment tx able to spend all of them. Bridge txs protect parties from having to keep track of

$O(n^3)$ commitment txs. They all have the same output, unifying the interface between the virtualisation and the payment txs, making virtual channel updates as cheap as simple channel updates.

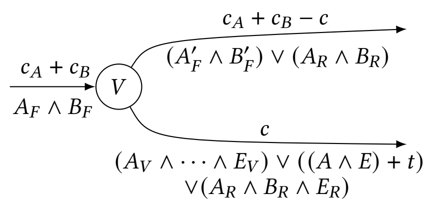

Figure 5: A–E virtual channel: A's initiator transaction. Spends the funding output of the A–B channel. A can use it if B has not published a virtual transaction.

Some considerations remain to ascertain the scheme security. Firstly, we must ensure that no intermediary can publish more than one virtual tx to protect the endpoints from an unbounded sequence of virtual txs preventing them from accessing their funding output indefinitely: malicious parties can fabricate arbitrarily many virtual outputs using their own, external to the protocol, coins, therefore if all virtual outputs were identical, an adversary could publish a perpetual stream of merge-intervals txs, spending one valid and one fabricated virtual output. This is safeguarded by specifying on each virtual output the exact sequence of parties that have already published a virtual tx and only allowing the parties at the two edges of the sequence to extend it with their virtual tx. If all intermediaries publish a virtual tx, then the last virtual output that was published is not spendable by another virtual tx. This ensures that the endpoints will eventually obtain a funding output. Preventing this attack means that intermediaries need to store $O(n^3)$ virtual txs for a virtual channel over n parties. Secondly, we have to carefully select the exact values of timelocks to ensure that each party has enough time to act. The timelocks increase linearly with the depth of the recursion. The exact values are shown in Sec. 4 and Appx. F.

We now return to the opening procedure. After the 5 parties set up the virtual txs, they revoke their previous commitment txs. They do this by signing the relevant revocation txs, just like for a simple channel. It takes $12 \cdot (n-1)$ messages, i.e., 6 messages per endpoint and 12 messages per intermediary, to set up the virtual layer.

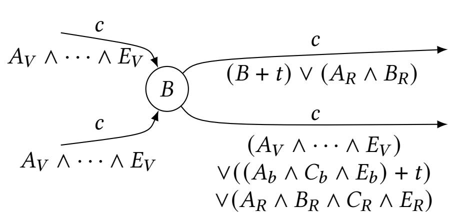

Figure 6: A–E virtual channel: One of B's merge intervals transactions. Spends the virtual outputs of A's and C's virtual transactions. B can use it if both A and C have already published their initiator or extend-interval transactions.

At last, the 5 parties have set up virtual layer: Both A and E can unilaterally force the funding output of their virtual channel on-chain, irrespective of the actions of the rest of the parties. Likewise, honest intermediaries can unilaterally retrieve their funds.

A and E finally exchange commitment transactions for their new channel, thus concluding its opening. A and E can pay each other over their virtual channel exactly like they would over a simple channel; we refer the reader to the relevant description above.

Note that funding outputs use the `ANYPREVOUT` flag, thus ensuring that a single pair of commitment txs can spend any of the funding outputs. If `ANYPREVOUT` were not used, each virtual layer would need a copy of the entire set of discussed txs for each possible funding output of its base layer, resulting in exponential storage requirements. To make matters worse, a payment over (A, E) would need A and E to renegotiate exponentially many commitment txs, as well as recalculate all their downstream txs, which would in turn need interaction with intermediaries of all virtual channels built over (A, E), completely defeating the essence of payment channels.

Cooperative closing. To enhance usability, our protocol enables closing the virtual channel off-chain if all parties cooperate. To do this, the endpoints first let the intermediaries know their final virtual channel balance. Then the parties of each base channel create new commitment txs for their channels, moving the collateral back into the channel: the "left" party gets A's coins and the "right" one gets E's. Thus all intermediaries "break even" across their two channels. Once they do this, the parties revoke all virtual txs, using a logic similar to the revocation procedure of simple channels but scaled up to all parties. This is why all virtual tx outputs (Figs. 3–6) have a spending method with $A_R \dots E_R$ keys.

What if one party does not cooperate? Then one or more of the other parties must close unilaterally on-chain. Fig. 7 shows how this would play out if A and D initiated this procedure.

Our protocol is recursive because both simple and virtual channels are ultimately represented by a funding output that either already is or can be put on-chain, therefore new virtual channels can be built on either. Both simple and virtual channels avoid key reuse on-chain, thus ensuring party privacy from on-chain observers. See Appx. E for more details.

3 Model

3.1 $\mathcal{G}_{\text{Ledger}}$ Functionality

In this work we embrace the Universal Composition (UC) framework [16] together with its global subroutines extension, UCGS [8], to model parties, network interactions, adversarial influence and corruptions, as well as formalise and prove security — more details on UC in Appx. B. We model the Bitcoin ledger with the $\mathcal{G}_{\text{Ledger}}$ functionality as defined in [9, 11]. $\mathcal{G}_{\text{Ledger}}$ 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

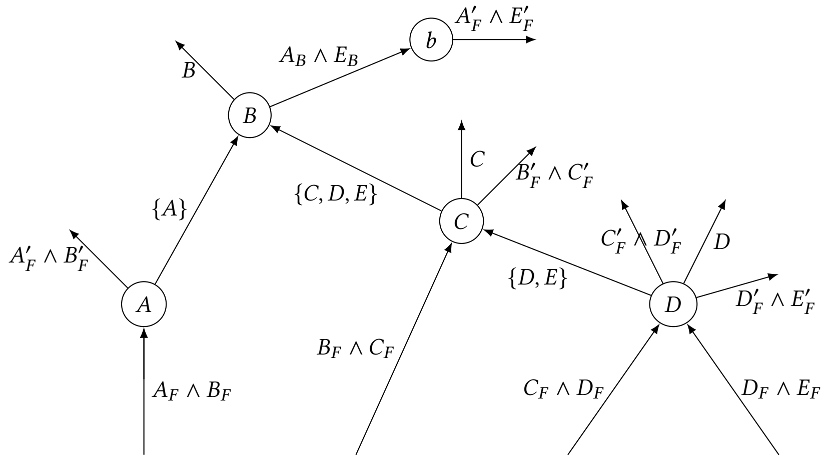

Figure 7: 4 simple channels supporting a virtual one. A **and** D **start closing by publishing their initiator virtual txs, then** C **publishes its suitable extend-interval virtual tx, after which** B **publishes its suitable merge-intervals virtual tx. No party stays inactive. The virtual transactions** A–D **form the virtual layer. Virtual outputs are marked with the set (interval) of parties that have already published a tx. *Bridge* txs like** b **are used by** A **and** E **to convert the various virtual outputs into the same funding output, as `ANYPREVOUT` only works across identical outputs.**

window, and it cannot be reordered or removed. This is named persistence.

Moreover, $\mathcal{G}_{\text{Ledger}}$ needs the $\mathcal{G}_{\text{CLOCK}}$ functionality [37], which models the notion of time. Any $\mathcal{G}_{\text{CLOCK}}$ participant can request to read the current time 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. Both $\mathcal{G}_{\text{Ledger}}$ and $\mathcal{G}_{\text{CLOCK}}$ are global functionalities [8] and therefore can be accessed directly by the environment. The definitions of $\mathcal{G}_{\text{Ledger}}$ and $\mathcal{G}_{\text{CLOCK}}$ can be found in Appx. ??.

3.2 Modelling time

The protocol (Π_{Chan}) and functionality ($\mathcal{G}_{\text{Chan}}$) defined in this work do not use $\mathcal{G}_{\text{CLOCK}}$ directly. The only notion of time is provided by the blockchain height, as reported by $\mathcal{G}_{\text{Ledger}}$. We thus omit it in our lemmas and theorems statements to simplify notation; it should normally appear as a hybrid together with $\mathcal{G}_{\text{Ledger}}$.

Our protocol does not impose any additional synchrony assumptions beyond what is needed by the underlying blockchain, i.e., the adversary can delay protocol messages arbitrarily long. The protocol is robust against such delays, as an honest party can unilaterally prevent loss of funds even if some of its messages are dropped by $\mathcal{A}$, given that the party can communicate with $\mathcal{G}_{\text{Ledger}}$. In other words, no extra synchrony assumptions to those required by $\mathcal{G}_{\text{Ledger}}$ are needed. We also note that, following the conventions of single-threaded UC execution model, the duration of local computation is not taken into account (as long as it does not exceed its polynomial bound).

4 Protocol Pseudocode

We here present a simplified version of the Π_{Chan} protocol. We omit complications imposed by UC. Appx. F contains the full UC protocol and Appx. E.3 its in-depth description in prose.

Process Π_{Chan} – self is P

- At the beginning of each activation:
  - **if** we have not been activated for more than p blocks **then**
    - We are negligent // no balance security guarantees
- Open channel with counterparty P': // A, B of Fig. 2 are P, P' resp.
  - Generate funding (P_F) and revocation (P_R) keypairs.
  - Exchange funding, revocation & own (P) public keys with P'.
  - **if** opening virtual (off-chain) channel **then**
    - Run next bullet "Host a virtual channel" as endpoint.
  - Exchange & verify signatures by P_F and P'_F on C_P and $C_{P'}$.
  - **if** opening simple (on-chain) channel **then**
    - Prepare and submit funding tx (F) to ledger and wait for its inclusion. // only one party funds the channel, so the funding tx needs only the funder's signature
    - $t_P \leftarrow s + p$ // simple channel timelock
    - // s: max blocks before submitted tx enters ledger
- Host virtual channel of c coins (endpoint/intermediary): // Fig. 3–7
  - Ensure we have at least c coins.
  - Generate one new funding keypair (P'_F), $O(n^2)$ virtual keypairs (P_V) (1 per hop and party, to control which virtual txs can spend which) and one virtual revocation keypair (P_R).
  - // Revocation keys in virtual and commitment txs are distinct, but we reuse notation in Fig. 2 and Figs. 3–7 for simplicity.
  - Exchange these public keys with all base channel parties.
  - Generate and sign new commitment txs (C_P, $C_{P'}$) with our base channel counterparty/ies (1 if endpoint, 2 if intermediary), using the new funding and latest revocation keys and reducing by c the balance of the party "closer" to the funder.
  - Exchange signatures with counterparty/ies and verify them.
  - Generate and sign all $O(n^3)$ virtual and bridge txs (b of Fig. 7) with the virtual (P_V) and bridge (P_B) keys.
  - Exchange signatures among all base channel parties and verify that all our virtual txs have signed virtual inputs.

Exchange with counterparty/ies and verify signatures for the funding inputs of our initiator and extend-interval txs.
Exchange with counterparty/ies and verify signatures for the revocation txs of the previous channel state.
if we are intermediary **then**
  $t_P \leftarrow \max\{t$ of left channel, t of right channel$\}$
else // we are endpoint
  $t_P \leftarrow p + \sum_{j=2}^{n-1}(s-1+t_j)$ // max delay is O(sum of intermediaries' delays). Occurs when we use initiator tx and each intermediary uses extend-interval tx sequentially.

- React if counterparty publishes virtual tx:
  Publish our only valid virtual tx. // if both counterparties have published, this is a merge-intervals tx, otherwise it is an extend-interval tx.
- Pay x coins to P' over our (simple or virtual) channel:
  Ensure we have at least x coins.
  if we host another virtual channel **then**
    Ensure new balance prevents griefing. // cf. E.3
  Generate and sign new commitment txs (C_P, $C_{P'}$), with x coins less for the payer and x coins more for the payee.
  Exchange and verify signatures by funding keys (P_F, P'_F).
  Sign revocation txs (R_P, $R_{P'}$) corresponding to old commitment txs with revocation keys (P_R, P'_R).
  Generate next revocation keypairs.
  Exchange and verify revocation signatures and public keys.
- Close virtual channel unilaterally:
  Publish initiator & bridge tx. // Funding output is on-chain
  Publish our latest commitment tx on-chain.
- Close virtual channel cooperatively: // Only if not hosting
  Endpoints send their balance (c_1, c_2) to all parties.
  Parties ensure endpoints agree and $c_1 + c_2 = c$.
  All parties generate and sign new commitment txs with:
  - the funding keys used before opening virtual channel,
  - the new revocation keys, and
  - c_1 more coins to party closer to funder, c_2 to the other.

  All parties generate new revocation keypairs.
  All pairs exchange & verify sigs & new revocation public keys.
  All parties generate and sign revocation txs for the old virtual, bridge and commitment txs with their virtual revocation keys.
  All pairs exchange and verify these signatures.
- Punish malicious counterparties: // Run every p blocks
  if an old commitment tx is on-chain **then**
    Sign w/ revocation key & publish corresp. revocation tx.
  if the ledger contains an old virtual or bridge tx **then**
    Sign w/ revocation key & publish corresp. revocation tx(s).

Figure 8: High level pseudocode of the Elmo protocol

5 Security

Before providing the UC-based security guarantees, it is useful to obtain concrete properties directly from our protocol, essentially providing standard property-based security in addition to our UC guarantees. We first delineate the security guarantees Elmo provides by proving Lemma 5.1 which discusses the conservation of funds. The formal statements (G.1 and G.2) along with all proofs are deferred to Appx. G. Informally, it establishes that if an honest, non-negligent party (Fig. 4, 1st bullet) was implicated in a channel that has now been unilaterally closed, then the party will have at least the expected funds on-chain.

Lemma 5.1 (Real world balance security (informal)). *Consider a real world execution with $P \in \{Alice, Bob\}$ an honest, non-negligent party. Assume that all of the following are true:*

- *P opened the channel, with initial balance c,*
- *P is the host of n channels, each funded with f_i coins,*
- *P has cooperatively closed k channels, where the i-th channel transferred r_i coins from the hosted virtual channel to P,*
- *P has sent m payments, each involving d_i coins,*
- *P has received l payments, each involving e_i coins.*

If P closes unilaterally, eventually there will be $h \in \mathbb{N}^*$ outputs on-chain spendable only by P, each of value c_i, such that

$$\sum_{i=1}^{h} c_i \geq c - \sum_{i=1}^{n} f_i + \sum_{i=1}^{k} r_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \ . \qquad (1)$$

The expected funds are [initial balance - funds for hosted virtuals + funds returned from hosted virtuals - outbound payments + inbound payments]. Note that the funds for hosted virtuals only refer to those funds used by the funder of the virtual channel, not the rest of the base parties. A similar property-based guarantee is provided for the ideal-world functionality $\mathcal{G}_{\text{Chan}}$ in Lemma G.2.
Proof Sketch [Lemma 5.1]. All execution paths are followed, keeping track of the resulting balance in each case and concluding that balance is secure in all cases, except if signatures are forged. □
It is important to note that in fact our protocol Π_{Chan} (Appx. F) provides a stronger guarantee: a party can always unilaterally close its channel and obtain the expected funds on-chain within a known number of blocks. This stronger guarantee is sufficient to make Elmo reliable enough for real-world applications. However an ideal world functionality with such guarantees would have to be aware of specific txs and signatures, making it as complicated as the protocol, thus violating the spirit of the simulation-based security paradigm.

$\mathcal{G}_{\text{Chan}}$ (Appx. D) halts on security breaches (e.g., lower than expected balance). We prove the "no-halt" Lemma (G.3), which informally states that if an ideal party is honest, $\mathcal{G}_{\text{Chan}}$ does not halt with overwhelming probability.

Since $\mathcal{G}_{\text{Chan}}$ corresponds to a single channel, which in turn can form the base of multiple independent virtual channels and thus needs to be accessible by all of them, $\mathcal{G}_{\text{Chan}}$ is a global functionality, i.e., it can communicate with entities outside the (single-channel) protocol. The alternative of modelling all channels within a single protocol [39] leads to a monolithic, hard-to-reuse ideal functionality.

As per Def. 19 of [15], a *subroutine respecting* protocol must not pass input to a party of a different session. In order to open a virtual channel however, Π_{Chan} passes inputs to a Π_{Chan} instance of another session, thus Π_{Chan} is not subroutine respecting. To address this, we first add a superscript to Π_{Chan}, i.e., Π^n_{Chan}. Π^1_{Chan} is always a simple channel. This is done by ignoring instructions to open on top of other channels. As for higher superscripts, $\forall n \in \mathbb{N}^*, \Pi^{n+1}_{\text{Chan}}$ is the same as Π_{Chan} but with base channels of a maximum superscript n. It then holds that $\forall n \in \mathbb{N}^*, \Pi^n_{\text{Chan}}$ is $(\mathcal{G}_{\text{Ledger}}, \Pi^1_{\text{Chan}}, \dots, \Pi^{n-1}_{\text{Chan}})$-*subroutine respecting* [8]. The same superscript trick is done to $\mathcal{G}_{\text{Chan}}$, thus the composition theorem of [8] is applicable (Appx. G). To the best of the authors' knowledge, this recursion-based proof technique for UC security is novel. It is of independent interest and can be reused to prove UC security in protocols that may use copies of themselves as subroutines. Theorems G.4 and G.5 (Appx. G) state that $\forall n \geq 1, \Pi^n_{\text{Chan}}$ UC-realises $\mathcal{G}^n_{\text{Chan}}$. Furthermore, all ideal global subroutines can be replaced with their real counterparts (Lemma G.6 and Theorem G.7).

6 Efficiency Evaluation & Simulations

We offer here a cost and efficiency comparison of this work with LVPC [36] and Donner [3]. We focus on these due to their exclusive support of virtual channels over any number of base channels. We remind that LVPC achieves this via recursion, while Donner because it is variadic (cf. Table 1).

We count the communication, storage and on-chain cost of a virtual channel in each protocol. We also simulate the execution of a large number of payments among many parties and derive payment latency and fees – contrary to our theoretical analysis, fees are included in our simulations to make the comparison more robust. We thus obtain an end-to-end understanding of both the requirements and benefits of each protocol.

Cost calculation. Consider the setting of 1 funder (P_1), 1 fundee (P_n) and $n-2$ intermediaries ($P_2, \dots, P_{n-1}$) where P_i has a base channel with each of P_{i-1}, P_{i+1}. We compare the costs of off-chain opening (Table 2) and on-chain closing unilaterally (Table 3).

Regarding opening, in Table 2 we calculate for each of the 3 protocols the number of communication rounds required, the total size of outgoing messages as well as the amount of space for storing channel data. We calculate funder, fundee and intermediary requirements, along with the aggregate for all parties. The communication rounds for a party are calculated as its [#incoming messages + #outgoing messages]/2. The size of outgoing messages and the stored data are measured in raw bytes. The data is counted as the sum of the relevant channel identifiers (8 bytes each, as defined by the Lightning Network specification[2]), transaction output identifiers (36 bytes), secret keys (32 bytes each), public keys (32 bytes each, compressed form – these double as party identifiers), Schnorr signatures (64 bytes each), coins (8 bytes each), times and timelocks (both 4 bytes each). UC-specific data is ignored.

For LVPC, multiple different topologies can support a virtual channel between P_1 and P_n (all of which need $n-1$ base channels). We here consider the case in which the funder P_1 first opens one virtual channel with P_3 on top of channels (P_1, P_2) and (P_2, P_3), then another virtual channel with P_4 over (P_1, P_3) and (P_3, P_4) and so on up to the (P_1, P_n) channel, opened over (P_1, P_{n-1}) and (P_{n-1}, P_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).

A subtle byproduct of the above topology is that during the opening phase of LVPC every intermediary P_i acts both as a fundee in its virtual channel with the funder P_1 and as an intermediary in the virtual channel of P_1 with the next party P_{i+1}. The above does not apply to the first intermediary P_2, since it already has a channel with P_1 before the protocol starts. Table 2 shows the total cost of intermediaries $P_3, \dots, P_{n-1}$. The first intermediary P_2 incurs instead [intermediary's costs - fundee's costs] for all three measured quantities.

For Elmo, the data are derived assuming a virtual channel opens directly on top of $n-1$ base channels. In other words the channel considered is opened without the help of recursion and only leverages the variadic property of Elmo. In Table 2 the resources calculated for Elmo are exact for $n \geq 4$ parties, whereas for $n = 3$ they slightly overestimate.

For the closing comparison, we calculate on-chain transactions' size in vbytes[3], which map directly to on-chain fees and thus are preferable to raw bytes. Using vbytes also ensures our comparison remains up-to-date irrespective of the network congestion and bitcoin-to-fiat currency exchange rate at the time of reading. We use a suitable tool[4] to aid size calculation. For the case of intermediaries, in order to only show the costs incurred due to supporting a virtual channel, we subtract the cost the intermediary would pay to close its channel if it was not supporting any virtual channel.

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.

In Table 3 we calculate for each of the three protocols the worst-case on-chain cost for a party in order to unilaterally close its channel. The cost is measured both in the number of transactions and in their total size.

For the two endpoints (funder and fundee), we show the cost of unilaterally closing the virtual channel, whereas for each intermediary we report the cost of closing a base channel. We also present the worst-case total on-chain cost, aggregated over all parties. Note that the latter cost is not simply the sum of the worst-case costs of all parties, as one party's worst case is not necessarily the worst case of another. This cost rather represents the maximum possible load an instantiation of each protocol could add to the blockchain when closing.

We note that Elmo exploits MuSig2 [48, 52] to reduce both its on-chain and storage footprint: the n signatures that are needed to spend each virtual and bridge output can be securely reduced to a single aggregate signature. The same cannot be said for Donner, since this technique cannot optimise away the n outputs of the funder's transaction $\mathtt{tx}^{\mathsf{vc}}$. Likewise LVPC cannot gain a linear improvement with this optimisation, since each of its relevant transactions ("split", "merge" and "refund") needs constant signatures.

We furthermore note that, since human connections form a small world [49], we expect that in practice the need for virtual channels with a large number of intermediaries will be exceedingly rare. This is corroborated by the fact that LN only supports payments of up to 20 hops without impact to its usefulness. Therefore, the asymptotic network and storage complexity are not as relevant as the concrete costs for specific, low values of n. Under this light, the overhead of Elmo is tolerable. For example, the total size of messages sent and received for a funder when opening an Elmo channel of length $n = 6$ are less than 3 times those of Donner and slightly cheaper than LVPC (Table 2).

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

[3]https://en.bitcoin.it/wiki/Weight_units

[4]https://jlopp.github.io/bitcoin-transaction-size-calculator/

Open											
	Funder			Fundee			Intermediary			Total	
	party rounds	size		party rounds	size		party 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	2	$184n + 829$	$1332.5k + 43n + 125.5$	1	$43n + 192.5$	$1332.5k + 43n + 125.5$	1	547	$1332.5k + 43n + 125.5$	$774n - 71$	$1332.5kn + 43n^2 + 125.5n$
Elmo	6	$32n^3 - 128n^2 + 544n - 276$	$\frac{128}{3}n^3 - 128n^2 + \frac{1276}{3}n + 220$	6	$32n^3 - 128n^2 + 544n - 340$	$\frac{128}{3}n^3 - 128n^2 + \frac{1276}{3}n + 220$	12	$96n^3 - 256n^2 + 404n - 40$	$96n^3 - 256n^2 + 468n + 88$	$96n^4 - 384n^3 + 724n^2 + 240n - 792$	$96n^4 - \frac{1088}{3}n^3 + 660n^2 + \frac{8}{3}n + 520$

Table 2: Open efficiency comparison of virtual channel protocols with n parties and k payments.

Unilateral Close								
	Intermediary		Funder		Fundee		Total	
	#txs	size	#txs	size	#txs	size	#txs	size
LVPC	3	627	2	383	2	359	$2n - 2$	$435n - 510.5$
Donner	1	204.5	4	$704 + 43n$	1	136.5	$2n$	$458n - 26$
Elmo	1	297.5	3	376	3	376	$n + 1$	$254.5n - 133$

Table 3: On-chain worst-case closing efficiency comparison of virtual channel protocols with n parties.

Moreover, Elmo can be deployed on commodity hardware in practice: To set up and maintain a channel of length $n = 6$, an intermediary needs to send less than 14 KiB over the network and store less than 15 KiB — the endpoints need even less resources.

Payment simulations. We implemented a simulation framework[5] in which a list of randomly generated payments are carried out. A single simulation is parametrised by a list of payments (sender, receiver, value triples), the protocol (Elmo, Donner, LVPC, LN or on-chain only), which future payments each payer knows and the utility function it maximises. The knowledge function defines which future payments inform each decision. Several knowledge functions are provided, such as full knowledge of all future payments and knowledge of the payer's next m payments. This degree of freedom is included as parties that know their future steps can better plan their channel configuration.

The utility of a payment is high when its latency and fees are low, it increases the payer's network centrality, and reduces distance from other parties. We weigh low latency and fees most, then small distance and high centrality last. Latency here is the time that passes until a payment is finalized. This depends on whether the party decides to do an on-chain transaction, open a new channel, or do an off-chain transaction if possible. Note that the first two are bound by the ~10′ latency of Bitcoin blocks. We measure latency in seconds. Recognising the arbitrary nature of the concrete weights, we chose them before running our simulations in order to minimise bias. Each payment is carried out by dry-running all known future payments with the three possible payment kinds (simply on-chain, opening a new channel, using existing channels), comparing their utility and executing the best one.

Our simulation framework is of independent interest, as it is flexible and reusable for a variety of payment network protocol evaluations. We here show the performance of the 3 protocols with respect to the metrics payment channels aim to improve, namely payment latency (Fig. 9) and fees (Fig. 10). We have designed 3 workloads: "power law", in which incoming payments follow a zipf [56] distribution, "preferred receiver", in which each party has a preferred payee which receives half of the payments, and "uniform", where payments are chosen uniformly at random.

Due to the privacy guarantees of LN, we are unable to obtain real-world off-chain payment data. We therefore generate payments randomly. More specifically, we provide three different payment workloads to mimic different usage schemes: For the first, each party has a *preferred receiver*, chosen uniformly at the beginning, which it pays half the time, the other half choosing the payee uniformly at random. Each payment value is chosen uniformly at random from the $[0, \max]$ range, for $\max = \frac{\text{(initial coins)} \cdot \text{\#players}}{\text{\#payments}}$. We employ 1000 parties, with a knowledge function disclosing to each party its next $m = 100$ payments, as it appeared this is a realistic knowledge function for this case. This scenario occurs when new users are onboarded with the intent to primarily pay a single counterparty, but sporadically pay others as well. For the second, in an attempt to emulate real-world payment distributions, the value and number of incoming payments of each player are drawn from the zipf [56] distibution with parameter 2, which corresponds to real-world *power-law* distributions with a heavy tail [12]. Each payment value is chosen according to the zipf(2.16) distribution which corresponds to the 80/20 rule [25], moved to have a mean equal to $\frac{\max}{2}$. We consider 500 parties, and a knowledge function with $m = 10$, as this is more aligned with real-world scenarios. For the third, all choices are made *uniformly at random*, with each payment chosen uniformly from $[0, \max]$, employing a total of 3000 parties, again with each knowing its next $m = 10$ payments. For all scenarios the payer of each payment is chosen uniformly at random, no channels exist initially, and all parties initially own the same amount of coins on-chain. A payer funds a new channel with the minimum of all the on-chain funds of the payer and the sum of the known future payments to the same payee plus 10 times the current payment value. Virtual channels are preferred to simple ones when possible. The coins that fund new virtual channels are essentially decided

[5] gitlab.com/anonymised-submission-8778e084/virtual-channels-simulation

in the same way as funding coins for simple channels. The only difference is that the availability of funding from the underlying channel is artificially limited so that it does not deplete too fast. This ensures that the underlying channel can be used as a base for several virtual channels, as well as for more transactions. The authors would use this heuristic to allocate their own funds in real-life use cases. The fees for new virtual channels are decided as follows: There is a fixed base fee that has to be payed for each intermediary. Furthermore, each intermediary gets a small fixed proportion of the number of coins that will be put on the virtual channel. The number of parties is chosen to ensure the simulation completes within a reasonable length of time.

In the simulation we do not close any channels, since this would require complicated heuristics of when to close a channel. One reason to avoid it is simplicity. Furthermore in the real world parties will act according to the characteristics of the payment network. Hence the heuristics should to some extent depend also on these characteristics and therefore would be easy to skew in favor of whichever payment network one prefers.

In order to avoid bias, we simulate each protocol with the same payments. We simulate each scenario with 20 distinct sets of payments and keep the average. In Figs. 9 and 10, scale does not begin at zero for better visibility. Payment delays are calculated based on which protocol is used and how the payment is performed. Average latency is high as it describes the whole run, including slow on-chain payments and channel openings. Total fees are calculated by summing the fee of each "basic" event (e.g., paying an intermediary for its service). None of the 3 protocols provide fee recommendations, so we use the same baseline fees for the same events in all 3 to avoid bias. These fees are not systematically chosen, therefore Fig. 10 provides relative, not absolute, fees.

As Fig. 9 shows, delays are primarily influenced by the payment distribution and only secondarily by the protocol: Preferred receiver is the fastest and uniform is the slowest. This is reasonable: In the preferred receiver scenario at least half of each party's payments can be performed over a single channel, thus on-chain actions are reduced. On the other hand, in the uniform scenario payments are spread over all parties evenly, so channels are not as well utilised.

As evidenced, Elmo is the best or on par with the best protocol in every case. We attribute this to the flexibility of Elmo, as it is both variadic and recursive, thus able to exploit the cheapest payment method in all scenarios. In particular, Donner is consistently the most fee-heavy protocol and LVPC the slowest. Elmo experiences similar delays to Donner and slightly higher fees than LVPC.

7 Discussion and Future work

Domino attack. In [3] the Domino attack is presented. Briefly, it claims that a malicious virtual channel member can force other channels to close. To illustrate this, suppose A and E of Fig. 7 open channels (A, B), (D, E) and (A, E) with the sole intent of forcing channels (B, C) and (C, D) to close. We observe that, contrary to the attack goal, honest base parties are only forced to publish a single virtual transaction each, which places their funding outputs on-chain but does not cause the base channels to close. There is still a small downside: the channel capacity is reduced by the collateral, which is paid directly to one of the two base channel parties. Since no coins are stolen, the only cost to B, C and D is the on-chain fees of one transaction. This is an inherent but small risk of recursive channels, which must be taken into account when making one's channel the base of another. This risk can be eliminated by making an attacker pay the fees for the others' on-chain transactions. This fee need not apply in case of cooperative closing nor during normal operation and can be reduced for reputable parties. Suitable reputation systems, as well as mechanisms for assigning inactivity blame (i.e., proving which parties tried collaborative closing before closing unilaterally), which are needed to determine who must pay the fees, can be designed but are beyond the scope of this work.

