

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 equitably 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.

Index Terms—Payment channels; Blockchain; Security

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 $\sim 1\text{h}$ [50]) 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. Crucially, 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.

Before the advent of virtual channels, closing a payment channel involved posting the channel state on-chain. Closing should be efficient, i.e., needing $O(1)$ on-chain transactions, independent of how many payments have occurred 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 stop an honest party from accessing their funds. This functionality however raises an crucial design consideration: how to prevent malicious parties from posting old states of the channel. This issue can be addressed with a suitable use of *timelocks*, a native Bitcoin Script instruction 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 cost of bounding the overall lifetime of a channel. Using 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 send funds to each other as long as there is a route of payment channels that connects them. This mechanism has the downside of requiring all parties along the path to be actively involved for each payment. Instead, *virtual payment channels* suggest the more attractive approach of an one-time off-chain initialization to set up a virtual payment channel over the preexisting channels, which can then be used for direct payments with complexity—in the optimistic case— independent of the path length. When the virtual channel outlives 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 [27], [29], while later work [2] brought them closer to Bitcoin compatibility by leveraging adaptor signatures [1]. We note that Bitcoin imposes very strict design

limitations, as it does not support Turing-complete smart contracts nor covenants [52].

We call the parties of the underlying channels *intermediaries*. A virtual channel constructor can be thought of as an *operator* 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 for 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 incur 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 channels on Bitcoin [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 [29], [2], [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 when the ledger is consistently congested. Without recursiveness and when facing unresponsive channel parties, one has 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. To move only part the channel coins to a new channel, a party still has to close the entire original channel, even if the channel parties are collaborative. In contrast, 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.

A scenario that needs 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. She later 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 her coins from her virtual channel with Dave to a new one with Eve – this needs the recursive property.

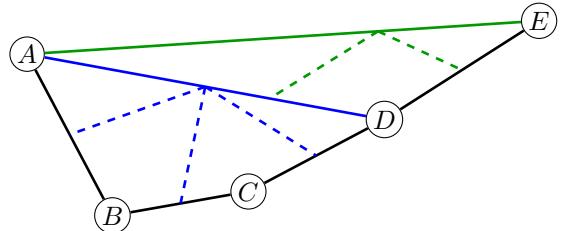


Figure 1: An Elmo network with 5 nodes, 4 black *simple* (i.e., on-chain) channels, 1 blue virtual channel built only on simple ones, and 1 green virtual channel built on the blue virtual and a simple one. Each virtual channel is connected with its *base* channels with dashed lines of the same colour.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

quent 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. C); our *global* (Appx. D) functionality $\mathcal{G}_{\text{Chan}}$ (Appx. J.1) represents a single channel (on globality, see [8] and Sec. 3). We also guarantee that the real-world protocol Π_{Chan} has *balance security* [31], i.e., that honest parties can close the channel and get the expected coins on-chain (Lemma 1). Elmo requires the ANYPREVOUT signature type, a candidate for inclusion in the next Bitcoin update [22]. Today, every signature of a Bitcoin transaction T must sign the hash of all transactions spent by T (more precisely, the spent *outpoints*), thus we currently cannot store a signed transaction T off-chain and decide only later which on-chain transaction(s) it will spend. An ANYPREVOUT signature fixes this by not including this hash in the signed message. This enables a transaction signed with an ANYPREVOUT signature to spend any output as long as it has a suitable script, even an output that was unknown at the time of signing.

We conjecture that without ANYPREVOUT no efficient virtual channel over Bitcoin can be built. In particular, if any protocol (i) has an efficient closing operation with $O(1)$ on-chain transactions, (ii) has parties locally store the channel state as transactions and signatures and (iii) does not need locking up coins as collateral (as [3] does), then each party needs space exponential in the number of intermediaries. The second 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 their counterparties' relevant signatures to ensure they can unilaterally exit later.

Related work. Perun [27], Thunderdome [6], and GSCN [29] build virtual state channels on the Turing-complete language of Ethereum. GSCN uses a per-channel functionality and a recursive argument similar to that of our UC-security analysis. However, their security argument is flawed, as they incorrectly argue that every level is subroutine respecting wrt 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, e.g., LN. 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 useful 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 Donner requires the funder to keep locked and unusable an amount of coins equal to those in the virtual channel, creating a considerable opportunity cost. As this is not needed in other constructions, the funder's collateral in Donner is double that of LVPC or Elmo. The collateral for all other parties is the same across LVPC, Donner, and Elmo. Additionally, contrary to Elmo, a Donner channel needs active periodic collaboration of all channel parties to refresh it, thus Donner channels do not have an unlimited lifetime. We conjecture that using external coins precludes variadic virtual channels with unlimited lifetime. This 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 cost. 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, and boasts a simpler design. Their work also introduces *Domino*, which we discuss in Appx. G. We briefly mention here that, contrary to what is claimed in [3], Domino does not force the base channels of an Elmo virtual channel to close.

We refer the reader to Appx. I 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. Future work can be found in Appx. H.

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*. Otherwise they can open it on top of a path of preexisting *base channels* $(P_1, P_2), (P_2, P_3), \dots, (P_{n-1}, P_n)$. Then (P_1, P_n) is a *virtual channel*. A channel is either simple or virtual, not both. Elmo is recursive, thus each base channel may itself be simple or virtual. To open a virtual channel, all parties on the path set aside funds in their

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

channels as collateral; to do this they create $O(n^3)$ so-called *virtual* and *bridge* transactions (txs) in total ($O(n^2)$ stored per party) that essentially tie the spending of two adjacent base channels into a single atomic action. These txs are kept off-chain. The bridge txs have a *funding* output with the sum of P_1 and P_n 's channel balance. To complete channel opening, P_1 and P_n then apply the logic of simple channels (see below) on top of the (off-chain) funding output. 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.

In more detail, the channel opening protocol steps (Appx. L, Fig. 38) are as follows: the two counterparties (i) create new signing keypairs and exchange the resulting public keys, then (ii) if the channel is virtual, they prepare the underlying base channels, next (iii) they exchange signatures for their initial *commitment* txs (see Simple channel below) 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. A **payment** over an established channel (see Simple channel below) follows a procedure inspired by LN.

Parties P_1, \dots, P_n can **optimistically close** a virtual channel fully off-chain. At a high level, parties controlling base channels *revoke* their *virtual* txs and the 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 channel state) in their base channels $((P_1, P_2) \text{ and } (P_{n-1}, P_n))$ respectively).

Finally, to **unilaterally close** a virtual channel (P_1, P_n) in case cooperation fails, either P_1 or P_n sign and publish some txs on-chain. In the simplest case, P_1 publishes her virtual tx. This prompts P_2 to publish her relevant virtual tx and so on up to P_{n-1} . The funding output of (P_1, P_n) is then on-chain and closing can proceed as for simple channels. If instead any intermediary stays inactive, then a timelock expires turning an output into 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. We assume that parties already have established secure communication channels. In practice, peer discovery can be done as in LN.

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. B for more details.

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*.

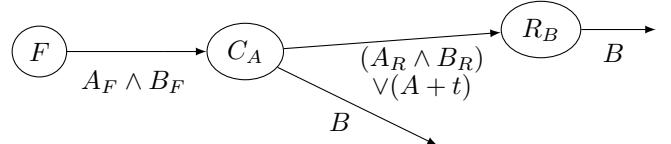


Figure 2: A - B simple channel: Funding (F), A 's commitment (C_A) and B 's revocation (R_B) txs. $A_F \wedge B_F$ is the funding output. The (symmetric) B 's commitment and A 's revocation txs are omitted. A_F is A 's funding key, A_R her revocation key, $A \wedge B$ needs a signature by both A and B . $A + t$ needs a signature by A after relative timelock t . The top C_A output is spendable either by both A_R and B_R , or by $A + t$, the “or” is denoted with \vee .

The funding tx F moves A 's initial coins to a 2-of-2 *multisig*, i.e., an output that needs signatures from 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 , Fig. 2) spends F 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 the countdown starts at the moment of publication). This is so that B has time to *punish* A if she cheats. B 's commitment tx C_B is symmetric. We recommend using $t = 5$ days as a default, the same as LN.

When A *pays* c coins to B , 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 have c more. They sign them and swap signatures. To ensure only one set of commitment txs is valid, they then revoke their previous ones by building and

signing *revocation* txs for the previous commitment txs. B 's revocation tx (R_B , 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 coins. A payment needs 3 messages in total. Note that, after generating the new commitment txs and before revoking the old 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.

To unilaterally *close* (A, B) , A or B publish the latest commitment tx on-chain and wait for timelock expiration. The last commitment tx is not revoked, thus punishment is impossible. Simple Elmo channels work like LN simplified.

Virtual channel. Let channels $(A, B), \dots, (D, E)$ be open and each “left” party own at least c coins. Let us see how to open the virtual channel (A, E) with A initially having c coins using $(A, B), \dots, (D, E)$ as base channels. Each base channel may be simple or virtual. If any one is virtual, then (A, E) is recursive. Thanks to the similarity of all layers, the description below is identical in both cases. First, the 5 parties 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).

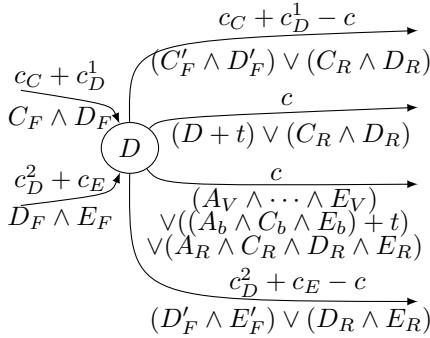


Figure 3: $A-E$ virtual channel: D 's initiator virtual tx. Spends the funding outputs of the $C-D$ and $D-E$ channels. D can use it if neither C nor E have published a virtual tx yet. c_D^1/c_D^2 : D 's coins in $C-D/D-E$ channel. A_V : A 's virtual key. A_b : A 's bridge key.

Next, parties generate and sign these virtual txs and exchange the signatures. These txs form Elmo's core. Intuitively, they force each intermediary to interact with both of its base channels instead of one at a time, so that virtual channel information flows along the base channels. There are 3 virtual tx types – looking ahead, we explain them based on how they help close the channel on-chain. The

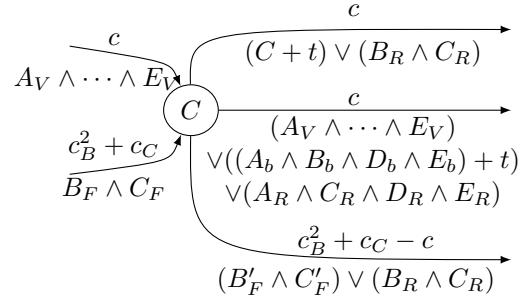


Figure 4: $A-E$ virtual channel: One of C 's virtual extend-interval txs. Spends the virtual output of D 's initiator tx and the funding output of the $B-C$ channel. C can use it if D has already published its initiator tx and B has not published a virtual tx yet.

initiator virtual tx spends the two funding outputs of an intermediary (say D) and needs to be signed by C_F , D_F , and E_F . It has 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 by the top input of an *extend-interval* tx, the second type of virtual tx (Fig. 4). It needs signatures from the *virtual keys* $A_V \dots E_V$ of all parties. This tx also spends the other 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 (Fig. 5). The format of an endpoint's initiator tx is different. 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 stores. 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 $A_V \dots E_V$. 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 have two properties: First, each intermediary can publish at most 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 because the unclaimed virtual output turns into the funding output of the virtual channel after a timelock, see, e.g., Fig. 3, output 3, spending method 2.

Keys $A_b \dots E_b$ (Figs. 3–6) can be spent by *bridge* txs, the output of which funds (A, E) . Bridge txs allows endpoints to only store a single commitment tx instead of

$O(n^3)$. Any virtual output may end up funding the virtual channel, but each virtual output has a different script, thus no single commitment tx is able to spend all of them. Bridge txs fix this by having the same output. This makes virtual channel updates as cheap as simple channel updates.

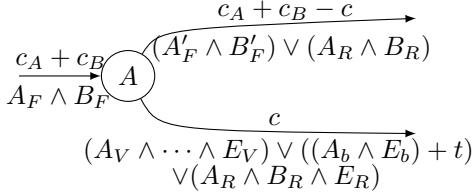


Figure 5: $A\text{-}E$ virtual channel: A 's virtual initiator tx. Spends the funding output of the $A\text{-}B$ channel. A can use it if B has not published a virtual tx.

Some considerations remain to ascertain the scheme's security. Firstly, if all virtual outputs were identical, an adversary could fabricate arbitrarily many virtual outputs using its own, external to the protocol, coins and publish a perpetual stream of merge-intervals txs that spend one valid and one fabricated virtual output. This would prevent an honest endpoint from accessing their funding output. We avoid this 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 published one cannot be spent by another virtual tx. This ensures that the endpoints will eventually obtain a funding output, but needs intermediaries to store $O(n^2)$ virtual txs for a virtual channel over n parties. Secondly, we have to carefully select the values of timelocks to ensure that each party has enough time to act. Timelocks increase linearly with the recursion depth. The exact values are shown in Appx. L.

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 — see Sec. 4 for more details.

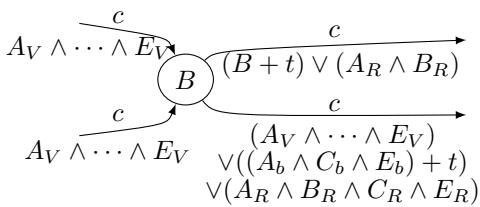


Figure 6: $A\text{-}E$ virtual channel: One of B 's merge-intervals txs. Spends the virtual outputs of A 's and C 's virtual txs. B can use it if both A and C have already published their initiator or extend-interval tx.

At last, the virtual layer is set up: Both A and E can unilaterally force a funding output for channel (A, E) on-chain, no matter what the rest of the parties do. 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. A for the protocol pseudocode and Appx. K for more details.

3. 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 1 which discusses the conservation of funds. The formal statements (3 and 4) along with all proofs are deferred to Appx. M. Informally, it establishes that if an honest was implicated in a channel that has now been unilaterally closed, then the party will have at least the expected funds on-chain.