Furthermore, a simple modification to Elmo eliminates the channel capacity reduction under a Domino attack, while also reducing the on-chain cost of unilateral closing: from each virtual tx, we eliminate the output that directly pays a party (e.g., 1st output of Fig. 4) and move its coins into the funding output of this transaction (e.g., 3rd output of Fig. 4). We further ensure at the protocol level that the base party that owns these coins never allows its channel balance to fall below the collateral, until the supported virtual channel closes. This change ensures that the collateral automatically becomes available to use in the base channel after the virtual one closes, keeping more funds off-chain after a Domino attack. This approach however has the drawback that, depending on which of the two parties first spends the funding output (with a virtual tx), the funds allocation in the channel differs. Base parties thus would have to maintain two sets of commitment and revocation txs, one for each case. Since this overhead encumbers the optimistic, cooperative case and only confers advantages in the pessimistic case, we choose not to adopt this approach into our design.

Future work. A number of features can be added to our protocol for additional efficiency, usability and flexibility. First of all, 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 and cooperative closing subprotocol with the ability to cooperatively open and close multiple virtual channels in the same layer, either simultaneously or by amending an existing virtualisation layer.

In this work a channel can be funded by only one of the two endpoints. This limitation simplifies the execution model and analysis, but can be lifted at the cost of additional protocol complexity.

Furthermore, as it currently stands, the timelocks calculated for the virtual channels are based on p (Figure 33) and s (Figure 37), 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 A.1 (Appx. A.3) 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 the counterparties agree. We leave this usability-augmenting protocol feature as future work.

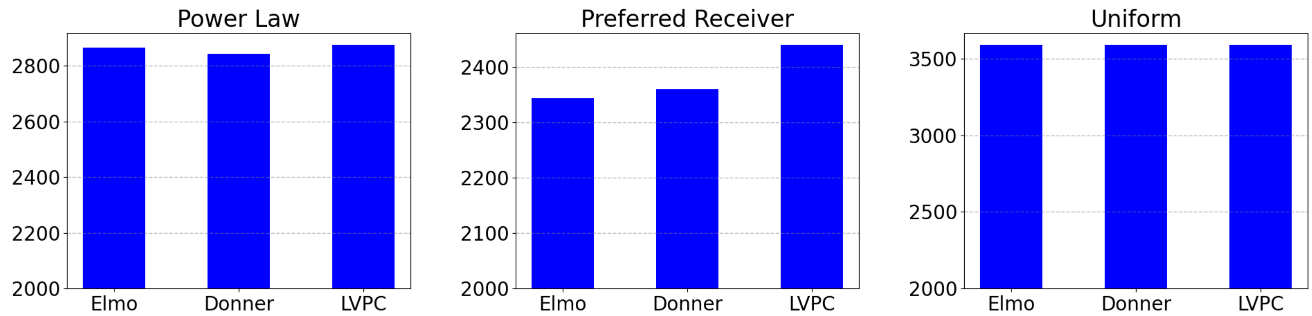

Figure 9: Average per-payment delay (both on- and off-chain) in sec. Less is better.

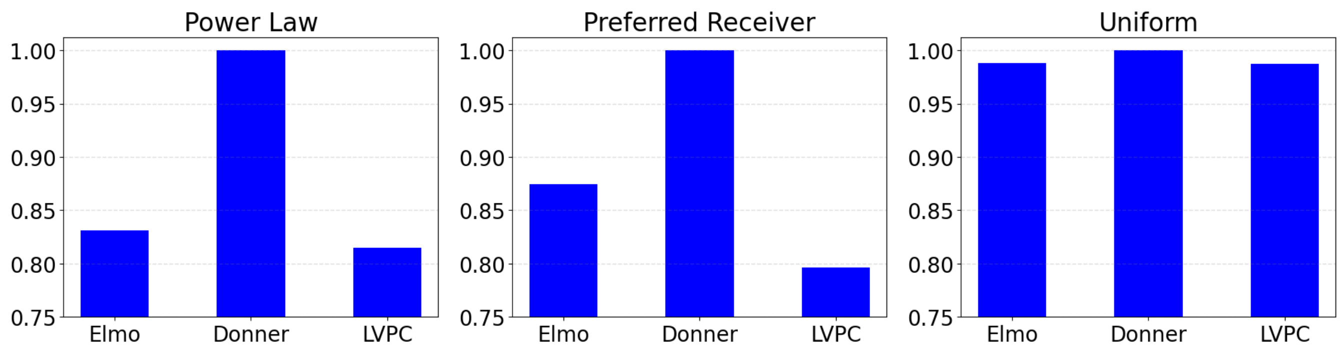

Figure 10: Average per-payment relative fee. Less is better.

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 channel fund 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.

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 and allow human users to choose which method to use in each case. Likewise, optimistic cooperative on-chain closing of simple channels could be done just like in Lightning, obviating the need to wait for the revocation timelock to expire and reducing on-chain costs if the counterparty is cooperative.

What is more, any deployment of the protocol has to explicitly handle the issue of tx fees. These include miner fees for on-chain txs 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. Additionally, a fee structure that takes into account the opportunity cost of base parties locking collateral for a potentially long time is needed. A straightforward mechanism is for parties to agree on a time-based fee schedule and periodically update their base channels to reflect contingent payments by the endpoints. We leave the relevant incentive analysis as future work.

In order to increase readability and to keep focus on the salient points of the construction, our protocol does not exploit various possible optimisations. These include allowing parties to stay offline for longer [4], and some techniques employed in Lightning that drastically reduce storage requirements, such as storage of per-update secrets in $O(\log n)$ space[6], and other improvements to our novel virtual subprotocol.

As mentioned before, we conjecture that a variadic virtual channel protocol with unlimited lifetime needs each party to store an exponential number of signatures if `ANYPREVOUT` is not available. We leave proof of this as future work. Furthermore, the formal verification of the UC security proof is deferred to such a time when a practical framework for mechanised UC proofs becomes available.

Last but not least, the current analysis gives no privacy guarantees for the protocol, as it does not employ onion packets [20] like Lightning. Furthermore, $\mathcal{G}_{\text{Chan}}$ leaks all messages to the ideal adversary therefore theoretically no privacy is offered at all. Nevertheless,

[6]https://github.com/lightning/bolts/blob/master/03-transactions.md#efficient-per-commitment-secret-storage

onion packets can be incorporated in the current construction. Intuitively our construction leaks less data than Lightning for the same multi-hop payments, as intermediaries in our case are not notified on each payment, contrary to multi-hop payments in 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 Elmo, a construction for the establishment and optimistic teardown of payment channels without posting transactions on-chain. Such a virtual channel can be opened over a path of base channels of any length, i.e., the constructor is *variadic*.

The base channels themselves can be virtual, therefore our construction is *recursive*. A key performance characteristic of our protocol is its optimal round complexity for on-chain channel closing: one transaction is required by any party to turn the virtual channel into a simple one and one more transaction is needed to close it.

We formally described the protocol in the UC setting, provided a suitable ideal functionality and finally proved the indistinguishability of the protocol and functionality, along with the balance security properties that ensure no loss of funds. This is achieved through the use of the `ANYPREVOUT` sighash flag, which is a feature that will likely be added in the next Bitcoin update.

References

[1] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. 2020. Generalized Bitcoin-Compatible Channels. *IACR Cryptol. ePrint Arch.* (2020), 476.

[2] Lukas Aumayr, Matteo Maffei, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. 2021. Bitcoin-Compatible Virtual Channels. In *2021 IEEE Symposium on Security and Privacy (SP)*. 901–918. https://doi.org/10.1109/SP40001.2021.00097

[3] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2023. Breaking and Fixing Virtual Channels: Domino Attack and Donner. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/breaking-and-fixing-virtual-channels-domino-attack-and-donner/

[4] Lukas Aumayr, Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. 2022. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 179–192. https://doi.org/10.1145/3548606.3559370

[5] Georgia Avarikioti, Eleftherios Kokoris Kogias, Roger Wattenhofer, and Dionysis Zindros. 2020. Brick: Asynchronous Payment Channels. arXiv:1905.11360 [cs.DC]

[6] Zeta Avarikioti, Yuheng Wang, and Yuyi Wang. 2025. Thunderdome: Timelock-Free Rationally-Secure Virtual Channels. arXiv:2501.14418 [cs.CR] https://arxiv.org/abs/2501.14418

[7] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. 2014. Enabling blockchain innovations with pegged sidechains.

[8] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. 2020. 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*. 1–30. https://doi.org/10.1007/978-3-030-64381-2_1

[9] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 913–930.

[10] Christian Badertscher, Julia Hesse, and Vassilis Zikas. 2021. On the (Ir)Replaceability of Global Setups, or How (Not) to Use a Global Ledger. In *Theory of Cryptography*, Kobbi Nissim and Brent Waters (Eds.). Springer International Publishing, Cham, 626–657.

[11] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a transaction ledger: A composable treatment. In *Annual International Cryptology Conference*. Springer, 324–356.

[12] Andrei Z. Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet L. Wiener. 2000. Graph structure in the Web. *Comput. Networks* 33, 1-6 (2000), 309–320. https://doi.org/10.1016/S1389-1286(00)00083-9

[13] Conrad Burchert, Christian Decker, and Roger Wattenhofer. 2018. Scalable funding of Bitcoin micropayment channel networks. In *The Royal Society*. https://doi.org/10.1098/rsos.180089

[14] Vitalik Buterin. [n. d.]. On-chain scaling to potentially 500 tx/sec through mass tx validation. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477.

[15] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Paper 2000/067. https://eprint.iacr.org/2000/067 https://eprint.iacr.org/2000/067.

[16] Ran Canetti. 2001. 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*. 136–145. https://doi.org/10.1109/SFCS.2001.959888

[17] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. [n. d.]. Hydra: Fast Isomorphic State Channels. Cryptology ePrint Archive, 2020/299.

[18] Manuel M. T. Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompson, Rebecca Valentine, and Philip Wadler. 2019. Functional blockchain contracts. https://iohk.io/en/research/library/papers/functional-blockchain-contracts/.

[19] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. 2016. On scaling decentralized blockchains. In *Financial Cryptography and Data Security*. Springer, 106–125.

[20] George Danezis and Ian Goldberg. 2009. Sphinx: A compact and provably secure mix format. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 269–282.

[21] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. [n. d.]. eltoo: A Simple Layer2 Protocol for Bitcoin.

[22] Christian Decker and Anthony Towns. [n. d.]. SIGHASH_ANYPREVOUT for Taproot Scripts.

[23] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*. Springer, 3–18.

[24] Mo Dong, Qingkai Liang, Xiaozhou Li, and Junda Liu. 2018. Celer Network: Bring Internet Scale to Every Blockchain. arXiv:1810.00037 [cs.NI]

[25] Rosie Dunford, Quanrong Su, and Ekraj Tamang. 2014. The Pareto Principle. *The Plymouth Student Scientist* 7 (2014), 140–148. https://doi.org/10.4135/9781412950596.n394

[26] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT*, Yuval Ishai and Vincent Rijmen (Eds.). 625–656. https://doi.org/10.1007/978-3-030-17653-2_21

[27] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2019. Perun: Virtual Payment Hubs over Cryptocurrencies. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 344–361. https://doi.org/10.1109/SP.2019.00020

[28] Stefan Dziembowski, Grzegorz Fabiański, Sebastian Faust, and Siavash Riahi. [n. d.]. Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma. Cryptology ePrint Archive, Report 2020/175.

[29] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Computer and Communications Security, CCS*. 949–966. https://doi.org/10.1145/3243734.3243856

[30] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *Conference on Computer and Communications Security, SIGSAC (CCS '19)*. 801–815. https://doi.org/10.1145/3319535.3345666

[31] Grzegorz Fabiański, Rafał Stefański, and Orfeas Stefanos Thyfronitis Litos. 2025. A Formally Verified Lightning Network. In *Financial Cryptography and Data Security*. Springer International Publishing, Cham.

[32] P. Gaži, A. Kiayias, and D. Zindros. 2019. Proof-of-Stake Sidechains. In *Symposium on Security and Privacy, SP*. 677–694. https://doi.org/10.1109/SP.2019.00040

[33] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 473–489. https://doi.org/10.1145/3133956.3134093

[34] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. 2020. SoK: Layer-Two Blockchain Protocols. In *Financial Cryptography and Data Security FC*. 201–226. https://doi.org/10.1007/978-3-030-51280-4_12

[35] Jona Harris and Aviv Zohar. 2020. Flood & Loot: A Systemic Attack on The Lightning Network. In *Conference on Advances in Financial Technologies (AFT)*.

202–213. https://doi.org/10.1145/3419614.3423248
[36] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. 2020. Lightweight Virtual Payment Channels. In *Cryptology and Network Security*. 365–384.
[37] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*. 477–498. https://doi.org/10.1007/978-3-642-36594-2_27
[38] Rami Khalil and Arthur Gervais. 2017. Revive: Rebalancing Off-Blockchain Payment Networks. In *Conference on Computer and Communications Security, CCS*. 439–453. https://doi.org/10.1145/3133956.3134033
[39] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. 2020. A Composable Security Treatment of the Lightning Network. In *Computer Security Foundations Symposium, CSF*. 334–349. https://doi.org/10.1109/CSF49147.2020.00031
[40] Aggelos Kiayias and Dionysis Zindros. 2019. Proof-of-Work Sidechains. In *Workshop on Trusted Smart Contracts*. 21–34. https://doi.org/10.1007/978-3-030-43725-1_3
[41] Georgios Konstantopoulos. 2019. Plasma Cash: Towards more efficient Plasma constructions. arXiv:1911.12095 [cs.CR]
[42] Junmo Lee, Seongjun Kim, Sanghyeon Park, and Soo-Mook Moon. 2020. RouTEE: A Secure Payment Network Routing Hub using Trusted Execution Environments. arXiv:2012.04254 [cs.CR]
[43] Jinghui Liao, Fengwei Zhang, Wenhai Sun, and Weisong Shi. 2022. Speedster: An Efficient Multi-party State Channel via Enclaves. In *Asia Conference on Computer and Communications Security, ASIACCS*. 637–651. https://doi.org/10.1145/3488932.3523259
[44] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. 2016. Teechan: Payment Channels Using Trusted Execution Environments. *CoRR* abs/1612.07766 (2016). arXiv:1612.07766
[45] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. 63–79. https://doi.org/10.1145/3341301.3359627
[46] Yehuda Lindell. 2017. How to Simulate It - A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*. 277–346. https://doi.org/10.1007/978-3-319-57048-8_6
[47] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2019. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *Network and Distributed System Security Symposium, NDSS*.
[48] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. 2019. Simple Schnorr multi-signatures with applications to Bitcoin. *Des. Codes Cryptogr.* 87, 9 (2019), 2139–2164. https://doi.org/10.1007/s10623-019-00608-x
[49] S Milgram. 1967. The small world problem. *Psychology Today* 1 (May 1967), 61–67.
[50] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. 2017. Sprites and State Channels: Payment Networks that Go Faster than Lightning. arXiv:1702.05812.
[51] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.
[52] Jonas Nick, Tim Ruffing, and Yannick Seurin. 2021. MuSig2: Simple Two-Round Schnorr Multi-signatures. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12825)*, Tal Malkin and Chris Peikert (Eds.). Springer, 189–221. https://doi.org/10.1007/978-3-030-84242-0_8
[53] Optimism. [n. d.]. Optimistic rollup overview. https://github.com/ethereum-optimism/optimistic-specs/blob/0e9673af0f2cafd89ac7d6c0e5d8bed7c67b74ca/overview.md.
[54] J. Poon and V. Buterin. [n. d.]. Plasma: Scalable Autonomous Smart Contracts.
[55] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf.
[56] David M. W. Powers. 1998. Applications and Explanations of Zipf's Law. In *New Methods in Language Processing and Computational Natural Language Learning*.
[57] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. 2016. Flare: An approach to routing in lightning network. (2016).
[58] Siavash Riahi and Orfeas Stefanos Thyfronitis Litos. 2024. Bitcoin Clique: Channel-Free Off-Chain Payments Using Two-Shot Adaptor Signatures. In *Applied Cryptography and Network Security*, Christina Pöpper and Lejla Batina (Eds.). Springer Nature Switzerland, Cham, 28–50.
[59] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia C. Fanti, and Pramod Viswanath. 2018. Routing Cryptocurrency with the Spider Network. *CoRR* abs/1809.05088 (2018). arXiv:1809.05088
[60] Jeremy Spilman. 2013. Anti dos for tx replacement. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html.
[61] Gavin Wood. [n. d.]. Ethereum: A secure decentralised generalised transaction ledger. ([n. d.]).
[62] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. 2019. SoK: Hardware Security Support for Trustworthy Execution. arXiv:1910.04957 [cs.CR]

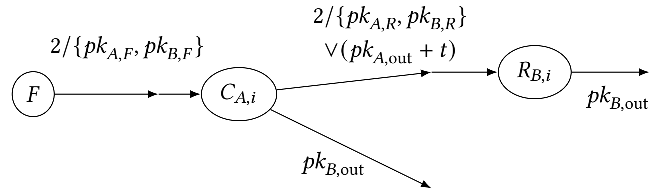

Figure 11: Funding, commitment, and revocation transactions. Inputs and outputs are represented by separate arrows: The input of $C_{A,i}$ spending the output of F is shown as two connected arrows.

A In-Depth Protocol Description

Let us first introduce some notation and concepts used, among others, in figures with transactions. Reflecting the UTXO model, each transaction is represented by a circular, 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 of 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 ($m \leq n$), i.e., a spending condition 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 make use of the `ANYPREVOUT` hash type.

A.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 – we refer the reader to Sec. 4 for an overview of the opening procedure. There are three sets of transactions at play: A *funding* transaction that is put on-chain, *commitment* transactions that are stored off-chain and spend the funding output on channel closure and off-chain *revocation* transactions that spend commitment outputs in case of misbehaviour (cf. Figure 11).

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 per 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. 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. In particular, Elmo does not use Hashed TimeLocked Contracts (HTLCs), which enable multi-hop payments in LN.

A.2 Virtual Channels

In order to gain intuition on how virtual channels work, we will first go in depth over the data each party stores locally while the channel is open. Consider $n-1$ simple channels between n honest parties as before. P_1, the funder, and P_n, the fundee, 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 (cf. Sec. 4), they will all hold off-chain a number of new, *virtual* transactions that spend their respective funding transactions. The *virtual* transactions can be spent by *bridge* transactions which in turn are spendable by new commitment transactions in a manner that ensures fair funds allocation for all honest parties. *Bridge* transactions take advantage of `ANYPREVOUT` to ensure that each of P_1, P_n only needs to maintain a single commitment transaction.

In particular, apart from the transactions of simple channels (i.e., commitment and revocation txs), 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 (cf. Figures 12, 61). If the initiator transaction ends up on-chain honestly, 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 functions exactly in the same manner as in a simple channel.

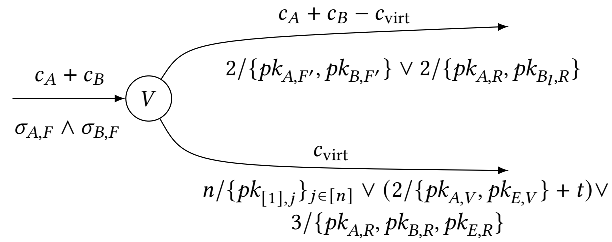

Figure 12: $A-E$ virtual channel: A's initiator transaction. Spends the funding output of the $A-B$ channel. Can be used if B has not published a virtual transaction yet.

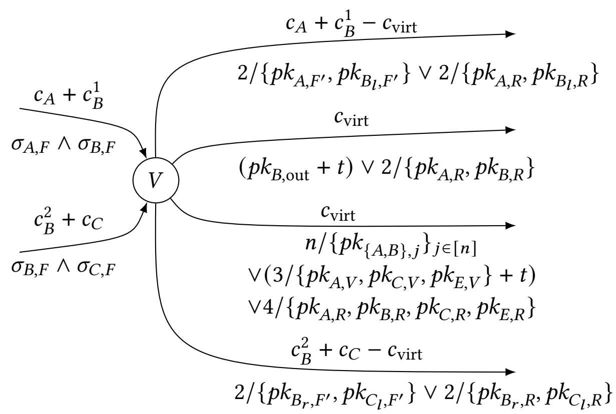

Figure 13: $A-E$ virtual channel: B's initiator transaction. Spends the funding outputs of the $A-B$ and $B-C$ channels. Can be used if neither A nor C have published a virtual transaction yet.

Intermediaries on the other hand store three sets of virtual transactions (Figure 60): *initiator* (Figure 13), *extend-interval* (Figure 14) and *merge-intervals* (Figure 15). 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*, with 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 honest intermediary to close either of its channels and retrieve its collateral.

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 funding 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 retrieve its collateral. Such a transaction consumes two outputs: the only available funding output and a suitable virtual output, as discussed below. An extend-interval tx

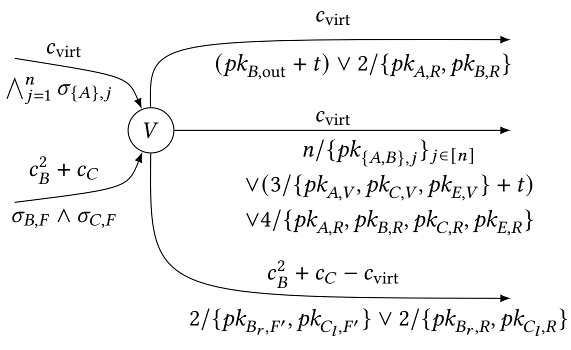

Figure 14: $A - E$ virtual channel: One of B's extend-interval transactions. σ is the signature. Spends the virtual output of A's initiator transaction and the funding output of the B-C channel. Can be used if A has already published its initiator transaction and C has not published a virtual transaction yet.

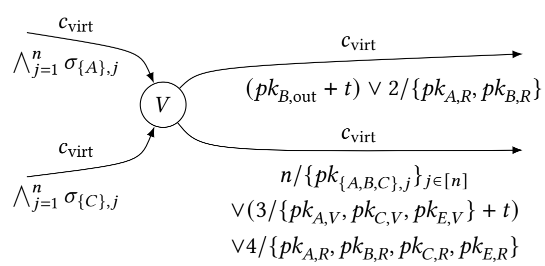

Figure 15: A–E virtual channel: One of B's merge intervals transactions. Spends the virtual outputs of A's and C's virtual transactions. Usable if both A and C have already published their initiator transactions.

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 retrieve its collateral. 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.

Note that each output of a virtual transaction has a *revocation* spending method which needs a signature from every party that could end up owning the output coins: each funding output is signed by the two parties of the corresponding channel, each refund output is signed by the transaction owner and the party to the left (giving c_{virt} coins to the left party if the owner acts maliciously), whereas each virtual output is signed by the transaction owner, the right party and the two virtual channel parties. If the owner acts maliciously, c_{virt} are given to the right party. The virtual channel parties have to sign as well since this output may end up funding their channel – lack of such signatures would allow two colluding intermediaries to claim the virtual output for themselves.

The revocation spending conditions take precedence over others because (i) they do not have a timelock and (ii) any other spending condition without a timelock (e.g., the n-of-n multisig of an initiator transaction) is transitively spendable by a transaction in which the only non-timelocked spending condition is the revocation.

Each virtual transaction is accompanied by a *bridge* transaction. Any virtual output may end up funding the virtual channel, but the various virtual outputs do not have the same script, thus there cannot be a single commitment transaction able to spend all of them. Without the bridge transaction, the parties of the virtual channel would have to keep track of $O(n^3)$ commitment transactions to be able to close their channel securely in every case, making channel updates expensive. This is fixed by the bridge transactions, which all have exactly the same output, unifying the interface between the virtualisation and the payment transactions and thus making virtual channel updates as cheap as simple channel updates.

To understand why this multitude of virtual transactions is needed, we now zoom out from the individual party and discuss the dynamic of unilateral closing 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 as collateral. 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 extension rule 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 to fund 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. Thanks to Schnorr signatures and the ability to aggregate them [48, 52] however, the on-chain footprint of the n signatures is reduced to that of a single signature. 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 A.1 of Subsec. A.3. 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 with an interval that includes all parties, which are just funding outputs for the virtual channel: an interval with all parties cannot be further extended, therefore one spending method and the timelock are dropped.

We here note that a typical extend-interval and merge-intervals transaction has 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. In order for the received signatures for virtual and commitment txs to be valid for multiple previous outputs, the previously proposed `ANYPREVOUT` sighash flag [22] is needed to be added to Bitcoin. We conjecture that, if this flag is not available, then it is impossible to build variadic recursive virtual channels for which each party only needs to (i) publish $O(1)$ on-chain transactions to open or close a channel and (ii) store a subexponential (in the number of intermediaries, payments and recursion layers) number of $O(1)$-sized transactions off-chain.[7] 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 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.

Let a single *channel round* be a series of messages starting from the funder and hop by hop reaching the fundee and back again. For the actual protocol that establishes a virtual channel 6 channel rounds are needed (cf. Figure 36). The first communicates parties' identities, their funding keys, revocation 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.

Cooperative closing is quite intuitive (cf. Figures 53, 54, 55, 56 and 72). It can be initiated by any party, one and a half communication rounds are needed. The funder builds new commitment txs, which once again spend the funding outputs that the virtual txs spent before, just like prior to opening the virtual channel. In particular, these new txs make the base channels independent once more. The funder sends its signature on the new commitment tx the to the first intermediary; the latter similarly builds, signs and sends a new commitment tx signature to the second intermediary and so on until the fundee. The fundee responds with its own commitment tx signatures, along with signatures revoking the previous commitment tx and virtual txs. This is repeated backwards until revocations reach the funder. Finally the funder sends its revocation to its neighbour and it to the next, until the revocations reach the fundee. The channel has now closed cooperatively.

At a high level, this procedure works without risk for the same reasons that a channel update does: Each party signs a new commitment transaction that guarantees it the same amount of funds as the last state before cooperatively closing did. It then revokes the state it had before closing only after receiving signatures for all relevant new commitment transactions. Furthermore, it only considers the closing complete if it receives revocations for all states before closing. If anything goes wrong in the process, the party can always unilaterally close, either in the last state before closing, or using the new commitment txs.

As for the unilateral closing, let us now turn to an example in order to better grasp how our construction plays out on-chain in practice (Figure 16). 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 unilaterally 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. No more virtual transactions can be published. Now A can spend the virtual output of the last extend-interval transaction with the relevant bridge transaction, which can then be spent by A's or E's latest commitment transaction. 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 the relevant bridge transaction and this with the latest commitment transaction, thus eventually A can close its virtual channel in all cases.

Remark. In order to support a virtual channel, base parties have to lock collateral for a potentially long time. A fee structure that takes this opportunity cost into consideration would bolster participation. A straightforward mechanism is for parties to agree when opening the virtual channel on a time-based fee schedule and periodically update their base channels to reflect contingent payments by the endpoints. In case of lack of cooperation for an update, a party can simply close its base channel. The details of such a scheme are outside the scope of this work.

[7] To see why, consider a virtual channel over $k + 1$ players who close the channel non-cooperatively via on-chain interaction. Assuming the $(k + 1)$-th party goes last, the protocol should be able to accommodate any possible activation sequence for the first k parties. Consecutive pairs of parties $(i, i + 1)$ need to be reactive to each other's posted transactions since they share a base channel. It follows that for each i we can assign either "L" or "R" signifying the directionality of reaction, resulting in a total of 2^{k-1} different sequences. Without `ANYPREVOUT`, the $(k + 1)$-th party needs a different transaction to interact with the outcome of each sequence, hence blowing up its local storage. The formalization of this argument is outside the scope of the present work.

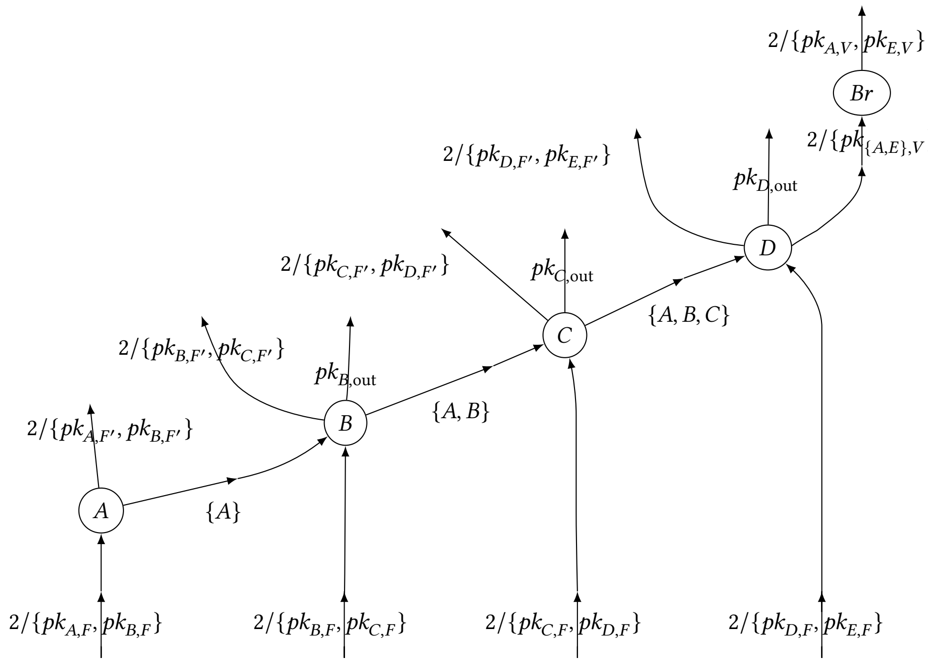

Figure 16: 4 simple channels supporting a virtual. A starts closing by publishing its initiator tx, then parties B–D each publishes its extend-interval tx with the relevant interval. No party is negligent. Virtual outputs are marked with their interval.

A.3 Liveness

PROPOSITION A.1. *Consider a synchronised honest party that submits a transaction* `tx` *to the ledger functionality [9] 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 $q = \lceil(\texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2})/\texttt{minTime}_{\texttt{window}}\rceil$.*

PROOF. Consider τ_h^U to be the round that a party U becomes aware of the h-th block in the `state`. It follows that $\tau_h \leq \tau_h^U$ where τ_h is the round block h enters `state`. Note that by time $\tau_h + \texttt{maxTime}_{\texttt{window}}$ another `windowSize` blocks are added to `state` and thus $\tau_h^U \leq \tau_h + \texttt{maxTime}_{\texttt{window}}$.