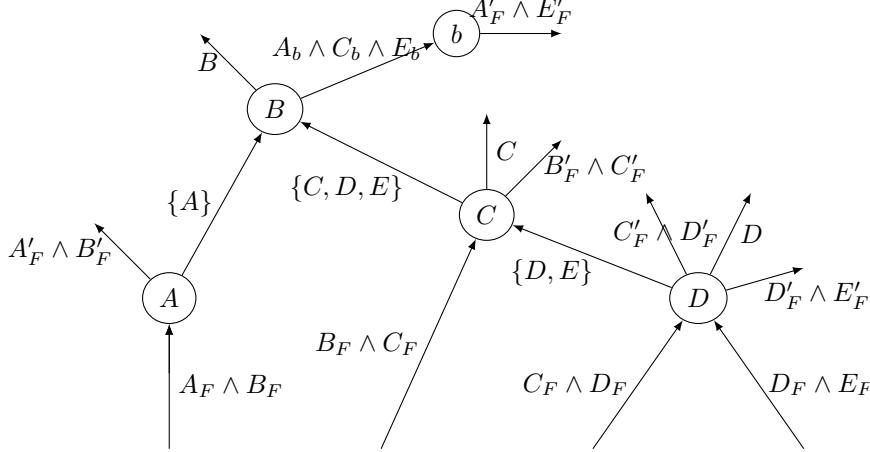


Figure 7: Example of on-chain (unilateral) closing of virtual channel (A, E) , built on top of simple channels $(A, B), \dots, (D, E)$. A and D start closing by publishing their initiator virtual txs (Fig. 5 and 3 respectively), then C publishes its suitable extend-interval virtual tx (Fig. 4), after which B publishes its suitable merge-intervals virtual tx (Fig. 6). 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 such as b are used by A and E to convert the various virtual outputs into the same funding output $A'_F \wedge E'_F$, as ANYPREVOUT only works across identical outputs.

Lemma 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 funder 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 funded 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 performs a successful unilateral close, eventually there will be $h \in \mathbb{N}^*$ outputs on-chain spendable only by P , each of value c_i , such that

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

The expected funds are [initial balance - funds for funded virtuals + funds returned from funded virtuals - outbound payments + inbound payments]. A similar property-based guarantee is provided for the ideal-world functionality $\mathcal{G}_{\text{Chan}}$ in Lemma 4.

Proof Sketch [Lemma 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. \square

It is important to note that in fact our protocol Π_{Chan} (Appx. L) provides a stronger guarantee: a party can always successfully 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. J) halts on security breaches (e.g., lower than expected balance). We prove the “no-halt” Lemma (5), which informally states that if an ideal party is honest, $\mathcal{G}_{\text{Chan}}$ does not halt with overwhelming probability.

4. 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 – 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). See Appx. E for comparison details and choices.

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

Open													
	Funder			Fundee			Intermediary			Total			
	party rounds	size	sent	stored	party rounds	size	sent	stored	party rounds	size	sent	stored	
LVPC	8($n - 2$)	1381($n - 2$)	3005($n - 2$)		7	1254	2936		16	2989	6385	4370 $n - 8740$	9390 $n - 18780$
Donner	2	184 $n + 829$	1332.5 $k + 43n + 125.5$		1	43 $n + 192.5$	1332.5 $k + 43n + 125.5$		1	547	1332.5 $k + 43n + 125.5$	774 $n - 71$	1332.5 $kn + 43n^2 + 125.5n$
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	2 $n - 2$	435 $n - 510.5$	
Donner	1	204.5	4	704 + 43 n	1	136.5	2 n	458 $n - 26$	
Elmo	1	297.5	3	376	3	376	$n + 1$	254.5 $n - 133$	

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

Delays (sec) and Relative Fees							
	Power Law		Preferred Receiver		Uniform		Fees
	Delay (sec)	Fees	Delay (sec)	Fees	Delay (sec)	Fees	
LVPC	2873	0.81	2440	0.80	3589	0.99	
Donner	2842	1	2361	1	3589	1	
Elmo	2864	0.83	2344	.87	3589	0.99	

Table 4: Average per-payment delay (both on- and off-chain) and relative fee. Less is better. Blue is best, red is worst.

funder, fundee and intermediary requirements, along with the aggregate for all parties.

Regarding closing, 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 [47], [51] 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 tx^{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 [48], 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). Moreover, Elmo can be deployed on commodity hardware: To set up and keep a channel of length $n = 6$ active, 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 framework¹ in which a list of randomly generated payments are carried out. A simulation run 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. Our framework is of independent interest, as it can be used for evaluating various payment networks.

We here show the performance of the 3 protocols with respect to the metrics payment channels aim to improve, namely payment latency and fees (Table 4). 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. See Appx. F for more details.

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.

1. gitlab.com/anonymised-submission-8778e084/virtual-channels-simulation

References

- [1] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sánchez, and S. Riahi, “Generalized bitcoin-compatible channels,” *IACR Cryptol. ePrint Arch.*, p. 476, 2020.
- [2] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sánchez, “Bitcoin-compatible virtual channels,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 901–918.
- [3] L. Aumayr, P. Moreno-Sánchez, A. Kate, and M. Maffei, “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, 2023. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/breaking-and-fixing-virtual-channels-domino-attack-and-donner/>
- [4] L. Aumayr, S. A. K. Thyagarajan, G. Malavolta, P. Moreno-Sánchez, and M. Maffei, “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*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 179–192. [Online]. Available: <https://doi.org/10.1145/3548606.3559370>
- [5] G. Avarikioti, E. K. Kogias, R. Wattenhofer, and D. Zindros, “Brick: Asynchronous payment channels,” 2020.
- [6] Z. Avarikioti, Y. Wang, and Y. Wang, “Thunderdome: Timelock-free rationally-secure virtual channels,” 2025. [Online]. Available: <https://arxiv.org/abs/2501.14418>
- [7] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, “Enabling blockchain innovations with pegged sidechains,” 2014.
- [8] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas, “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*, 2020, pp. 1–30.
- [9] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas, “Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 913–930.
- [10] C. Badertscher, J. Hesse, and V. Zikas, “On the (ir)replaceability of global setups, or how (not) to use a global ledger,” in *Theory of Cryptography*, K. Nissim and B. Waters, Eds. Cham: Springer International Publishing, 2021, pp. 626–657.
- [11] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, “Bitcoin as a transaction ledger: A composable treatment,” in *Annual International Cryptology Conference*. Springer, 2017, pp. 324–356.
- [12] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener, “Graph structure in the web,” *Comput. Networks*, vol. 33, no. 1-6, pp. 309–320, 2000. [Online]. Available: [https://doi.org/10.1016/S1389-1286\(00\)00083-9](https://doi.org/10.1016/S1389-1286(00)00083-9)
- [13] C. Burchert, C. Decker, and R. Wattenhofer, “Scalable funding of bitcoin micropayment channel networks,” in *The Royal Society*, 2018.
- [14] V. Buterin, “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] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” *Cryptology ePrint Archive*, Paper 2000/067, 2000, <https://eprint.iacr.org/2000/067>. [Online]. Available: <https://eprint.iacr.org/2000/067>
- [16] ———, “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*, 2001, pp. 136–145.
- [17] M. M. T. Chakravarty, S. Coretti, M. Fitzi, P. Gazi, P. Kant, A. Kiayias, and A. Russell, “Hydra: Fast isomorphic state channels,” *Cryptology ePrint Archive*, 2020/299.
- [18] M. M. T. Chakravarty, R. Kireev, K. MacKenzie, V. McHale, J. Müller, A. Nemish, C. Nester, M. Peyton Jones, S. Thompson, R. Valentine, and P. Wadler, “Functional blockchain contracts,” 2019, <https://iohk.io/en/research/library/papers/functional-blockchain-contracts/>.
- [19] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer et al., “On scaling decentralized blockchains,” in *Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.
- [20] G. Danezis and I. Goldberg, “Sphinx: A compact and provably secure mix format,” in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 269–282.
- [21] C. Decker, R. Russell, and O. Osuntokun, “eltoo: A simple layer2 protocol for bitcoin.”
- [22] C. Decker and A. Towns, “Sighash_anyprevout for taproot scripts.”
- [23] C. Decker and R. Wattenhofer, “A fast and scalable payment network with bitcoin duplex micropayment channels,” in *Symposium on Self-Stabilizing Systems*. Springer, 2015, pp. 3–18.
- [24] M. Dong, Q. Liang, X. Li, and J. Liu, “Celer network: Bring internet scale to every blockchain,” 2018.
- [25] R. Dunford, Q. Su, and E. Tamang, “The pareto principle,” *The Plymouth Student Scientist*, vol. 7, pp. 140–148, 2014. [Online]. Available: <http://hdl.handle.net/10026.1/14054>
- [26] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, “Multi-party virtual state channels,” in *International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT*, Y. Ishai and V. Rijmen, Eds., 2019, pp. 625–656.
- [27] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, “Perun: Virtual payment hubs over cryptocurrencies,” in *IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2019, pp. 344–361.
- [28] S. Dziembowski, G. Fabiański, S. Faust, and S. Riahi, “Lower bounds for off-chain protocols: Exploring the limits of plasma,” *Cryptology ePrint Archive*, Report 2020/175.
- [29] S. Dziembowski, S. Faust, and K. Hostáková, “General state channel networks,” in *Computer and Communications Security, CCS*, 2018, pp. 949–966.
- [30] C. Egger, P. Moreno-Sánchez, and M. Maffei, “Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks,” in *Conference on Computer and Communications Security, SIGSAC*, ser. CCS ’19, 2019, pp. 801–815.
- [31] G. Fabiański, R. Stefański, and O. S. Thyfronitis Litos, “A formally verified lightning network,” in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2025.
- [32] P. Gaži, A. Kiayias, and D. Zindros, “Proof-of-stake sidechains,” in *Symposium on Security and Privacy, SP*, 2019, pp. 677–694.
- [33] M. Green and I. Miers, “Bolt: Anonymous payment channels for decentralized currencies,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 473–489.
- [34] L. Gudeon, P. Moreno-Sánchez, S. Roos, P. McCorry, and A. Gervais, “Sok: Layer-two blockchain protocols,” in *Financial Cryptography and Data Security FC*, 2020, pp. 201–226.
- [35] J. Harris and A. Zohar, “Flood & loot: A systemic attack on the lightning network,” in *Conference on Advances in Financial Technologies*, ser. AFT, 2020, pp. 202–213.
- [36] M. Jourenko, M. Larangeira, and K. Tanaka, “Lightweight virtual payment channels,” in *Cryptology and Network Security*, 2020, pp. 365–384.

- [37] R. Khalil and A. Gervais, “Revive: Rebalancing off-blockchain payment networks,” in *Conference on Computer and Communications Security, CCS*, 2017, pp. 439–453.
- [38] A. Kiayias and O. S. T. Litos, “A composable security treatment of the lightning network,” in *Computer Security Foundations Symposium, CSF*, 2020, pp. 334–349.
- [39] A. Kiayias and D. Zindros, “Proof-of-work sidechains,” in *Workshop on Trusted Smart Contracts*, 2019, pp. 21–34.
- [40] G. Konstantopoulos, “Plasma cash: Towards more efficient plasma constructions,” 2019.
- [41] J. Lee, S. Kim, S. Park, and S.-M. Moon, “Routee: A secure payment network routing hub using trusted execution environments,” 2020.
- [42] J. Liao, F. Zhang, W. Sun, and W. Shi, “Speedster: An efficient multi-party state channel via enclaves,” in *Asia Conference on Computer and Communications Security, ASIACCS*, 2022, pp. 637–651.
- [43] J. Lind, I. Eyal, P. R. Pietzuch, and E. G. Sirer, “Teechan: Payment channels using trusted execution environments,” *CoRR*, vol. abs/1612.07766, 2016.
- [44] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, “Teechain: A secure payment network with asynchronous blockchain access,” in *Symposium on Operating Systems Principles*, ser. SOSP ’19, 2019, pp. 63–79.
- [45] Y. Lindell, “How to simulate it - A tutorial on the simulation proof technique,” in *Tutorials on the Foundations of Cryptography*, 2017, pp. 277–346.
- [46] G. Malavolta, P. Moreno-Sánchez, C. Schneidewind, A. Kate, and M. Maffei, “Anonymous multi-hop locks for blockchain scalability and interoperability,” in *Network and Distributed System Security Symposium, NDSS*, 2019.
- [47] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, “Simple schnorr multi-signatures with applications to bitcoin,” *Des. Codes Cryptogr.*, vol. 87, no. 9, pp. 2139–2164, 2019. [Online]. Available: <https://doi.org/10.1007/s10623-019-00608-x>
- [48] S. Milgram, “The small world problem,” *Psychology Today*, vol. 1, pp. 61–67, May 1967.
- [49] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, “Sprites and state channels: Payment networks that go faster than lightning,” 2017, arXiv:1702.05812.
- [50] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [51] J. Nick, T. Ruffing, and Y. Seurin, “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*, ser. Lecture Notes in Computer Science, T. Malkin and C. Peikert, Eds., vol. 12825. Springer, 2021, pp. 189–221. [Online]. Available: https://doi.org/10.1007/978-3-030-84242-0_8
- [52] R. O’Connor and M. Piekska, “Enhancing bitcoin transactions with covenants,” in *Financial Cryptography and Data Security*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 191–198.
- [53] Optimism, “Optimistic rollup overview,” <https://github.com/ethereum-optimism/optimistic-specs/blob/0e9673af0f2cafdb89ac7d6c0e5d8bed7c67b74ca/overview.md>.
- [54] J. Poon and V. Buterin, “Plasma: Scalable autonomous smart contracts.”
- [55] J. Poon and T. Dryja, “The bitcoin lightning network: Scalable off-chain instant payments,” <https://lightning.network/lightning-network-paper.pdf>, January 2016.
- [56] D. M. W. Powers, “Applications and explanations of Zipf’s law,” in *New Methods in Language Processing and Computational Natural Language Learning*, 1998.
- [57] P. Prihodko, S. Zhigulin, M. Sahno, A. Ostrovskiy, and O. Osuntokun, “Flare: An approach to routing in lightning network,” 2016.
- [58] S. Riahi and O. S. T. Litos, “Bitcoin clique: Channel-free off-chain payments using two-shot adaptor signatures,” in *Applied Cryptography and Network Security*, C. Pöpper and L. Batina, Eds. Cham: Springer Nature Switzerland, 2024, pp. 28–50.
- [59] V. Sivaraman, S. B. Venkatakrishnan, M. Alizadeh, G. C. Fanti, and P. Viswanath, “Routing cryptocurrency with the spider network,” *CoRR*, vol. abs/1809.05088, 2018.
- [60] J. Spilman, “Anti dos for tx replacement,” <https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html>, April 2013.
- [61] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger.”
- [62] L. Zhao, H. Shuang, S. Xu, W. Huang, R. Cui, P. Bettadpur, and D. Lie, “Sok: Hardware security support for trustworthy execution,” 2019.

Appendix A. Protocol Pseudocode

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

Process $\Pi_{\text{Chan}} - \text{self is } P$

- 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 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 virtual and bridge txs (b of Fig. 7) for all parties ($O(n^3)$ txs) 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**
- $t_P \leftarrow \max\{t \text{ of left channel}, t \text{ of right channel}\}$
 - else // we are endpoint**
 - $t_P \leftarrow p + \sum_{j=2}^{n-1} (s - 1 + t_j)$ // max delay is $O(\text{sum of intermediaries' delays})$. 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. K.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

Appendix B. 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. $m/\{pk_1, \dots, pk_n\}$ is an m -of- n multisig ($m \leq n$), i.e., a spending condition that needs signatures from m distinct keys among sk_1, \dots, 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.

B.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 Appx. A 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 9).

In particular, there is a single on-chain funding transaction that spends $c_A + c_B$ coins (originally belonging to

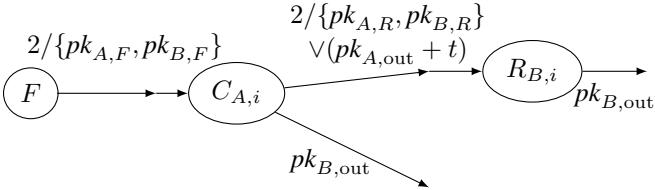


Figure 9: Funding, commitment, and revocation transactions. Inputs and outputs are represented by separate arrows: The input of $C_{A,i}$ spending the output of F is shown as two connected arrows.

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 j th revocation tx held by an endpoint spends the output carrying the counterparty's funds in the counterparty's j th 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.

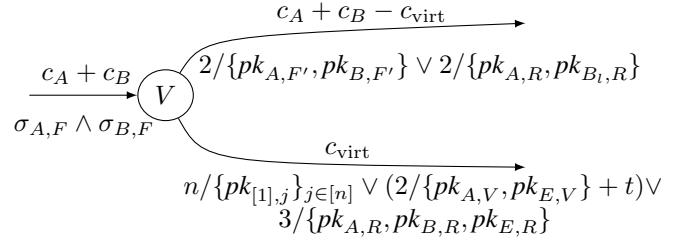


Figure 10: $A - E$ virtual channel: A 's initiator transaction. Spends the funding output of the $A - B$ channel. Can be used if B has not published a virtual transaction yet.

B.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. Appx. A), 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 10, 59). 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.

Intermediaries on the other hand store three sets of virtual transactions (Figure 58): *initiator* (Figure 11), *extend-interval* (Figure 12) and *merge-intervals* (Figure 13). 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.

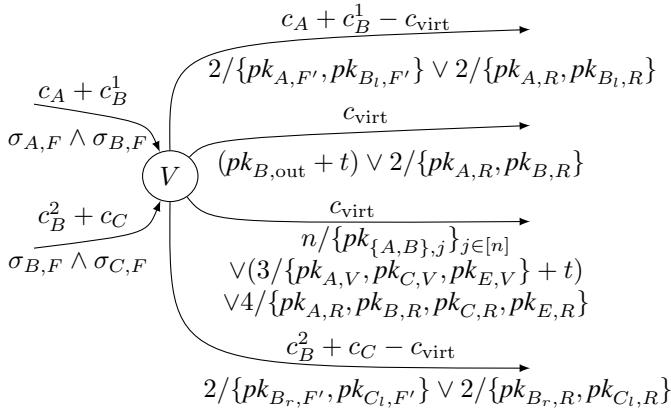


Figure 11: $A - E$ virtual channel: B 's initiator transaction. Spends the funding outputs of the $A - B$ and $B - C$ channels. Can be used if neither A nor C have published a virtual transaction yet.

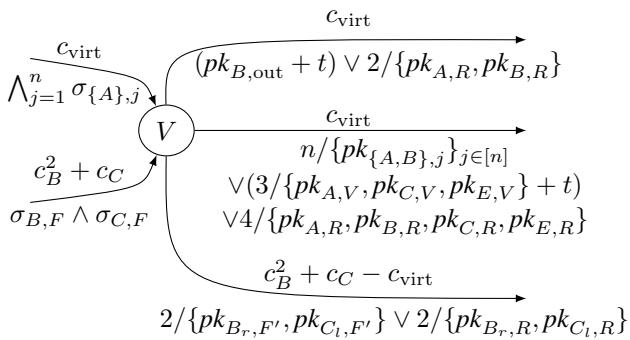


Figure 12: $A - E$ virtual channel: One of B 's extend-interval transactions. σ is the signature. Spends the virtual output of A 's initiator transaction and the funding output of the $B - C$ channel. Can be used if A has already published its initiator transaction and C has not published a virtual transaction yet.

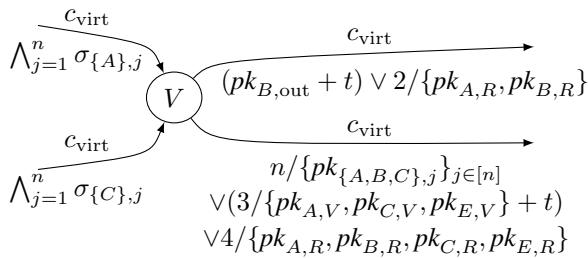


Figure 13: $A - E$ virtual channel: One of B 's merge-intervals transactions. Spends the virtual outputs of A 's and C 's virtual transactions. Usable if both A and C have already published their initiator transactions.

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 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-timedlocked 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 [47], [51] 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 2 of Appx. B.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 j th 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.² 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 34). 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 51, 52, 53, 54 and 70). 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 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

² 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.

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 14). 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.

B.3. Liveness

Proposition 2. Consider a synchronised honest party that submits a transaction tx to the ledger functional-

ity [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 $s = (2+q)\text{windowSize}$ and $q = \lceil(\maxTime_{\text{window}} + \frac{\text{Delay}}{2})/\minTime_{\text{window}}\rceil$.

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 $\tau_h + \maxTime_{\text{window}}$ another windowSize blocks are added to state and thus $\tau_h^U \leq \tau_h + \maxTime_{\text{window}}$.

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

Appendix C. 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 [45]. For a particular real world protocol, the main goal of UC is to 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

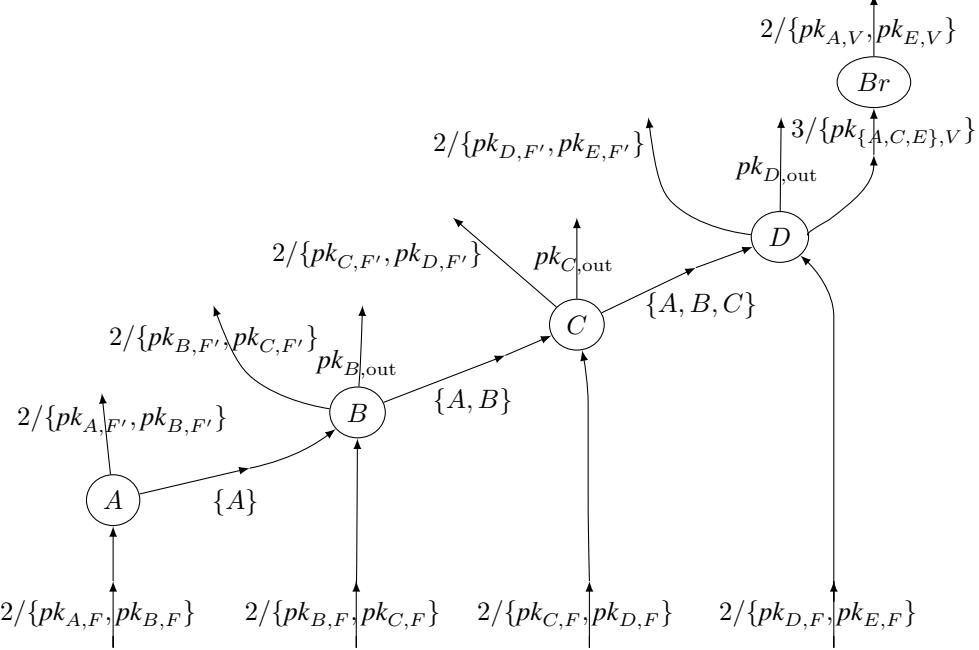


Figure 14: 4 simple channels supporting a virtual. A starts closing by publishing its initiator tx, then parties $B-D$ each publishes its extend-interval tx with the relevant interval. No party is negligent. Virtual outputs are marked with their interval.

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

Appendix D. On Globality

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 [38] 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_{Chan}^n . Π_{Chan}^1 is always a simple channel. This is done by ignoring instructions to OPEN on top of other channels. As for higher superscripts, $\forall n \in \mathbb{N}^*$, Π_{Chan}^{n+1} is the same as Π_{Chan} but with base channels of a maximum superscript n . It then holds that $\forall n \in \mathbb{N}^*$, Π_{Chan}^n is $(\mathcal{G}_{\text{Ledger}}, \Pi_{\text{Chan}}^1, \dots, \Pi_{\text{Chan}}^{n-1})$ -*subroutine respecting* [8]. The same superscript trick is done to $\mathcal{G}_{\text{Chan}}$, thus the composition theorem of [8] is applicable (Appx. M). 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 6 and 7 (Appx. M) state that $\forall n \geq 1$, Π_{Chan}^n UC-realises $\mathcal{G}_{\text{Chan}}^n$. Furthermore, all ideal global subroutines can be replaced with their real counterparts (Lemma 8 and Theorem 9).

Appendix E. Cost calculation details

E.1. Details on off-chain opening efficiency comparison (Table 2)

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 specification³), 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),

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

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.

E.2. Details on on-chain closing efficiency comparison (Table 3)

For the closing comparison, we measure on-chain transactions' size in vbytes⁴, 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 tool⁵ 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.

Appendix F. Payment simulation details

Several knowledge functions are provided, such as full knowledge of all future payments and knowledge of the

4. https://en.bitcoin.it/wiki/Weight_units

5. <https://jlopp.github.io/bitcoin-transaction-size-calculator/>

payer’s next m payments. 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 $\sim 10'$ 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.

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 $[0, \text{max}]$ range, for $\text{max} = \frac{(\text{initial coins}) \cdot \# \text{players}}{\# \text{payments}}$. 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] distribution with parameter 2, which corresponds to real-world *power-law* distributions with a heavy tail [12]. Each payment value is chosen according to the $\text{zipf}(2.16)$ distribution which corresponds to the 80/20 rule [25], moved to have a mean equal to $\frac{\text{max}}{2}$. 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 $[0, \text{max}]$, 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. The coins that fund new virtual channels are essentially decided 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. 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 Table 4 provides relative, not absolute, fees.

As Table 4 shows, delays are primarily influenced by the payment distribution and only secondarily by the protocol: The preferred receiver is the fastest and the 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.

Appendix G. Discussion on Domino

In [3] a process called “Domino attack” is presented, which erroneously claims that a malicious virtual channel member can force others’ channels to close. According to Domino, if A and E of Fig. 7 unilaterally close virtual channel (A, E) , they then force base channels (B, C) and (C, D) to close. This is not the case. Contrary to Domino’s goal, honest parties B, C, D are only forced to publish a single virtual tx each, which simply places their funding outputs on-chain; channels (B, C) and (C, D) remain open. Contrary to the authors’ claims, Domino is not an attack against Elmo. 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 tx. 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 txs. 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.

A simple modification to Elmo eliminates the channel capacity reduction under a Domino scenario, 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 scenario. 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.

Appendix H. 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 we only allow a channel to be funded by 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 31) and s (Figure 35), 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 2 (Appx. B.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.

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 $O(\log n)$ space⁶, 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

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

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, 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.

Appendix I. Further 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.

Various attacks have been identified against LN. The wormhole attack [46] 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 [59], [57], [41] is another research area that aims to improve network efficiency without sacrificing privacy. Actively rebalancing channels [37] can further increase network efficiency by reducing unavailable routes due to lack of well-balanced funds.

An alternative 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.

Bolt [33] constructs privacy-preserving payment channels enabling both direct payments and payments with a single untrusted intermediary. Sprites [49] 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 [42], [26].

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 [43], [44], [42], [41] 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.

Solutions alternative to PCNs include sidechains (e.g., [7], [32], [39]), commit-chains (e.g., [54], [58]) and non-custodial chains (e.g., [54], [40], [28]). These approaches offer more efficient payment methods, at the cost of requiring a distinguished mediator, additional trust, 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.

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.

Appendix J. Functionalities & Simulator

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

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

- On receiving input (msg) by \mathcal{E} addressed to $P \in \{\text{Alice}, \text{Bob}\}$, handle it according to the corresponding rule in Fig. 16, 17, 18, 19 or 20 (if any) and subsequently send (RELAY, msg , P , \mathcal{E} , input) to \mathcal{A} .
- On receiving (msg) by party R addressed to $P \in \{\text{Alice}, \text{Bob}\}$ by means of mode $\in \{\text{output, network}\}$, handle it according to the corresponding rule in Fig. 16, 17, 18, 19 or 20 (if any) and subsequently send (RELAY, msg , P , \mathcal{E} , mode) to \mathcal{A} . // all messages are relayed to \mathcal{A}
- On receiving (RELAY, msg , P , R , mode) by \mathcal{A} (mode $\in \{\text{input, output, network}\}$, $P \in \{\text{Alice}, \text{Bob}\}$), 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. 16, 17, 18, 19 or 20 (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 15

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

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

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

- 5: On (BECAME CORRUPTED OR NEGLIGENT, P) by \mathcal{A} or on output (ENABLER USED REVOCATION) by $host_P$ when in any state:
- 6: $State_P \leftarrow \text{IGNORED}$
- 7: On (INIT, pk) by P when $State_P = \text{UNINIT}$:
- 8: $pk_P \leftarrow pk$
- 9: $State_P \leftarrow \text{INIT}$
- 10: On (OPEN, x , “ledger”, ...) by Alice when $State_A = \text{INIT}$:
- 11: store x
- 12: $State_A \leftarrow \text{TENTATIVE BASE OPEN}$
- 13: On (BASE OPEN) by \mathcal{A} when $State_A = \text{TENTATIVE BASE OPEN}$:
- 14: $\text{balance}_A \leftarrow x$
- 15: $\text{layer}_A \leftarrow 0$
- 16: $State_A \leftarrow \text{OPEN}$
- 17: On (BASE OPEN) by \mathcal{A} when $State_B = \text{INIT}$:
- 18: $\text{layer}_B \leftarrow 0$
- 19: $State_B \leftarrow \text{OPEN}$
- 20: On (OPEN, x , $\text{hops} \neq \text{“ledger”}$, ...) by Alice when $State_A = \text{INIT}$:
- 21: store x
- 22: $\text{enabler}_A \leftarrow \text{hops}[0].\text{left}$
- 23: add enabler_A to Alice's kindred parties
- 24: $State_A \leftarrow \text{PENDING VIRTUAL OPEN}$
- 25: On output (FUNDED, $host$, ...) to Alice by enabler_A when $State_A = \text{PENDING VIRTUAL OPEN}$:
- 26: $host_A \leftarrow \text{host}[0].\text{left}$
- 27: $State_A \leftarrow \text{TENTATIVE VIRTUAL OPEN}$
- 28: On output (FUNDED, $host$, ...) to Bob by ITI $R \in \{\mathcal{G}_{\text{Chan}}, \text{LN}\}$ when $State_B = \text{INIT}$:
- 29: $\text{enabler}_B \leftarrow R$
- 30: add enabler_B to Bob's kindred parties
- 31: $host_B \leftarrow host$
- 32: $State_B \leftarrow \text{TENTATIVE VIRTUAL OPEN}$
- 33: On (VIRTUAL OPEN) by \mathcal{A} when $State_P = \text{TENTATIVE VIRTUAL OPEN}$:
- 34: if $P = \text{Alice}$ then $\text{balance}_P \leftarrow x$
- 35: $\text{layer}_P \leftarrow 0$
- 36: $State_P \leftarrow \text{OPEN}$