Suppose U submits the transaction `tx` to the ledger at time τ_h^U. Observe that as long as $\tau_h + \texttt{maxTime}_{\texttt{window}}$ is `Delay/2` before the time that block with index $h + t - 2\texttt{windowSize}$ enters `state`, then `tx` is guaranteed to enter the `state` in a block with index up to $h + t$ where since $\texttt{advBlcks}_{\texttt{window}} < \texttt{windowSize}$. It follows we need $\tau_h + \texttt{maxTime}_{\texttt{window}} < \tau_{h+t-2\texttt{windowSize}} - \frac{\texttt{Delay}}{2}$. Let $r = \lceil(\texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2})/\texttt{minTime}_{\texttt{window}}\rceil$. Recall that in a period of $\texttt{minTime}_{\texttt{window}}$ rounds at most `windowSize` blocks enter `state`. As a result $r \cdot \texttt{windowSize}$ blocks require at least $r \cdot \texttt{minTime}_{\texttt{window}} \geq \texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2}$ rounds. We deduce that if $t \geq (2+r)\texttt{windowSize}$ the inequality follows. □

B Universal Composition Framework

In this work we embrace the Universal Composition (UC) framework [16] 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 [46]. 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 $\mathcal{E}$.

C Further Related Work

Solutions alternative to PCNs include sidechains (e.g., [7, 32, 40]), commit-chains (e.g., [54, 58]) and non-custodial chains (e.g., [28, 41, 54]). These approaches offer more efficient payment methods, at the cost of requiring a distinguished mediator, additional tust, or on-chain checkpointing. Furthermore, they do not enable payments between different instances of the same protocol. Due to their conceptual and interface differences as well as differing levels of software maturity, dedicated user studies need to be carried out in order to compare the usability and overall costs of each approach under various usage patterns. Rollups [14, 53] are incompatible with Bitcoin, as they only optimise computation, not storage, whereas Bitcoin has by design minimal computation needs.

Various attacks have been identified against LN. The wormhole attack [47] against LN allows colluding parties in a multi-hop payment to steal the fees of the intermediaries between them and Flood & Loot attacks [35] analyses an attack in which too many channels are forced to close in a short amount of time, harming blockchain liveness and enabling a malicious party to steal off-chain funds.

To the best of our knowledge, no formal treatment of the privacy of LN exists. Nevertheless, it intuitively improves upon the privacy of on-chain Bitcoin transactions, as LN payments do not leave a permanent record: only intermediaries of each payment are informed. It can be argued that Elmo further improves privacy, as payments are hidden from the intermediaries of a virtual channel.

Payment routing [42, 57, 59] is another research area that aims to improve network efficiency without sacrificing privacy. Actively rebalancing channels [38] can further increase network efficiency by reducing unavailable routes due to lack of well-balanced funds.

Bolt [33] constructs privacy-preserving payment channels enabling both direct payments and payments with a single untrusted intermediary. Sprites [50] leverages the scripting language of Ethereum to decrease the time collateral is locked compared to LN.

State channels are a generalisation of payment channels, which enable off-chain execution of any smart contract supported by the underlying blockchain, not just payments. Generalized Bitcoin-Compatible Channels [1] enable the creation of state channels on Bitcoin, extending channel functionality from simple payments to arbitrary Bitcoin scripts. Since Elmo only pertains to payment, not state, channels, we choose not to build it on top of [1]. State channels can also be extended to more than two parties [26, 43].

BDW [13] 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. Such proposals are complementary to virtual channels and, depending on the use case, could be more efficient. In comparison to Elmo, BDW is less flexible: coins in a BDW pool can only be exchanged with members of that pool. ACMU [30] allows for multi-path atomic payments with reduced collateral, enabling new applications such as crowdfunding conditional on reaching a funding target.

TEE-based [62] solutions [42–45] improve the throughput and efficiency of PCNs by an order of magnitude or more, at the cost of having to trust TEEs. Brick [5] uses a partially trusted committee to extend PCNs to fully asynchronous networks.

Donner [3] is technically insecure since any state update to a base channel invalidates the corresponding tx^r. There is a straightforward fix, which however adds an overhead to each payment over a base channel: On every payment, the two base channel parties must update their tx^r to spend the α output of the new state. Potential intermediaries must consider this overhead and possibly increase the fees they require from the endpoints. This per-payment overhead can be avoided by using `ANYPREVOUT` in the α output.

D Functionalities & Simulator

D.1 $\mathcal{G}_{\text{Chan}}$ functionality

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. 18, 19, 20, 21 or 22 (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` $\in$ {output, network}, handle it according to the corresponding rule in Fig. 18, 19, 20, 21 or 22 (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` $\in$ {input, output, 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. 18, 19, 20, 21 or 22 (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

Figure 17

Note that in UCGS [8], 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

$P \in \{Alice, Bob\}$

```
On first activation: // before handing the message
    pk_P ← ⊥; balance_P ← 0; State_P ← UNINIT
    enabler_P ← ⊥ // if we are a virtual channel, the ITI of P's base channel
    host_P ← ⊥ // if we are a virtual channel, the ITI of the common host of this channel and P's base channel

On (BECAME CORRUPTED OR NEGLIGENT, P) by A or on output (ENABLER USED REVOCATION) by host_P when in any state:
    State_P ← IGNORED

On (INIT, pk) by P when State_P = UNINIT:
    pk_P ← pk
    State_P ← INIT

On (OPEN, x, "ledger", ...) by Alice when State_A = INIT:
    store x
    State_A ← TENTATIVE BASE OPEN

On (BASE OPEN) by A when State_A = TENTATIVE BASE OPEN:
    balance_A ← x
    layer_A ← 0
    State_A ← OPEN

On (BASE OPEN) by A when State_B = INIT:
    layer_B ← 0
    State_B ← OPEN

On (OPEN, x, hops ≠ "ledger", ...) by Alice when State_A = INIT:
    store x
    enabler_A ← hops[0].left
    add enabler_A to Alice's kindred parties
    State_A ← PENDING VIRTUAL OPEN

On output (FUNDED, host, ...) to Alice by enabler_A when State_A = PENDING VIRTUAL OPEN:
    host_A ← host[0].left
    State_A ← TENTATIVE VIRTUAL OPEN

On output (FUNDED, host, ...) to Bob by ITI R ∈ {G_Chan, LN} when State_B = INIT:
    enabler_B ← R
    add enabler_B to Bob's kindred parties
    host_B ← host
    State_B ← TENTATIVE VIRTUAL OPEN

On (VIRTUAL OPEN) by A when State_P = TENTATIVE VIRTUAL OPEN:
    if P = Alice then balance_P ← x
    layer_P ← 0
    State_P ← OPEN
```

Figure 18: State machine in Fig. 23, 24, 25 and 30

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: $\texttt{balance}_P \leftarrow \texttt{balance}_P - x$
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: $\texttt{balance}_P \leftarrow \texttt{balance}_P + x$
20: // if $\bar{P}$ honest, this state transition happens simultaneously with l. 15
21: $State_P \leftarrow$ OPEN
22: **end if**

Figure 19: State machine in Fig. 26

Functionality $\mathcal{G}_{\text{Chan}}$ – funding state machine

$P \in \{Alice, Bob\}$

1: On input (FUND ME, $x, \ldots$) by ITI $R \in \{\mathcal{G}_{\text{Chan}}$, 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:
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 `msg` = (INIT, ..., `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** $State_{\bar{P}} \in$ {IGNORED, SYNC HELP FUND} **then**
27: $\texttt{balance}_P \leftarrow \texttt{balance}_P - x$
28: $\texttt{host}_P \leftarrow \texttt{host}$
29: // if $\bar{P}$ honest, this state transition happens simultaneously with l. 38
30: $\texttt{layer}_P \leftarrow \texttt{layer}_P + 1$
31: $State_P \leftarrow$ OPEN
32: **end if**

33: When $State_P$ = SYNC HELP FUND:
34: **if** $State_{\bar{P}} \in$ {IGNORED, SYNC FUND} **then**
35: $\texttt{host}_P \leftarrow \texttt{host}$
36: // if $\bar{P}$ honest, this state transition happens simultaneously with l. 31
37: $\texttt{layer}_P \leftarrow \texttt{layer}_P + 1$
38: $State_P \leftarrow$ OPEN
39: **end if**

Figure 20: State machine in Fig. 27

Functionality $\mathcal{G}_{\text{Chan}}$ – force close state machine

$P \in \{Alice, Bob\}$

1: On (FORCECLOSE) by P when $State_P$ = OPEN:
2: $State_P \leftarrow$ 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, $\texttt{balance}_P$, pk_P, $\texttt{balance}_{\bar{P}}$, $pk_{\bar{P}}$)
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

coins ⟵ sum of values of outputs exclusively spendable or spent by pk_P in Σ
balance ⟵ balance_P
for all P's kindred parties R **do**
    input (BALANCE) to R as P and extract balance_R, pk_R from response
    balance ⟵ balance + balance_R
    coins ⟵ coins + sum of values of outputs exclusively spendable or spent by pk_R in Σ
end for
if coins $\geq$ balance **then**
    $State_P$ ⟵ CLOSED
else // balance security is broken
    halt
end if

Figure 21

Functionality $\mathcal{G}_{\text{Chan}}$ – cooperative close state machine

$P \in \{Alice, Bob\}$

On (COOP CLOSING, P, x) by $\mathcal{A}$ when $State_P$ = OPEN:
    store x
    $State_P$ ⟵ COOP CLOSING

On (COOP CLOSED, P) by $\mathcal{A}$ when $State_P$ = COOP CLOSING:
    if $\text{layer}_P = 0$ **then** // P's channel, which is virtual, is cooperatively closed
        $State_P$ ⟵ COOP CLOSED
    else // the virtual channel for which P's channel is base is cooperatively closed
        $\text{layer}_P \longleftarrow \text{layer}_P - 1$
        $\text{balance}_P \longleftarrow \text{balance}_P + x$
        $State_P$ ⟵ OPEN
    end if

Figure 22

D.2 Simulator $\mathcal{S}$ for $\mathcal{G}$

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

- On receiving (RELAY, in_msg, P, R, in_mode) by $\mathcal{G}_{\text{Chan}}$ (in_mode ∈ {input, output, network}, $P \in \{Alice, Bob\}$), handle (in_msg) with the simulated party P as if it was received from R by means of in_mode. In case simulated P does not exist yet, initialise it as an LN ITI. If there is a resulting message out_msg that is to be sent by simulated P to R' by means of out_mode ∈ {input, output, network}, send (RELAY, out_msg, P, R', 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 (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.

Figure 31

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.

On (CORRUPT) by $\mathcal{A}$, addresed to P:
    // After executing this code and getting control back from $\mathcal{G}_{\text{Chan}}$ (which always happens, cf. Fig. 17), deliver (CORRUPT) to simulated P (cf. Fig. 31).
    send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$

When simulated P sets variable negligent to True (Fig. 33, l. 7/Fig. 34, l. 26):
    send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$

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

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

When the last of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties moves to the OPEN $State$ for the first time (Fig. 37, l. 19/Fig. 39, l. 16/Fig. 40, l. 18):
    if hops = "ledger" **then**
        send (INFO, BASE OPEN) to $\mathcal{G}_{\text{Chan}}$
    else
        send (INFO, VIRTUAL OPEN) to $\mathcal{G}_{\text{Chan}}$
    end if

When (both $\mathcal{G}_{\text{Chan}}$'s simulated parties are honest and complete sending and receiving a payment (Fig. 45, 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. 45, l. 6 is executed by one party, but its counterparty is corrupted before executing Fig. 45, l. 21
    send (INFO, PAY) to $\mathcal{G}_{\text{Chan}}$

When honest P executes Fig. 42, l. 21 or (when honest P executes Fig. 42, l. 19 and $\bar{P}$ is corrupted): // in the first case if $\bar{P}$ is honest, it has already moved to the new host, (Fig 68, ll. 7, 23): lifting to next layer is done

send (INFO, FUND) to $\mathcal{G}_{\text{Chan}}$

When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P moves to the COOP CLOSING state (Fig. 55, l. 4, Fig. 56, ll. 6, 12, Fig. 72, ll. 11, 24):

if triggered by Fig. 55, l. 4 or Fig. 56, l. 6 **then** // P is funder or fundee

send (INFO, COOP CLOSING, P, $-c_P$) to $\mathcal{G}_{\text{Chan}}$ // coin value extracted from simulated P

else if triggered by Fig. 56, l. 12 **then** // P is funder's base

send (INFO, COOP CLOSING, P, c'_1) to $\mathcal{G}_{\text{Chan}}$

else if triggered by Fig. 72, l. 11 **then** // P is an intermediary farther from funder than $\bar{P}$

send (INFO, COOP CLOSING, P, c'_2) to $\mathcal{G}_{\text{Chan}}$

else if triggered by Fig. 72, l. 24 **then** // P is an intermediary closer to funder than $\bar{P}$

send (INFO, COOP CLOSING, P, $c'_1 - c_{\text{virt}}$) to $\mathcal{G}_{\text{Chan}}$

end if

When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P completes cooperative closing (Fig. 56, l. 45, Fig. 72, l. 187, Fig. 72, l. 150, Fig. 72, or l. 134):

send (INFO, COOP CLOSED, P) to $\mathcal{G}_{\text{Chan}}$

When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P moves to the CLOSED state (Fig. 49, l. 8 or l. 11):

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

Figure 32

D.3 The Ledger and Clock Functionalities

We next provide the complete description of the ledger and the clock functionalities that are drawn from the UC formalisation of [9, 11].

The key characteristics of the functionality are as follows. The variable `state` maintains the current immutable state of the ledger. An honest, synchronised party considers finalised a prefix of `state` (specified by a pointer position $\mathtt{pt}_i$ for party U_i below). The functionality has a parameter `windowSize` such that no finalised prefix of any player will be shorter than $|\mathtt{state}| - \mathtt{windowSize}$. On any input originating from an honest party the functionality will run the ExtendPolicy function that ensures that a suitable sequence of transactions will be "blockified" and added to `state`. Honest parties may also find themselves in a desynchronised state: this is when honest parties lose access to some of their resources. The resources that are necessary for proper ledger maintenance and that the functionality keeps track of are the global random oracle $\mathcal{G}_{\text{RO}}$ and the clock $\mathcal{G}_{\text{CLOCK}}$. If an honest party maintains registration with all the resources then after `Delay` clock ticks it necessarily becomes synchronised.

The progress of the `state` variable is guaranteed via the ExtendPolicy function that is executed when honest parties submit inputs to the functionality. While we do not specify ExtendPolicy in our paper (we refer to the citations above for the full specification) it is sufficient to note that ExtendPolicy guarantees the following properties:

(1) in a period of time equal to $\mathtt{maxTime}_{\mathtt{window}}$, a number of blocks at least `windowSize` are added to `state`.
(2) in a period of time equal to $\mathtt{minTime}_{\mathtt{window}}$, no more blocks may be added to `state` if `windowSize` blocks have been already added.
(3) each window of `windowSize` blocks has at most $\mathtt{advBlcks}_{\mathtt{window}}$ adversarial blocks included in it.
(4) any transaction that (i) is submitted by an honest party earlier than $\frac{\mathtt{Delay}}{2}$ rounds before the time that the block that is `windowSize` positions before the head of the `state` was included, and (ii) is valid with respect to an honest block that extends `state`, then it must be included in such block.

Given a synchronised honest party, we say that a transaction `tx` is finalised when it becomes a part of `state` in its view.

Functionality $\mathcal{G}_{\text{LEDGER}}$

General: The functionality is parameterized by four algorithms, Validate, ExtendPolicy, Blockify, and predict-time, along with three parameters: $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$, and $\mathcal{S}_{\text{initStake}} := \{(U_1, s_1), \ldots, (U_n, s_n)\}$. The functionality manages variables `state` (the immutable state of the ledger), `NxtBC` (a list of transaction identifiers to be added to the ledger), `buffer` (the set of pending transactions), τ_L (the rules under which the state is extended), and $\vec{\tau}_{\mathtt{state}}$ (the time sequence where all immutable blocks where added). The variables are initialized as follows: $\mathtt{state} := \vec{\tau}_{\mathtt{state}} := \mathtt{NxtBC} := \varepsilon$, $\mathtt{buffer} := \emptyset$, $\tau_L = 0$. For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer $\mathtt{pt}_i$ (initially set to 1) and a current-state view $\mathtt{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\vec{I}_H^T$ (initially $\vec{I}_H^T := \varepsilon$).

Party Management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (as discussed below). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a (currently unregistered) honest party is registered at the ledger, *if it is registered with the clock and the global RO already,* then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$.

Handling initial stakeholders: If during round $\tau = 0$, the ledger did not received a registration from each initial stakeholder, i.e., $U_p \in \mathcal{S}_{\text{initStake}}$, the functionality halts.

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and upon receiving response $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ set $\tau_L := \tau$ and do the following if $\tau > 0$ (otherwise, ignore input):

(1) Updating synchronized/desynchronized party set:
  (a) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) to the ledger, the clock, and the GRO since time $\tau' < \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
  (b) For any synchronized party $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if U_p is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{U_p\}$.

(2) If I was received from an honest party $U_p \in \mathcal{P}$:

(a) Set $\vec{I}_H^T := \vec{I}_H^T || (I, U_p, \tau_L)$;

(b) Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) :=$ $\mathsf{ExtendPolicy}(\vec{I}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \vec{\tau}_{\mathtt{state}})$ and if $\vec{N} \neq \varepsilon$ set $\mathtt{state} := \mathtt{state}||\mathsf{Blockify}(\vec{N}_1)|| \ldots ||\mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathtt{state}} := \vec{\tau}_{\mathtt{state}}||\tau_L^\ell$, where $\tau_L^\ell = \tau_L|| \ldots, ||\tau_L$.

(c) For each $\mathtt{BTX} \in \mathtt{buffer}$: if $\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 0$ then delete $\mathtt{BTX}$ from $\mathtt{buffer}$. Also, reset $\mathtt{NxtBC} := \varepsilon$.

(d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathtt{state}| - \mathtt{pt}_j > \mathtt{windowSize}$ or $\mathtt{pt}_j < |\mathtt{state}_j|$, then set $\mathtt{pt}_k := |\mathtt{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

(3) If the calling party U_p is *stalled or time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:

– *Submitting a transaction:*
If $I = (\text{SUBMIT}, \text{sid}, \mathtt{tx})$ and is received from a party $U_p \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party U_p) do the following

(a) Choose a unique transaction ID txid and set $\mathtt{BTX} := (\mathtt{tx}, \text{txid}, \tau_L, U_p)$

(b) If $\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 1$, then $\mathtt{buffer} := \mathtt{buffer} \cup \{\mathtt{BTX}\}$.

(c) Send $(\text{SUBMIT}, \mathtt{BTX})$ to $\mathcal{A}$.

– *Reading the state:*
If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\mathtt{state}_p := \mathtt{state}|_{\min\{\mathtt{pt}_p, |\mathtt{state}|\}}$ and return $(\text{READ}, \text{sid}, \mathtt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\mathtt{state}, \mathtt{buffer}, \vec{I}_H^T)$ to $\mathcal{A}$.

– *Maintaining the ledger state:*
If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\vec{I}_H^T$ as above) $\text{predict-time}(\vec{I}_H^T) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

– *The adversary proposing the next block:*
If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \ldots, \text{txid}_\ell))$ is sent from the adversary, update $\mathtt{NxtBC}$ as follows:

(a) Set listOfTxid $\leftarrow \epsilon$

(b) For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \mathtt{buffer}$ with ID $\text{txid} = \text{txid}_i$ then set listOfTxid := listOfTxid$||\text{txid}_i$.

(c) Finally, set $\mathtt{NxtBC} := \mathtt{NxtBC}||(\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.

– *The adversary setting state-slackness:*
If $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\mathtt{pt}}_{i_1}), \ldots, (U_{i_\ell}, \widehat{\mathtt{pt}}_{i_\ell}))$, with $\{U_{p_{i_1}}, \ldots, U_{p_{i_\ell}}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:

(a) If for all $j \in [\ell] : |\mathtt{state}| - \widehat{\mathtt{pt}}_{i_j} \leq \mathtt{windowSize}$ and $\widehat{\mathtt{pt}}_{i_j} \geq |\mathtt{state}_{i_j}|$, set $\mathtt{pt}_{i_1} := \widehat{\mathtt{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$.

(b) Otherwise set $\mathtt{pt}_{i_j} := |\mathtt{state}|$ for all $j \in [\ell]$.

– *The adversary setting the state for desychronized parties:*
If $I = (\text{DESYNC-STATE}, (U_{i_1}, \mathtt{state}'_{i_1}), \ldots, (U_{i_\ell}, \mathtt{state}'_{i_\ell}))$, with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathtt{state}_{i_j} := \mathtt{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.

Functionality Functionality $\mathcal{G}_{\text{CLOCK}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $U_p = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each session sid the clock maintains a variable τ_{sid}. For each identity $U_p := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable d_{U_p}. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})}$ (all integer variables are initially 0).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $U_p \in \mathcal{P}$ set $d_{U_p} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return $(\text{CLOCK-READ}, \text{sid}, \tau_{\text{sid}})$ to the requestor (where sid is the sid of the calling instance).

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\text{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_{U_p} = 1$ for all honest parties $U_p = (\cdot, \text{sid}) \in \mathcal{P}$, then set $\tau_{\text{sid}} := \tau_{\text{sid}} + 1$ and reset $d_{(\mathcal{F},\text{sid})} := 0$ and $d_{U_p} := 0$ for all parties $U_p = (\cdot, \text{sid}) \in \mathcal{P}$.

E Model & Construction

E.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 [8] to formulate the protocol and its security. We list the ideal-world global functionality $\mathcal{G}_{\text{Chan}}$ in Appx. D (Figures 17-21) and a simulator $\mathcal{S}$ (Figures 31-32), along with a real-world protocol Π_{Chan} (Figures 33-73) that UC-realises $\mathcal{G}_{\text{Chan}}$ (Theorem G.5). We give a self-contained description in this section, while pointing to figures in Appx. D and F, in case the reader is interested in a pseudocode style specification.

As in previous formulations, (e.g., [39]), 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 (cf. Lemma 5.1). 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.

E.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 (cf. Figures 23, 24, 25, 26, 27, 28, 30) 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 $\mathcal{G}_{\text{Chan}}$, cf. Theorems G.4 and G.5) 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 often participates 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 [8].

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 [8] to circumvent the issue. More specifically, we say that a simple channel functionality is of "level" 1, which is written as $\mathcal{G}^1_{\text{Chan}}$. Inductively, a virtual channel functionality that is based on channels of any "level" up to and including $n-1$ (but no further) has a "level" n, which we write as $\mathcal{G}^n_{\text{Chan}}$. Then $\mathcal{G}^n_{\text{Chan}}$ is $(\mathcal{G}_{\text{Ledger}}, \mathcal{G}^1_{\text{Chan}}, \dots, \mathcal{G}^{n-1}_{\text{Chan}})$-subroutine respecting, according to the definition of [8]. 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.

We could instead contain all the channels in a single, monolithic functionality (following the approach of [39]) 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 [8]. We note that the ideal functionality defined in [1] 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.

E.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 33-52) and the novel virtual layer subprotocol, named VIRT (Figures 58-73). 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 33).

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 the VIRT subprotocol description) in case it is a virtual channel, the identities of the base channel owners (Figure 40), *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 (only for virtual channels). 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 35).

The next step prepares the base channels (Figure 36) 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 42 and 43). 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 37). After that, in case the channel is simple the funding transaction is put on-chain (Figure 38) 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 47) and subsequently the payer is instructed by $\mathcal{E}$ to commence the payment (Figure 46).

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 [21].

Subsequently each of the two parties builds the new commitment transaction of its counterparty and signs it. It also generates a new revocation keypair for the next update and sends over the generated signature and public key. 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 and the public key 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 49). It also keeps track of whether the party is activated often enough and marks it as negligent otherwise (Figure 33). 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.

Cooperative closing involves both LN (Figures 53-56) and VIRT (Figure 72) subprotocols. Any party can initiate it by asking the virtual channel fundee. The latter signs the last coin balance and sends it to the funder, who first ensures the fundee signed the correct balance, then signs it as well. Its enabler (i.e., the kindred party that is a member of the 1st base channel) generates and signs a new commitment tx in which it adds the funder's coins to its own and the fundee's coins to its counterparty's, while using the funding keys that were used before opening the virtual channel. It also generates a new revocation keypair for the next channel update and sends the revocation public key with the signature and the final virtual channel balance to its counterparty. The latter verifies the signature and that the two virtual channel parties agree on their final balance. If all goes well, it passes control to its kindred party that is a member of the next channel in sequence. If no verification fails, the process repeats until the fundee is reached. Now a backwards sequence of messages begins, in which each party that previously did verification now provides a signature for the new commitment tx, along with a revocation signature for the old commitment tx and a new revocation public key for the next update. Each receiver verifies the signatures and "passes the baton" to its kindred party closer to the funder. When the funder is reached, the last series of messages begins. Now each party that has not yet sent a revocation does so. Once the chain of messages reaches the fundee, the channel has successfully closed cooperatively. In total, each LN party sends and stores 2 signatures, 1 private key and 1 public key. The associated behaviour of the VIRT subprotocol is discussed later.

Alternatively, when either party is instructed by $\mathcal{E}$ to unilaterally close the channel (Figure 51), 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 49).

VIRT subprotocol. This subprotocol acts as a mediator between the base channels and the Lightning-based logic. Put simply, its main 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 58), 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 59) 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 66) every party receives and verifies all signatures for all inputs of its virtual and bridge transactions that spend a virtual output. It also produces and sends its own such signatures to the other parties. Each party generates and circulates $S = 2(n-2) + (i-3)(n-i) + (i-1)(n-i-2) + \chi_{i=3}(2(n-i)-1) + \chi_{i=n-2}(2i-3) + 3 + \sum_{i=2}^{n-2}(n-3+\chi_{i=2}+\chi_{i=n-1}+2(i-2+\chi_{i=2})(n-i-1+\chi_{i=n-1})) \in O(n^3)$ 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 bridge transactions and virtual outputs of all parties' virtual transactions – we remind that each virtual output can be spent either by a n-of-n multisig via a new virtual transaction, or by a 4-of-4 multisig via its bridge transaction. On a related note, the total number of virtual and bridge transactions for which each party needs to store signatures is 2 for the two endpoints (Figure 61) and $2(n-2+\chi_{i=2}+\chi_{i=n-1}+(i-2+\chi_{i=2})(n-i-1+\chi_{i=n-1})) \in O(n^2)$ for the i-th intermediary (Figure 60). 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 67). 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 68) 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.

In case of a cooperative closing, VIRT orchestrates the hosted LN ITIs, instructing them to perform the actions discussed previously. It also is responsible for sending the actual messages to the host of the next counterparty and receiving its responses. Apart from controlling the flow of messages, a VIRT ITI also generates revocation signatures to invalidate its virtual and bridge transactions and verifies the respective revocation signatures generated by its counterparty VIRT ITI, thereby ensuring that, moving forward, the use of an old virtual or bridge transaction can be punished.

On the other hand, when VIRT is instructed to unilaterally close by party R (Figure 70), 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 71). 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 wins the race to the chain, then our VIRT party has to try again with another, compatible virtual transaction.