Figure 16: State machine in Fig. 21, 22, 23 and 28

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

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

- 1: On (PAY, x) by P when $State_P = \text{OPEN}$: // 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  $State_{\bar{P}} \in \{\text{IGNORED}, (\text{SYNC GET PAID}, x)\}$  then
13:      $balance_{\bar{P}} \leftarrow balance_{\bar{P}} - x$ 
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  $State_{\bar{P}} \in \{\text{IGNORED}, (\text{SYNC PAY}, x)\}$  then
19:      $balance_{\bar{P}} \leftarrow balance_{\bar{P}} + x$ 
20:     // if  $\bar{P}$  honest, this state transition happens
       simultaneously with l. 15
21:      $State_P \leftarrow$  OPEN
22:   end if

```

Figure 17: State machine in Fig. 24

```

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  $State_{\bar{P}} \in \{\text{IGNORED}, \text{SYNC HELP FUND}\}$  then
27:      $balance_{\bar{P}} \leftarrow balance_{\bar{P}} - x$ 
28:      $host_{\bar{P}} \leftarrow host$ 
29:     // if  $\bar{P}$  honest, this state transition happens
       simultaneously with l. 38
30:      $layer_{\bar{P}} \leftarrow layer_{\bar{P}} + 1$ 
31:      $State_P \leftarrow$  OPEN
32:   end if
33: When  $State_P =$  SYNC HELP FUND:
34:   if  $State_{\bar{P}} \in \{\text{IGNORED}, \text{SYNC FUND}\}$  then
35:      $host_{\bar{P}} \leftarrow host$ 
36:     // if  $\bar{P}$  honest, this state transition happens
       simultaneously with l. 31
37:      $layer_{\bar{P}} \leftarrow layer_{\bar{P}} + 1$ 
38:      $State_P \leftarrow$  OPEN
39:   end if

```

Figure 18: State machine in Fig. 25

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

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

- On input (FUND ME, x, \dots) by ITI $R \in \{\mathcal{G}_{\text{Chan}}, \text{LN}\}$ when $State_P =$ OPEN:
 - store x
 - add R to P 's kindred parties
 - $State_P \leftarrow$ PENDING FUND
- When $State_P =$ PENDING FUND:
 - if we intercept the command “define new VIRT ITI host” by \mathcal{A} , routed through P then
 - store host
 - $State_P \leftarrow$ TENTATIVE FUND
 - continue executing \mathcal{A} 's command
 - end if
- On (FUND) by \mathcal{A} when $State_P =$ TENTATIVE FUND:
 - $State_P \leftarrow$ SYNC FUND
- When $State_P =$ OPEN:
 - if we intercept the command “define new VIRT ITI host” by \mathcal{A} , routed through P then
 - store host

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

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

- On (FORCECLOSE) by P when $State_P =$ OPEN:
 - $State_P \leftarrow$ CLOSING
- On input (BALANCE) by R addressed to P where R is kindred with P :
 - if $State_P \notin \{\text{UNINIT}, \text{INIT}, \text{PENDING VIRTUAL OPEN}, \text{TENTATIVE VIRTUAL OPEN}, \text{TENTATIVE BASE OPEN}, \text{IGNORED}, \text{CLOSED}\}$ then
 - reply (MY BALANCE, $balance_P, pk_P, balance_{\bar{P}}, pk_{\bar{P}}$)
 - else
 - reply (MY BALANCE, 0, $pk_P, 0, pk_{\bar{P}}$)
 - end if
- On (FORCECLOSE, P) by \mathcal{A} when $State_P \notin \{\text{UNINIT}, \text{INIT}, \text{PENDING VIRTUAL OPEN}, \text{TENTATIVE VIRTUAL OPEN}, \text{TENTATIVE BASE OPEN}, \text{IGNORED}\}$:
 - input (READ) to $\mathcal{G}_{\text{Ledger}}$ as P and assign output to Σ
 - $coins \leftarrow$ sum of values of outputs exclusively spendable or spent by pk_P in Σ
 - $balance \leftarrow balance_P$
 - for all P 's kindred parties R do
 - input (BALANCE) to R as P and extract

```

balanceR,  $pk_R$  from response
15:   balance  $\leftarrow$  balance + balanceR
16:   coins  $\leftarrow$  coins + sum of values of outputs
      exclusively spendable or spent by  $pk_R$  in  $\Sigma$ 
17:   end for
18:   if coins  $\geq$  balance then
19:     StateP  $\leftarrow$  CLOSED
20:   else // balance security is broken
21:     halt
22:   end if

```

Figure 19

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

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

- 1: On (COOP CLOSING, P , x) by \mathcal{A} when State_P = OPEN:
 - 2: store x
 - 3: State_P \leftarrow COOP CLOSING
- 4: On (COOP CLOSED, P) by \mathcal{A} when State_P = COOP CLOSING:
 - 5: **if** layer_P = 0 **then** // P 's channel, which is virtual, is cooperatively closed
 - 6: State_P \leftarrow COOP CLOSED
 - 7: **else** // the virtual channel for which P 's channel is base is cooperatively closed
 - 8: layer_P \leftarrow layer_P - 1
 - 9: balance_P \leftarrow balance_P + x
 - 10: State_P \leftarrow OPEN
 - 11: **end if**

Figure 20

J.2. Simulator \mathcal{S} for $\mathcal{G}_{\text{Chan}}$

Simulator \mathcal{S} – general message handling rules

- On receiving (RELAY, in_msg, P , R , in_mode) by $\mathcal{G}_{\text{Chan}}$ ($\text{in_mode} \in \{\text{input, output, network}\}$, $P \in \{\text{Alice}, \text{Bob}\}$), 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 $\in \{\text{input, output, network}\}$, 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 29

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

- “ P ” refers one of the parties that correspond to $\mathcal{G}_{\text{Chan}}$.
 - When an action in this Figure interrupts an ITI simulation, continue simulating from the interruption location once action is over/ $\mathcal{G}_{\text{Chan}}$ hands control back.
- 1: On (CORRUPT) by \mathcal{A} , addressed to P :
 - 2: // After executing this code and getting control back from $\mathcal{G}_{\text{Chan}}$ (which always happens, cf. Fig. 15), deliver (CORRUPT) to simulated P (cf. Fig. 29).
 - 3: send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$
 - 4: When simulated P sets variable negligent to True (Fig. 31, l. 7/Fig. 32, l. 26):
 - 5: send (INFO, BECAME CORRUPTED OR NEGLIGENT, P) to $\mathcal{G}_{\text{Chan}}$
 - 6: When simulated honest Alice receives (OPEN, x , hops, ...) by \mathcal{E} :
 - 7: store hops // will be used to inform $\mathcal{G}_{\text{Chan}}$ once the channel is open
 - 8: When simulated honest Bob receives (OPEN, x , hops, ...) by Alice:
 - 9: **if** Alice is corrupted **then** store hops // if Alice is honest, we already have hops. If Alice became corrupted after receiving (OPEN, ...), overwrite hops
 - 10: When the last of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties moves to the OPEN State for the first time (Fig. 35, l. 19/Fig. 37, l. 16/Fig. 38, l. 18):
 - 11: **if** hops = “ledger” **then**
 - 12: send (INFO, BASE OPEN) to $\mathcal{G}_{\text{Chan}}$
 - 13: **else**
 - 14: send (INFO, VIRTUAL OPEN) to $\mathcal{G}_{\text{Chan}}$
 - 15: **end if**
 - 16: When (both $\mathcal{G}_{\text{Chan}}$'s simulated parties are honest and complete sending and receiving a payment (Fig. 43, 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. 43, l. 6 is executed by one party, but its counterparty is corrupted before executing Fig. 43, l. 21)
 - 17: send (INFO, PAY) to $\mathcal{G}_{\text{Chan}}$

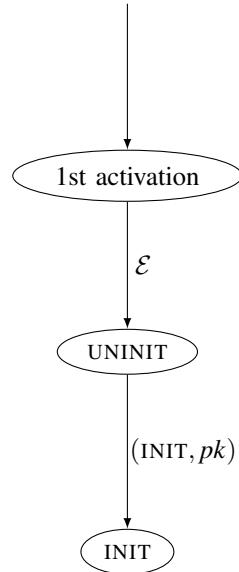


Figure 21: $\mathcal{G}_{\text{Chan}}$ state machine up to INIT (both parties)

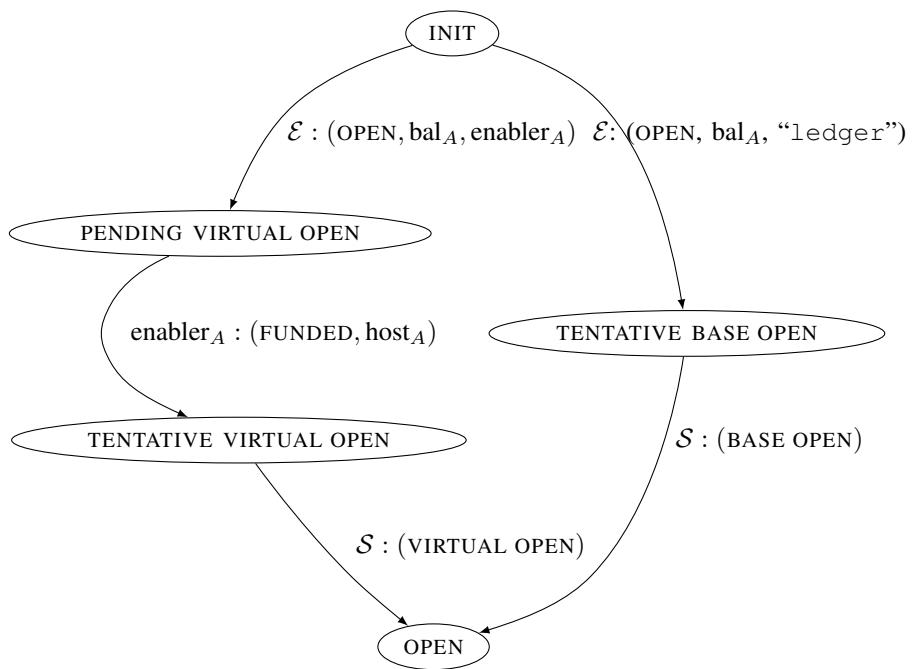


Figure 22: $\mathcal{G}_{\text{Chan}}$ state machine from INIT up to OPEN (funder)

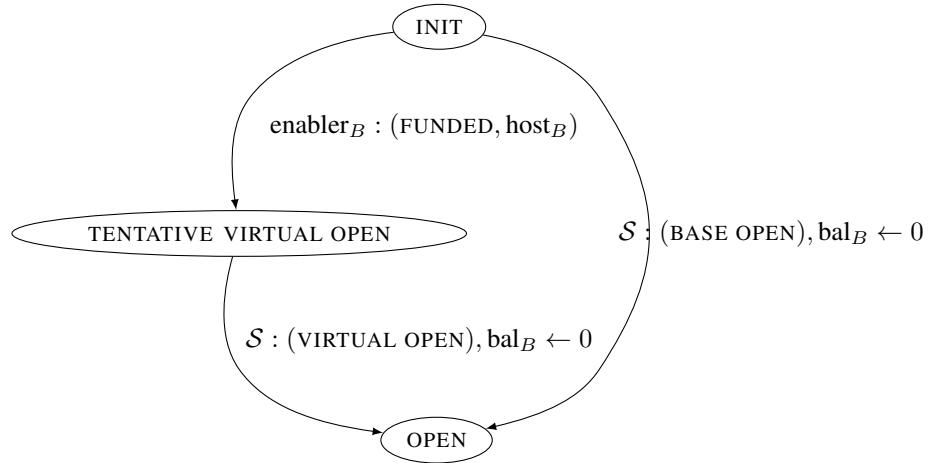


Figure 23: $\mathcal{G}_{\text{Chan}}$ state machine from INIT up to OPEN (fundee)

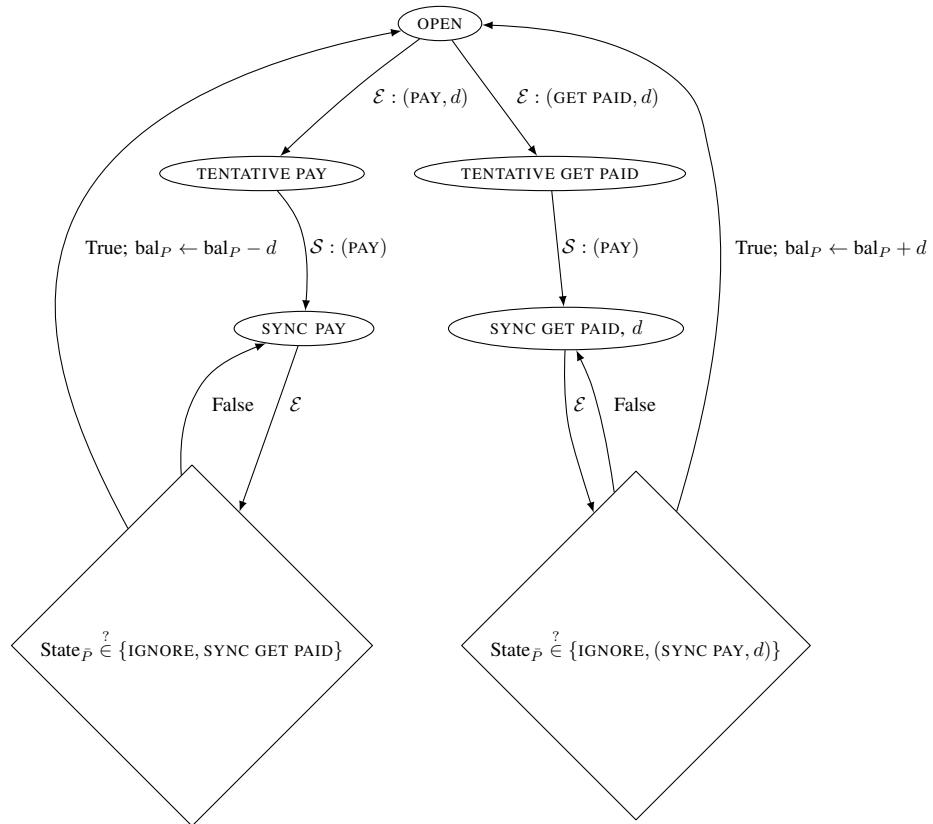


Figure 24: $\mathcal{G}_{\text{Chan}}$ state machine for payments (both parties)

- 18: When honest P executes Fig. 40, l. 21 or (when honest P executes Fig. 40, 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 66, ll. 7, 23): lifting to next layer is done
- 19: send (INFO, FUND) to $\mathcal{G}_{\text{Chan}}$
- 20: When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P moves to the COOP CLOSING state (Fig. 53, l. 4, Fig. 54, ll. 6, 12, Fig. 70, ll. 11, 24):
- 21: **if** triggered by Fig. 53, l. 4 or Fig. 54, l. 6 **then** // P is funder or fundee
- 22: send (INFO, COOP CLOSING, $P, -c_P$) to $\mathcal{G}_{\text{Chan}}$ // coin value extracted from simulated P
- 23: **else if** triggered by Fig. 54, l. 12 **then** // P is funder's base
- 24: send (INFO, COOP CLOSING, P, c'_1) to $\mathcal{G}_{\text{Chan}}$
- 25: **else if** triggered by Fig. 70, l. 11 **then** // P is an intermediary farther from funder than \bar{P}
- 26: send (INFO, COOP CLOSING, P, c'_2) to $\mathcal{G}_{\text{Chan}}$
- 27: **else if** triggered by Fig. 70, l. 24 **then** // P is an intermediary closer to funder than \bar{P}
- 28: send (INFO, COOP CLOSING, $P, c'_1 - c_{\text{virt}}$) to $\mathcal{G}_{\text{Chan}}$
- 29: **end if**
- 30: When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P completes cooperative closing (Fig. 54, l. 45, Fig. 70, l. 187, Fig. 70, l. 150, Fig. 70, or l. 134):
- 31: send (INFO, COOP CLOSED, P) to $\mathcal{G}_{\text{Chan}}$
- 32: When one of the honest simulated $\mathcal{G}_{\text{Chan}}$'s parties P moves to the CLOSED state (Fig. 47, l. 8 or l. 11):
- 33: send (INFO, FORCECLOSE, P) to $\mathcal{G}_{\text{Chan}}$

Figure 30

J.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 [11], [9].

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 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 $|\text{state}| - \text{windowSize}$. 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}_{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 $\text{maxTime}_{\text{window}}$, a number of blocks at least `windowSize` are added to `state`.
- 2) in a period of time equal to $\text{minTime}_{\text{window}}$, no more blocks may be added to `state` if `windowSize` blocks have been already added.
- 3) each window of `windowSize` blocks has at most `advBlcks_{\text{window}}` adversarial blocks included in it.
- 4) any transaction that (i) is submitted by an honest party earlier than $\frac{\text{Delay}}{2}$ 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:

$\text{windowSize}, \text{Delay} \in \mathbb{N}$, and

$S_{\text{initStake}} := \{(U_1, s_1), \dots, (U_n, s_n)\}$. 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 $\vec{\tau}_{\text{state}}$ (the time sequence where all immutable blocks were added). The variables are initialized as follows:

$\text{state} := \vec{\tau}_{\text{state}} := \text{NxtBC} := \varepsilon$, $\text{buffer} := \emptyset$, $\tau_L = 0$.

For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer pt_i (initially set to 1) and a current-state view $\text{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

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 $\mathcal{P}_{DS} \subseteq \mathcal{H}$ (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 receive 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 $\tau' < \tau_L - \text{Delay}$. 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 $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T \parallel (I, U_p, \tau_L)$;
 - b) Compute $\vec{N} = (\vec{N}_1, \dots, \vec{N}_\ell) := \text{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \text{state}, \text{NxtBC}, \text{buffer}, \vec{\tau}_{\text{state}})$ and if $\vec{N} \neq \varepsilon$ set $\text{state} := \text{state} \parallel \text{Blockify}(\vec{N}_1) \parallel \dots \parallel \text{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\text{state}} := \vec{\tau}_{\text{state}} \parallel \tau_L^\ell$, where $\tau_L^\ell = \tau_L \parallel \dots \parallel \tau_L$.
 - c) For each BTX $\in \text{buffer}$: if $\text{Validate}(\text{BTX}, \text{state}, \text{buffer}) = 0$ then delete BTX from buffer. Also, reset $\text{NxtBC} := \varepsilon$.
 - d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\text{state}| - \text{pt}_j > \text{windowSize}$ or $\text{pt}_j < |\text{state}_j|$, then set $\text{pt}_k := |\text{state}|$ 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 $I = (\text{SUBMIT}, \text{sid}, \text{tx})$ 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 $\text{BTX} := (\text{tx}, \text{txid}, \tau_L, U_p)$
 - b) If $\text{Validate}(\text{BTX}, \text{state}, \text{buffer}) = 1$, then $\text{buffer} := \text{buffer} \cup \{\text{BTX}\}$.
 - c) Send $(\text{SUBMIT}, \text{BTX})$ to \mathcal{A} .
 - *Reading the state:* If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\text{state}_p := \text{state}|_{\min\{\text{pt}_p, |\text{state}|\}}$ and return $(\text{READ}, \text{sid}, \text{state}_p)$ to the requester. If the requester is \mathcal{A} then send $(\text{state}, \text{buffer}, \vec{\mathcal{I}}_H^T)$ 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 $\vec{\mathcal{I}}_H^T$ as above) $\text{predict-time}(\vec{\mathcal{I}}_H^T) = \hat{\tau} > \tau_L$ 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 $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \dots, \text{txid}_\ell))$ is sent from the adversary, update NxtBC as follows:

- a) Set $\text{listOfTxid} \leftarrow \varepsilon$
 - b) For $i = 1, \dots, \ell$ do: if there exists $\text{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \text{buffer}$ with $\text{ID txid} = \text{txid}_i$ then set $\text{listOfTxid} := \text{listOfTxid} \parallel \text{txid}_i$.
 - c) Finally, set $\text{NxtBC} := \text{NxtBC} \parallel (\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, \text{ok})$ to \mathcal{A} .
- *The adversary setting state-slackness:* If $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\text{pt}}_{i_1}), \dots, (U_{i_\ell}, \widehat{\text{pt}}_{i_\ell}))$, with $\{U_{p_{i_1}}, \dots, U_{p_{i_\ell}}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary \mathcal{A} do the following:
 - a) If for all $j \in [\ell]$:

$$|\text{state}| - \widehat{\text{pt}}_{i_j} \leq \text{windowSize}$$
 and

$$\widehat{\text{pt}}_{i_j} \geq |\text{state}_{i_j}|$$
, set $\text{pt}_{i_1} := \widehat{\text{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, \text{ok})$ to \mathcal{A} .
 - b) Otherwise set $\text{pt}_{i_j} := |\text{state}|$ for all $j \in [\ell]$.
 - *The adversary setting the state for desynchronized parties:* If $I = (\text{DESYNC-STATE}, (U_{i_1}, \text{state}'_{i_1}), \dots, (U_{i_\ell}, \text{state}'_{i_\ell}))$, with $\{U_{i_1}, \dots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary \mathcal{A} , set $\text{state}_{i_j} := \text{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, \text{ok})$ 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 $U_p = (\cdot, \text{sid}) \in \mathcal{P}$.

Appendix K. Model & Construction

K.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. J (Figures 15-19) and a simulator \mathcal{S} (Figures 29-30), along with a real-world protocol Π_{Chan} (Figures 31-71) that UC-realises $\mathcal{G}_{\text{Chan}}$ (Theorem 7). We give a self-contained description in this section, while pointing to figures in Appx. J and L, in case the reader is interested in a pseudocode style specification.

As in previous formulations, (e.g., [38]), 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 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.

K.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 21, 22, 23, 24, 25, 26, 28) 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 6 and 7) 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 $\mathcal{G}_{\text{Chan}}^1$. 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 $\mathcal{G}_{\text{Chan}}^n$. Then $\mathcal{G}_{\text{Chan}}^n$ is $(\mathcal{G}_{\text{Ledger}}, \mathcal{G}_{\text{Chan}}^1, \dots, \mathcal{G}_{\text{Chan}}^{n-1})$ -subroutine respecting, according to the definition of [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 Sec. 3 in more detail.

We could instead contain all the channels in a single, monolithic functionality (following the approach of [38]) 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.

K.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 31-50) and the novel virtual layer subprotocol, named VIRT (Figures 56-71). 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 31).

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 38), *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 33).

The next step prepares the base channels (Figure 34) 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 40 and 41). 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 35). After that, in case the channel is simple the funding transaction is put on-chain (Figure 36) 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 45) and subsequently the payer is instructed by \mathcal{E} to commence the payment (Figure 44).

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 corre-

sponding revocation transaction in case one is found (Figure 47). It also keeps track of whether the party is activated often enough and marks it as negligent otherwise (Figure 31). 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 51–54) and VIRT (Figure 70) 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 49), 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 commitment 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 47).

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 56), 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 57) 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 64) 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 59) 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 58). 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 65). 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 66) 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 counterpart 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 counterpart 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 68), 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 69). 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.

Appendix L. Protocol

Process LN – init

```

1: // When not specified, input comes from and output goes to  $\mathcal{E}$ .
2: // The ITI knows whether it is Alice (funder) or Bob (fundee). The activated party is  $P$  and the counterparty is  $\bar{P}$ .
3: On every activation, before handling the message:
4:   if last_poll ≠ ⊥ ∧ State ≠ CLOSED then // channel is open
5:     input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
6:     if last_poll +  $p < |\Sigma|$  then //  $p$  is a global parameter
7:       negligent ← True
8:     end if
9:   end if
10:  if State = WAITING FOR NOTHING REVOKED ∧ 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
11:    State ← BASE PUNISHED
12:  end if
13: On (INIT,  $pk_{P,\text{out}}$ ):
14:   ensure State = ⊥
15:   State ← INIT
16:   hosting ← False

```

```

17:   store  $pk_{P,\text{out}}$ 
18:    $(c_A, c_B, \text{locked}_A, \text{locked}_B) \leftarrow (0, 0, 0, 0)$ 
19:    $(\text{paid}_\text{out}, \text{paid}_\text{in}) \leftarrow (\emptyset, \emptyset)$ 
20:   negligent ← False
21:   last_poll ← ⊥
22:   output (INIT OK)

23: On (TOP UP):
24:   ensure  $P = Alice$  // activated party is the funder
25:   ensure State = INIT
26:    $(sk_{P,\text{chain}}, pk_{P,\text{chain}}) \leftarrow \text{KEYGEN}()$ 
27:   input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
28:   output (TOP UP TO,  $pk_{P,\text{chain}}$ )
29:   while  $\neg \exists \text{tx} \in \Sigma, c_{P,\text{chain}} : (c_{P,\text{chain}}, pk_{P,\text{chain}}) \in \text{tx.outputs}$ 
30:     // while waiting, all other messages by  $P$  are ignored
31:     wait for input (CHECK TOP UP)
32:     input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
33:   end while
34:   State ← TOPPED UP
35:   output (TOP UP OK,  $c_{P,\text{chain}}$ )

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

```

Figure 31

Process LN – methods used by VIRT

```

1: REVOKEPREVIOUS():
2:   ensure
3:      $State \in \{\text{WAITING FOR (OUTBOUND) REVOCATION}, \text{HOST PUBLISHED}\}$ 
4:      $R_{\bar{P},i} \leftarrow \text{TX } \{ \text{input: } C_{P,i}.\text{outputs}.P, \text{output: } (C_{P,i}.\text{outputs}.P.\text{value}, pk_{\bar{P},\text{out}}) \}$ 
5:      $\text{sig}_{A,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$ 
6:     if State = WAITING FOR REVOCATION then
7:       State ← WAITING FOR INBOUND REVOCATION
8:     else // State = WAITING FOR OUTBOUND REVOCATION
9:       i ← i + 1
10:      State ← WAITING FOR HOSTS READY
11:    end if
12:    hostP ← host'P // forget old host, use new host instead
13:    layer ← layer + 1
14:    return  $\text{sig}_{P,R,i}$ 

15: PROCESSREMOTEREVOCATION( $\text{sig}_{\bar{P},R,i}$ ):
16:   ensure
17:      $State = \{\text{WAITING FOR (INBOUND) REVOCATION}, \text{HOST PUBLISHED}\}$ 
18:      $R_{P,i} \leftarrow \text{TX } \{ \text{input: } C_{\bar{P},i}.\text{outputs}.\bar{P}, \text{output: } (C_{\bar{P},i}.\text{outputs}.\bar{P}.\text{value}, pk_{P,\text{out}}) \}$ 
19:     ensure  $\text{VERIFY}(R_{P,i}, \text{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}) = \text{True}$ 
20:     if State = WAITING FOR REVOCATION then
21:       State ← WAITING FOR OUTBOUND REVOCATION

```

```

20:   else // State = WAITING FOR INBOUND REVOCATION
21:     i ← i + 1
22:     State ← WAITING FOR HOSTS READY
23:   end if
24:   return (OK)

25: NEGIGENT():
26:   negligent ← True
27:   return (OK)

```

Figure 32

Process LN.EXCHANGEOPENKEYS()

```

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

```

Figure 33

Process LN.PREPAREBASE()

```

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

```

Figure 34

Process LN.EXCHANGEOPENSIGS()

```

1: //  $s = (2 + q)windowSize$ , where  $q$  and windowSize are defined in Proposition 2
2:  $C_{A,0} \leftarrow TX$  {input:  $(c, 2/\{pk_{A,F}, pk_{B,F}\})$ , outputs:  $(c, (pk_{A,out} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\}), (0, pk_{B,out})$ }
3:  $C_{B,0} \leftarrow TX$  {input:  $(c, 2/\{pk_{A,F}, pk_{B,F}\})$ , outputs:  $(c, pk_{A,out}), (0, (pk_{B,out} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$ }
4:  $sig_{A,C,0} \leftarrow SIGN(C_{B,0}, sk_{A,F})$ 
5: State ← WAITING FOR COMM SIG
6: send (FUNDING CREATED,  $(c, pk_{A,chain}), sig_{A,C,0}$ ) to fundee
7: ensure State = WAITING FOR COMM SIG // if opening virtual channel, we have received (FUNDED, host_fundee) by hops[-1].right (Fig 37, l. 3)
8: if hops = "ledger" then // opening base channel
9:    $F \leftarrow TX$  {input:  $(c, pk_{A,chain})$ , output:  $(c, 2/\{pk_{A,F}, pk_{B,F}\})$ }
10: end if
11:  $C_{B,0} \leftarrow TX$  {input:  $(c, 2/\{pk_{A,F}, pk_{B,F}\})$ , outputs:  $(c, pk_{A,out}), (0, (pk_{B,out} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\})$ }
12: ensure VERIFY( $C_{B,0}, sig_{A,C,0}, pk_{A,F}$ ) = True
13:  $C_{A,0} \leftarrow TX$  {input:  $(c, 2/\{pk_{A,F}, pk_{B,F}\})$ , outputs:  $(c, (pk_{A,out} + (p + s)) \vee 2/\{pk_{A,R,1}, pk_{B,R,1}\}), (0, pk_{B,out})$ }
14:  $sig_{B,C,0} \leftarrow SIGN(C_{A,0}, sk_{B,F})$ 
15: if hops = "ledger" then // opening base channel
16:   State ← WAITING TO CHECK FUNDING
17: else // opening virtual channel
18:    $ca \leftarrow c; cb \leftarrow 0; i \leftarrow 0$ 
19:   State ← OPEN
20: end if
21: reply (FUNDING SIGNED,  $sig_{B,C,0}$ )
22: ensure State = WAITING FOR COMM SIG
23: ensure VERIFY( $C_{A,0}, sig_{B,C,0}, pk_{B,F}$ ) = True

```

Figure 35

Process LN.COMMITBASE()