F Protocol

Process LN – init

// When not specified, input comes from and output goes to $\mathcal{E}$.
// The ITI knows whether it is *Alice* (funder) or *Bob* (fundee). The activated party is P and the counterparty is $\bar{P}$.
On every activation, before handling the message:
    if `last_poll` $\neq \bot \land$ *State* $\neq$ CLOSED **then** // channel is open
        input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
        if `last_poll` $+ p < |\Sigma|$ **then** // p is a global parameter
            `negligent` $\leftarrow$ True
        end if
    end if
    if *State* = WAITING FOR NOTHING REVOKED $\land$ activation is not caused by output (NOTHING REVOKED), received by a member of the list of old hosts **then** // the only way for this case to be true is if the old host punished a misbehaving counterparty
        State $\leftarrow$ BASE PUNISHED
    end if

On (INIT, $pk_{P,\text{out}}$):
    ensure *State* = $\bot$
    State $\leftarrow$ INIT
    `hosting` $\leftarrow$ False
    store $pk_{P,\text{out}}$
    $(c_A, c_B, \texttt{locked}_A, \texttt{locked}_B) \leftarrow (0, 0, 0, 0)$
    $(\texttt{paid\_out}, \texttt{paid\_in}) \leftarrow (\emptyset, \emptyset)$
    `negligent` $\leftarrow$ False
    `last_poll` $\leftarrow \bot$
    output (INIT OK)

On (TOP UP):
    ensure P = *Alice* // activated party is the funder
    ensure *State* = INIT
    $(sk_{P,\text{chain}}, pk_{P,\text{chain}}) \leftarrow$ KEYGEN()
    input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
    output (TOP UP TO, $pk_{P,\text{chain}}$)
    while $\neg\exists \text{tx} \in \Sigma, c_{P,\text{chain}} : (c_{P,\text{chain}}, pk_{P,\text{chain}}) \in \text{tx.outputs}$ **do**
        // while waiting, all other messages by P are ignored

wait for input (CHECK TOP UP)
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
end while
State ← TOPPED UP
output (TOP UP OK, $c_{P,\text{chain}}$)

On (BALANCE):
ensure *State* ∈ {OPEN, CLOSED}
output (BALANCE, c_A, $pk_{A,\text{out}}$, c_B, $pk_{B,\text{out}}$, $\texttt{locked}_A$, $\texttt{locked}_B$)

Figure 33

Process LN – methods used by VIRT

REVOKEPREVIOUS():
ensure *State* ∈ WAITING FOR (OUTBOUND) REVOCATION
$R_{\bar{P},i}$ ← TX {input: $C_{P,i}$.outputs.P, output: ($C_{P,i}$.outputs.P.value, $pk_{\bar{P},\text{out}}$)}
$\text{sig}_{A,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$
if *State* = WAITING FOR REVOCATION **then**
State ← WAITING FOR INBOUND REVOCATION
else // *State* = WAITING FOR OUTBOUND REVOCATION
$i \leftarrow i + 1$
State ← WAITING FOR HOSTS READY
end if
$\texttt{host}_P \leftarrow \texttt{host}'_P$ // forget old host, use new host instead
$\texttt{layer} \leftarrow \texttt{layer} + 1$
return $\text{sig}_{P,R,i}$

PROCESSREMOTEREVOCATION($\text{sig}_{\bar{P},R,i}$):
ensure *State* = WAITING FOR (INBOUND) REVOCATION
$R_{P,i}$ ← TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: ($C_{\bar{P},i}$.outputs.$\bar{P}$.value, $pk_{P,\text{out}}$)}
ensure VERIFY($R_{P,i}$, $\text{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
if *State* = WAITING FOR REVOCATION **then**
State ← WAITING FOR OUTBOUND REVOCATION
else // *State* = WAITING FOR INBOUND REVOCATION
$i \leftarrow i + 1$
State ← WAITING FOR HOSTS READY
end if
return (OK)

NEGLIGENT():
$\texttt{negligent}$ ← True
return (OK)

Figure 34

Process LN.EXCHANGEOPENKEYS()

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

Figure 35

Process LN.PREPAREBASE()

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

Figure 36

Process LN.EXCHANGEOPENSIGS()

// $s = (2 + q)\texttt{windowSize}$, where q and $\texttt{windowSize}$ are defined in Proposition A.1
$C_{A,0}$ ← TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, (pk_{A,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$, $(0, pk_{B,\text{out}})$}
$C_{B,0}$ ← TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, pk_{A,\text{out}})$, $(0, (pk_{B,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$}
$\text{sig}_{A,C,0} \leftarrow \text{SIGN}(C_{B,0}, sk_{A,F})$
State ← WAITING FOR COMM SIG
send (FUNDING CREATED, $(c, pk_{A,\text{chain}})$, $\text{sig}_{A,C,0}$) to $\texttt{fundee}$
ensure *State* = WAITING FOR COMM SIG // if opening virtual channel, we have received (FUNDED, $\texttt{host\_fundee}$) by $\texttt{hops[-1].right}$ (Fig 39, l. 3)
if $\texttt{hops}$ = "$\texttt{ledger}$" **then** // opening base channel
F ← TX {input: $(c, pk_{A,\text{chain}})$, output: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$}
end if
$C_{B,0}$ ← TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, pk_{A,\text{out}})$, $(0, (pk_{B,\text{out}} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$}

ensure VERIFY($C_{B,0}$, $\text{sig}_{A,C,0}$, $pk_{A,F}$) = True
$C_{A,0} \leftarrow$ TX {input: $(c, 2/\{pk_{A,F}, pk_{B,F}\})$, outputs: $(c, (pk_{A,\text{out}} + (p+s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$, $(0, pk_{B,\text{out}})$}
$\text{sig}_{B,C,0} \leftarrow$ SIGN($C_{A,0}$, $sk_{B,F}$)
if hops = "ledger" **then** // opening base channel
State ← WAITING TO CHECK FUNDING
else // opening virtual channel
$c_A \leftarrow c$; $c_B \leftarrow 0$; $i \leftarrow 0$
State ← OPEN
end if
reply (FUNDING SIGNED, $\text{sig}_{B,C,0}$)
ensure *State* = WAITING FOR COMM SIG
ensure VERIFY($C_{A,0}$, $\text{sig}_{B,C,0}$, $pk_{B,F}$) = True

Figure 37

Process LN.COMMITBASE()

$\text{sig}_F \leftarrow$ SIGN(F, $sk_{A,\text{chain}}$)
input (SUBMIT, (F, sig_F)) to $\mathcal{G}_{\text{Ledger}}$ // enter "while" below before sending
while $F \notin \Sigma$ **do**
wait for input (CHECK FUNDING) // ignore all other messages
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
end while

Figure 38

Process LN – external open messages for *Bob*

On output (FUNDED, host$_P$, funder_layer, t_P) by hops[-1].right:
ensure *State* = WAITING FOR FUNDED
store host$_P$ // we will talk directly to host$_P$
layer ← funder_layer
State ← WAITING FOR COMM SIG
reply (FUND ACK)

On output (CHECK KEYS, (pk_1, pk_2)) by hops[-1].right:
ensure *State* = WAITING FOR CHECK KEYS
ensure $pk_1 = pk_{A,F} \wedge pk_2 = pk_{B,F}$
State ← WAITING FOR FUDNED
reply (KEYS OK)

On input (CHECK FUNDING):
ensure *State* = WAITING TO CHECK FUNDING
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
if $F \in \Sigma$ **then**
State ← OPEN
reply (OPEN OK)
end if

Figure 39

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

// fundee is *Bob*
ensure we run the code of *Alice* // activated party is the funder
if hops = "ledger" **then** // opening base channel
ensure *State* = TOPPED UP
ensure $c = c_{A,\text{chain}}$
else // opening virtual channel
ensure len(hops) ≥ 2 // cannot open a virtual over 1 channel
end if
LN.EXCHANGEOPENKEYS()
LN.PREPAREBASE()
LN.EXCHANGEOPENSIGS()
if hops = "ledger" **then**
LN.COMMITBASE()
end if
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
last_poll ← $|\Sigma|$
$c_A \leftarrow c$; $c_B \leftarrow 0$; $i \leftarrow 0$
State ← OPEN
output (OPEN OK, c, fundee, hops)

Figure 40

Process LN.UPDATEFORVIRTUAL()

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

Figure 41

Process LN – virtualise start and end

On input (FUND ME, fundee, hops, c_{virt}, $pk_{A,V}$, $pk_{B,V}$) by funder:

ensure *State* = OPEN

ensure c_P − locked$_P \geq c_{\text{virt}}$

State ← VIRTUALISING

$(sk'_{P,F}, pk'_{P,F})$ ← KEYGEN()

define new VIRT ITI host$'_P$

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

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

LN.UPDATEFORVIRTUAL()

State ← WAITING FOR REVOCATION

input (HOST ME, funder, fundee, host$'_{\bar{P}}$, 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 host$'_P$

On output (HOSTS READY, t_P) by host$_P$: // host$_P$ is the new host, renamed in Fig. 34, l. 12

ensure *State* = WAITING FOR HOSTS READY

State ← OPEN

hosting ← True

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

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

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

output (FUNDED, host$_P$, layer, t_P) to fundee and expect reply (FUND ACK)

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

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

output (FUNDED, host$_P$, layer, t_P) to funder // do not expect reply by funder

end if

reply (HOST ACK)

Figure 42

Process LN – virtualise hops

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

ensure *State* = OPEN

ensure $c_{\bar{P}}$ − locked$_{\bar{P}} \geq c_{\text{virt}}$; $1 \leq i \leq n$

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

State ← VIRTUALISING

locked$_{\bar{P}}$ ← locked$_{\bar{P}}$ + c_{virt} // 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

$(sk'_{P,F}, pk'_{P,F})$ ← KEYGEN()

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

define new VIRT ITI host$'_P$

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, $pk_{\text{sib},\bar{P},F}$)

input (INIT, host$_P$, 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)

else // we are the last hop

input (INIT, host$_P$, 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)

end if

State ← WAITING FOR REVOCATION

send (VIRTUALISING ACK, host$'_P$, $pk'_{P,F}$) to $\bar{P}$

On input (VIRTUALISING, host_sibling, $(sk'_{P,F}, pk'_{P,F})$, $pk_{\text{sib},\bar{P},F}$, $pk_{\text{sib,out}}$, hops, fundee, c_{virt}, $c_{\text{sib,rem}}$, c_{sib}, i, n) by sibling:

ensure *State* = OPEN

ensure c_P − locked$_P \geq c_{\text{virt}}$

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

State ← VIRTUALISING

locked$_P$ ← locked$_P$ + c_{virt}

define new VIRT ITI host$'_P$

send (VIRTUALISING, host$'_P$, $pk'_{P,F}$, hops, fundee, c_{virt}, $i+1$, n) to hops[0].right and expect reply (VIRTUALISING ACK, host$'_{\bar{P}}$, $pk'_{\bar{P},F}$)

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

LN.UPDATEFORVIRTUAL()

input (INIT, host$_P$, 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)

State ← WAITING FOR REVOCATION

output (VIRTUALISING ACK, host$'_P$, $pk'_{\bar{P},F}$) to sibling

Figure 43

Process LN.SIGNATURESROUNDTRIP()

$C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ 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

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

$(sk_{P,R,i+2}, pk_{P,R,i+2})$ ← KEYGEN()

State ← WAITING FOR COMMITMENT SIGNED

send (PAY, x, sig$_{P,C,i+1}$, $pk_{P,R,i+2}$) to $\bar{P}$

// P refers to payer and $\bar{P}$ to payee both in local and remote code

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

$C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$ 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

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

$C_{P,i+1} \leftarrow C_{P,i}$ 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

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

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

sig$_{\bar{P},R,i}$ ← SIGN($R_{P,i}$, $sk_{\bar{P},R,i}$)

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

State ← WAITING FOR PAY REVOCATION

reply (COMMITMENT SIGNED, $\mathrm{sig}_{\bar{P},C,i+1}$, $\mathrm{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i+2}$)
ensure *State* = WAITING FOR COMMITMENT SIGNED
$C_{P,i+1} \leftarrow C_{P,i}$ 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

Figure 44

Process LN.REVOCATIONSTRIP()

ensure VERIFY($C_{P,i+1}$, $\mathrm{sig}_{\bar{P},C,i+1}$, $pk_{\bar{P},F}$) = True
$R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: ($c_{\bar{P}}$, $pk_{P,\mathrm{out}}$)}
ensure VERIFY($R_{P,i}$, $\mathrm{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
$R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.P, output: (c_P, $pk_{\bar{P},\mathrm{out}}$)}
$\mathrm{sig}_{P,R,i} \leftarrow$ SIGN($R_{\bar{P},i}$, $sk_{P,R,i}$)
add x to `paid_out`
$c_P \leftarrow c_P - x$; $c_{\bar{P}} \leftarrow c_{\bar{P}} + x$; $i \leftarrow i + 1$
State $\leftarrow$ OPEN
if $\texttt{host}_P \neq$ "`ledger`" $\wedge$ we have a `host_sibling` **then** // we are intermediary channel
    input (NEW BALANCE, c_P, $c_{\bar{P}}$) to $\texttt{host}_P$
    relay message as input to `sibling` // run by VIRT
    relay message as output to `guest` // run by VIRT
    store new sibling balance and reply (NEW BALANCE OK)
    output (NEW BALANCE OK) to `sibling` // run by VIRT
    output (NEW BALANCE OK) to `guest` // run by VIRT
end if
send (REVOKE AND ACK, $\mathrm{sig}_{P,R,i}$) to $\bar{P}$
ensure *State* = WAITING FOR PAY REVOCATION
$R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.P, output: (c_P, $pk_{\bar{P},\mathrm{out}}$)}
ensure VERIFY($R_{\bar{P},i}$, $\mathrm{sig}_{P,R,i}$, $pk_{P,R,i}$) = True
add x to `paid_in`
$c_P \leftarrow c_P - x$; $c_{\bar{P}} \leftarrow c_{\bar{P}} + x$; $i \leftarrow i + 1$
State $\leftarrow$ OPEN
if $\texttt{host}_P \neq$ "`ledger`" $\wedge$ $\bar{P}$ has a `host_sibling` **then** // we are intermediary channel
    input (NEW BALANCE, $c_{\bar{P}}$, c_P) to $\texttt{host}_{\bar{P}}$
    relay message as input to `sibling` // run by VIRT
    relay message as output to `guest` // run by VIRT
    store new sibling balance and reply (NEW BALANCE OK)
    output (NEW BALANCE OK) to `sibling` // run by VIRT
    output (NEW BALANCE OK) to `guest` // run by VIRT
end if

Figure 45

Process LN – On (PAY, x):

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

Figure 46

Process LN – On (GET PAID, y):

ensure *State* = OPEN $\wedge$ $c_{\bar{P}} \geq y$
if $\texttt{host}_P \neq$ "`ledger`" $\wedge$ P has a `host_sibling` **then** // we are intermediary channel
    ensure $c_P + y \leq c_{\mathrm{sib,rem}} \wedge c_{\mathrm{sib}} \leq c_{\bar{P}} - y$ // avoid value loss by griefing attack
end if
store y
State $\leftarrow$ WAITING TO GET PAID

Figure 47

Process LN – On (CHECK FOR LATERAL CLOSE):

if $\texttt{host}_P \neq$ "`ledger`" **then**
    input (CHECK FOR LATERAL CLOSE) to $\texttt{host}_P$
end if

Figure 48

Process LN – On (CHECK CHAIN FOR CLOSED):

ensure *State* $\notin$ {$\perp$, INIT, TOPPED UP} // channel open
// even virtual channels check $\mathcal{G}_{\mathrm{Ledger}}$ directly. This is intentional
input (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign reply to Σ
`last_poll` $\leftarrow |\Sigma|$
if $\exists 0 \leq j < i : C_{\bar{P},j} \in \Sigma$ **then** // counterparty has closed maliciously
    State $\leftarrow$ CLOSING
    LN.SUBMITANDCHECKREVOCATION(j)
    State $\leftarrow$ CLOSED
    output (CLOSED)
else if $C_{P,i} \in \Sigma \vee C_{\bar{P},i} \in \Sigma$ **then**
    State $\leftarrow$ CLOSED
    output (CLOSED)
else
    `state_before_checking_revoked` $\leftarrow$ *State*
    for each `host` in list of old hosts **do**
        State $\leftarrow$ WAITING FOR NOTHING REVOKED
        input (CHECK FOR REVOKED) to `host` and expect output (NOTHING REVOKED)
        State $\leftarrow$ `state_before_checking_revoked`
    end for
end if

Figure 49

Process LN.SUBMITANDCHECKREVOCATION(j)

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

Figure 50

Process LN – On (FORCECLOSE):

ensure *State* $\notin \{\perp$, INIT, TOPPED UP, CLOSED, BASE PUNISHED$\}$ // channel open
if `host`$_P \neq$ "`ledger`" **then** // we have a virtual channel
  State $\leftarrow$ HOST CLOSING
  input (FORCECLOSE) to `host`$_P$ and keep relaying any (CHECK IF CLOSING) or (FORCECLOSE) input to `host`$_P$ until receiving output (CLOSED) by `host`$_P$
  `host`$_P \leftarrow$ "`ledger`"
end if
State $\leftarrow$ CLOSING
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
if $C_{\bar{P},i} \in \Sigma$ **then** // counterparty has closed honestly
  no-op // do nothing
else if $\exists 0 \leq j < i : C_{\bar{P},j} \in \Sigma$ **then** // counterparty has closed maliciously
  LN.SUBMITANDCHECKREVOCATION(j)
else // counterparty is idle
  while $\neg\exists$ unspent `output` $\in \Sigma$ that $C_{P,i}$ can spend **do** // possibly due to an active timelock
    wait for input (CHECK VIRTUAL) // ignore other messages
    input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
  end while
  $\mathrm{sig}'_{P,C,i} \leftarrow$ SIGN$(C_{P,i}, sk_{P,F})$
  input (SUBMIT, $(C_{P,i}, \mathrm{sig}_{P,C,i}, \mathrm{sig}'_{P,C,i})$) to $\mathcal{G}_{\text{Ledger}}$
end if

Figure 51

Process LN – punishment

On output (ENABLER USED REVOCATION) by `host`$_P$:
  State $\leftarrow$ BASE PUNISHED

Figure 52

Process LN – On (COOPCLOSE):

// any endpoint or intermediary can initiate virtual channel closing
1: ensure `host`$_P \neq$ "`ledger`"
2: ensure *State* = OPEN
3: `we_are_close_initiator` $\leftarrow$ True
4: **if** `hosting` = True $\vee$ we have received OPEN from $\mathcal{E}$ while *State* was TOPPED UP **then** // we are not the `fundee` of a channel that is not the base of any other channel
5:   **if** `hosting` = True **then** // we are not the `funder` of the channel to be closed
6:     the next time we are activated, if we are not activated by output (CHECK COOP CLOSE, ...) from `host`$_P$, set `we_are_close_initiator` $\leftarrow$ False
7:   **else** // we are the `funder` of the channel to be closed
8:     the next time we are activated, if we are not activated by output (COOP CLOSE, ...) from $\bar{P}$, set `we_are_close_initiator` $\leftarrow$ False
9:   **end if**
10:   send (COOP CLOSE) to `fundee`
11: **else** // we are the `fundee` of a channel that is not the base of any other channel
12:   the next time we are activated, if we are not activated by output (CHECK COOP CLOSE FUNDEE, ...) from `host`$_P$, set `we_are_close_initiator` $\leftarrow$ False
13:   `close_initiator` $\leftarrow P$
14:   execute code of Fig. 55
15: **end if**

Figure 53

Process LN – On (COOPCLOSED) by R:

if `hosting` = True **then** // we are intermediary
  ensure *State* = OPEN
else // we are endpoint
  ensure *State* = COOP CLOSED
end if
ensure `we_are_close_initiator` = True
ensure that the last cooperatively closed channel in which we acted as a base had R as its `fundee`
`we_are_close_initiator` $\leftarrow$ False
output (COOPCLOSED)

Figure 54

Process LN – On (COOP CLOSE) by R:

// also executed when we are instructed to close a channel cooperatively by $\mathcal{E}$– cf. Fig. 53, l. 14
ensure we are `fundee`
ensure `hosting` $\neq$ True
ensure *State* = OPEN
State $\leftarrow$ COOP CLOSING
`close_initiator` $\leftarrow R$
`sig_bal` $\leftarrow ((c_{\bar{P}}, c_P),$ SIGN$((c_{\bar{P}}, c_P), sk_{P,F}))$
State $\leftarrow$ WAITING TO REVOKE VIRT COMM
send (COOP CLOSE, `sig_bal`) to $\bar{P}$

Figure 55

Process LN – On (COOP CLOSE, $\mathtt{sig\_bal}_{\bar{P}}$) by $\bar{P}$:

ensure we are funder
ensure *State* = OPEN
parse $\mathtt{sig\_bal}_{\bar{P}}$ as $((c'_1, c'_2), \mathrm{sig}_{\bar{P}})$
ensure $c_P = c'_1 \wedge c_{\bar{P}} = c'_2 \wedge \textsc{verify}((c'_1, c'_2), \mathrm{sig}_{\bar{P}}, pk_{\bar{P},F}) = \mathrm{True}$
$\mathtt{sig\_bal} \leftarrow ((c_P, c_{\bar{P}}), \textsc{sign}((c_P, c_{\bar{P}}), sk_{P,F}), \mathrm{sig}_{\bar{P}})$
State ← COOP CLOSING
input (COOP CLOSE, $\mathtt{sig\_bal}$) to $\mathtt{host}_P$
ensure *State* = OPEN // executed by $\mathtt{host}_P$
State ← COOP CLOSING
output (COOP CLOSE SIGN COMM FUNDER, (c'_1, c'_2)) to $\mathtt{guest}$
ensure *State* = OPEN // executed by guest of $\mathtt{host}_P$
State ← COOP CLOSING
remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$, $pk'_{\bar{P},F}$
$C_{\bar{P},i+1} \leftarrow \mathrm{TX}$ {input: $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$,
outputs: $(c_P + c'_1, pk_{P,\mathrm{out}})$,
$(c_{\bar{P}} + c'_2, (pk_{\bar{P},\mathrm{out}} + (p+s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$}
$\mathrm{sig}_{P,C,i+1} \leftarrow \textsc{sign}(C_{\bar{P},i+1}, sk'_{P,F})$
$(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \textsc{keyGen}()$
input (NEW COMM TX, $\mathrm{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to $\mathtt{host}_P$
rename received signature to $\mathrm{sig}_{1,\mathrm{right},C}$ // executed by $\mathtt{host}_P$
rename received public key to $pk_{1,\mathrm{right},R}$
send (COOP CLOSE, $\mathtt{sig\_bal}$, $\mathrm{sig}_{1,\mathrm{right},C}$, $pk_{1,\mathrm{right},R}$) to $\bar{P}$ and expect reply (COOP CLOSE BACK, ($\mathtt{right\_comms\_revkeys}$, $\mathtt{right\_revocations}$)
$R_{\mathrm{loc,virt}} \leftarrow \mathrm{TX}$ {input: $(c_{\mathrm{virt}}, 2/\{pk_{1,\mathrm{rev}}, pk_{2,\mathrm{rev}}\})$, output: $(c_{\mathrm{virt}}, pk_{1,\mathrm{out}})$}
extract $\mathrm{sig}_{2,\mathrm{right,rev,virt}}$ from $\mathtt{right\_revocations}$
ensure $\textsc{verify}(R_{\mathrm{loc,virt}}, \mathrm{sig}_{2,\mathrm{right,rev,virt}}, pk_{2,\mathrm{rev}}) = \mathrm{True}$
$R_{\mathrm{loc,fund}} \leftarrow \mathrm{TX}$ {input: $(c_P + c_{\bar{P}}, 2/\{pk_{1,\mathrm{rev}}, pk_{2,\mathrm{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{1,\mathrm{out}})$}
extract $\mathrm{sig}_{2,\mathrm{right,rev,fund}}$ from $\mathtt{right\_revocations}$
ensure $\textsc{verify}(R_{\mathrm{loc,fund}}, \mathrm{sig}_{2,\mathrm{right,rev,fund}}, pk_{2,\mathrm{rev}}) = \mathrm{True}$
extract $\mathrm{sig}_{2,\mathrm{right},R}$ from $\mathtt{right\_revocations}$
extract $\mathrm{sig}_{2,\mathrm{right},C}$ from $\mathtt{right\_comms\_revkeys}$
extract $pk_{2,R}$ from $\mathtt{right\_comms\_revkeys}$
output (VERIFY REVOKE, $\mathrm{sig}_{2,\mathrm{right},C}$, $\mathrm{sig}_{2,\mathrm{right},R}$, $pk_{2,R}$, $\mathtt{host}_P$) to $\mathtt{guest}$
store $\mathrm{sig}_{2,\mathrm{right},C}$ as $\mathrm{sig}_{\bar{P},C,i+1}$ // executed by guest of $\mathtt{host}_P$
store $\mathrm{sig}_{2,\mathrm{right},R}$ as $\mathrm{sig}_{\bar{P},R,i}$
store received public key as $pk_{\bar{P},R,i+2}$
$C_{P,i+1} \leftarrow \mathrm{TX}$ {input: $(c_P + c_{\bar{P}} + c'_1 + c'_2)$, outputs:
$(c_P + c'_1, (pk_{P,\mathrm{out}} + (p+s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$,
$(c_{\bar{P}} + c'_2, pk_{\bar{P},\mathrm{out}})$}
ensure $\textsc{verify}(C_{P,i+1}, \mathrm{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = \mathrm{True}$
$R_{P,i} \leftarrow \mathrm{TX}$ {input: $C_{\bar{P},i}.\mathrm{outputs}.\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\mathrm{out}})$}
ensure $\textsc{verify}(R_{P,i}, \mathrm{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = \mathrm{True}$
input (VERIFIED) to $\mathtt{host}_P$
extract $\mathrm{sig}_{n,\mathrm{left},R}$ from $\mathtt{right\_revocations}$ // executed by $\mathtt{host}_P$
output (VERIFY REVOCATION, $\mathrm{sig}_{n,\mathrm{left},R}$) to $\mathtt{funder}$
$R_{P,i} \leftarrow \mathrm{TX}$ {input: $C_{\bar{P},i}.\mathrm{outputs}.\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\mathrm{out}})$}
ensure $\textsc{verify}(R_{P,i}, \mathrm{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = \mathrm{True}$
$R_{\bar{P},i} \leftarrow \mathrm{TX}$ {input: $C_{P,i}.\mathrm{outputs}.P$, output: $(c_P, pk_{\bar{P},\mathrm{out}})$}
$\mathrm{sig}_{P,R,i} \leftarrow \textsc{sign}(R_{\bar{P},i}, sk_{P,R,i})$
State ← COOP CLOSED // in LN, only virtual channels can end up in this state
input (COOP CLOSE REVOCATION, $\mathrm{sig}_{P,R,i}$) to $\mathtt{host}_P$
output (COOP CLOSE REVOCATIONS, $\mathtt{host}_P$) to $\mathtt{guest}$ // executed by $\mathtt{host}_P$
$R_{\bar{P},i} \leftarrow \mathrm{TX}$ {input: $C_{P,i}.\mathrm{outputs}.P$, output: $(c_P, pk_{\bar{P},\mathrm{out}})$} // executed by guest of $\mathtt{host}_P$
$\mathrm{sig}_{P,R,i} \leftarrow \textsc{sign}(R_{\bar{P},i}, sk_{P,R,i})$
add $sk_{P,F}, pk_{P,F}, pk_{\bar{P},F}$ to list of old enable channel funding keys
add $\mathtt{host}_P$ to list of old hosts
assign received host to $\mathtt{host}_P$
$c_P \leftarrow c_P + c'_1; c_{\bar{P}} \leftarrow c_{\bar{P}} + c'_2$
$\mathtt{layer} \leftarrow \mathtt{layer} - 1$
$\mathtt{locked}_P \leftarrow \mathtt{locked}_P - c_{\mathrm{virt}}$
State ← OPEN
input (REVOCATION, $\mathrm{sig}_{P,R,i}$) to last old host
rename received signature to $\mathrm{sig}_{1,\mathrm{right},R}$ // executed by $\mathtt{host}_P$
$R_{\mathrm{rem,virt}} \leftarrow \mathrm{TX}$ {input: $(c_{\mathrm{virt}}, 4/\{pk_{1,\mathrm{rev}}, pk_{1,\mathrm{rev}}, pk_{2,\mathrm{rev}}, pk_{n,\mathrm{rev}}\})$, output: $(c_{\mathrm{virt}}, pk_{2,\mathrm{out}})$}
$\mathrm{sig}_{1,\mathrm{right,rev,virt}} \leftarrow \textsc{sign}(R_{\mathrm{rem,virt}}, sk_{1,\mathrm{rev}})$
$R_{\mathrm{rem,fund}} \leftarrow \mathrm{TX}$ {input: $(c_P + c_{\bar{P}}, 2/\{pk_{1,\mathrm{rev}}, pk_{2,\mathrm{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{2,\mathrm{out}})$}
$\mathrm{sig}_{1,\mathrm{right,rev,fund}} \leftarrow \textsc{sign}(R_{\mathrm{rem,fund}}, sk_{1,\mathrm{rev}})$
for all $j \in \{2, \ldots, n\}$ **do**
    $R_{j,\mathrm{left}} \leftarrow \mathrm{TX}$ {input: $(c_{\mathrm{virt}}, 4/\{pk_{1,\mathrm{rev}}, pk_{j-1,\mathrm{rev}}, pk_{j,\mathrm{rev}}, pk_{n,\mathrm{rev}}\})$, output: $(c_{\mathrm{virt}}, pk_{j,\mathrm{out}})$}
    $\mathrm{sig}_{1,j,\mathrm{left,rev}} \leftarrow \textsc{sign}(R_{j,\mathrm{left}}, sk_{1,\mathrm{rev}})$
end for
State ← COOP CLOSED
send (COOP CLOSE REVOCATIONS, ($\mathrm{sig}_{1,\mathrm{right},R}$, $\mathrm{sig}_{1,\mathrm{right,rev,virt}}$, $\mathrm{sig}_{1,\mathrm{right,rev,fund}}$, $(\mathrm{sig}_{1,j,\mathrm{left,rev}})_{j \in \{2,\ldots,n\}}$)) to $\bar{P}$

Figure 56

Process LN – On (CORRUPT) by $\mathcal{A}$ or kindred party R:

// This is executed by the shell – cf. [16]
if *State* ≠ CORRUPTED **then**
    State ← CORRUPTED
    for $S \in$ set of kindred parties **do**
        input (CORRUPT) to S and expect reply (OK)
    end for
end if
reply (OK)

Figure 57

Process VIRT

On every activation, before handling the message:
    if $\mathtt{last\_poll} \neq \bot$ **then** // virtual layer is ready
        input (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign ouput to Σ
        if $\mathtt{last\_poll} + p < |\Sigma|$ **then**

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

ensure P.NEGLIGENT() returns (OK)

end for

end if

end if

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

On input (INIT, 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:

ensure $1 < i \le n$ // host_funder ($i = 1$) is initialised with HOST ME

ensure side ∈ {"left", "right"}

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

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

$pk_{\text{myRem,fund,new}} \leftarrow pk_{\text{rem,fund,new}}$

if $i < n$ **then** // we are not last hop

$pk_{\text{sibRem,fund,new}} \leftarrow pk_{\text{sib,rem,fund,new}}$

end if

if side = "left" **then**

side′ ← "right"; myRem ← $i - 1$; sibRem ← $i + 1$

$pk_{i,\text{out}} \leftarrow pk_{\text{loc,out}}$

$(sk_{i,j,k}, pk_{i,j,k})_{j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}} \leftarrow \text{KEYGEN}()^{(n-2)(n-1)}$

$(sk_{i,\text{rev}}, pk_{i,\text{rev}}) \leftarrow \text{KEYGEN}()$

else // side = "right"

side′ ← "left"; myRem ← $i + 1$; sibRem ← $i - 1$

// sibling will send keys in KEYS AND COINS FORWARD

end if

$(sk_{i,\text{side,fund,old}}, pk_{i,\text{side,fund,old}}) \leftarrow (sk_{\text{loc,fund,old}}, pk_{\text{loc,fund,old}})$

$pk_{\text{myRem,side}',\text{fund,old}} \leftarrow pk_{\text{rem,fund,old}}$

$(c_{i,\text{side}}, c_{\text{myRem,side}'}, t_{i,\text{side}}) \leftarrow (c_P, c_{\bar{P}}, t_P)$

last_poll ← ⊥

output (HOST INIT OK) to guest

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:

last_poll ← ⊥

$i \leftarrow 1$

$c_{1,\text{right}} \leftarrow c_P$; $c_{2,\text{left}} \leftarrow c_{\bar{P}}$

$(sk_{1,j,k}, pk_{1,j,k})_{j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}} \leftarrow \text{KEYGEN}()^{(n-2)(n-1)}$

$(sk_{1,\text{rev}}, pk_{1,\text{loc,rev}}) \leftarrow \text{KEYGEN}()$

ensure VIRT.CIRCULATEKEYSCOINSTIMES() returns (OK)

ensure VIRT.CIRCULATEVIRTUALSIGS() returns (OK)

ensure VIRT.CIRCULATEFUNDINGSIGS() returns (OK)

ensure VIRT.CIRCULATEREVOCATIONS() returns (OK)

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

Figure 58

Process VIRT.CIRCULATEKEYSCOINSTIMES(left_data):

if left_data is given as argument **then** // we are not host_funder

parse left_data as $((pk_{j,\text{fund,new}})_{j\in[i-1]}$, $(pk_{j,\text{left,fund,old}})_{j\in\{2,\dots,i-1\}}$, $(pk_{j,\text{right,fund,old}})_{j\in[i-1]}$, $(pk_{j,\text{out}})_{j\in[i-1]}$, $(c_{j,\text{left}})_{j\in\{2,\dots,i-1\}}$, $(c_{j,\text{right}})_{j\in[i-1]}$, $(t_j)_{j\in[i-1]}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $(pk_{h,j,k})_{h\in[i-1],j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}}$, $(pk_{h,\text{loc,rev}})_{h\in[i-1]}$, $(pk_{h,\text{rem,rev}})_{h\in[i-1]})$

if we have a sibling **then** // we are not host_fundee

input (KEYS AND COINS FORWARD, (left_data, $(sk_{i,\text{left,fund,old}}, pk_{i,\text{left,fund,old}})$, $pk_{i,\text{out}}$, $c_{i,\text{left}}$, $t_{i,\text{left}}$, $(sk_{i,j,k}, pk_{i,j,k})_{j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}}$, $(sk_{i,\text{rev}}, pk_{i,\text{rev}})$)) to sibling

store input as left_data and parse it as $((pk_{j,\text{fund,new}})_{j\in[i-1]}$, $(pk_{j,\text{left,fund,old}})_{j\in\{2,\dots,i\}}$, $(pk_{j,\text{right,fund,old}})_{j\in[i-1]}$, $(pk_{j,\text{out}})_{j\in[i]}$, $(c_{j,\text{left}})_{j\in\{2,\dots,i\}}$, $(c_{j,\text{right}})_{j\in[i-1]}$, $(t_j)_{j\in[i-1]}$, $sk_{i,\text{left,fund,old}}$, $t_{i,\text{left}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $(pk_{h,j,k})_{h\in[i],j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}}$, $(sk_{i,j,k})_{j\in\{2,\dots,n-1\},k\in[n]\setminus\{j\}}$, $(pk_{h,\text{rev}})_{h\in[i]}$, $sk_{i,\text{rev}}$

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

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

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

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

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

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)

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

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

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

else // we are host_fundee

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

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

end if

else // we are host_funder

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

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

return (OK)

end if

Figure 59

Process virt

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

  ensure $1 < i < n$
  ensure $c_{\text{rem,left}} \geq c_{\text{virt}} \wedge c_{\text{loc,left}} \geq c_{\text{virt}}$ // left parties fund virtual channel
  ensure $c_{\text{rem,left}} \geq c_{\text{loc,right}} \wedge c_{\text{rem,right}} \geq c_{\text{loc,left}}$ // avoid griefing attack
  $c_{\text{left}} \leftarrow c_{\text{rem,left}} + c_{\text{loc,left}}$; $c_{\text{right}} \leftarrow c_{\text{loc,right}} + c_{\text{rem,right}}$
  `left_old_fund` $\leftarrow 2/\{pk_{\text{rem,left,fund,old}}, pk_{\text{loc,left,fund,old}}\}$
  `right_old_fund` $\leftarrow 2/\{pk_{\text{loc,right,fund,old}}, pk_{\text{rem,right,fund,old}}\}$
  `left_new_fund` $\leftarrow 2/\{pk_{\text{rem,left,fund,new}}, pk_{\text{loc,left,fund,new}}\} \vee 2/\{pk_{\text{left,rev}}, pk_{\text{loc,rev}}\}$
  `right_new_fund` $\leftarrow 2/\{pk_{\text{loc,right,fund,new}}, pk_{\text{rem,right,fund,new}}\} \vee 2/\{pk_{\text{loc,rev}}, pk_{\text{right,rev}}\}$
  `virt_fund` $\leftarrow 2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$
  `revocation` $\leftarrow 4/\{pk_{\text{funder,rev}}, pk_{\text{loc,rev}}, pk_{\text{right,rev}}, pk_{\text{fundee,rev}}\}$
  `refund` $\leftarrow (pk_{\text{loc,out}} + (p + s)) \vee 2/\{pk_{\text{left,rev}}, pk_{\text{loc,rev}}\}$
  for all $j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1, j\}$ **do**
    $all_{j,k} \leftarrow n/\{pk_{1,j,k}, \ldots, pk_{n,j,k}\} \wedge$ "k"
  end for
  if $i = 2$ **then**
    $all_{2,1} \leftarrow n/\{pk_{1,2,1}, \ldots, pk_{n,2,1}\} \wedge$ "1"
  end if
  if $i = n - 1$ **then**
    $all_{n-1,n} \leftarrow n/\{pk_{1,n-1,n}, \ldots, pk_{n,n-1,n}\} \wedge$ "n"
  end if
  if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
  if $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
  $\texttt{bridge}_1 \leftarrow 4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge$ "1" // We reuse the $pk_{j,2,1}$ keys for all `bridge`s to avoid new keys. $pk_{i,2,1}$ is not needed because i is not a beneficiary of the bridge tx.
  $\texttt{revocation}_1 \leftarrow 4/\{pk_{\text{funder,rev}}, pk_{\text{loc,rev}}, pk_{\text{right,rev}}, pk_{\text{fundee,rev}}\} \wedge$ "1"
  for all $k \in \{m, \ldots, l\} \setminus \{i\}$ **do**
    $\texttt{bridge}_{2,k} \leftarrow 4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge$ "2,k"
    $\texttt{revocation}_{2,k} \leftarrow 4/\{pk_{\text{funder,rev}}, pk_{\text{loc,rev}}, pk_{\text{right,rev}}, pk_{\text{fundee,rev}}\} \wedge$ "2,k"
  end for
  for all $(k_1, k_2) \in \{m, \ldots, i-1\} \times \{i+1, \ldots, l\}$ **do**
    $\texttt{bridge}_{3,k_1,k_2} \leftarrow 4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge$ "3,k_1,k_2"
    $\texttt{revocation}_{3,k_1,k_2} \leftarrow 4/\{pk_{\text{funder,rev}}, pk_{\text{loc,rev}}, pk_{\text{right,rev}}, pk_{\text{fundee,rev}}\} \wedge$ "3,k_1,k_2"
  end for
  // After funding is complete, A_j has the signature of all other parties for all $all_{j,k}$ and `bridge` inputs, but other parties do not have A_j's signature for this input, therefore only A_j can publish it.
  // $\text{TX}_{i,j,k}$:= i-th move, j, k input interval start and end. j, k unneeded for $i = 1$, k unneeded for $i = 2$.
  $\text{TX}_1 \leftarrow$ TX:
    inputs:
      (c_{left}, `left_old_fund`),
      (c_{right}, `right_old_fund`)
    outputs:
      ($c_{\text{left}} - c_{\text{virt}}$, `left_new_fund`),
      ($c_{\text{right}} - c_{\text{virt}}$, `right_new_fund`),
      (c_{virt}, `refund`),
      (c_{virt},
        (if $(i - 1 > 1)$ then $all_{i-1,i}$ else False)
        $\vee$ (if $(i + 1 < n)$ then $all_{i+1,i}$ else False)
        $\vee \texttt{revocation}_1$
        $\vee$ (
          if $(i - 1 = 1 \wedge i + 1 = n)$ then $\texttt{bridge}_1$
          else if $(i - 1 > 1 \wedge i + 1 = n)$ then $\texttt{bridge}_1 + t_{i-1}$
          else if $(i - 1 = 1 \wedge i + 1 < n)$ then $\texttt{bridge}_1 + t_{i+1}$
          else /*$i - 1 > 1 \wedge i + 1 < n$*/ $\texttt{bridge}_1 + \max(t_{i-1}, t_{i+1})$
)
)
  $B_1 \leftarrow$ TX:
    input:
      (c_{virt}, $\texttt{bridge}_1$)
    output:
      (c_{virt}, `revocation` $\vee$ `virt_fund`)
  if $i = 2$ **then**
    $\text{TX}_{2,1} \leftarrow$ TX:
      inputs:
        (c_{virt}, $all_{2,1}$),
        (c_{right}, `right_old_fund`)
      outputs:
        ($c_{\text{right}} - c_{\text{virt}}$, `right_new_fund`),
        (c_{virt}, `refund`),
        (c_{virt},
          (if $(n > 3)$ then $(all_{3,2} \vee \texttt{revocation}_{2,1} \vee (\texttt{bridge}_{2,1} + t_3))$
          else $\texttt{revocation}_{2,1} \vee \texttt{bridge}_{2,1}$)
)
    $B_{2,1} \leftarrow$ TX:
      input:
        (c_{virt}, $\texttt{bridge}_{2,1}$)
      output:
        (c_{virt}, `revocation` $\vee$ `virt_fund`)
  end if
  if $i = n - 1$ **then**
    $\text{TX}_{2,n} \leftarrow$ TX:
      inputs:
        (c_{left}, `left_old_fund`),
        (c_{virt}, $all_{n-1,n}$)
      outputs:
        ($c_{\text{left}} - c_{\text{virt}}$, `left_new_fund`),
        (c_{virt}, `refund`),
        (c_{virt},

(if $(n-2>1)$ then $(all_{n-2,n-1} \lor \texttt{revocation}_{2,n} \lor (\texttt{bridge}_{2,n} + t_{n-2}))$
else $\texttt{revocation}_{2,n} \lor \texttt{bridge}_{2,n}$)
)
$B_{2,n} \leftarrow$ TX:
input:
$(c_{\text{virt}}, \texttt{bridge}_{2,n})$
output:
$(c_{\text{virt}}, \texttt{revocation} \lor \texttt{virt\_fund})$
end if
for all $k \in \{2, \ldots, i-1\}$ **do** // $2(i-2)$ txs
$\text{TX}_{2,k} \leftarrow$ TX:
inputs:
$(c_{\text{virt}}, all_{i,k})$,
$(c_{\text{right}}, \texttt{right\_old\_fund})$
outputs:
$(c_{\text{right}} - c_{\text{virt}}, \texttt{right\_new\_fund})$,
$(c_{\text{virt}}, \texttt{refund})$,
$(c_{\text{virt}},$
(if $(k-1>1)$ then $all_{k-1,i}$ else False)
$\lor$ (if $(i+1<n)$ then $all_{i+1,k}$ else False)
$\lor \texttt{revocation}_{2,k}$
$\lor$ (
if $(k-1=1 \land i+1=n)$ then $\texttt{bridge}_{2,k}$
else if $(k-1>1 \land i+1=n)$ then $\texttt{bridge}_{2,k} + t_{k-1}$
else if $(k-1=1 \land i+1<n)$ then $\texttt{bridge}_{2,k} + t_{i+1}$
else /*$k-1>1 \land i+1<n$*/
$\texttt{bridge}_{2,k} + \max(t_{k-1}, t_{i+1})$
)
)
$B_{2,k} \leftarrow$ TX:
input:
$(c_{\text{virt}}, \texttt{bridge}_{2,k})$
output:
$(c_{\text{virt}}, \texttt{revocation} \lor \texttt{virt\_fund})$
end for
for all $k \in \{i+1, \ldots, n-1\}$ **do** // $2(n-i-1)$ txs
$\text{TX}_{2,k} \leftarrow$ TX:
inputs:
$(c_{\text{left}}, \texttt{left\_old\_fund})$
$(c_{\text{virt}}, all_{i,k})$,
outputs:
$(c_{\text{left}} - c_{\text{virt}}, \texttt{left\_new\_fund})$,
$(c_{\text{virt}}, \texttt{refund})$,
$(c_{\text{virt}},$
(if $(i-1>1)$ then $all_{i-1,k}$ else False)
$\lor$ (if $(k+1<n)$ then $all_{k+1,i}$ else False)
$\lor \texttt{revocation}_{2,k}$
$\lor$ (
if $(i-1=1 \land k+1=n)$ then $\texttt{bridge}_{2,k}$
else if $(i-1>1 \land k+1=n)$ then $\texttt{bridge}_{2,k} + t_{i-1}$
else if $(i-1=1 \land k+1<n)$ then $\texttt{bridge}_{2,k} + t_{k+1}$
else /*$i-1>1 \land k+1<n$*/
$\texttt{bridge}_{2,k} + \max(t_{i-1}, t_{k+1})$
)
)
$B_{2,k} \leftarrow$ TX:
input:
$(c_{\text{virt}}, \texttt{bridge}_{2,k})$
output:
$(c_{\text{virt}}, \lor\ \texttt{revocation} \lor \texttt{virt\_fund})$
end for
for all $(k_1, k_2) \in \{m, \ldots, i-1\} \times \{i+1, \ldots, l\}$ **do** // $(i-m) \cdot (l-i)$ txs
$\text{TX}_{3,k_1,k_2} \leftarrow$ TX:
inputs:
$(c_{\text{virt}}, all_{i,k_1})$,
$(c_{\text{virt}}, all_{i,k_2})$
outputs:
$(c_{\text{virt}}, \texttt{refund})$,
$(c_{\text{virt}},$
(if $(k_1-1>1)$ then $all_{k_1-1,\min(k_2,n-1)}$ else False)
$\lor$ (if $(k_2+1<n)$ then $all_{k_2+1,\max(k_1,2)}$ else False)
$\lor \texttt{revocation}_{3,k_1,k_2}$
$\lor$ (
if $(k_1-1 \leq 1 \land k_2+1 \geq n)$ then $\texttt{bridge}_{3,k_1,k_2}$
else if $(k_1-1>1 \land k_2+1 \geq n)$ then $\texttt{bridge}_{3,k_1,k_2} + t_{k_1-1}$
else if $(k_1-1 \leq 1 \land k_2+1<n)$ then $\texttt{bridge}_{3,k_1,k_2} + t_{k_2+1}$
else /*$k_1-1>1 \land k_2+1<n$*/
$\texttt{bridge}_{3,k_1,k_2} + \max(t_{k_1-1}, t_{k_2+1})$
)
)
$B_{3,k_1,k_2} \leftarrow$ TX:
input:
$(c_{\text{virt}}, \texttt{bridge}_{3,k_1,k_2})$
output:
$(c_{\text{virt}}, \lor\ \texttt{revocation} \lor \texttt{virt\_fund})$
end for
return (
TX_1, B_1,
$(\text{TX}_{2,k}, B_{2,k})_{k \in \{m,\ldots,l\} \setminus \{i\}}$,
$(\text{TX}_{3,k_1,k_2}, B_{3,k_1,k_2})_{(k_1,k_2) \in \{m,\ldots,i-1\} \times \{i+1,\ldots,l\}}$
)

Figure 60

Process VIRT

// 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.

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):

  ensure $i \in \{1, n\}$

  ensure $c_{\text{left}} \geq c_{\text{virt}}$ // left party funds virtual channel

  $c_{\text{tot}} \leftarrow c_{\text{left}} + c_{\text{right}}$

  old_fund $\leftarrow 2/\{pk_{\text{left,fund,old}}, pk_{\text{right,fund,old}}\}$

  new_fund $\leftarrow 2/\{pk_{\text{left,fund,new}}, pk_{\text{right,fund,new}}\} \vee 2/\{pk_{\text{left,rev}}, pk_{\text{right,rev}}\}$

  virt_fund $\leftarrow 2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$

  revocation $\leftarrow$ revocation$_1$ $\leftarrow 2/\{pk_{\text{interm,rev}}, pk_{\text{endpoint,rev}}\}$

  if $i = 1$ **then** // funder's tx

    $all \leftarrow n/\{pk_{\text{all},1}, \ldots, pk_{\text{all},n}\} \wedge$ "1"

    bridge $\leftarrow 2/\{pk_{all,2}, pk_{all,n}\} \wedge$ "1" // We reuse the $pk_{all,j}$ keys to avoid new keys

  else // $i = n$, fundee's tx

    $all \leftarrow n/\{pk_{\text{all},1}, \ldots, pk_{\text{all},n}\} \wedge$ "n"

    bridge $\leftarrow 2/\{pk_{all,1}, pk_{all,n-1}\} \wedge$ "1"

  end if

  $\text{TX}_1 \leftarrow$ TX: // endpoints only have an "initiator" tx

    input:

      (c_{tot}, old_fund)

    outputs:

      ($c_{\text{tot}} - c_{\text{virt}}$, new_fund),

      (c_{virt}, $all \vee$ revocation$_1$ $\vee$ (bridge $+ t$))

  $B_1 \leftarrow$ TX:

    input:

      (c_{virt}, bridge)

    output:

      (c_{virt}, revocation $\vee$ virt_fund)

  return TX_1, B_1

Figure 61

Process VIRT.SIBLINGSIGS()

parse input as $\text{sigs}_{\text{byLeft}}$

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

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

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

// notation: sig(TX, pk) := sig with ANYPREVOUT flag such that VERIFY(TX, sig, pk) = True

ensure that the following signatures are present in $\text{sigs}_{\text{byLeft}}$ and store them:

- // $(l - m) \cdot (i - 1)$ signatures

  $\forall k \in \{m, \ldots, l\} \setminus \{i\}, \forall j \in [i - 1]$:

    sig($\text{TX}_{i,2,k}$, $pk_{j,i,k}$)

- // $2 \cdot (i - m) \cdot (l - i) \cdot (i - 1)$ signatures

  $\forall k_1 \in \{m, \ldots, i - 1\}, \forall k_2 \in \{i + 1, \ldots, l\}, \forall j \in [i - 1]$:

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

$\text{sigs}_{\text{toRight}} \leftarrow \text{sigs}_{\text{byLeft}}$

if $i + 1 = n$ **then** // next hop is host_fundee

  $\text{TX}_{n,1}, B_{n,1} \leftarrow$ VIRT.GETENDPOINTTX(n, n, c_{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})

  add SIGN($B_{n,1}$, $sk_{i,2,1}$, ANYPREVOUT) to $\text{sigs}_{\text{toRight}}$

end if

for all $j \in \{2, \ldots, n - 1\} \setminus \{i\}$ **do**

  if $j = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$

  if $j = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$

  $(\text{TX}_{j,1}, B_{j,1}, (\text{TX}_{j,2,k}, B_{j,2,k})_{k \in \{m',\ldots,l'\} \setminus \{i\}}, (\text{TX}_{j,3,k_1,k_2}, B_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m',\ldots,i-1\} \times \{i+1,\ldots,l'\}}) \leftarrow$ GETMIDTXS(j, n, c_{virt}, $c_{j-1,\text{right}}$, $c_{j,\text{left}}$, $c_{j,\text{right}}$, $c_{j+1,\text{left}}$, $pk_{j-1,\text{right,fund,old}}$, $pk_{j,\text{left,fund,old}}$, $pk_{j,\text{right,fund,old}}$, $pk_{j+1,\text{left,fund,old}}$, $pk_{j-1,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j+1,\text{fund,new}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $pk_{j,\text{out}}$, $pk_{1,\text{rev}}$, $pk_{j-1,\text{rev}}$, $pk_{j,\text{rev}}$, $pk_{j+1,\text{rev}}$, $pk_{n,\text{rev}}$, $(pk_{k,p,s})_{k \in [n], p \in [n-1] \setminus \{1\}, s \in [n-1] \setminus \{1,p\}}$, $(pk_{k,2,1})_{k \in [n]}$, $(pk_{k,n-1,n})_{k \in [n]}$, $(t_k)_{k \in [n-1] \setminus \{1\}}$)

  if $j = i - 1$ **then**

    ensure that the following signatures are present in $\text{sigs}_{\text{byLeft}}$ and store them:

    - // 2 signatures

        sig($B_{i-1,1}$, $pk_{1,2,1}$), sig($B_{i-1,1}$, $pk_{i-1,2,1}$)

    - // $2(l' - m')$ signatures

        $\forall k \in \{m', \ldots, l'\} \setminus \{i\}$:

          sig($B_{i-1,2,k}$, $pk_{1,2,1}$), sig($B_{i-1,2,k}$, $pk_{i-1,2,1}$)

    - // $2(i - m') \cdot (l' - i)$ signatures

        $\forall k_1 \in \{m', \ldots, i - 1\}, \forall k_2 \in \{i + 1, \ldots, l'\}$:

          sig($B_{i-1,3,k_1,k_2}$, $pk_{1,2,1}$), sig($B_{i-1,3,k_1,k_2}$, $pk_{i-1,2,1}$)

  end if

  if $j < i$ **then** sigs $\leftarrow \text{sigs}_{\text{toLeft}}$ **else** sigs $\leftarrow \text{sigs}_{\text{toRight}}$

  if $j \in \{i - 1, i + 1\}$ **then**

    add SIGN($B_{j,1}$, $sk_{i,2,1}$, ANYPREVOUT) to sigs

  end if

  for all $k \in \{m', \ldots, l'\} \setminus \{j\}$ **do**

    add SIGN($\text{TX}_{j,2,k}$, $sk_{i,j,k}$, ANYPREVOUT) to sigs

    if $j \in \{i - 1, i + 1\}$ **then**

      add SIGN($B_{j,2,k}$, $sk_{i,2,1}$, ANYPREVOUT) to sigs

    end if

  end for

  for all $k_1 \in \{m', \ldots, j - 1\}, k_2 \in \{j + 1, \ldots, l'\}$ **do**

    add SIGN($\text{TX}_{j,3,k_1,k_2}$, sk_{i,j,k_1}, ANYPREVOUT) to sigs

    add SIGN($\text{TX}_{j,3,k_1,k_2}$, sk_{i,j,k_2}, ANYPREVOUT) to sigs

    if $j \in \{i - 1, i + 1\}$ **then**

      add SIGN($B_{j,3,k_1,k_2}$, $sk_{i,2,1}$, ANYPREVOUT) to sigs

    end if

  end for

end for

call $\bar{P}$.circulateVirtualSigs($\text{sigs}_{\text{toRight}}$) and assign returned value to $\text{sigs}_{\text{byRight}}$
output (virtualSigsBack, $\text{sigs}_{\text{toLeft}}$, $\text{sigs}_{\text{byRight}}$)

Figure 62

Process virt.intermediarySigs()

if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
if $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
$(\text{TX}_{i,1}, B_{i,1}, (\text{TX}_{i,2,k}, B_{i,2,k})_{k \in \{m,\dots,l\} \setminus \{i\}}, (\text{TX}_{i,3,k_1,k_2}, B_{i,3,k_1,k_2})_{(k_1,k_2) \in \{m,\dots,i-1\} \times \{i+1,\dots,l\}}) \leftarrow$ virt.getMidTXs(i, n, c_{virt}, $c_{i-1,\text{right}}$, $c_{i,\text{left}}$, $c_{i,\text{right}}$, $c_{i+1,\text{left}}$, $pk_{i-1,\text{right,fund,old}}$, $pk_{i,\text{left,fund,old}}$, $pk_{i,\text{right,fund,old}}$, $pk_{i+1,\text{left,fund,old}}$, $pk_{i-1,\text{fund,new}}$, $pk_{i,\text{fund,new}}$, $pk_{i+1,\text{fund,new}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $pk_{i,\text{out}}$, $pk_{1,\text{rev}}$, $pk_{j-1,\text{rev}}$, $pk_{j,\text{rev}}$, $pk_{j+1,\text{rev}}$, $pk_{n,\text{rev}}$, $(pk_{h,j,k})_{h \in [n], j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,j\}}$, $(pk_{h,2,1})_{h \in [n]}$, $(pk_{h,n-1,n})_{h \in [n]}$, $(t_h)_{h \in [n-1] \setminus \{1\}}$)
// not verifying our signatures in $\text{sigs}_{\text{byLeft}}$, our (trusted) `sibling` will do that
input (virtual sigs forward, $\text{sigs}_{\text{byLeft}}$) to `sibling`
virt.siblingSigs()
$\text{sigs}_{\text{toLeft}} \leftarrow \text{sigs}_{\text{byRight}} + \text{sigs}_{\text{toLeft}}$
if $i = 2$ **then** // previous hop is `host_funder`
    $\text{TX}_{1,1}, B_{1,1} \leftarrow$ virt.getEndpointTX(1, n, c_{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)
    add sign($B_{1,1}$, $sk_{i,2,1}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
end if
return $\text{sigs}_{\text{toLeft}}$

Figure 63

Process virt.hostFundeeSigs()

$\text{TX}_{n,1}, B_{n,1} \leftarrow$ virt.getEndpointTX(n, n, c_{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})
ensure that sig($B_{n,1}$, $pk_{1,2,1}$), sig($B_{n,1}$, $pk_{n-1,2,1}$) are present in $\text{sigs}_{\text{byLeft}}$ and store them
$\text{sigs}_{\text{toLeft}} \leftarrow \emptyset$
for all $j \in [n-1] \setminus \{1\}$ **do**
    if $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
    if $j = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
    $(\text{TX}_{j,1}, B_{j,1}, (\text{TX}_{j,2,k}, B_{j,2,k})_{k \in \{m,\dots,l\} \setminus \{j\}}, (\text{TX}_{j,3,k_1,k_2}, B_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m,\dots,j-1\} \times \{j+1,\dots,l\}}) \leftarrow$ virt.getMidTXs(j, n, c_{virt}, $c_{j-1,\text{right}}$, $c_{j,\text{left}}$, $c_{j,\text{right}}$, $c_{j+1,\text{left}}$, $pk_{j-1,\text{right,fund,old}}$, $pk_{j,\text{left,fund,old}}$, $pk_{j,\text{right,fund,old}}$, $pk_{j+1,\text{left,fund,old}}$, $pk_{j-1,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j+1,\text{fund,new}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $pk_{j,\text{out}}$, $pk_{1,\text{rev}}$, $pk_{j-1,\text{rev}}$, $pk_{j,\text{rev}}$, $pk_{j+1,\text{rev}}$, $pk_{n,\text{rev}}$, $(pk_{h,s,k})_{h \in [n], s \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,s\}}$, $(pk_{h,2,1})_{h \in [n]}$, $(pk_{h,n-1,n})_{h \in [n]}$, $(t_h)_{h \in [n-1] \setminus \{1\}}$)
    for all $k \in \{m, \dots, l\} \setminus \{j\}$ **do**
        ensure that sig($B_{j,2,k}$, $pk_{1,2,1}$), sig($B_{j,2,k}$, $pk_{n-1,2,1}$) are present in $\text{sigs}_{\text{byLeft}}$ and store them
        add sign($\text{TX}_{j,2,k}$, $sk_{n,j,k}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
        add sign($B_{j,2,k}$, $sk_{n,2,1}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
    end for
    for all $k_1 \in \{m, \dots, j-1\}, k_2 \in \{j+1, \dots, l\}$ **do**
        ensure that sig($B_{j,3,k_1,k_2}$, $pk_{1,2,1}$), sig($B_{j,3,k_1,k_2}$, $pk_{n-1,2,1}$) are present in $\text{sigs}_{\text{byLeft}}$ and store them
        add sign($\text{TX}_{j,3,k_1,k_2}$, sk_{n,j,k_1}, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
        add sign($\text{TX}_{j,3,k_1,k_2}$, sk_{n,j,k_2}, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
        add sign($B_{j,3,k_1,k_2}$, $sk_{n,2,1}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toLeft}}$
    end for
end for
return $\text{sigs}_{\text{toLeft}}$

Figure 64

Process virt.hostFunderSigs()

$\text{sigs}_{\text{toRight}} \leftarrow \emptyset$
for all $j \in [n-1] \setminus \{1\}$ **do**
    if $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
    if $j = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
    $(\text{TX}_{j,1}, B_{j,1}, (\text{TX}_{j,2,k}, B_{j,2,k})_{k \in \{m,\dots,l\} \setminus \{j\}}, (\text{TX}_{j,3,k_1,k_2}, B_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m,\dots,j-1\} \times \{j+1,\dots,l\}}) \leftarrow$ virt.getMidTXs(j, n, c_{virt}, $c_{j-1,\text{right}}$, $c_{j,\text{left}}$, $c_{j,\text{right}}$, $c_{j+1,\text{left}}$, $pk_{j-1,\text{right,fund,old}}$, $pk_{j,\text{left,fund,old}}$, $pk_{j,\text{right,fund,old}}$, $pk_{j+1,\text{left,fund,old}}$, $pk_{j-1,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j,\text{fund,new}}$, $pk_{j+1,\text{fund,new}}$, $pk_{\text{left,virt}}$, $pk_{\text{right,virt}}$, $pk_{j,\text{out}}$, $pk_{1,\text{rev}}$, $pk_{j-1,\text{rev}}$, $pk_{j,\text{rev}}$, $pk_{j+1,\text{rev}}$, $pk_{n,\text{rev}}$, $(pk_{h,s,k})_{h \in [n], s \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,s\}}$, $(pk_{h,2,1})_{h \in [n]}$, $(pk_{h,n-1,n})_{h \in [n]}$, $(t_h)_{h \in [n-1] \setminus \{1\}}$)
    for all $k \in \{m, \dots, l\} \setminus \{j\}$ **do**
        add sign($\text{TX}_{j,2,k}$, $sk_{1,j,k}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toRight}}$
        add sign($B_{j,2,k}$, $sk_{1,2,1}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toRight}}$
    end for
    for all $k_1 \in \{m, \dots, j-1\}, k_2 \in \{j+1, \dots, l\}$ **do**
        add sign($\text{TX}_{j,3,k_1,k_2}$, sk_{1,j,k_1}, `ANYPREVOUT`) to $\text{sigs}_{\text{toRight}}$
        add sign($\text{TX}_{j,3,k_1,k_2}$, sk_{1,j,k_2}, `ANYPREVOUT`) to $\text{sigs}_{\text{toRight}}$
        add sign($B_{j,3,k_1,k_2}$, $sk_{1,2,1}$, `ANYPREVOUT`) to $\text{sigs}_{\text{toRight}}$
    end for
end for
call virt.circulateVirtualSigs($\text{sigs}_{\text{toRight}}$) of $\bar{P}$ and assign output to $\text{sigs}_{\text{byRight}}$
$\text{TX}_{1,1}, B_{1,1} \leftarrow$ virt.getEndpointTX(1, n, c_{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)
ensure that sig($B_{1,1}$, $pk_{2,2,1}$), sig($B_{1,1}$, $pk_{n,2,1}$) are present in $\text{sigs}_{\text{byRight}}$ and store them
for all $j \in [n-1] \setminus \{1\}$ **do**
    if $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
    if $j = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
    for all $k \in \{m, \dots, l\} \setminus \{j\}$ **do**
        ensure that sig($B_{j,2,k}$, $pk_{2,2,1}$), sig($B_{j,2,k}$, $pk_{n,2,1}$) are present in $\text{sigs}_{\text{byRight}}$ and store them
    end for
    for all $k_1 \in \{m, \dots, j-1\}, k_2 \in \{j+1, \dots, l\}$ **do**

ensure that $\mathrm{sig}(B_{j,3,k_1,k_2}, pk_{2,2,1})$, $\mathrm{sig}(B_{j,3,k_1,k_2}, pk_{n,2,1})$ are present in $\mathrm{sigs}_{\mathrm{byRight}}$ and store them
end for
end for
return (OK)