```

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

```

Figure 36

Process LN – external open messages for *Bob*

```

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

7: On output (CHECK KEYS, (pk1, pk2)) by
   hops[-1].right:
8:   ensure State = WAITING FOR CHECK KEYS
9:   ensure pk1 = pkA,F ∧ pk2 = pkB,F
10:  State ← WAITING FOR FUNDED
11:  reply (KEYS OK)

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

```

Figure 37

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

```

1: // fundee is Bob
2: ensure we run the code of Alice // activated party is the
   funder
3: if hops = "ledger" then // opening base channel
4:   ensure State = TOPPED UP
5:   ensure c = cA,chain
6: else // opening virtual channel
7:   ensure len(hops) ≥ 2 // cannot open a virtual over 1
   channel
8: end if
9: LN.EXCHANGEOPENKEYS()
10: LN.PREPAREBASE()
11: LN.EXCHANGEOPENSIGS()
12: if hops = "ledger" then
13:   LN.COMMITBASE()
14: end if
15: input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
16: last_poll ← | $\Sigma$ |
17: cA ← c; cB ← 0; i ← 0
18: State ← OPEN
19: output (OPEN OK, c, fundee, hops)

```

Figure 38

Process LN.UPDATEFORVIRTUAL()

```

1: C̄P,i+1 ← C̄P,i with pk'P,F, pk'̄P,F, pk̄P,R,i+1 and
   pkP,R,i+1 instead of pkP,F, pk̄P,F, pk̄P,R,i and pkP,R,i

```

respectively, reducing the input and *P*'s output by c_{virt}

```

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

```

Figure 39

Process LN – virtualise start and end

```

1: On input (FUND ME, fundee, hops, cvirt, pkA,V,
   pkB,V) by funder:
2:   ensure State = OPEN
3:   ensure cP - lockedP ≥ cvirt
4:   State ← VIRTUALISING
5:   (sk'P,F, pk'P,F) ← KEYGEN()
6:   define new VIRT ITI host'P
7:   send (VIRTUALISING, host'P, pk'P,F, hops,
   fundee, cvirt, 2, len(hops)) to  $\bar{P}$  and expect reply
   (VIRTUALISING ACK, host'̄P, pk'̄P,F)
8:   ensure pk'̄P,F is different from pk̄P,F and all older
    $\bar{P}$ 's funding public keys
9:   LN.UPDATEFORVIRTUAL()
10:  State ← WAITING FOR REVOCATION
11:  input (HOST ME, funder, fundee, host'̄P, hostP,
   cP, c̄P, cvirt, pkA,V, pkB,V, (sk'P,F, pk'P,F), (skP,F,
   pkP,F), pk̄P,F, pk'̄P,F, pkP,out, len(hops)) to host'P
12: On output (HOSTS READY, tP) by hostP: // hostP is
   the new host, renamed in Fig. 32, l. 12
13:  ensure State = WAITING FOR HOSTS READY
14:  State ← OPEN
15:  hosting ← True
16:  move skP,F, pkP,F, pk̄P,F to list of old funding keys
17:  (skP,F, pkP,F) ← (sk'P,F, pk'P,F); pk̄P,F ← pk'̄P,F
18:  if len(hops) = 1 then // we are the last hop
19:    output (FUNDED, hostP, layer, tP) to
   fundee and expect reply (FUND ACK)
20:  else if we have received input FUND ME just before
   we moved to the VIRTUALISING state then // we are the
   first hop
21:    cP ← cP - cvirt
22:    output (FUNDED, hostP, layer, tP) to

```

```

funder // do not expect reply by funder
23:   end if
24:   reply (HOST ACK)

```

Figure 40

```

expect reply (HOST INIT OK)
28:   State ← WAITING FOR REVOCATION
29:   output (VIRTUALISING ACK, host'P, pk'P,F) to
      sibling

```

Figure 41

Process LN – virtualise hops

```

1: On (VIRTUALISING, host'P, pk'P,F, hops, fundee,
   cvirt, i, n) by  $\bar{P}$ :
2:   ensure State = OPEN
3:   ensure  $c_{\bar{P}} - \text{locked}_{\bar{P}} \geq c_{\text{virt}}$ ;  $1 \leq i \leq n$ 
4:   ensure pk'P,F is different from pk'P,F and all older
    $\bar{P}$ 's funding public keys
5:   State ← VIRTUALISING
6:   lockedP ← lockedP + cvirt // if  $\bar{P}$  is hosting the
   funder,  $\bar{P}$  will transfer cvirt coins instead of locking
   them, but the end result is the same
7:   ( $sk'_{P,F}, pk'_{P,F}$ ) ← KEYGEN()
8:   if len(hops) > 1 then // we are not the last hop
9:     define new VIRT ITI host'P
10:    input (VIRTUALISING, host'P, ( $sk'_{P,F}, pk'_{P,F}$ ),
        pk'P,F, pkP,out, hops[1:], fundee, cvirt,  $c_{\bar{P}}$ ,  $c_P$ , i, n)
        to hops[1].left and expect reply (VIRTUALISING ACK,
        host_sibling, pksib,bar,P,F)
11:    input (INIT, hostP, host'P, host_sibling,
        ( $sk'_{P,F}, pk'_{P,F}$ ), pk'P,F, pksib,bar,P,F, ( $sk_{P,F}, pk_{P,F}$ ), pk'P,F,
        pkP,out,  $c_P$ ,  $c_{\bar{P}}$ , cvirt, i,  $t_P$ , "left", n) to host'P and
        expect reply (HOST INIT OK)
12:   else // we are the last hop
13:     input (INIT, hostP, host'P, fundee=fundee,
        ( $sk'_{P,F}, pk'_{P,F}$ ), pk'P,F, ( $sk_{P,F}, pk_{P,F}$ ), pk'P,F, pkP,out,
         $c_P$ ,  $c_{\bar{P}}$ , cvirt,  $t_P$ , i, "left", n) to new VIRT ITI host'P
        and expect reply (HOST INIT OK)
14:   end if
15:   State ← WAITING FOR REVOCATION
16:   send (VIRTUALISING ACK, host'P, pk'P,F) to  $\bar{P}$ 

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

```

Process LN.SIGNATURESROUNDTRIP()

```

1:  $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$  with  $pk_{P,R,i+1}$  and  $pk_{\bar{P},R,i+1}$  instead of
    $pk_{P,R,i}$  and  $pk_{\bar{P},R,i}$  respectively, and  $x$  coins moved from
   P's to  $\bar{P}$ 's output
2:  $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk_{P,F})$  // kept by  $\bar{P}$ 
3: ( $sk_{P,R,i+2}, pk_{P,R,i+2}$ ) ← KEYGEN()
4: State ← WAITING FOR COMMITMENT SIGNED
5: send (PAY,  $x$ ,  $\text{sig}_{P,C,i+1}, pk_{P,R,i+2}$ ) to  $\bar{P}$ 
6: // P refers to payer and  $\bar{P}$  to payee both in local and
   remote code
7: ensure State = WAITING TO GET PAID  $\wedge x = y$ 
8:  $C_{\bar{P},i+1} \leftarrow C_{\bar{P},i}$  with  $pk_{P,R,i+1}$  and  $pk_{\bar{P},R,i+1}$  instead of
    $pk_{P,R,i}$  and  $pk_{\bar{P},R,i}$  respectively, and  $x$  coins moved from
   P's to  $\bar{P}$ 's output
9: ensure VERIFY( $C_{\bar{P},i+1}, \text{sig}_{P,C,i+1}, pk_{P,F}$ ) = True
10:  $C_{P,i+1} \leftarrow C_{P,i}$  with  $pk_{\bar{P},R,i+1}$  and  $pk_{P,R,i+1}$  instead of
     $pk_{\bar{P},R,i}$  and  $pk_{P,R,i}$  respectively, and  $x$  coins moved from
    P's to  $\bar{P}$ 's output
11:  $\text{sig}_{\bar{P},C,i+1} \leftarrow \text{SIGN}(C_{P,i+1}, sk_{\bar{P},F})$  // kept by P
12:  $R_{P,i} \leftarrow \text{TX}\{\text{input: } C_{\bar{P},i}.\text{outputs.}\bar{P}, \text{output: } (c_{\bar{P}},$ 
     $pk_{P,out})\}$ 
13:  $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{P,i}, sk_{P,R,i})$ 
14: ( $sk_{\bar{P},R,i+2}, pk_{\bar{P},R,i+2}$ ) ← KEYGEN()
15: State ← WAITING FOR PAY REVOCATION
16: reply (COMMITMENT SIGNED,  $\text{sig}_{\bar{P},C,i+1}, \text{sig}_{\bar{P},R,i},$ 
     $pk_{\bar{P},R,i+2}$ )
17: ensure State = WAITING FOR COMMITMENT SIGNED
18:  $C_{P,i+1} \leftarrow C_{P,i}$  with  $pk_{\bar{P},R,i+1}$  and  $pk_{P,R,i+1}$  instead of
     $pk_{\bar{P},R,i}$  and  $pk_{P,R,i}$  respectively, and  $x$  coins moved from
    P's to  $\bar{P}$ 's output

```

Figure 42

Process LN.REVOCATIONSROUNDTRIP()

```

1: ensure VERIFY( $C_{P,i+1}, \text{sig}_{\bar{P},C,i+1}, pk_{\bar{P},F}$ ) = True
2:  $R_{P,i} \leftarrow \text{TX}\{\text{input: } C_{\bar{P},i}.\text{outputs.}\bar{P}, \text{output: } (c_{\bar{P}},$ 
     $pk_{P,out})\}$ 
3: ensure VERIFY( $R_{P,i}, \text{sig}_{\bar{P},R,i}, pk_{\bar{P},R,i}$ ) = True
4:  $R_{\bar{P},i} \leftarrow \text{TX}\{\text{input: } C_{P,i}.\text{outputs.}\bar{P}, \text{output: } (c_P,$ 
     $pk_{\bar{P},out})\}$ 
5:  $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$ 
6: add  $x$  to paid_out
7:  $c_P \leftarrow c_P - x; c_{\bar{P}} \leftarrow c_{\bar{P}} + x; i \leftarrow i + 1$ 
8: State ← OPEN
9: if hostP ≠ "ledger"  $\wedge$  we have a host_sibling
   then // we are intermediary channel
10:   input (NEW BALANCE,  $c_P, c_{\bar{P}}$ ) to hostP
11:   relay message as input to sibling // run by VIRT

```

```

12:   relay message as output to guest // run by VIRT
13:   store new sibling balance and reply (NEW BALANCE
    OK)
14:   output (NEW BALANCE OK) to sibling // run by
    VIRT
15:   output (NEW BALANCE OK) to guest // run by VIRT
16: end if
17: send (REVOKE AND ACK,  $\text{sig}_{P,R,i}$ ) to  $\bar{P}$ 
18: ensure State = WAITING FOR PAY REVOCATION
19:  $R_{\bar{P},i} \leftarrow \text{TX}$  {input:  $C_{P,i}.\text{outputs}.P$ , output: ( $c_P$ ,
 $\text{pk}_{\bar{P},\text{out}}$ )}
20: ensure  $\text{VERIFY}(R_{\bar{P},i}, \text{sig}_{P,R,i}, \text{pk}_{P,R,i}) = \text{True}$ 
21: add  $x$  to paid_in
22:  $c_P \leftarrow c_P - x; c_{\bar{P}} \leftarrow c_{\bar{P}} + x; i \leftarrow i + 1$ 
23: State  $\leftarrow$  OPEN
24: if  $\text{host}_P \neq \text{"ledger"} \wedge \bar{P}$  has a host_sibling
    then // we are intermediary channel
25:   input (NEW BALANCE,  $c_{\bar{P}}$ ,  $c_P$ ) to  $\text{host}_{\bar{P}}$ 
26:   relay message as input to sibling // run by VIRT
27:   relay message as output to guest // run by VIRT
28:   store new sibling balance and reply (NEW BALANCE
    OK)
29:   output (NEW BALANCE OK) to sibling // run by
    VIRT
30:   output (NEW BALANCE OK) to guest // run by VIRT
31: end if

```

Figure 43

Process LN – On (PAY, x):

```

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

```

Figure 44

Process LN – On (GET PAID, y):

```

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

```

Figure 45

Process LN – On (CHECK FOR LATERAL CLOSE):

```

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

```

Figure 46

Process LN – On (CHECK CHAIN FOR CLOSED):

```

1: ensure State  $\notin \{\perp, \text{INIT}, \text{TOPPED UP}\}$  // channel open
2: // even virtual channels check  $\mathcal{G}_{\text{Ledger}}$  directly. This is
    intentional
3: input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign reply to  $\Sigma$ 
4: last_poll  $\leftarrow |\Sigma|$ 
5: if  $\exists 0 \leq j < i : C_{\bar{P},j} \in \Sigma$  then // counterparty has closed
    maliciously
6:   State  $\leftarrow$  CLOSING
7:   LN.SUBMITANDCHECKREVOCATION( $j$ )
8:   State  $\leftarrow$  CLOSED
9:   output (CLOSED)
10: else if  $C_{P,i} \in \Sigma \vee C_{\bar{P},i} \in \Sigma$  then
11:   State  $\leftarrow$  CLOSED
12:   output (CLOSED)
13: else
14:   state_before_checking_revoked  $\leftarrow$  State
15:   for each host in list of old hosts do
16:     State  $\leftarrow$  WAITING FOR NOTHING REVOKED
17:     input (CHECK FOR REVOKED) to host and expect
        output (NOTHING REVOKED)
18:
19:   State  $\leftarrow$  state_before_checking_revoked
20: end if

```

Figure 47

Process LN.SUBMITANDCHECKREVOCATION(j):

```

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

```

Figure 48

Process LN – On (FORCECLOSE):

```

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

```

Figure 49

Process LN – punishment