Figure 65

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

if $1 < i < n$ **then** // we are not host_funder nor host_fundee
return VIRT.INTERMEDIARYSIGS()
else if $i = 1$ **then** // we are host_funder
return VIRT.HOSTFUNDERSIGS()
else if $i = n$ **then** // we are host_fundee
return VIRT.HOSTFUNDEESIGS()
end if // it is always $1 \leq i \leq n$ – cf. Fig. 58, l. 12 and l. 37

Figure 66

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

if $1 < i < n$ **then** // we are not endpoint
if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
if $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
ensure that the following signatures are present in $\mathrm{sigs}_{\mathrm{byLeft}}$ and store them:
- // 1 signature

$\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i-1,\mathrm{right,fund,old}})$
- // $n - 3 + \chi_{i=2} + \chi_{i=n-1}$ signatures

$\forall k \in \{m, \dots, l\} \setminus \{i\}$
$\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i-1,\mathrm{right,fund,old}})$
input (VIRTUAL BASE SIG FORWARD, $\mathrm{sigs}_{\mathrm{byLeft}}$) to sibling
extract and store $\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i-1,\mathrm{right,fund,old}})$ and $\forall k \in \{m, \dots, l\} \setminus \{i\}$ $\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i-1,\mathrm{right,fund,old}})$ from $\mathrm{sigs}_{\mathrm{byLeft}}$ // same signatures as sibling
$\mathrm{sigs}_{\mathrm{toRight}} \leftarrow \{\mathrm{SIGN}(\mathrm{TX}_{i+1,1}, sk_{i,\mathrm{right,fund,old}}, \mathrm{ANYPREVOUT})\}$
if $i + 1 < n$ **then**
if $i + 1 = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$
for all $k \in \{2, \dots, l'\}$ **do**
add $\mathrm{SIGN}(\mathrm{TX}_{i+1,2,k}, sk_{i,\mathrm{right,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toRight}}$
end for
else // $i + 1 = n$
add $\mathrm{SIGN}(\mathrm{TX}_{n,1}, sk_{i,\mathrm{right,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toRight}}$
end if
call VIRT.CIRCULATEFUNDINGSIGS($\mathrm{sigs}_{\mathrm{toRight}}$) of $\bar{P}$ and assign returned values to $\mathrm{sigs}_{\mathrm{byRight}}$
ensure that the following signatures are present in $\mathrm{sigs}_{\mathrm{byRight}}$ and store them:
- // 1 signature

$\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i+1,\mathrm{left,fund,old}})$
- // $n - 3 + \chi_{i=2} + \chi_{i=n-1}$ signatures

$\forall k \in \{m, \dots, l\} \setminus \{i\}$
$\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i+1,\mathrm{right,fund,old}})$
output (VIRTUAL BASE SIG BACK, $\mathrm{sigs}_{\mathrm{byRight}}$)
extract and store $\mathrm{sig}(\mathrm{TX}_{i,1}, pk_{i+1,\mathrm{right,fund,old}})$ and $\forall k \in \{m, \dots, l\} \setminus \{i\}$ $\mathrm{sig}(\mathrm{TX}_{i,2,k}, pk_{i+1,\mathrm{right,fund,old}})$ from $\mathrm{sigs}_{\mathrm{byRight}}$ // same signatures as sibling
$\mathrm{sigs}_{\mathrm{toLeft}} \leftarrow \{\mathrm{SIGN}(\mathrm{TX}_{i-1,1}, sk_{i,\mathrm{left,fund,old}}, \mathrm{ANYPREVOUT})\}$
if $i - 1 > 1$ **then**
if $i - 1 = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$
for all $k \in \{m', \dots, n - 1\}$ **do**
add $\mathrm{SIGN}(\mathrm{TX}_{i-1,2,k}, sk_{i,\mathrm{left,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toLeft}}$
end for
else // $i - 1 = 1$
add $\mathrm{SIGN}(\mathrm{TX}_{1,1}, sk_{i,\mathrm{left,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toLeft}}$
end if
return $\mathrm{sigs}_{\mathrm{toLeft}}$
else if $i = 1$ **then** // we are host_funder
$\mathrm{sigs}_{\mathrm{toRight}} \leftarrow \{\mathrm{SIGN}(\mathrm{TX}_{2,1}, sk_{1,\mathrm{right,fund,old}}, \mathrm{ANYPREVOUT})\}$
if $2 = n - 1$ **then** $l' \leftarrow n$ **else** $l' \leftarrow n - 1$
for all $k \in \{3, \dots, l'\}$ **do**
add $\mathrm{SIGN}(\mathrm{TX}_{2,2,k}, sk_{1,\mathrm{right,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toRight}}$
end for
call VIRT.CIRCULATEFUNDINGSIGS($\mathrm{sigs}_{\mathrm{toRight}}$) of $\bar{P}$ and assign returned value to $\mathrm{sigs}_{\mathrm{byRight}}$
ensure that $\mathrm{sig}(\mathrm{TX}_{1,1}, pk_{2,\mathrm{left,fund,old}})$ is present in $\mathrm{sigs}_{\mathrm{byRight}}$ and store it
return (OK)
else if $i = n$ **then** // we are host_fundee
ensure $\mathrm{sig}(\mathrm{TX}_{n,1}, pk_{n-1,\mathrm{right,fund,old}})$ is present in $\mathrm{sigs}_{\mathrm{byLeft}}$ and store it
$\mathrm{sigs}_{\mathrm{toLeft}} \leftarrow \{\mathrm{SIGN}(\mathrm{TX}_{n-1,1}, sk_{n,\mathrm{left,fund,old}}, \mathrm{ANYPREVOUT})\}$
if $n - 1 = 2$ **then** $m' \leftarrow 1$ **else** $m' \leftarrow 2$
for all $k \in \{m', \dots, n - 2\}$ **do**
add $\mathrm{SIGN}(\mathrm{TX}_{n-1,2,k}, sk_{n,\mathrm{left,fund,old}}, \mathrm{ANYPREVOUT})$ to $\mathrm{sigs}_{\mathrm{toLeft}}$
end for
return $\mathrm{sigs}_{\mathrm{toLeft}}$
end if // it is always $1 \leq i \leq n$ – cf. Fig. 58, l. 12 and l. 37

Figure 67

Process VIRT.CIRCULATEREVOCATIONS(revoc_by_prev)

if revoc_by_prev is given as argument **then** // we are not host_funder
ensure guest.PROCESSREMOTEREVOCATION(revoc_by_prev) returns (OK)
else // we are host_funder
revoc_for_next ← guest.REVOKEPREVIOUS()
input (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign ouput to Σ
last_poll ← $|\Sigma|$
call VIRT.CIRCULATEREVOCATIONS(revoc_for_next) of $\bar{P}$ and assign returned value to revoc_by_next
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)
return (OK)

end if
if we have a `sibling` **then** // we are not `host_fundee` nor `host_funder`
input (VIRTUAL REVOCATION FORWARD) to `sibling`
`revoc_for_next` ← `guest`.REVOKEPREVIOUS()
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign ouput to Σ
`last_poll` ← $|\Sigma|$
call VIRT.CIRCULATEREVOCATIONS(`revoc_for_next`) of $\bar{P}$ and assign output to `revoc_by_next`
ensure `guest`.PROCESSREMOTEREVOCATION(`revoc_by_next`) returns (OK)
output (HOSTS READY, t_i) to `guest` and expect reply (HOST ACK)
output (VIRTUAL REVOCATION BACK)
end if
`revoc_for_prev` ← `guest`.REVOKEPREVIOUS()
if $1 < i < n$ **then** // we are intermediary
output (HOSTS READY, t_i) to `guest` and expect reply (HOST ACK)
// p is every how many blocks we have to check the chain
else // we are `host_fundee`, case of `host_funder` covered earlier
output (HOSTS READY, $p + \sum_{j=2}^{n-1}(s - 1 + t_j)$) to `guest` and expect reply (HOST ACK)
end if
return `revoc_for_prev`

Figure 68

Process VIRT – poll

On input (CHECK FOR LATERAL CLOSE) by $R \in \{$`guest`, `funder`, `fundee`$\}$:
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
$k_1 \leftarrow 0$
if $\text{TX}_{i-1,1}$ is defined and $\text{TX}_{i-1,1} \in \Sigma$ **then**
$k_1 \leftarrow i - 1$
end if
for all $k \in [i-2]$ **do**
if $\text{TX}_{i-1,2,k}$ is defined and $\text{TX}_{i-1,2,k} \in \Sigma$ **then**
$k_1 \leftarrow k$
end if
end for
$k_2 \leftarrow 0$
if $\text{TX}_{i+1,1}$ is defined and $\text{TX}_{i+1,1} \in \Sigma$ **then**
$k_2 \leftarrow i + 1$
end if
for all $k \in \{i+2, \dots, n\}$ **do**
if $\text{TX}_{i+1,2,k}$ is defined and $\text{TX}_{i+1,2,k} \in \Sigma$ **then**
$k_2 \leftarrow k$
end if
end for
`last_poll` ← $|\Sigma|$
if $k_1 > 0 \lor k_2 > 0$ **then** // at least one neighbour has published its TX
ignore all messages except for (CHECK IF CLOSING) by R
State ← CLOSING
sigs ← $\emptyset$
end if
if $k_1 > 0 \land k_2 > 0$ **then** // both neighbours have published their TXs
add $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$ to sigs
add $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$ to sigs
add SIGN($\text{TX}_{i,3,k_1,k_2}$, sk_{i,i,k_1}, ANYPREVOUT) to sigs
add SIGN($\text{TX}_{i,3,k_1,k_2}$, sk_{i,i,k_2}, ANYPREVOUT) to sigs
input (SUBMIT, $\text{TX}_{i,3,k_1,k_2}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
else if $k_1 > 0$ **then** // only left neighbour has published its TX
add $(\text{sig}(\text{TX}_{i,2,k_1}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$ to sigs
add SIGN($\text{TX}_{i,2,k_1}$, sk_{i,i,k_1}, ANYPREVOUT) to sigs
add SIGN($\text{TX}_{i,2,k_1}$, $sk_{i,\text{left,fund,old}}$, ANYPREVOUT) to sigs
input (SUBMIT, $\text{TX}_{i,2,k_1}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
else if $k_2 > 0$ **then** // only right neighbour has published its TX
add $(\text{sig}(\text{TX}_{i,2,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$ to sigs
add SIGN($\text{TX}_{i,2,k_2}$, sk_{i,i,k_2}, ANYPREVOUT) to sigs
add SIGN($\text{TX}_{i,2,k_2}$, $sk_{i,\text{right,fund,old}}$, ANYPREVOUT) to sigs
input (SUBMIT, $\text{TX}_{i,2,k_2}$, sigs) to $\mathcal{G}_{\text{Ledger}}$
end if

On input (CHECK FOR REVOKED) by $R \in \{$`guest`, `funder`, `fundee`$\}$:
input (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign output to Σ
if $\text{TX}_{i-1,1} \in \Sigma \lor \exists k \in \mathbb{N} : \text{TX}_{i-1,2,k} \in \Sigma$ **then** // left counterparty maliciously published old virtual tx
if $\exists k \in \mathbb{N} : \text{TX}_{i-1,2,k} \in \Sigma$ **then** // exactly one of the two pairs is valid. That is OK
$(R_a, sk_a, R_b, sk_b) \leftarrow (R_{i-1,2,k}, sk_{i,2,1}, R_{\text{loc,left,virt}}, sk_{i,\text{rev}})$
else
$(R_a, sk_a, R_b, sk_b) \leftarrow (R_{i-1,1}, sk_{i,2,1}, R_{\text{loc,left,virt}}, sk_{i,\text{rev}})$
end if
input (SUBMIT, $(R_a, R_b, R_{\text{loc,left,virt}}, R_{\text{loc,left,fund}})$, (SIGN($R_a$, sk_a), (SIGN(R_b, sk_b), SIGN($R_{\text{loc,left,virt}}$, $sk_{i,\text{rev}}$), SIGN($R_{\text{loc,left,fund}}$, $sk_{i,\text{rev}}$))) to $\mathcal{G}_{\text{Ledger}}$
end if
if $\text{TX}_{i+1,1} \in \Sigma \lor \exists k \in \mathbb{N} : \text{TX}_{i+1,2,k} \in \Sigma$ **then** // right counterparty maliciously published old virtual tx
input (SUBMIT, $(R_{\text{loc,right,virt}}, R_{\text{loc,right,fund}})$, (SIGN($R_{\text{loc,right,virt}}$, $sk_{i,\text{rev}}$), SIGN($R_{\text{loc,right,fund}}$, $sk_{i,\text{rev}}$))) to $\mathcal{G}_{\text{Ledger}}$
end if
output (NOTHING REVOKED) to R

Figure 69

Process VIRT – On input (FORCECLOSE) by R:

// At most one of `funder`, `fundee` is defined
ensure $R \in \{$`guest`, `funder`, `fundee`$\}$
if *State* = CLOSED **then** output (CLOSED) to R
if *State* = GUEST PUNISHED **then** output (GUEST PUNISHED) to R
ensure *State* $\in$ {OPEN, CLOSING}
if `host`$_P$ $\neq \mathcal{G}_{\text{Ledger}}$ **then** // `host`$_P$ is a VIRT
ignore all messages except for output (CLOSED) by `host`$_P$. Also relay to `host`$_P$ any (CHECK IF CLOSING) or (FORCECLOSE) input received
input (FORCECLOSE) to `host`$_P$
end if
// if we have a `host`$_P$, continue from here on output (CLOSED) by it

send (READ) to $\mathcal{G}_{\text{Ledger}}$ as R and assign reply to Σ
if $i \in \{1, n\} \wedge (\text{TX}_{(i-1)+\frac{2}{n-1}(n-i),1} \in \Sigma \vee \exists k \in [n] : \text{TX}_{(i-1)+\frac{2}{n-1}(n-i),2,k} \in \Sigma)$ **then** // we are an endpoint and our counterparty has closed – 1st subscript of TX is 2 if $i = 1$ and $n - 1$ if $i = n$
    ignore all messages except for (CHECK IF CLOSING) and (FORCECLOSE) by R
    State ← CLOSING
    give up execution token // control goes to $\mathcal{E}$
end if
let TX_p be the unique transaction 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 Σ in a valid way // ignore invalid subscript combinations
let `sigs` be the set of stored signatures that sign TX_p
add SIGN(TX_p, $sk_{i,\text{left,fund,old}}$, ANYPREVOUT), SIGN($\text{TX}_p$, $sk_{i,\text{right,fund,old}}$, ANYPREVOUT), $(\text{SIGN}(\text{TX}_p, sk_{i,j,k}, \text{ANYPREVOUT}))_{j,k \in [n]}$ to `sigs` // ignore invalid signatures
ignore all messages except for (CHECK IF CLOSING) by R
State ← CLOSING
send (SUBMIT, TX_p, `sigs`) to $\mathcal{G}_{\text{Ledger}}$

Figure 70

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

ensure *State* = CLOSING
ensure $R \in \{$`guest`, `funder`, `fundee`$\}$
send (READ) to $\mathcal{G}_{\text{Ledger}}$ as R and assign reply to Σ
if $i = 1$ **then** // we are `host_funder`
    ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\text{virt}}$ coins and a $2/\{pk_{1,\text{fund,new}}, pk_{2,\text{fund,new}}\}$ spending method with expired/non-existent timelock in Σ // new base funding output
    ensure that there either exists an output with c_{virt} coins and a $2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$ spending method with expired/non-existent timelock in Σ /*virtual funding output by a "bridge" tx*/ or a $\texttt{bridge}_p$ output. In the latter case, collect all B_p's signatures in `sigs`, add SIGN(B_p, $sk_{1,2,1}$, ANYPREVOUT) (or, if $p = n$, 1, SIGN(B_p, $sk_{1,n-1,n}$, ANYPREVOUT) instead) to `sigs`, send (SUBMIT, B_p, `sigs`) to $\mathcal{G}_{\text{Ledger}}$ and keep waiting here for (CHECK IF CLOSING) by R until B_p is in Σ returned by sending (READ) to $\mathcal{G}_{\text{Ledger}}$.
else if $i = n$ **then** // we are `host_fundee`
    ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\text{virt}}$ coins and a $2/\{pk_{n-1,\text{fund,new}}, pk_{n,\text{fund,new}}\}$ spending method with expired/non-existent timelock in Σ // new base funding output
    ensure that there either exists an output with c_{virt} coins and a $2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$ spending method with expired/non-existent timelock in Σ /*virtual funding output by a "bridge" tx*/ or a $\texttt{bridge}_p$ output. In the latter case, collect all B_p's signatures in `sigs`, add SIGN(B_p, $sk_{1,2,1}$, ANYPREVOUT) (or, if $p = n$, 1, SIGN(B_p, $sk_{1,n-1,n}$, ANYPREVOUT) instead) to `sigs`, send (SUBMIT, B_p, `sigs`) to $\mathcal{G}_{\text{Ledger}}$ and keep waiting here for (CHECK IF CLOSING) by R until B_p is in Σ returned by sending (READ) to $\mathcal{G}_{\text{Ledger}}$.
else // we are intermediary
    if `side` = "left" **then** $j \leftarrow i - 1$ **else** $j \leftarrow i + 1$ // `side` is defined for all intermediaries – cf. Fig. 58, l. 11
    ensure that there exists an output with $c_P + c_{\bar{P}} - c_{\text{virt}}$ coins and a $2/\{pk_{i,\text{fund,new}}, pk_{j,\text{fund,new}}\}$ spending method with expired/non-existent timelock in Σ
    ensure that there either exists an output with c_{virt} coins and a $pk_{i,\text{out}}$ spending method with expired/non-existent timelock in Σ /*virtual funding output by a "bridge" tx*/ or a $\texttt{bridge}_{i-1,p}$ output. In the latter case, collect all $B_{i-1,p}$'s signatures in `sigs`, add SIGN($B_{i-1,p}$, $sk_{1,2,1}$, ANYPREVOUT) (or, if $i - 1, p = n$, 1, SIGN($B_{i-1,p}$, $sk_{1,n-1,n}$, ANYPREVOUT) instead) to `sigs`, send (SUBMIT, $B_{i-1,p}$, `sigs`) to $\mathcal{G}_{\text{Ledger}}$ and keep waiting here for (CHECK IF CLOSING) by R until $B_{i-1,p}$ is in Σ returned by sending (READ) to $\mathcal{G}_{\text{Ledger}}$.
end if
State ← CLOSED
output (CLOSED) to R

Figure 71

Process VIRT – cooperative closing

// we are left intermediary or host of fundee
On (COOP CLOSE, `sig_bal`, `left_comms_revkeys`) by $\bar{P}$:
ensure *State* = OPEN
parse `sig_bal` as (c'_1, c'_2), sig_1, sig_2
ensure $c_{\text{virt}} = c'_1 + c'_2$
ensure VERIFY((c'_1, c'_2), sig_1, $pk_{\text{left,virt}}$) = True
ensure VERIFY((c'_1, c'_2), sig_2, $pk_{\text{right,virt}}$) = True
State ← COOP CLOSING
extract $\text{sig}_{i-1,\text{right},C}$, $pk_{i-1,\text{right},R}$ from `left_comms_revkeys`
if $i < n$ **then** M ← CHECK COOP CLOSE **else** M ← CHECK COOP CLOSE FUNDEE
output (M, (c'_1, c'_2), $\text{sig}_{i-1,\text{right},C}$, $pk_{i-1,\text{right},R}$) to `guest`
ensure *State* = OPEN // executed by `guest`
State ← COOP CLOSING
store received signature as $\text{sig}_{\bar{P},C,i+1}$ // in guests, i is the current state number
store received revocation key as $pk_{\bar{P},R,i+2}$
remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$ and $pk'_{\bar{P},F}$
$C_{P,i+1}$ ← TX {input: $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c'_2, (pk_{P,\text{out}} + (p + s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\})$, $(c_{\bar{P}} + c'_1, pk_{\bar{P},\text{out}})$}
ensure VERIFY($C_{P,i+1}$, $\text{sig}_{\bar{P},C,i+1}$, $pk'_{\bar{P},F}$) = True
input (COOP CLOSE CHECK OK) to host_P
if $i < n$ **then** // we are intermediary
    input (COOP CLOSE, `left_comms_keys`) to `sibling`
    ensure *State* = OPEN // executed by `sibling`
    State ← COOP CLOSING
    output (COOP CLOSE SIGN COMM, (c'_1, c'_2)) to `guest`
    ensure *State* = OPEN // executed by `guest` of `sibling`
    State ← COOP CLOSING
    remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}$, $pk'_{P,F}$ and $pk'_{\bar{P},F}$
    $C_{\bar{P},i+1}$ ← TX {input: $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c'_1, pk_{P,\text{out}})$, $(c_{\bar{P}} + c'_2, (pk_{\bar{P},\text{out}} + (p + s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$}

$\mathsf{sig}_{P,C,i+1} \leftarrow \textsc{sign}(C_{\bar{P},i+1}, sk'_{P,F})$
$(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \textsc{KeyGen}()$
input ($\textsc{new comm tx}$, $\mathsf{sig}_{P,C,i+1}$, $pk_{P,R,i+2}$) to $\texttt{host}_P$
rename received signature to $\mathsf{sig}_{i,\text{right},C}$ // executed by $\texttt{sibling}$
rename received public key to $pk_{i,\text{right},R}$ // in hosts, i is our hop number
send ($\textsc{coop close}$, $\texttt{sig\_bal}$, ($\texttt{left\_comms\_keys}$, $\mathsf{sig}_{i,\text{right},C}$, $pk_{i,\text{right},R}$) to $\bar{P}$ and expect reply ($\textsc{coop close back}$, ($\texttt{right\_comms\_revkeys}$, $\texttt{right\_revocations}$))
$R_{\text{loc,right,virt}} \leftarrow$ TX {input: $(c_{\text{virt}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i,\text{out}})$}
extract $\mathsf{sig}_{i+1,\text{right,rev,virt}}$ from $\texttt{right\_revocations}$
ensure $\textsc{verify}(R_{\text{loc,right,virt}}, \mathsf{sig}_{i+1,\text{right,rev,virt}}, pk_{i+1,\text{rev}}) = \text{True}$
$R_{\text{loc,right,fund}} \leftarrow$ TX {input: $(c_P + c_{\bar{P}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{i,\text{out}})$}
extract $\mathsf{sig}_{i+1,\text{right,rev,fund}}$ from $\texttt{right\_revocations}$
ensure $\textsc{verify}(R_{\text{loc,right,fund}}, \mathsf{sig}_{i+1,\text{right,rev,fund}}, pk_{i+1,\text{rev}}) = \text{True}$
extract $\mathsf{sig}_{i+1,\text{left},C}$ from $\texttt{right\_comms\_revkeys}$
extract $\mathsf{sig}_{i+1,\text{left},R}$ from $\texttt{right\_revocations}$
extract $pk_{i+1,\text{left},R}$ from $\texttt{right\_comms\_revkeys}$
output ($\textsc{verify comm rev}$, $\mathsf{sig}_{i+1,\text{left},C}$, $\mathsf{sig}_{i+1,\text{left},R}$, $pk_{i+1,\text{left},R}$) to $\texttt{guest}$
store received public key as $pk_{\bar{P},R,i+2}$ // executed by $\texttt{guest}$ of $\texttt{sibling}$
store $\mathsf{sig}_{i+1,\text{left},C}$ as $\mathsf{sig}_{\bar{P},C,i+1}$, $pk_{\bar{P},R,i+2}$
$C_{P,i+1} \leftarrow$ TX {input: $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs: $(c_P + c_1, (pk_{P,\text{out}} + (p+s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})$, $(c_{\bar{P}} + c'_2, pk_{\bar{P},\text{out}})$}
ensure $\textsc{verify}(C_{P,i+1}, \mathsf{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = \text{True}$
store $\mathsf{sig}_{i+1,\text{left},R}$ as $\mathsf{sig}_{\bar{P},R,i}$
$R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: $(c_P + c_{\bar{P}}, pk_{P,\text{out}})$}
ensure $\textsc{verify}(R_{P,i}, \mathsf{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = \text{True}$
input ($\textsc{comm rev verified}$) to $\texttt{host}_P$
output ($\textsc{coop close back}$, $\texttt{right\_comms\_revkeys}$, $\texttt{right\_revocations}$) to $\texttt{sibling}$ // executed by $\texttt{sibling}$
$R_{\text{loc,left,virt}} \leftarrow$ 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 $\bar{P}$
extract $\mathsf{sig}_{n,i,\text{left,rev,virt}}$ from $\texttt{right\_revocations}$
ensure $\textsc{verify}(R_{\text{loc,left}}, \mathsf{sig}_{n,\text{left,rev}}, pk_{n,\text{rev}}) = \text{True}$
if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
if $i = n-1$ **then** $l \leftarrow n$ **else** $l \leftarrow n-1$
ensure that the following signatures are present in $\texttt{right\_revocations}$ and store them:

- // 1 signature

$\mathrm{sig}(R_{i-1,1}, pk_{n,\text{rev}})$

- // $l - m$ signatures

$\forall k \in \{m, \ldots, l\} \setminus \{i\}$:
$\mathrm{sig}(R_{i-1,2,k}, pk_{n,\text{rev}})$

- // $(i-m)\cdot(l-i)$ signatures

$\forall k_1 \in \{m, \ldots, i-1\}, \forall k_2 \in \{i+1, \ldots, l\}$:
$\mathrm{sig}(R_{i-1,3,k_1,k_2}, pk_{n,\text{rev}})$

else // $i = n$, we are host of fundee
output ($\textsc{revoke}$) to $\texttt{fundee}$
$R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.P, output: $(c_P, pk_{\bar{P},\text{out}})$} // executed by $\texttt{fundee}$
$\mathsf{sig}_{P,R,i} \leftarrow \textsc{sign}(R_{\bar{P},i}, sk_{P,R,i})$
$\texttt{virtual\_revocation\_sigs} \leftarrow \emptyset$
for $j \in [n-1]$ **do**
$R_{j,1} \leftarrow$ TX {input: $\mathrm{TX}_{j,1}.\texttt{revocation}_1$, output: $(c_{\text{virt}}, pk_{j+1,\text{out}})$}
$\mathsf{sig}_{j,R,1,i} \leftarrow \textsc{sign}(R_{j,1}, sk_{i,\text{rev}})$; $\texttt{virtual\_revocation\_sigs} \leftarrow \texttt{virtual\_revocation\_sigs} \cup \mathsf{sig}_{j,R,1,i}$
if $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
if $j = n-1$ **then** $l \leftarrow n$ **else** $l \leftarrow n-1$
for $k \in \{m, \ldots, l\}$ **do**
$R_{j,2,k} \leftarrow$ TX {input: $\mathrm{TX}_{j,2,k}.\texttt{revocation}_{2,k}$, output: $(c_{\text{virt}}, pk_{j+1,\text{out}})$}
$\mathsf{sig}_{j,R,2,k,i} \leftarrow \textsc{sign}(R_{j,2,k}, sk_{i,\text{rev}})$; $\texttt{virtual\_revocation\_sigs} \leftarrow \texttt{virtual\_revocation\_sigs} \cup \mathsf{sig}_{j,R,2,k,i}$
end for
for $k_1 \in \{m, \ldots, j-1\}, k_2 \in \{j+1, \ldots, l\}$ **do**
$R_{j,3,k_1,k_2} \leftarrow$ TX {input: $\mathrm{TX}_{j,3,k_1,k_2}.\texttt{revocation}_{3,k_1,k_2}$, output: $(c_{\text{virt}}, pk_{j+1,\text{out}})$}
$\mathsf{sig}_{j,R,3,k_1,k_2,i} \leftarrow \textsc{sign}(R_{j,3,k_1,k_2}, sk_{i,\text{rev}})$; $\texttt{virtual\_revocation\_sigs} \leftarrow \texttt{virtual\_revocation\_sigs} \cup \mathsf{sig}_{j,R,3,k_1,k_2,i}$
end for
end for
input ($\textsc{revocations}$, $\mathsf{sig}_{P,R,i}$, $\texttt{virtual\_revocation\_sigs}$) to $\texttt{host}_P$
rename received signature $\mathsf{sig}_{P,R,i}$ to $\mathsf{sig}_{n,\text{right},R}$
for all $j \in \{2, \ldots, n\}$ **do**
$R_{j,\text{left}} \leftarrow$ TX {input: $(c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{n,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{j,\text{out}})$}
$\mathsf{sig}_{n,j,\text{left,rev}} \leftarrow \textsc{sign}(R_{j,\text{left}}, sk_{n,\text{rev}})$
end for
end if
output ($\textsc{new comm rev}$) to $\texttt{guest}$
$C_{\bar{P},i+1} \leftarrow$ 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 $\texttt{guest}$
$\mathsf{sig}_{P,C,i+1} \leftarrow \textsc{sign}(C_{\bar{P},i+1}, sk'_{P,F})$
$R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.P, output: $(c_P + c_{\bar{P}}, pk_{\bar{P},\text{out}})$}
$\mathsf{sig}_{P,R,i} \leftarrow \textsc{sign}(R_{\bar{P},i}, sk_{P,R,i})$
$(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \textsc{KeyGen}()$
input ($\textsc{new comm rev}$, $\mathsf{sig}_{P,C,i+1}$, $\mathsf{sig}_{P,R,i}$, $pk_{P,R,i+2}$) to $\texttt{host}_P$
rename $\mathsf{sig}_{P,C,i+1}$ to $\mathsf{sig}_{i,\text{left},C}$
rename $\mathsf{sig}_{P,R,i}$ to $\mathsf{sig}_{i,\text{left},R}$
rename received public key to $pk_{i,\text{left},R}$
$R_{\text{rem,left,virt}} \leftarrow$ TX {input: $(c_{\text{virt}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i-1,\text{out}})$}
$\mathsf{sig}_{i,\text{left,rev,virt}} \leftarrow \textsc{sign}(R_{\text{rem,left,virt}}, sk_{i,\text{rev}})$
$R_{\text{rem,left,fund}} \leftarrow$ TX {input: $(c_P + c_{\bar{P}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{i-1,\text{out}})$}
$\mathsf{sig}_{i,\text{left,rev,fund}} \leftarrow \textsc{sign}(R_{\text{rem,left,fund}}, sk_{i,\text{rev}})$
if $i < n$ **then** // we are intermediary
$M \leftarrow$ ($\textsc{coop close back}$, (($\texttt{right\_comms\_revkeys}$, $\mathsf{sig}_{i,\text{left},C}$, $pk_{i,\text{left},R}$), ($\texttt{right\_revocations}$, $\mathsf{sig}_{i,\text{left,rev,virt}}$, $\mathsf{sig}_{i,\text{left,rev,fund}}$, $\mathsf{sig}_{i,\text{left},R}$)))
else // $i = n$, we are host of fundee
$M \leftarrow$ ($\textsc{coop close back}$, ($\mathsf{sig}_{i,\text{left},C}$, $pk_{i,\text{left},R}$, $\mathsf{sig}_{n,\text{left},R}$), ($\mathsf{sig}_{n,\text{left,rev,virt}}$, $\mathsf{sig}_{n,\text{left,rev,fund}}$, $(\mathsf{sig}_{n,j,\text{left,rev}})_{j \in \{2,\ldots,n\}}$), $\texttt{virtual\_rev\_sigs}$)
end if

send M to $\bar{P}$ and expect reply (COOP CLOSE REVOCATIONS, `left_revocations`)
extract $\mathrm{sig}_{i-1,\mathrm{right},R}$, $\mathrm{sig}_{1,i,\mathrm{right,rev}}$, $\mathrm{sig}_{i-1,\mathrm{right,rev}}$ from `left_revocations`
ensure VERIFY($R_{\mathrm{loc,left,virt}}$, $\mathrm{sig}_{1,\mathrm{right,rev}}$, $pk_{1,\mathrm{rev}}$) = True
ensure VERIFY($R_{\mathrm{loc,left,virt}}$, $\mathrm{sig}_{i-1,\mathrm{right,rev}}$, $pk_{i-1,\mathrm{rev}}$) = True
$R_{\mathrm{loc,left,fund}} \leftarrow$ TX {input: $(c_P + c_{\bar{P}}, 2/\{pk_{i-1,\mathrm{rev}}, pk_{i,\mathrm{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{i,\mathrm{out}})$} // the input corresponds to the revocation path of the right funding output of all virtual txs owned by $\bar{P}$
extract $\mathrm{sig}_{i-1,\mathrm{left,rev,fund}}$ from `left_revocations`
ensure VERIFY($R_{\mathrm{loc,left,fund}}$, $\mathrm{sig}_{i-1,\mathrm{left,rev,fund}}$, $pk_{i-1,\mathrm{rev}}$) = True
if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
if $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
ensure that the following signatures are present in `left_revocations` and store them:

- // 2 signatures

    $\mathrm{sig}(R_{i-1,1}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{i-1,1}, pk_{i-1,\mathrm{rev}})$

- // $2(l - m)$ signatures

    $\forall k \in \{m, \dots, l\} \setminus \{i\}$:
        $\mathrm{sig}(R_{i-1,2,k}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{i-1,2,k}, pk_{i-1,\mathrm{rev}})$

- // $2(i - m) \cdot (l - i)$ signatures

    $\forall k_1 \in \{m, \dots, i-1\}, \forall k_2 \in \{i+1, \dots, l\}$:
        $\mathrm{sig}(R_{i-1,3,k_1,k_2}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{i-1,3,k_1,k_2}, pk_{i-1,\mathrm{rev}})$

output (VERIFY REV, $\mathrm{sig}_{i-1,\mathrm{right},R}$, `host`$_P$) to `guest`
store received signature as $\mathrm{sig}_{barP,R,i}$ // executed by `guest`
$R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: $(c_P + c_{\bar{P}}, pk_{P,\mathrm{out}})$}
ensure VERIFY($R_{P,i}$, $\mathrm{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
add `host`$_P$ to list of old hosts
assign received host to `host`$_P$
$i \leftarrow i + 1$; $c_P \leftarrow c_P + c'_2$; $c_{\bar{P}} \leftarrow c_{\bar{P}} + c'_1$
add $sk_{P,F}, pk_{P,F}, pk_{\bar{P},F}$ to list of old enabler channel funding keys
$(sk_{P,F}, pk_{P,F}) \leftarrow (sk'_{P,F}, pk'_{P,F})$
`layer` $\leftarrow$ `layer` $- 1$
`locked`$_P \leftarrow$ `locked`$_P - c_{\mathrm{virt}}$
State $\leftarrow$ OPEN
`hosting` $\leftarrow$ False
input (REV VERIFIED) to last old host
State $\leftarrow$ COOP CLOSED
if $i < n$ **then** // we are intermediary
    send (COOP CLOSE REVOCATIONS, `left_revocations`) to `sibling`
    output (COOP CLOSE REVOCATIONS, `host`$_P$) to `guest` // executed by `sibling`
    $R_{\bar{P},i} \leftarrow$ TX {input: $C_{P,i}$.outputs.P, output: $(c_P, pk_{\bar{P},\mathrm{out}})$} // executed by `guest` of `sibling`
    $\mathrm{sig}_{P,R,i} \leftarrow$ SIGN($R_{\bar{P},i}$, $sk_{P,R,i}$)
    add `host`$_P$ to list of old hosts
    assign received host to `host`$_P$
    $i \leftarrow i + 1$; $c_P \leftarrow c_P + c'_1$; $c_{\bar{P}} \leftarrow c_{\bar{P}} + c'_2$
    add $sk_{P,F}, pk_{P,F}, pk_{\bar{P},F}$ to list of old enabler channel funding keys
    $(sk_{P,F}, pk_{P,F}) \leftarrow (sk'_{P,F}, pk'_{P,F})$
    `layer` $\leftarrow$ `layer` $- 1$
    `locked`$_P \leftarrow$ `locked`$_P - c_{\mathrm{virt}}$
    State $\leftarrow$ OPEN
    `hosting` $\leftarrow$ False
    input (REVOCATION, $\mathrm{sig}_{P,R,i}$) to last old host
    rename received signature to $\mathrm{sig}_{i,\mathrm{right},R}$ // executed by `sibling`
    $R_{\mathrm{rem,right,virt}} \leftarrow$ TX {input: $(c_{\mathrm{virt}}, 4/\{pk_{1,\mathrm{rev}}, pk_{i,\mathrm{rev}}, pk_{i+1,\mathrm{rev}}, pk_{n,\mathrm{rev}}\})$, output: $(c_{\mathrm{virt}}, pk_{i+1,\mathrm{out}})$}
    $\mathrm{sig}_{i,\mathrm{right,rev,virt}} \leftarrow$ SIGN($R_{\mathrm{rem,right,virt}}$, $sk_{i,\mathrm{rev}}$)
    $R_{\mathrm{rem,right,fund}} \leftarrow$ TX {input: $(c_P + c_{\bar{P}}, 2/\{pk_{i,\mathrm{rev}}, pk_{i+1,\mathrm{rev}}\})$, output: $(c_P + c_{\bar{P}}, pk_{i+1,\mathrm{out}})$}
    $\mathrm{sig}_{i,\mathrm{right,rev,fund}} \leftarrow$ SIGN($R_{\mathrm{rem,right,fund}}$, $sk_{i,\mathrm{rev}}$)
    $R_{i,1} \leftarrow$ TX {input: $\mathrm{TX}_{i,1}$.`revocation`$_1$, output: $(c_{\mathrm{virt}}, pk_{i+1,\mathrm{out}})$}
    $\mathrm{sig}_{i,R,1,i} \leftarrow$ SIGN($R_{i,1}$, $sk_{i,\mathrm{rev}}$); `left_revocations` $\leftarrow$ `left_revocations` $\cup\ \mathrm{sig}_{i,R,1,i}$
    if $i = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
    if $i = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
    for $k \in \{m, \dots, l\}$ **do**
        $R_{i,2,k} \leftarrow$ TX {input: $\mathrm{TX}_{i,2,k}$.`revocation`$_{2,k}$, output: $(c_{\mathrm{virt}}, pk_{i+1,\mathrm{out}})$}
        $\mathrm{sig}_{i,R,2,k,i} \leftarrow$ SIGN($R_{i,2,k}$, $sk_{i,\mathrm{rev}}$); `left_revocations` $\leftarrow$ `left_revocations` $\cup\ \mathrm{sig}_{i,R,2,k,i}$
    end for
    for $k_1 \in \{m, \dots, i-1\}, k_2 \in \{i+1, \dots, l\}$ **do**
        $R_{i,3,k_1,k_2} \leftarrow$ TX {input: $\mathrm{TX}_{i,3,k_1,k_2}$.`revocation`$_{3,k_1,k_2}$, output: $(c_{\mathrm{virt}}, pk_{i+1,\mathrm{out}})$}
        $\mathrm{sig}_{i,R,3,k_1,k_2,i} \leftarrow$ SIGN($R_{i,3,k_1,k_2}$, $sk_{i,\mathrm{rev}}$); `left_revocations` $\leftarrow$ `left_revocations` $\cup\ \mathrm{sig}_{i,R,3,k_1,k_2,i}$
    end for
    send (COOP CLOSE REVOCATIONS, (`left_revocations`, $\mathrm{sig}_{i,\mathrm{right},R}$, $\mathrm{sig}_{i,\mathrm{right,rev,virt}}$, $\mathrm{sig}_{i,\mathrm{right,rev,fund}}$) to $\bar{P}$)
else // $i = n$, we are host of funder
    extract $\mathrm{sig}_{1,\mathrm{right},R}$ from `left_revocations`
    output (VERIFY REVOCATION, $\mathrm{sig}_{1,\mathrm{right},R}$) to `fundee`
    store received signature as $\mathrm{sig}_{\bar{P},R,i}$ // executed by `fundee`
    $R_{P,i} \leftarrow$ TX {input: $C_{\bar{P},i}$.outputs.$\bar{P}$, output: $(c_{\bar{P}}, pk_{P,\mathrm{out}})$}
    ensure VERIFY($R_{P,i}$, $\mathrm{sig}_{\bar{P},R,i}$, $pk_{\bar{P},R,i}$) = True
    for $j \in [n - 1]$ **do**
        if $j = 2$ **then** $m \leftarrow 1$ **else** $m \leftarrow 2$
        if $j = n - 1$ **then** $l \leftarrow n$ **else** $l \leftarrow n - 1$
        ensure that the following signatures are present in `left_revocations` and store them: // exclude signatures by $j + 1$ if $j = n - 1$

- // 3 signatures

    $\mathrm{sig}(R_{j,1}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{j,1}, pk_{j,\mathrm{rev}})$, $\mathrm{sig}(R_{j,1}, pk_{j+1,\mathrm{rev}})$

- // $3(l - m)$ signatures

    $\forall k \in \{m, \dots, l\} \setminus \{i\}$:
        $\mathrm{sig}(R_{j,2,k}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{j,2,k}, pk_{j,\mathrm{rev}})$, $\mathrm{sig}(R_{j,2,k}, pk_{j+1,\mathrm{rev}})$

- // $3(i - m) \cdot (l - i)$ signatures

    $\forall k_1 \in \{m, \dots, i-1\}, \forall k_2 \in \{i+1, \dots, l\}$:
        $\mathrm{sig}(R_{j,3,k_1,k_2}, pk_{1,\mathrm{rev}})$, $\mathrm{sig}(R_{j,3,k_1,k_2}, pk_{j,\mathrm{rev}})$, $\mathrm{sig}(R_{j,3,k_1,k_2}, pk_{j+1,\mathrm{rev}})$

    end for
    State $\leftarrow$ COOP CLOSED
    if `close_initiator` = P **then** // $\mathcal{E}$ instructed us to close the channel
        execute code of Fig. 54
    else // $\mathcal{E}$ instructed another party to close the channel
        send (COOPCLOSED) to `close_initiator`
    end if
end if

Figure 72

Process VIRT – punishment handling

On input (USED REVOCATION) by guest: // (USED REVOCATION) by funder/fundee is ignored
    State ← GUEST PUNISHED
    input (USED REVOCATION) to $\texttt{host}_P$, expect reply (USED REVOCATION OK)
    if funder or fundee is defined **then**
        output (ENABLER USED REVOCATION) to it
    else // sibling is defined
        output (ENABLER USED REVOCATION) to sibling
    end if

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

On output (USED REVOCATION) by $\texttt{host}_P$:
    State ← GUEST PUNISHED
    if funder or fundee is defined **then**
        output (ENABLER USED REVOCATION) to it
    else // sibling is defined
        output (ENABLER USED REVOCATION) to sibling
    end if

Figure 73

G Omitted Theorems and Proofs

LEMMA G.1 (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, $c, \dots$) by either $\mathcal{E}$ or $\bar{P}$,*
- *P [has received (FUND ME, $f_i, \dots$) as input by another LN ITI while State was OPEN and subsequently P transitioned to OPEN State] n times,*
- *P [has received (CHECK COOP CLOSE FUNDEE, $(\_, r_i), \dots$) as output by $\texttt{host}_P$ while State was OPEN and subsequently P transitioned to OPEN State] j times,*
- *P [has received (COOP CLOSE SIGN COMM FUNDER, $(l_i, \_)$) as output by $\texttt{host}_P$ while State was OPEN and subsequently P transitioned to OPEN State] k 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 (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 $\texttt{host}_P \neq$ "ledger" the output of $\texttt{host}_P$ 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*

$$\sum_{i=1}^{h} c_i \geq \phi \cdot c - \sum_{i=1}^{n} f_i + \sum_{i=1}^{j} r_i + \sum_{i=1}^{k} l_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i \qquad (2)$$

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

- *Assume that, at some particular instant during the execution,*
  (1) $\texttt{host}_P \neq$ "ledger",
  (2) P has State OPEN.

  Consider two alternative series of subsequent execution steps:
  (1) The guest *of $\texttt{host}_P$ (call them S) receives (FORCECLOSE) by $\mathcal{E}$. From that point onward, all protocol parties (even corrupted ones) honestly follow the protocol. Eventually a total of c_b coins is exclusively spendable by $pk_{R,\text{out}}$, where R is a machine kindred to S. Additionally, there is at least one funding output of P's channel $(c_P + c_{\bar{P}}, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ that is on-chain and unspent.*
  (2) P receives either (COOPCLOSE) by $\mathcal{E}$ or (COOP CLOSE,...) by some other ITI, and P's variable hosting *is False. Subsequently, P's State transitions to COOP CLOSED and then the State of S transitions to OPEN. The next time S is activated is via a (FORCECLOSE) input by $\mathcal{E}$ and eventually a total of c_t coins is exclusively spendable by $pk_{R,\text{out}}$.*

  It then holds that

$$c_t - c_b \geq \phi \cdot c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i + \sum_{i=1}^{j} r_i + \sum_{i=1}^{k} l_i \qquad (3)$$

with overwhelming probability in the security parameter.

PROOF. [Proof of Lemma G.1] 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 "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 hops $\neq$ "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. 45, l. 21) or sent (Fig. 45, l. 6) respectively by P. Moving on to the funding of new virtual channels, whenever P funds a new virtual channel (Fig. 42, l. 21), $[-c_{\text{virt}}]$ is appended to C and whenever P helps with the opening of a new virtual channel, but does not fund it (Fig. 42, l. 24), $[0]$ is appended to C. In case of cooperatively closing a channel (Figs. 53-56 & 72) to which P's channel is base, if this channel was initially funded by P, when the closing procedure completes (Fig. 56, l. 53) $[c'_1]$ is appended to C. Likewise, if in the closed virtual channel P was the base of the `fundee` (Fig. 72, l. 171), then $[c'_2]$ (Fig. 72, l. 9) is appended to C. In case P was a left intermediary for the closed virtual channel (Fig. 72, l. 10), then $[c'_2]$ is appended to C. Lastly, in case P was a right intermediary for the closed virtual channel (Fig. 72, l. 23), then $[c'_1 - c_{\text{virt}}]$ 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 virtual layer that is created or torn down cooperatively. 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 (2) is equal to $\sum\limits_{s \in C} \sum\limits_{x \in s} x$, as all inbound and outbound payment values, new channel funding values and cooperative closing refunds 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` = "`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, $c, \ldots$) while in the INIT *State*. In case $P = Alice$, this message must have been received as input by $\mathcal{E}$ (Fig. 40, l. 1), or in case $P = Bob$, this message must have been received via the network by $\bar{P}$ (Fig. 35, l. 3).
- It has received $pk_{\bar{P},F}$. In case $P = Bob$, $pk_{\bar{P},F}$ must have been contained in the (OPEN, $\ldots$) message by $\bar{P}$ (Fig. 35, l. 3), otherwise if $P = Alice$ $pk_{\bar{P},F}$ must have been contained in the (ACCEPT CHANNEL, $\ldots$) message by $\bar{P}$ (Fig. 35, 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. 37, ll. 12 and 23).
- It has the transaction F in the $\mathcal{G}_{\text{Ledger}}$ state (Fig. 38, l. 3 or Fig. 39, 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. 37, ll. 4 or 14), and sent it to $\bar{P}$ (Fig. 37, 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. 37, 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 (2). 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. 37, l. 2 or 3). Additionally, if P receives (FORCECLOSE) by $\mathcal{E}$ while $H = ([], [g])$, it attempts to publish $C_{P,0}$ (Fig. 51, 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 (2). 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, $f, \ldots$) 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 $\texttt{host}_P$ 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 $\sum\limits_{i=1}^{h} c_i \geq \sum\limits_{s \in C'} \sum\limits_{x \in s} x$.* Furthermore, given that P moves to the OPEN state after the (FUND ME, $\ldots$) message, it also sends (FUNDED, $\ldots$) to R (Fig. 42, l. 22). 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 $\texttt{host}_R$ ($\texttt{host}_R = \texttt{host}_P$ – Fig. 39, 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 $\sum\limits_{i=1}^{k} c_i^R \geq \sum\limits_{s \in C_R} \sum\limits_{x \in s} x$.*
- If P receives (VIRTUALISING, $\ldots$) by $\bar{P}$ or `sibling`, 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 $\texttt{host}_P$ 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 $\sum\limits_{i=1}^{h} c_i \geq \sum\limits_{s \in C} \sum\limits_{x \in s} x$.* Furthermore, given that P moves to the OPEN state after the (VIRTUALISING, $\ldots$) message and in case it sends (FUNDED, $\ldots$) to some party R (Fig. 42, l. 19), 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 $\texttt{host}_R$ ($\texttt{host}_R = \texttt{host}_P$ – Fig. 39, 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 (CHECK COOP CLOSE, $\ldots$) by $\texttt{host}_P$, subsequently transitions back to OPEN (therefore moving to history (F, C') where $C' = C + [c'_2]$), and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by $\texttt{host}_P$ 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* $\sum_{i=1}^{h} c_i \geq \sum_{s \in C} \sum_{x \in s} x$.

- If P receives (COOP CLOSE SIGN COMM, ...) by $\texttt{host}_P$, subsequently transitions back to OPEN (therefore moving to history (F, C') where $C' = C + [c'_1 - c_{\text{virt}}]$), and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by $\texttt{host}_P$ 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, *there exists a local, kindred machine R that transitioned to the* OPEN *state after the last time control was obtained by one of P's kindred machines and before P transitioned to the* OPEN *state, such that R obtained* $c'_2 = c_{\text{virt}} - c'_1$ *coins during its last activation.* (In other words, P and R broke even on aggregate by first supporting the opening and then the cooperative closing of a virtual channel.)
- If P receives (COOP CLOSE SIG COMM FUNDER, ...) by $\texttt{host}_P$, subsequently transitions back to OPEN (therefore moving to history (F, C') where $C' = C + [c'_1]$) and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by $\texttt{host}_P$ 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* $\sum_{i=1}^{h} c_i \geq \sum_{s \in C} \sum_{x \in s} x$.
- If P receives (CHECK COOP CLOSE FUNDEE, ...) by $\texttt{host}_P$, subsequently transitions back to OPEN (therefore moving to history (F, C') where $C' = C + [c'_2]$) and finally receives (FORCECLOSE) by $\mathcal{E}$ and (CLOSED) by $\texttt{host}_P$ 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* $\sum_{i=1}^{h} c_i \geq \sum_{s \in C} \sum_{x \in s} x$.
- 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 $\texttt{host}_P$ (the latter only if $\texttt{host}_P \neq$ "$\texttt{ledger}$" 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 $\texttt{host}_P$ (the latter only if $\texttt{host}_P \neq$ "$\texttt{ledger}$" or equivalently $F = []$) 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* $\sum_{i=1}^{h} c_i \geq \sum_{s \in C'} \sum_{x \in s} x$.

Consider the first bullet. 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 → VIRTUALISING → WAITING FOR REVOCATION → WAITING FOR INBOUND REVOCATION → WAITING FOR HOSTS READY → OPEN. During this sequence of events, a new $\texttt{host}_P$ is defined (Fig. 42, l. 6), new commitment transactions are negotiated with $\bar{P}$ (Fig. 42, l. 9), control of the old funding output is handed over to $\texttt{host}_P$ (Fig. 42, l. 11), $\texttt{host}_P$ 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. 65 and 67) and the previous valid commitment transactions of both P and $\bar{P}$ are invalidated (Fig. 34, 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 $\texttt{host}_P$ (Fig. 51, l. 4). As per the Lemma conditions, $\texttt{host}_P$ 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. 71, l. 5 and l. 6 respectively).

If the $\texttt{host}$ of $\texttt{host}_P$ is "$\texttt{ledger}$", 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 $\texttt{host}_P \neq$ "$\texttt{ledger}$", after the funding procedure is complete, the new $\texttt{host}_P$ will have as its $\texttt{host}$ the old $\texttt{host}_P$ of P. If the (FORCECLOSE) sequence is initiated, the new $\texttt{host}_P$ will follow the same steps that will be described below once the old $\texttt{host}_P$ succeeds in closing the lower layer (Fig. 70, l. 6). The old $\texttt{host}_P$ 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 $\texttt{host}_P$ = "$\texttt{ledger}$".

Moving on, $\texttt{host}_P$ is either able to publish its $\text{TX}_{1,1}$ and $B_{1,1}$ (it has necessarily received valid signatures $\text{sig}(\text{TX}_{1,1}, pk_{\bar{P},F})$ (Fig. 67, l. 43), $\text{sig}(B_{1,1}, pk_{2,2,1})$ and $\text{sig}(B_{1,1}, pk_{n,n-1,n})$ (Fig. 65, l. 18) 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 publish $\text{TX}_{1,1}$ 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 $\texttt{host}_P$ has signed with $sk_{P,F}$ (Fig. 67, ll. 37-41). The output can be also spent by old, revoked commitment transactions, but in that case $\texttt{host}_P$ would not have output (CLOSED); P would have instead detected this triggered by a (CHECK CHAIN FOR CLOSED) message by $\mathcal{E}$ (Fig. 49) 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 $\mathrm{TX}_{1,1}$, $\mathrm{TX}_{2,1}$, $(\mathrm{TX}_{2,2,k})_{k>2}$ has a $(c_P + c_{\bar{P}} - f, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$ output (Fig. 61, l. 21 and Fig. 60, ll. 41 and 128) which will end up in $\mathcal{G}_{\mathrm{Ledger}}$ – call this output o_P. We will prove that at most $\sum\limits_{i=2}^{n-1}(t_i + p + s - 1)$ 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 $\mathcal{G}_{\mathrm{Ledger}}$. In case party $\bar{P}$ is idle, then $o_{1,2}$ is consumed by $\mathrm{TX}_{1,1}$, its virtual output is spent by $B_{1,1}$ and the timelock on the output of the latter expires, therefore the required output o_R is on-chain. In case $\bar{P}$ is active, exactly one of $\mathrm{TX}_{2,1}$, $(\mathrm{TX}_{2,2,k})_{k>2}$ or $(\mathrm{TX}_{2,3,1,k})_{k>2}$ is a descendant of $o_{1,2}$; if the transaction belongs to one of the two last transaction groups (with subscript g) then necessarily $\mathrm{TX}_{1,1}$ is on-chain in some block height h and given the timelock on the virtual output of $\mathrm{TX}_{1,1}$, $\bar{P}$'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 a `bridge` output which can be spent only by R_g or B_g. The P has never signed R_g, so only B_g can spend it. B_g has the required output o_R (without a timelock) and P publishes B_g (Fig. 71, l. 6). 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 $\mathrm{TX}_{1,1}$ 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 $\mathrm{TX}_{i,2,2}$ or $(\mathrm{TX}_{i,3,2,k})_{k>i}$ 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 is only spendable by a `bridge` tx, which R publishes (Fig. 71, l. 6) and which has 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 $\mathrm{TX}_{i,3,2,k}$ **then** $i \leftarrow k$ **else** $i \leftarrow i + 1$
13: **end if**
14: **end if**
15: **end loop**
16: // `maxDel` $\leq \sum\limits_{i=2}^{n-1}(t_i + p + s - 1)$

Figure 74

In every case o_P and o_R end up on-chain in at most s and $\sum\limits_{i=2}^{n-1}(t_i + p + s - 1)$ 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 $(c_P - f, pk_{P,\mathrm{out}})$ 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 $(c_P - f, pk_{P,\mathrm{out}})$ 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, $\mathtt{host}_P, \dots$) by P and before moving to the open state, it receives $\mathrm{sig}_{\bar{R},C,0} = \mathrm{sig}(C_{R,0}, pk_{\bar{R},F})$ and sends $\mathrm{sig}_{R,C,0} = \mathrm{sig}(C_{\bar{R},0}, pk_{R,F})$. 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 $\mathtt{host}_P$, R attempts to add $C_{R,0}$ to $\mathcal{G}_{\mathrm{Ledger}}$, 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 $(f, pk_{R,\mathrm{out}})$ 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 $(f, pk_{R,\mathrm{out}})$ 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 `fundee` (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 $o_{n-1,n}$, $\mathrm{TX}_{1,1}$ with $\mathrm{TX}_{n,1}$, $B_{1,1}$ with $B_{n,1}$, $\mathrm{TX}_{2,1}$ with $\mathrm{TX}_{n-1,1}$, $B_{2,1}$ with $B_{n-1,1}$, $(\mathrm{TX}_{2,2,k})_{k>2}$ with $(\mathrm{TX}_{n-1,2,k})_{k<n-1}$, $(B_{2,2,k})_{k>2}$ with $(B_{n-1,2,k})_{k<n-1}$, $(\mathrm{TX}_{2,3,1,k})_{k>2}$ with $(\mathrm{TX}_{n-1,3,n,k})_{k<n-1}$, $(B_{2,3,1,k})_{k>2}$ with $(B_{n-1,3,n,k})_{k<n-1}$, t_2 with t_{n-1}, $\mathrm{TX}_{i,3,2,k}$ with $\mathrm{TX}_{i,3,n-1,k}$, $B_{i,3,2,k}$ with $B_{i,3,n-1,k}$, i is initialized to $n - 2$ in l. 2 of Fig. 74, 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 `fundee` ($1 < i < n$), 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 $c_P = \sum\limits_{s \in C}\sum\limits_{x \in s} x$ 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 $\mathrm{TX}_{i,1}$, $(\mathrm{TX}_{i,2,k})_{k>i}$, $\mathrm{TX}_{i-1,1}$, or $(\mathrm{TX}_{i-1,2,k})_{k<i-1}$, 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 $pk_{P,\mathrm{out}}$ 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 VIRTUALISING 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 (Note that this output is also encumbered with a timelock, but the alternative spending method cannot be used as $\texttt{host}_P$ has not signed the required revocation transaction). 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 for which the revocation-based spending methd cannot be used since $\texttt{host}_P$ has not produced the corresponding signature for the revocation transaction. We have therefore covered all cases and proven the second bullet.

We now focus on the third bullet. Once more the induction hypothesis guarantees that before (CHECK COOP CLOSE, ...) 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 that have a sum value of $c_P = \sum\limits_{s \in C} \sum\limits_{x \in s} x$. When P receives (CHECK COOP CLOSE, ...), it moves to the COOP CLOSING state before returning to the OPEN state. It verifies the counterparty's signature on the new commitment transaction $C_{P,i+1}$, (Fig. 72, l. 16) which spends the latest old funding output (Fig. 72, l. 14), effectively removing one virtualisation layer. In $C_{P,i+1}$ P owns c'_2 more coins than before that moment (Fig. 72, l. 15). It then signs the corresponding commitment transaction for the counterparty (Fig. 72, l. 91) and expects a valid signature for the revocation transaction of the old commitment transaction of the counterparty (Fig. 72, l. 126). Once these are received, P transitions to the OPEN state. If the o_F output is spent while P is in the COOP CLOSING state, it can be spent by one of $C_{P,i+1}$ or some of $(C_{\bar{P},j})_{0 \leq j \leq i+1}$. If it is spent by $C_{P,i+1}$ or $C_{\bar{P},i+1}$, then these two transactions have a ($c_P + c'_2$, $pk_{P,\text{out}}$) 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 this or another of our past funding outputs 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. What is more, if o_F is spent by any virtual transaction, then $\texttt{host}_P$ will punish the publisher of such transaction with the corresponding virtual revocation transaction (Fig. 72, l. 35, l. 38, l. 62, l. 110, l. 111 and l. 114) at the latest when P receives (CHECK CHAIN FOR CLOSED) (Fig. 49, l. 17) – note that the latter message is received periodically by P, since it is a non-negligent party. The virtual revocation transaction gives a sum equal to the entirety of the channel's funds to P. 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. 72, l. 126) and moves to the OPEN state, the above analysis of what can happen when o_F is spent 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 $c_P + c'_2$ coins upon channel closure. We have therefore proven the third bullet.