```

1: On output (ENABLER USED REVOCATION) by  $\text{host}_P$ :
2:    $\text{State} \leftarrow \text{BASE PUNISHED}$ 

```

Figure 50

Process LN – On (COOPCLOSE):

```

// any endpoint or intermediary can initiate virtual channel
closing
1: ensure  $\text{host}_P \neq \text{"ledger"}$ 
2: ensure  $\text{State} = \text{OPEN}$ 
3:  $\text{we\_are\_close\_initiator} \leftarrow \text{True}$ 
4: if  $\text{hosting} = \text{True} \vee$  we have received OPEN from  $\mathcal{E}$ 
  while  $\text{State}$  was TOPPED UP then // we are not the
  fundee of a channel that is not the base of any other
  channel
5:   if  $\text{hosting} = \text{True}$  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
     $\text{host}_P$ , set  $\text{we\_are\_close\_initiator} \leftarrow \text{False}$ 
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

```

```

 $\text{we\_are\_close\_initiator} \leftarrow \text{False}$ 
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
     $\text{host}_P$ , set  $\text{we\_are\_close\_initiator} \leftarrow \text{False}$ 
13:    $\text{close\_initiator} \leftarrow \bar{P}$ 
14:   execute code of Fig. 53
15: end if

```

Figure 51

Process LN – On (COOPCLOSED) by R :

```

1: if  $\text{hosting} = \text{True}$  then // we are intermediary
2:   ensure  $\text{State} = \text{OPEN}$ 
3: else // we are endpoint
4:   ensure  $\text{State} = \text{COOP CLOSED}$ 
5: end if
6: ensure  $\text{we\_are\_close\_initiator} = \text{True}$ 
7: ensure that the last cooperatively closed channel in which
  we acted as a base had  $R$  as its fundee
8:  $\text{we\_are\_close\_initiator} \leftarrow \text{False}$ 
9: output (COOPCLOSED)

```

Figure 52

Process LN – On (COOP CLOSE) by R :

// also executed when we are instructed to close a channel
cooperatively by \mathcal{E} – cf. Fig. 51, l. 14

```

1: ensure we are fundee
2: ensure  $\text{hosting} \neq \text{True}$ 
3: ensure  $\text{State} = \text{OPEN}$ 
4:  $\text{State} \leftarrow \text{COOP CLOSING}$ 
5:  $\text{close\_initiator} \leftarrow R$ 
6:  $\text{sig\_bal} \leftarrow ((c_{\bar{P}}, c_P), \text{SIGN}((c_{\bar{P}}, c_P), sk_{P,F}))$ 
7:  $\text{State} \leftarrow \text{WAITING TO REVOKE VIRT COMM}$ 
8: send (COOP CLOSE,  $\text{sig\_bal}$ ) to  $\bar{P}$ 

```

Figure 53

Process LN – On (COOP CLOSE, $\text{sig_bal}_{\bar{P}}$) by \bar{P} :

```

1: ensure we are funder
2: ensure  $\text{State} = \text{OPEN}$ 
3: parse  $\text{sig\_bal}_{\bar{P}}$  as  $((c'_1, c'_2), \text{sig}_{\bar{P}})$ 
4: ensure
   $c_P = c'_1 \wedge c_{\bar{P}} = c'_2 \wedge \text{VERIFY}((c'_1, c'_2), \text{sig}_{\bar{P}}, pk_{\bar{P},F}) = \text{True}$ 
5:  $\text{sig\_bal} \leftarrow ((c_P, c_{\bar{P}}), \text{SIGN}((c_P, c_{\bar{P}}), sk_{P,F}), \text{sig}_{\bar{P}})$ 
6:  $\text{State} \leftarrow \text{COOP CLOSING}$ 
7: input (COOP CLOSE,  $\text{sig\_bal}$ ) to  $\text{host}_P$ 

```

```

8: ensure State = OPEN // executed by hostP
9: State ← COOP CLOSING
10: output (COOP CLOSE SIGN COMM FUNDER, (c1', c2')) to guest
11: ensure State = OPEN // executed by guest of hostP
12: State ← COOP CLOSING
13: remove most recent keys from list of old funding keys and assign them to skP,F, pkP,F, pk'P,F
14: CP,i+1 ← TX {input: (cP + cP' + c1' + c2', 2/{pkP,F, pk'P,F}), outputs: (cP + c1', pkP,out), (cP + c2', (pkP,out + (p + s)) ∨ 2/{pkP,R,i+1, pkP,R,i+1})}}
15: sigP,C,i+1 ← SIGN(CP,i+1, skP,F)
16: (skP,R,i+2, pkP,R,i+2) ← KEYGEN()
17: input (NEW COMM TX, sigP,C,i+1, pkP,R,i+2) to hostP
18: rename received signature to sig1,right,C // executed by hostP
19: rename received public key to pk1,right,R
20: send (COOP CLOSE, sigbal, sig1,right,C, pk1,right,R) to P and expect reply (COOP CLOSE BACK, (right_comms_revkeys, right_revocations)
21: Rloc,virt ← TX {input: (cvirt, 2/{pk1,rev, pk2,rev}), output: (cvirt, pk1,out)}
22: extract sig2,right,rev,virt from right_revocations
23: ensure VERIFY(Rloc,virt, sig2,right,rev,virt, pk2,rev) = True
24: Rloc,fund ← TX {input: (cP + cP', 2/{pk1,rev, pk2,rev}), output: (cP + cP', pk1,out)}
25: extract sig2,right,rev,fund from right_revocations
26: ensure VERIFY(Rloc,fund, sig2,right,rev,fund, pk2,rev) = True
27: extract sig2,right,R from right_revocations
28: extract sig2,right,C from right_comms_revkeys
29: extract pk2,R from right_comms_revkeys
30: output (VERIFY REVOKE, sig2,right,C, sig2,right,R, pk2,R, hostP) to guest
31: store sig2,right,C as sigP,C,i+1 // executed by guest of hostP
32: store sig2,right,R as sigP,R,i
33: store received public key as pkP,R,i+2
34: CP,i+1 ← TX {input: (cP + cP' + c1' + c2', outputs: (cP + c1', (pkP,out + (p + s)) ∨ 2/{pkP,R,i+1, pkP,R,i+1}), (cP' + c2', pk'P,out))}
35: ensure VERIFY(CP,i+1, sigP,C,i+1, pk'P,F) = True
36: RP,i ← TX {input: CP,i.outputs.P, output: (cP, pkP,out)}
37: ensure VERIFY(RP,i, sigP,R,i, pkP,R,i) = True
38: input (VERIFIED) to hostP
39: extract sign,left,R from right_revocations // executed by hostP
40: output (VERIFY REVOCATION, sign,left,R) to funder
41: RP,i ← TX {input: CP,i.outputs.P, output: (cP, pkP,out)}
42: ensure VERIFY(RP,i, sigP,R,i, pkP,R,i) = True
43: RP,i ← TX {input: CP,i.outputs.P, output: (cP, pk'P,out)}
44: sigP,R,i ← SIGN(RP,i, skP,R,i)
45: State ← COOP CLOSED // in LN, only virtual channels can end up in this state
46: input (COOP CLOSE REVOCATION, sigP,R,i) to hostP
47: output (COOP CLOSE REVOCATIONS, hostP) to guest // executed by hostP

```

```

48: RP,i ← TX {input: CP,i.outputs.P, output: (cP, pk'P,out)} // executed by guest of hostP
49: sigP,R,i ← SIGN(RP,i, skP,R,i)
50: add skP,F, pkP,F, pk'P,F to list of old enable channel funding keys
51: add hostP to list of old hosts
52: assign received host to hostP
53: cP ← cP + c1'; cP ← cP + c2'
54: layer ← layer - 1
55: lockedP ← lockedP - cvirt
56: State ← OPEN
57: input (REVOCATION, sigP,R,i) to last old host
58: rename received signature to sig1,right,R // executed by hostP
59: Rrem,virt ← TX {input: (cvirt, 4/{pk1,rev, pk2,rev, pkn,rev}), output: (cvirt, pk2,out)}
60: sig1,right,rev,virt ← SIGN(Rrem,virt, sk1,rev)
61: Rrem,fund ← TX {input: (cP + cP', 2/{pk1,rev, pk2,rev}), output: (cP + cP', pk2,out)}
62: sig1,right,rev,fund ← SIGN(Rrem,fund, sk1,rev)
63: for all j ∈ {2, …, n} do
64:   Rj,left ← TX {input: (cvirt, 4/{pk1,rev, pkj-1,rev, pkj,rev, pkn,rev}), output: (cvirt, pkj,out)}
65:   sig1,j,left,rev ← SIGN(Rj,left, sk1,rev)
66: end for
67: State ← COOP CLOSED
68: send (COOP CLOSE REVOCATIONS, (sig1,right,R, sig1,right,rev,virt, sig1,right,rev,fund, (sig1,j,left,rev)j ∈ {2, …, n})) to P

```

Figure 54

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

```

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

```

Figure 55

Process VIRT

```

1: On every activation, before handling the message:
2:   if last_poll ≠ ⊥ then // virtual layer is ready
3:     input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
4:     if last_poll +  $p < |\Sigma|$  then
5:       for  $P \in \{\text{guest, funder, fundee}\}$  do // at most 1 of funder, fundee is defined
6:         ensure  $P.\text{NEGLIGENT}()$  returns (OK)
7:       end for

```

```

8:     end if
9: end if

10: // guest is trusted to give sane inputs, therefore a state
    machine and input verification are redundant
11: On input (INIT, hostP,  $\bar{P}$ , sibling, fundee,
    ( $sk_{loc,fund,new}, pk_{loc,fund,new}$ ),  $pk_{rem,fund,new}$ ,
     $pk_{sib,rem,fund,new}$ , ( $sk_{loc,fund,old}, pk_{loc,fund,old}$ ),
     $pk_{rem,fund,old}, pk_{loc,out}$ ,  $c_P, c_{\bar{P}}, cvirt, t_P, i, side, n$ ) by
    guest:
12:     ensure  $1 < i \leq n$  // host_funder ( $i = 1$ ) is
    initialised with HOST ME
13:     ensure side ∈ {"left", "right"}
14:     store message contents and guest // sibling,
     $pk_{sib,\bar{P},F}$  are missing for endpoints, fundee is present
    only in last node
15:     ( $sk_{i,fund,new}, pk_{i,fund,new}$ ) ←
    ( $sk_{loc,fund,new}, pk_{loc,fund,new}$ )
16:      $pk_{myRem,fund,new} \leftarrow pk_{rem,fund,new}$ 
17:     if  $i < n$  then // we are not last hop
18:          $pk_{sibRem,fund,new} \leftarrow pk_{sib,rem,fund,new}$ 
19:     end if
20:     if side = "left" then
21:         side' ← "right"; myRem ←  $i - 1$ ;
    sibRem ←  $i + 1$ 
22:          $pk_{i,out} \leftarrow pk_{loc,out}$ 
23:         ( $sk_{i,j,k}, pk_{i,j,k}$ )j ∈ {2,...,n-1}, k ∈ [n] \ {j} ←
    KEYGEN()(n-2)(n-1)
24:         ( $sk_{i,rev}, pk_{i,rev}$ ) ← KEYGEN()
25:     else // side = "right"
26:         side' ← "left"; myRem ←  $i + 1$ ;
    sibRem ←  $i - 1$ 
27:         // sibling will send keys in KEYS AND COINS
        FORWARD
28:     end if
29:     ( $sk_{i,side,fund,old}, pk_{i,side,fund,old}$ ) ←
    ( $sk_{loc,fund,old}, pk_{loc,fund,old}$ )
30:      $pk_{myRem,side',fund,old} \leftarrow pk_{rem,fund,old}$ 
31:     ( $c_{i,side}, c_{myRem,side'}, t_{i,side}$ ) ← ( $c_P, c_{\bar{P}}, t_P$ )
32:     last_poll ← ⊥
33:     output (HOST INIT OK) to guest

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

```

Process VIRT.CIRCULATEKEYSCOINSTIMES(left_data):

```

1: if left_data is given as argument then // we are not
    host_funder
2:     parse left_data as (( $pk_{j,fund,new}$ )j ∈ {i-1},
    ( $pk_{j,lef,fund,old}$ )j ∈ {2,...,i-1}, ( $pk_{j,right,fund,old}$ )j ∈ {i-1},
    ( $pk_{j,out}$ )j ∈ {i-1}, ( $c_{j,lef}$ )j ∈ {2,...,i-1}, ( $c_{j,right}$ )j ∈ {i-1},
    ( $t_j$ )j ∈ {i-1},  $pk_{left,virt}, pk_{right,virt}$ ,
    ( $pk_{h,j,k}$ )h ∈ {i-1}, j ∈ {2,...,n-1}, k ∈ [n] \ {j},
    ( $pk_{h,loc,rev}$ )h ∈ {i-1}, ( $pk_{h,rem,rev}$ )h ∈ {i-1})
3:     if we have a sibling then // we are not
    host_funder
4:         input (KEYS AND COINS FORWARD,
    (left_data, ( $sk_{i,lef,fund,old}, pk_{i,lef,fund,old}$ ),  $pk_{i,out}$ ,
     $c_{i,lef}, t_{i,lef}$ , ( $sk_{i,j,k}, pk_{i,j,k}$ )j ∈ {2,...,n-1}, k ∈ [n] \ {j},
    ( $sk_{i,rev}, pk_{i,rev}$ )) to sibling
5:         store input as left_data and parse it as
    (( $pk_{j,fund,new}$ )j ∈ {i-1}, ( $pk_{j,lef,fund,old}$ )j ∈ {2,...,i},
    ( $pk_{j,right,fund,old}$ )j ∈ {i-1}, ( $pk_{j,out}$ )j ∈ {i}, ( $c_{j,lef}$ )j ∈ {2,...,i},
    ( $c_{j,right}$ )j ∈ {i-1}, ( $t_j$ )j ∈ {i-1},  $sk_{i,lef,fund,old}, t_{i,lef}$ ,
     $pk_{left,virt}, pk_{right,virt}$ , ( $pk_{h,j,k}$ )h ∈ {i}, j ∈ {2,...,n-1}, k ∈ [n] \ {j},
    ( $sk_{i,j,k}$ )j ∈ {2,...,n-1}, k ∈ [n] \ {j}, ( $pk_{h,rev}$ )h ∈ {i},  $sk_{i,rev}$ 
6:          $t_i \leftarrow \max(t_{i,lef}, t_{i,right})$ 
7:         replace  $t_{i,lef}$  in left_data with  $t_i$ 
8:         remove  $sk_{i,lef,fund,old}$ ,
    ( $sk_{i,j,k}$ )j ∈ {2,...,n-1}, k ∈ [n] \ {j},  $sk_{i,loc,rev}$  and  $sk_{i,rem,rev}$ 
    from left_data
9:         call
    VIRT.CIRCULATEKEYSCOINSTIMES(left_data) of  $\bar{P}$ 
    and assign returned value to right_data
10:        parse right_data as
    (( $pk_{j,fund,new}$ )j ∈ {i+1,...,n}, ( $pk_{j,lef,fund,old}$ )j ∈ {i+1,...,n},
    ( $pk_{j,right,fund,old}$ )j ∈ {i+1,...,n-1}, ( $pk_{j,out}$ )j ∈ {i+1,...,n},
    ( $c_{j,lef}$ )j ∈ {i+1,...,n}, ( $c_{j,right}$ )j ∈ {i+1,...,n-1},
    ( $t_j$ )j ∈ {i+1,...,n},
    ( $pk_{h,j,k}$ )h ∈ {i+1,...,n}, j ∈ {2,...,n-1}, k ∈ [n] \ {j},
    ( $pk_{h,rev}$ )h ∈ {i+1,...,n})
11:        output (KEYS AND COINS BACK, right_data,
    ( $sk_{i,right,fund,old}, pk_{i,right,fund,old}$ ),  $c_{i,right}, t_i$ )
12:        store output as right_data and parse it as
    (( $pk_{j,fund,new}$ )j ∈ {i+1,...,n}, ( $pk_{j,lef,fund,old}$ )j ∈ {i+1,...,n},
    ( $pk_{j,right,fund,old}$ )j ∈ {i,...,n-1}, ( $pk_{j,out}$ )j ∈ {i+1,...,n},
    ( $c_{j,lef}$ )j ∈ {i+1,...,n}, ( $c_{j,right}$ )j ∈ {i,...,n-1}, ( $t_j$ )j ∈ {i,...,n},
    ( $pk_{h,j,k}$ )h ∈ {i+1,...,n}, j ∈ {2,...,n-1}, k ∈ [n] \ {j},
    ( $pk_{h,loc,rev}$ )h ∈ {i+1,...,n}, ( $pk_{h,rem,rev}$ )h ∈ {i+1,...,n},
     $sk_{i,right,fund,old}$ )
13:        remove  $sk_{i,right,fund,old}$  from right_data
14:        return (right_data,  $pk_{i,fund,new}$ ,
     $pk_{i,lef,fund,old}, pk_{i,out}, c_{i,lef}$ )
15:        else // we are host_funder
16:            output (CHECK KEYS, ( $pk_{left,virt}, pk_{right,virt}$ )) to
    fundee and expect reply (KEYS OK)
17:            return ( $pk_{n,fund,new}, pk_{n,lef,fund,old}, pk_{n,out}$ ,
     $c_{n,lef}, t_n$ , ( $pk_{n,j,k}$ )j ∈ {2,...,n-1}, k ∈ [n] \ {j},  $pk_{n,loc,rev}$ ,
     $pk_{n,rem,rev}$ )
18:        end if
19:        else // we are host_funder
20:            call VIRT.CIRCULATEKEYSCOINSTIMES( $pk_{1,fund,new}$ ,
     $pk_{1,right,fund,old}, pk_{1,out}, c_{1,right}, t_1, pk_{left,virt}$ ,

```

Figure 56