We now focus on the fourth bullet. Once more the induction hypothesis guarantees that before (COOP CLOSE SIGN COMM, ...) 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 that have a sum value of $c_P = \sum\limits_{s \in C} \sum\limits_{x \in s} x$. When P receives (COOP CLOSE SIGN COMM, ...), it moves to the COOP CLOSING state before returning to the OPEN state. It signs the new commitment transaction for the counterparty (Fig. 72, l. 27) which spends the latest old funding output (Fig. 72, l. 25), effectively removing one virtualisation layer. In $C_{P,i+1}$ P owns $c_{\text{virt}} - c'_1$ less coins than before that moment (Fig. 72, l. 26) – note that P now lost access to c_{virt} coins from the refund output of its virtual transactions. It then verifies the counterparty's signatures on the corresponding new local commitment transaction $C_{P,i+1}$, (Fig. 72, l. 46) and on the revocation transaction of the old commitment transaction of the counterparty (Fig. 72, l. 49). Once these are received, P transitions to the OPEN state. If the o_F output is spent while P is in the COOP CLOSING state, it can be spent by one of $C_{P,i+1}$ or some of $(C_{\bar{P},j})_{0 \leq j \leq i+1}$. If it is spent by $C_{P,i+1}$ or $C_{\bar{P},i+1}$, then these two transactions have a ($c_P + c'_1$, $pk_{P,\text{out}}$) 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 this or another of our past funding outputs 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. Similarly to the previous bullet, if o_F is spent by any virtual transaction, then $\texttt{host}_P$ will punish the publisher and P will obtain a sum equal to the entirety of the channel's funds. 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. 72, l. 126) and moves to the OPEN 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 $c_P - c_{\text{virt}} + c'_1$ coins upon channel closure. This proves the first claim of the fourth bullet.

Regarding the second claim, we observe that P can only move to the OPEN state if previously a local kindred LN ITI R moves to the OPEN state as well. Via direct application of the previous claim of the currently analysed bullet, R has gained c'_2 coins in the process, therefore guaranteeing that P and R have on aggregate access to the same number of coins as before the cooperative closing. What is more, throughout the cooperative closing process both parties had access to at least c_P and c_R coins respectively, thus ensuring that no loss of coins is possible. We have now proven the fourth bullet.

Moving on to the fifth bullet, the same reasoning as that of the treatment of the previous bullet holds, albeit with the `guest`'s signature verifications as they appear in Fig. 56.

The first claim of the sixth bullet holds due to an argument identical to that provided for the third bullet, since in both cases the relevant parts of the protocol execution are the same. Note that `funder`'s signature for the revocation of the last commitment transaction of the virtual channel has not been yet verified, but this is of no consequence for our balance as all other revocation signatures have been already verified and the connection with the `funder` has been severed due to the successful cooperative closing.

Regarding now the seventh 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 $c_P = \sum\limits_{s \in C} \sum\limits_{x \in s} x$. (Note that $\sum\limits_{s \in C'} \sum\limits_{x \in s} x = d + \sum\limits_{s \in C} \sum\limits_{x \in s} x$.) 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. 44, l. 2) the new commitment transaction $C_{\bar{P},i+1}$ in which the counterparty owns d more coins than before that moment (Fig. 44, l. 1), sends the signature to the counterparty (Fig. 44, l. 5) and expects valid signatures on its own updated commitment transaction (Fig. 45, l. 1) and the revocation transaction for the old commitment transaction of the counterparty (Fig. 45, l. 3). Upon verifying them, P transitions to the OPEN state. 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. 45, l. 5) and send (Fig. 45, l. 17) the counterparty's revocation transaction for P's previous commitment transaction.

Similarly to previous bullets, if $\texttt{host}_P \neq$ "`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 $C_{P,i+1}$, $(C_{\bar{P},j})_{0 \le j \le i+1}$ 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 seventh bullet.

For the eighth 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 $c_P = \sum\limits_{s \in C} \sum\limits_{x \in s} x$. (Note that $e + \sum\limits_{s \in C'} \sum\limits_{x \in s} x = \sum\limits_{s \in C} \sum\limits_{x \in s} x$ 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. 47, l. 6). If subsequently it receives a valid signature for $C_{P,i+1}$ (Fig. 44, 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 $(C_{\bar{P},j})_{0 \le j \le i+1}$. 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 $(C_{\bar{P},j})_{0 \le j < i}$ 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. 45, 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 eighth bullet and with it the first bullet of the Lemma.

We now turn to proving the second bullet of the Lemma. We will take advantage of the results that have been derived earlier in this proof. If P is the `funder` of the virtual channel in process of cooperatively closing, it ensures that $c'_1 = c_P \wedge c'_2 = c_{\bar{P}}$ (Fig. 56, l. 4). If P is the `fundee`, it requests that the virtual channel be closed with the current honest coin balance (Fig. 55, l. 6), in which case it is $c'_1 = c_{\bar{P}} \wedge c'_2 = c_P$. Due to the arguments proving the first Lemma bullet, we know that

$$c_P = \sum_{s \in C} \sum_{x \in s} x \geq \phi \cdot c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i + \sum_{i=1}^{j} r_i + \sum_{i=1}^{k} l_i \ . \quad (4)$$

Just before the splitting of the two alternative scenarios, party S is entitled to c_b coins, since (i) in the first scenario all other parties honestly follow the protocol and thus they do not lose any coins to S and (ii) no action during the first scenario causes any transfer of coins. As we saw previously, if P transitions to the COOP CLOSED state, then S has also transitioned from the COOP CLOSING to the OPEN state and benefitted from an increase of the coins it can exclusively spend by c_P. It therefore holds that the difference of the coins $c_t - c_b$ that P owns at the end of the two scenarios is exactly c_P and due to (4) we can directly derive the required (3). The Lemma has now been proven. □

LEMMA G.2 (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, $c, \dots$) by $\mathcal{E}$ prior to transitioning to the OPEN State,*
- *P [has received (FUND ME, $f_i, \dots$) as input by another $\mathcal{G}_{\text{Chan}}$/LN ITI while $State_P$ was OPEN and P subsequently transitioned to OPEN State] n times,*
- *$\mathcal{G}_{\text{Ledger}}$ [has received (COOP CLOSING, P, r_i) by $\mathcal{S}$ while $State_P$ was OPEN and subsequently P transitioned to OPEN State] k 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:

$$\texttt{balance}_P = \phi \cdot c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i + \sum_{i=1}^{k} r_i \tag{5}$$

PROOF. [Proof of Lemma G.2] 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. 18, l. 1).
- If $P = Alice$ and it receives (OPEN, $c, \dots$) by $\mathcal{E}$, it stores c (Fig. 18, l. 11). If later $State_P$ becomes OPEN, $\mathcal{G}_{\text{Chan}}$ sets $\texttt{balance}_P \leftarrow c$ (Fig. 18, ll. 14 or 34). In contrast, if $P = Bob$, it is $\texttt{balance}_P = 0$ until at least the first transition of $State_P$ to OPEN (Fig. 18).
- Every time that P receives input (FUND ME, $f_i, \dots$) by another party while $State_P =$ OPEN, P stores f_i (Fig. 20, l. 1). The next time $State_P$ transitions to OPEN (if such a transition happens), $\texttt{balance}_P$ is decremented by f_i (Fig. 20, 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 $\mathcal{G}_{\text{Ledger}}$ receives (COOP CLOSING, P, r_i) by $\mathcal{S}$ while $State_P$ is OPEN, r_i is stored (Fig. 22, l. 1). The next time $State_P$ transitions to OPEN (if such a transition happens), $\texttt{balance}_P$ is incremented by r_i (Fig. 22, l. 9). Therefore, if this cycle happens $k \geq 0$ times, $\texttt{balance}_P$ will be incremented by $\sum_{i=1}^{k} r_i$ in total.
- Every time P receives input (PAY, d_i) by $\mathcal{E}$ while $State_P =$ OPEN, d_i is stored (Fig. 19, l. 2). The next time $State_P$ transitions to OPEN (if such a transition happens), $\texttt{balance}_P$ is decremented by d_i (Fig. 19, 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. 19, l. 7). The next time $State_P$ transitions to OPEN (if such a transition happens) $\texttt{balance}_P$ is incremented by e_i (Fig. 19, 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 $\texttt{balance}_P = c - \sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i + \sum_{i=1}^{k} r_i$ if $P = Alice$, or else if $P = Bob$, $\texttt{balance}_P = -\sum_{i=1}^{n} f_i - \sum_{i=1}^{m} d_i + \sum_{i=1}^{l} e_i + \sum_{i=1}^{k} r_i$. □

LEMMA G.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. 21 is executed negligibly often).*

PROOF. [Proof of Lemma G.3] We prove the Lemma in two steps. We first show that if the conditions of Lemma G.2 hold, then the conditions of Lemma G.1 for the real world execution with protocol LN and the same $\mathcal{E}$ and $\mathcal{A}$ hold as well for the same k, 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. 18, l. 5). The first case only happens when either P receives (CORRUPT) by $\mathcal{A}$ (Fig. 32, l. 1), which means that the simulated P is not honest anymore, or when P becomes $\texttt{negligent}$ (Fig. 32, l. 4), which means that the first condition of Lemma G.1 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. 73, ll. 1 or 12), so in case P receives (FORCECLOSE) by $\mathcal{E}$ the output of $\texttt{host}_P$ will be (GUEST PUNISHED) (Fig. 70, l. 4). In all cases, some condition of Lemma G.1 is violated.

For $State_P$ to become OPEN at least once, the following sequence of events must take place (Fig. 18): If $P = Alice$, it must receive (INIT, pk) by $\mathcal{E}$ when $State_P =$ UNINIT, then either receive (OPEN, c, $\mathcal{G}_{\text{Ledger}}, \dots$) by $\mathcal{E}$ and (BASE OPEN) by $\mathcal{S}$ or (OPEN, c, $\texttt{hops}$ ($\neq \mathcal{G}_{\text{Ledger}}$), $\dots$) by $\mathcal{E}$, (FUNDED, HOST, $\dots$) by $\texttt{hops[0].left}$ 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. 32, l. 10), therefore if the second condition of Lemma G.2 holds and $P = Alice$, then the second condition of Lemma G.1 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 G.2 holds only if the second condition of Lemma G.1

holds as well. We also observe that, if both parties are honest, they will transition to the OPEN state simultaneously.

Regarding the third Lemma G.2 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, $f, \dots$) 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 $State_{\bar{P}} = \text{IGNORED}$, then $State_P$ transitions directly back to OPEN. If on the other hand $State_{\bar{P}} = \text{OPEN}$ 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 $State_{\bar{P}} = \text{TENTATIVE HELP FUND}$, $\mathcal{G}_{\text{Chan}}$ also sets $State_{\bar{P}}$ to SYNC HELP FUND. Then both $State_{\bar{P}}$ and $State_P$ transition simultaneously to OPEN (Fig. 20). This sequence of events may repeat any $n \geq 0$ times. We observe that throughout these steps, honest simulated P has received (FUND ME, $f, \dots$) and that $\mathcal{S}$ only sends (FUND) when all honest simulated parties have transitioned to the OPEN state (Fig. 32, l. 18 and Fig. 42, l. 12), so the third condition of Lemma G.1 holds with the same n as that of Lemma G.2.

Moving on to the fourth Lemma G.2 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 $\mathcal{G}_{\text{Chan}}$ receives (COOP CLOSING, P, r) by $\mathcal{S}$, $State_P$ transitions to COOP CLOSING and subsequently when $\mathcal{S}$ sends (COOP CLOSED, P) to $\mathcal{G}_{\text{Chan}}$, if $\texttt{layer}_P = 0$ then $State_P$ transitions to COOP CLOSED, else $State_P$ transitions to OPEN. This sequence of events may repeat any $k \geq 0$ times. We observe that throughout these steps, honest simulated P has transitioned to the COOP CLOSING state and that $\mathcal{S}$ only sends (COOP CLOSED, P) when honest simulated P transitions to either OPEN or COOP CLOSED state, so the sum of j (from the fourth condition of Lemma G.1) plus k (from the fifth condition of Lemma G.1) is equal to the k of Lemma G.2.

Regarding the sixth Lemma G.2 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 $State_{\bar{P}} = \text{IGNORED}$, then $State_P$ transitions directly back to OPEN. If on the other hand $State_{\bar{P}} = \text{OPEN}$ 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 $State_{\bar{P}} = \text{TENTATIVE GET PAID}$, $\mathcal{G}_{\text{Chan}}$ also sets $State_{\bar{P}}$ to SYNC GET PAID. Then both $State_P$ and $State_{\bar{P}}$ transition simultaneously to OPEN (Fig. 19). 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. 32, l. 16), so the sixth condition of Lemma G.1 holds with the same m as that of Lemma G.2. As far as the seventh condition of Lemma G.2 goes, we observe that this case is symmetric to the one discussed for its sixth condition above if we swap P and $\bar{P}$, therefore we deduce that if Lemma G.2 holds with some l, then Lemma G.1 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. Additionally, we saw that if one party transitions from the COOP CLOSING state to the OPEN state, 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. 21, l. 18), we deduce that if the conditions of Lemma G.2 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 G.2 do not hold, then the check of Fig. 21, l. 18 never takes place. We first discuss the $State_P = \text{IGNORED}$ 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 $State_P \neq \text{IGNORED}$ (Fig. 21, l. 9). We deduce that, once $State_P = \text{IGNORED}$, 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. 21 without first having been in the OPEN state. Moreover if $P = Alice$, it is impossible to reach the OPEN state without receiving input (OPEN, $c, \dots$) by $\mathcal{E}$. Lastly, as we have observed already, the three last conditions of Lemma G.2 are always satisfied. We conclude that if the conditions to Lemma G.2 do not hold, then the check of Fig. 21, 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. □

THEOREM G.4 (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:*

$$\forall\, PPT\, \mathcal{A}, \exists\, PPT\, \mathcal{S} : \forall\, PPT\, \mathcal{E} \text{ it is}$$

$$EXEC^{\mathcal{G}_{\text{Ledger}}}_{\Pi^1_{\text{Chan}},\mathcal{A},\mathcal{E}} \approx EXEC^{\mathcal{G}^1_{\text{Chan}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}} \quad .$$

The corresponding proof stems from Lemma G.3, the fact that $\mathcal{G}_{\text{Chan}}$ is a simple relay and that $\mathcal{S}$ faithfully simulates Π_{Chan}. Lastly we prove that $\forall n \geq 2, \Pi^n_{\text{Chan}}$ UC-realises $\mathcal{G}^n_{\text{Chan}}$ in the presence of $\mathcal{G}^1_{\text{Chan}}, \dots, \mathcal{G}^{n-1}_{\text{Chan}}$ (leveraging the relevant definition from [8]).

THEOREM G.5 (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 $\mathcal{G}^1_{\text{Chan}}, \dots, \mathcal{G}^{n-1}_{\text{Chan}}$ and $\mathcal{G}_{\text{Ledger}}$, assuming the security of the underlying digital signature. Specifically,*

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

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

Proof. [Proof of Theorem G.4] By inspection of Figures 17 and 31 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 G.3, therefore the two executions are computationally indistinguishable. □

Proof. [Proof of Theorem G.5] The proof is exactly the same as that of Theorem G.4, replacing superscripts 1 for *n*. □

Since we use a global setup, proving UC-emulation is not enough. We further need to prove that all ideal global subroutines are *replaceable*, i.e., they can be replaced with their real counterparts. This guarantees that a real deployment will offer the same security guarantees as its idealized description.

For any $i \in [n]$, the individual global subroutines $\mathcal{G}_{\text{Ledger}}, \mathcal{G}^1_{\text{Chan}}, \dots, \mathcal{G}^i_{\text{Chan}}$ can be merged (as per Def. 4.1 of [10]) into the "global setup" $\mathcal{G}^i$. Likewise, the realisation Π_{Ledger} of $\mathcal{G}_{\text{Ledger}}$ [11] and $\Pi^1_{\text{Chan}}, \dots, \Pi^i_{\text{Chan}}$ can be merged into Π^i.

Lemma G.6. *For all $i \in [n]$, Π^i UC-emulates $\mathcal{G}^i$.*

Proof. [Proof of Lemma G.6] The following facts hold (note the inversion of the order of indices compared to the formulation of Theorem 4.3): $\mathcal{G}_{\text{Ledger}}$ and Π_{Ledger} are *subroutine respecting* (Def. A.6 [10]) as they do not accept/pass inputs or outputs from/to parties outside their session (this can be verified by inspection). $\mathcal{G}^1_{\text{Chan}}$ and Π^1_{Chan} are $\mathcal{G}_{\text{Ledger}}$- and Π_{Ledger}-subroutine respecting respectively (Def. 2.3 [10]), as they they do not accept/pass inputs or outputs from/to parties outside their session apart from $\mathcal{G}_{\text{Ledger}}$ and Π_{Ledger} respectively. For any $j \in [i] \setminus \{1\}$, $\mathcal{G}^j_{\text{Chan}}$ and Π^j_{Chan} are $\mathcal{G}^{j-1}$- and Π^{j-1}-subroutine respecting respectively, as they they do not accept/pass inputs or outputs from/to parties outside their session apart from $\mathcal{G}^{j-1}$ and Π^{j-1} respectively. Theorem 4.3 of [10] then implies the required result. □

Theorem G.7 (Full Replacement). *For all $i \in [n]$, the ideal global setup $\mathcal{G}^i$ can be replaced with Π^i.*

Proof. Simulator $\mathcal{S}$ (Figs. 31–32) is *$\mathcal{G}$-agnostic* (Def. 3.4 of [10]) as $\mathcal{S}$ is a relay between parties and the environment $\mathcal{E}$. This can be verified by inspection of Fig. 31. Thus Theorem 3.5 of [10], full replacement, applies inductively to $\mathcal{G}^i$ and Π^i. □

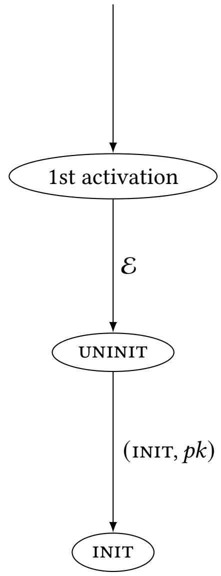

Figure 23: $\mathcal{G}_{\text{Chan}}$ state machine up to INIT (both parties)

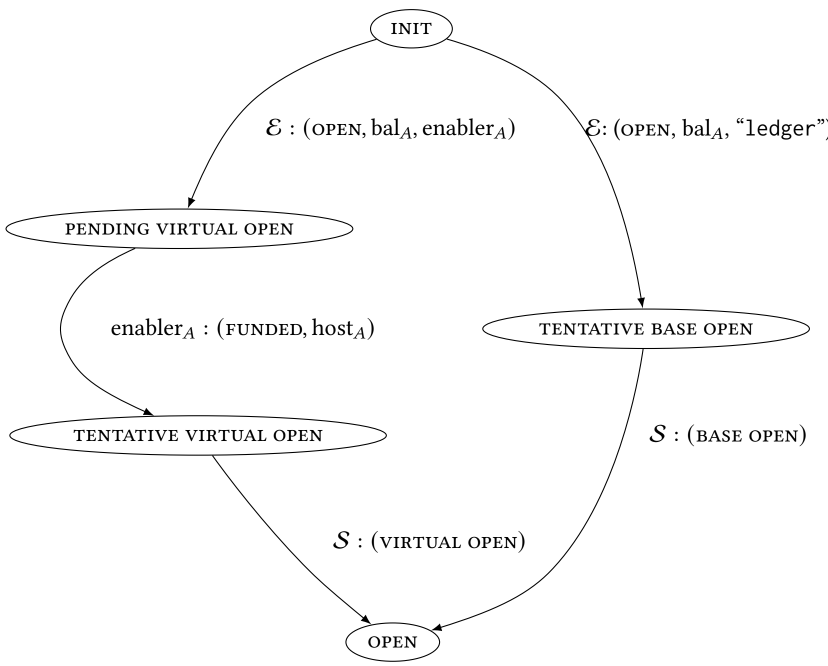

Figure 24: $\mathcal{G}_{\text{Chan}}$ state machine from INIT up to OPEN (funder)

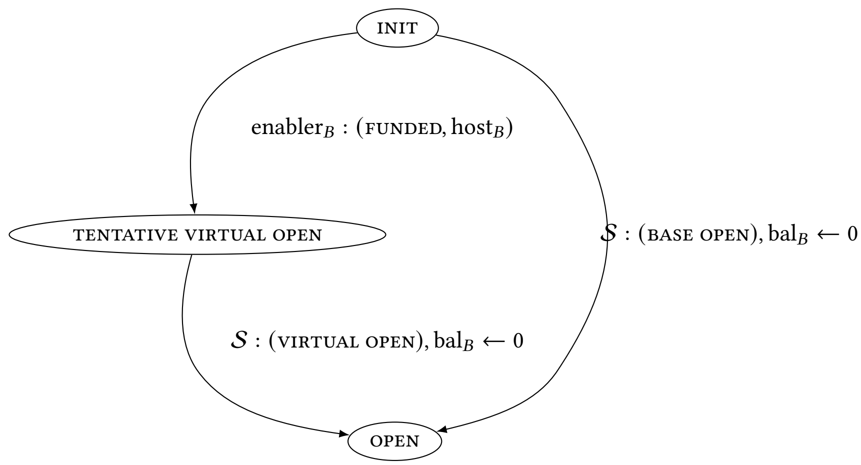

Figure 25: $\mathcal{G}_{\text{Chan}}$ state machine from INIT up to OPEN (fundee)

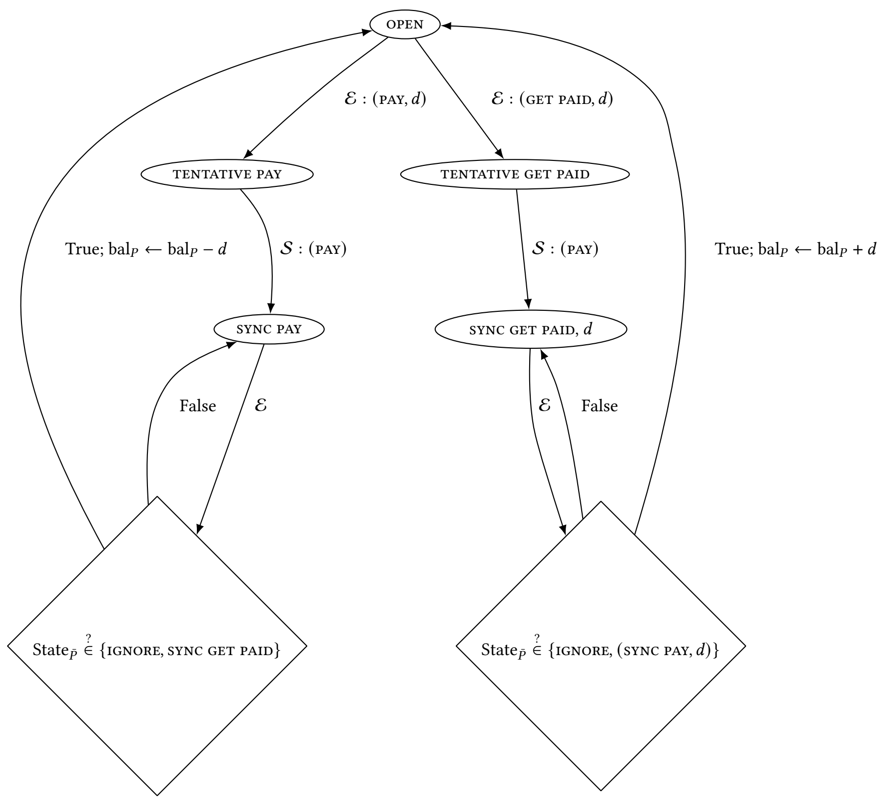

Figure 26: $\mathcal{G}_{\text{Chan}}$ state machine for payments (both parties)

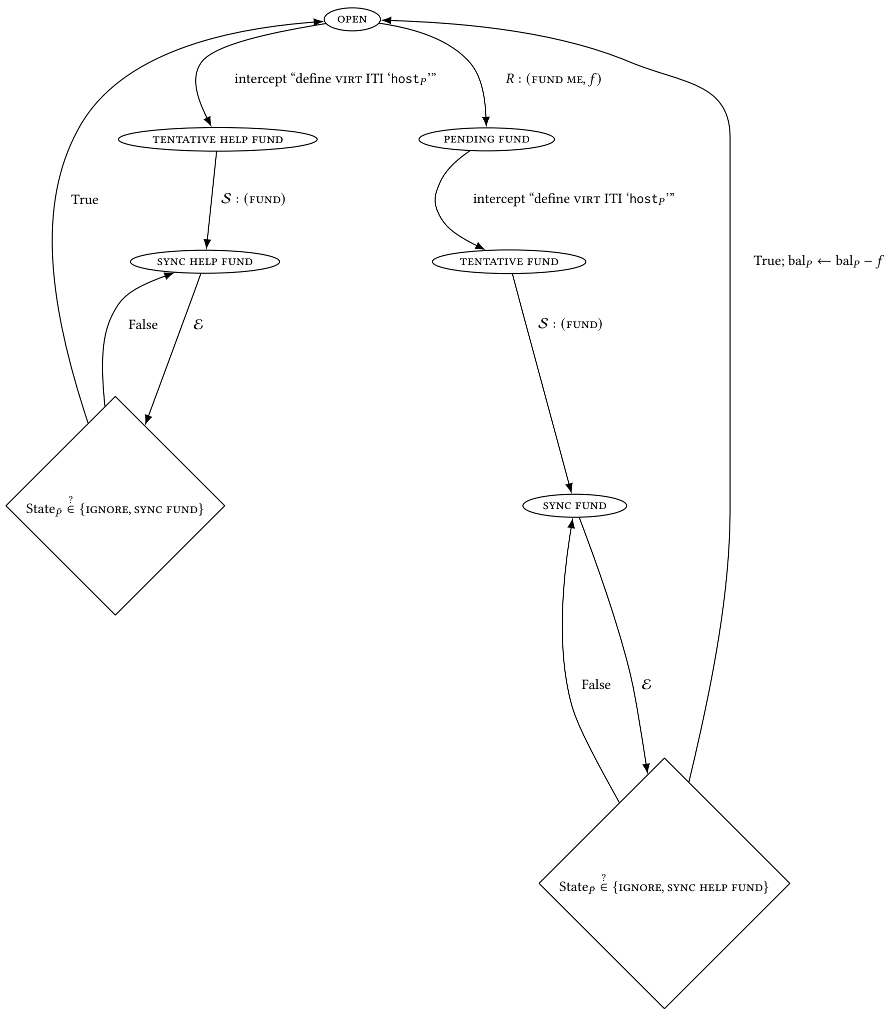

Figure 27: $\mathcal{G}_{\text{Chan}}$ state machine for funding new virtuals (both parties)

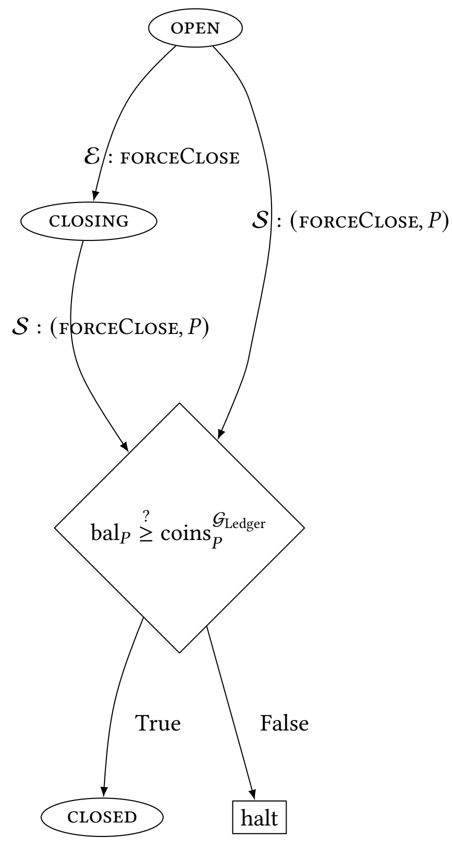

Figure 28: $\mathcal{G}_{\text{Chan}}$ state machine for channel closure (both parties)

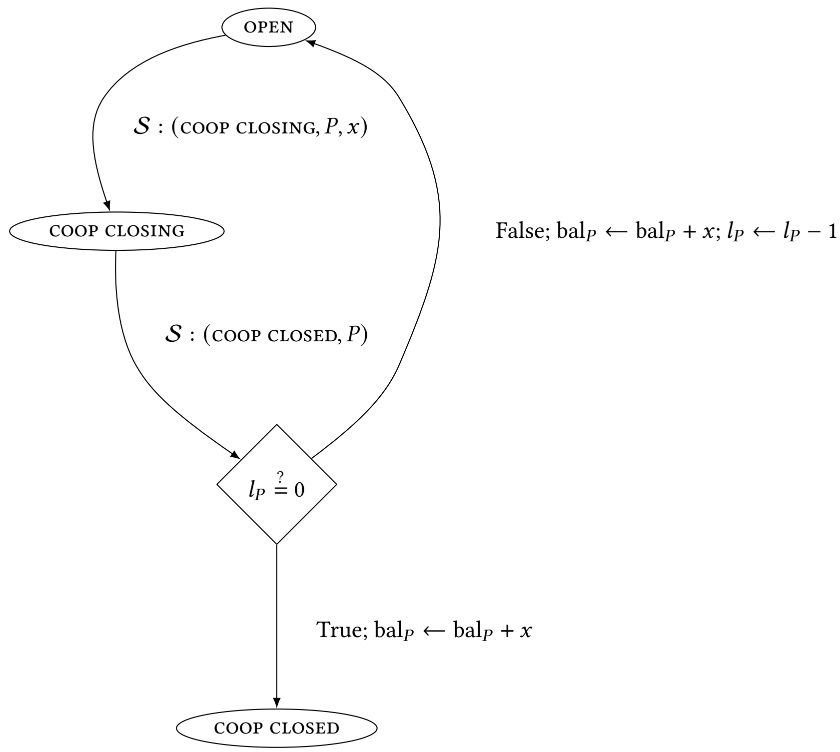

Figure 29: $\mathcal{G}_{\text{Chan}}$ state machine for cooperative channel closure (all parties)

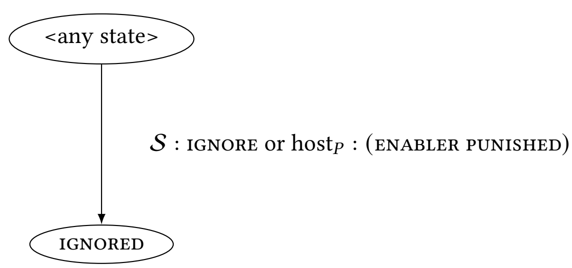

Figure 30: $\mathcal{G}_{\text{Chan}}$ state machine for corruption, negligence or punishment of the counterparty of a lower layer (both parties)