```

 $pk_{right,virt}, (pk_{1,j,k})_{j \in \{2, \dots, n-1\}, k \in [n] \setminus \{j\}}, pk_{1,loc,rev},$ 
 $pk_{1,rem,rev}$  of  $P$  and assign returned value to
right_data
21: parse right_data as  $((pk_{j,fund,new})_{j \in \{2, \dots, n\}},$ 
 $(pk_{j,loc,fund,old})_{j \in \{2, \dots, n\}}, (pk_{j,right,fund,old})_{j \in \{2, \dots, n-1\}},$ 
 $(pk_{j,out})_{j \in \{2, \dots, n\}}, (c_{j,loc})_{j \in \{2, \dots, n\}},$ 
 $(c_{j,right})_{j \in \{2, \dots, n-1\}}, (t_j)_{j \in \{2, \dots, n\}},$ 
 $(pk_{h,j,k})_{h \in \{2, \dots, n\}, j \in \{2, \dots, n-1\}, k \in [n] \setminus \{j\}},$ 
 $(pk_{h,loc,rev})_{h \in \{2, \dots, n\}}, (pk_{h,rem,rev})_{h \in \{2, \dots, n\}}$ 
22: return (OK)
23: end if

```

Figure 57

Process VIRT

```

1: GETMIDTXS( $i, n, c_{virt}, c_{rem, left}, c_{loc, left}, c_{loc, right},$ 
 $c_{rem, right}, pk_{rem, left, fund, old}, pk_{loc, left, fund, old},$ 
 $pk_{loc, right, fund, old}, pk_{rem, right, fund, old}, pk_{rem, left, fund, new},$ 
 $pk_{loc, left, fund, new}, pk_{loc, right, fund, new}, pk_{rem, right, fund, new},$ 
 $pk_{left, virt}, pk_{right, virt}, pk_{loc, out}, pk_{funder, rev}, pk_{left, rev},$ 
 $pk_{loc, rev}, pk_{right, rev}, pk_{fundee, rev},$ 
 $(pk_{h,j,k})_{h \in [n], j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1, j\}}, (pk_{h,2,1})_{h \in [n]},$ 
 $(pk_{h,n-1,n})_{h \in [n]}, (t_j)_{j \in [n-1] \setminus \{1\}}:$ 
2: ensure  $1 < i < n$ 
3: ensure  $c_{rem, left} \geq c_{virt} \wedge c_{loc, left} \geq c_{virt}$  // left parties
fund virtual channel
4: ensure  $c_{rem, left} \geq c_{loc, right} \wedge c_{rem, right} \geq c_{loc, left}$  //
avoid griefing attack
5:  $c_{left} \leftarrow c_{rem, left} + c_{loc, left};$ 
 $c_{right} \leftarrow c_{loc, right} + c_{rem, right}$ 
6:  $left\_old\_fund \leftarrow$ 
 $2/\{pk_{rem, left, fund, old}, pk_{loc, left, fund, old}\}$ 
7:  $right\_old\_fund \leftarrow$ 
 $2/\{pk_{loc, right, fund, old}, pk_{rem, right, fund, old}\}$ 
8:  $left\_new\_fund \leftarrow$ 
 $2/\{pk_{rem, left, fund, new}, pk_{loc, left, fund, new}\} \vee$ 
 $2/\{pk_{left, rev}, pk_{loc, rev}\}$ 
9:  $right\_new\_fund \leftarrow$ 
 $2/\{pk_{loc, right, fund, new}, pk_{rem, right, fund, new}\} \vee$ 
 $2/\{pk_{loc, rev}, pk_{right, rev}\}$ 
10:  $virt\_fund \leftarrow 2/\{pk_{left, virt}, pk_{right, virt}\}$ 
11: revocation  $\leftarrow$ 
 $4/\{pk_{funder, rev}, pk_{loc, rev}, pk_{right, rev}, pk_{fundee, rev}\}$ 
12: refund  $\leftarrow (pk_{loc, out} + (p+s)) \vee 2/\{pk_{left, rev}, pk_{loc, rev}\}$ 
13: for all  $j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1, j\}$  do
14:  $all_{j,k} \leftarrow n/\{pk_{1,j,k}, \dots, pk_{n,j,k}\} \wedge "k"$ 
15: end for
16: if  $i = 2$  then
17:  $all_{2,1} \leftarrow n/\{pk_{1,2,1}, \dots, pk_{n,2,1}\} \wedge "1"$ 
18: end if
19: if  $i = n-1$  then
20:  $all_{n-1,n} \leftarrow n/\{pk_{1,n-1,n}, \dots, pk_{n,n-1,n}\} \wedge "n"$ 
21: end if
22: if  $i = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
23: if  $i = n-1$  then  $l \leftarrow n$  else  $l \leftarrow n-1$ 
24: bridge1  $\leftarrow$ 
 $4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge "1"$  // We
reuse the  $pk_{j,2,1}$  keys for all bridges to avoid new keys.

```

$pk_{i,2,1}$ is not needed because i is not a beneficiary of the bridge tx.

```

25: revocation1  $\leftarrow$ 
 $4/\{pk_{funder, rev}, pk_{loc, rev}, pk_{right, rev}, pk_{fundee, rev}\} \wedge "1"$ 
26: for all  $k \in \{m, \dots, l\} \setminus \{i\}$  do
27: bridge2,k  $\leftarrow$ 
 $4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge "2, k"$ 
28: revocation2,k  $\leftarrow$ 
 $4/\{pk_{funder, rev}, pk_{loc, rev}, pk_{right, rev}, pk_{fundee, rev}\} \wedge "2, k"$ 
29: end for
30: for all  $(k_1, k_2) \in \{m, \dots, i-1\} \times \{i+1, \dots, l\}$  do
31: bridge3,k1,k2  $\leftarrow$ 
 $4/\{pk_{1,2,1}, pk_{i-1,2,1}, pk_{i+1,2,1}, pk_{n,2,1}\} \wedge "3, k_1, k_2"$ 
32: revocation3,k1,k2  $\leftarrow$ 
 $4/\{pk_{funder, rev}, pk_{loc, rev}, pk_{right, rev}, pk_{fundee, rev}\} \wedge "3, k_1, k_2"$ 
33: end for
34: // 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.
35: // TXi,j,k := i-th move, j, k input interval start and
end. j, k unneeded for  $i = 1$ , k unneeded for  $i = 2$ .
36: TX1  $\leftarrow$  TX:
37: inputs:
38:  $(c_{left}, left\_old\_fund),$ 
39:  $(c_{right}, right\_old\_fund)$ 
40: outputs:
41:  $(c_{left} - c_{virt}, left\_new\_fund),$ 
42:  $(c_{right} - c_{virt}, right\_new\_fund),$ 
43:  $(c_{virt}, refund),$ 
44:  $(c_{virt},$ 
45:  $\quad (\text{if } (i-1 > 1) \text{ then } all_{i-1,i} \text{ else False})$ 
46:  $\quad \vee (\text{if } (i+1 < n) \text{ then } all_{i+1,i} \text{ else False})$ 
47:  $\quad \vee revocation_1$ 
48:  $\quad \vee ($ 
49:  $\quad \quad \text{if } (i-1 = 1 \wedge i+1 = n) \text{ then}$ 
bridge1
50:  $\quad \quad \text{else if } (i-1 > 1 \wedge i+1 = n) \text{ then}$ 
bridge1 +  $t_{i-1}$ 
51:  $\quad \quad \text{else if } (i-1 = 1 \wedge i+1 < n) \text{ then}$ 
bridge1 +  $t_{i+1}$ 
52:  $\quad \quad \text{else } /*i-1 > 1 \wedge i+1 < n*/$ 
bridge1 + max( $t_{i-1}, t_{i+1}$ )
53:  $\quad )$ 
54:  $\quad )$ 
55:  $B_1 \leftarrow$  TX:
56: input:
57:  $(c_{virt}, bridge_1)$ 
58: output:
59:  $(c_{virt}, revocation \vee virt\_fund)$ 
60: if  $i = 2$  then
61: TX2,1  $\leftarrow$  TX:
62: inputs:
63:  $(c_{virt}, all_{2,1}),$ 
64:  $(c_{right}, right\_old\_fund)$ 
65: outputs:
66:  $(c_{right} - c_{virt}, right\_new\_fund),$ 
67:  $(c_{virt}, refund),$ 

```

```

68:           ( $c_{virt}$ ,
69:            (if ( $n > 3$ ) then
70:             ( $all_{3,2} \vee revocation_{2,1} \vee (bridge_{2,1} + t_3)$ )
71:             else  $revocation_{2,1} \vee bridge_{2,1}$ )
72:           )
73:            $B_{2,1} \leftarrow TX:$ 
74:           input:
75:             ( $c_{virt}$ ,  $bridge_{2,1}$ )
76:           output:
77:             ( $c_{virt}$ ,  $revocation \vee virt\_fund$ )
78:         end if
79:       if  $i = n - 1$  then
80:          $TX_{2,n} \leftarrow TX:$ 
81:         inputs:
82:           ( $c_{left}$ ,  $left\_old\_fund$ ),
83:           ( $c_{virt}$ ,  $all_{n-1,n}$ )
84:         outputs:
85:           ( $c_{left} - c_{virt}$ ,  $left\_new\_fund$ ),
86:           ( $c_{virt}$ ,  $refund$ ),
87:           ( $c_{virt}$ ,
88:            (if ( $n - 2 > 1$ ) then
89:              ( $all_{n-2,n-1} \vee revocation_{2,n} \vee (bridge_{2,n} + t_{n-2})$ )
90:              else  $revocation_{2,n} \vee bridge_{2,n}$ )
91:            )
92:            $B_{2,n} \leftarrow TX:$ 
93:           input:
94:             ( $c_{virt}$ ,  $bridge_{2,n}$ )
95:           output:
96:             ( $c_{virt}$ ,  $revocation \vee virt\_fund$ )
97:         end if
98:       for all  $k \in \{2, \dots, i - 1\}$  do //  $2(i - 2)$  txs
99:          $TX_{2,k} \leftarrow TX:$ 
100:        inputs:
101:          ( $c_{virt}$ ,  $all_{i,k}$ ),
102:          ( $c_{right}$ ,  $right\_old\_fund$ )
103:        outputs:
104:          ( $c_{right} - c_{virt}$ ,  $right\_new\_fund$ ),
105:          ( $c_{virt}$ ,  $refund$ ),
106:          ( $c_{virt}$ ,
107:           (if ( $k - 1 > 1$ ) then  $all_{k-1,i}$  else
108:            False)
109:            $\vee$  (if ( $i + 1 < n$ ) then  $all_{i+1,k}$  else
110:            False)
111:            $\vee revocation_{2,k}$ 
112:            $\vee$  (
113:             if ( $k - 1 = 1 \wedge i + 1 = n$ ) then
114:                $bridge_{2,k}$ 
115:             else if ( $k - 1 > 1 \wedge i + 1 = n$ )
116:               then  $bridge_{2,k} + t_{k-1}$ 
117:             else if ( $k - 1 = 1 \wedge i + 1 < n$ )
118:               then  $bridge_{2,k} + t_{i+1}$ 
119:             else /* $k - 1 > 1 \wedge i + 1 < n*/$ 
120:                $bridge_{2,k} + max(t_{k-1}, t_{i+1})$ 
121:             )
122:           )
123:            $B_{2,k} \leftarrow TX:$ 

```

```

117:           input:
118:             ( $c_{virt}$ ,  $bridge_{2,k}$ )
119:           output:
120:             ( $c_{virt}$ ,  $revocation \vee virt\_fund$ )
121:         end for
122:       for all  $k \in \{i + 1, \dots, n - 1\}$  do //  $2(n - i - 1)$  txs
123:          $TX_{2,k} \leftarrow TX:$ 
124:         inputs:
125:           ( $c_{left}$ ,  $left\_old\_fund$ )
126:           ( $c_{virt}$ ,  $all_{i,k}$ ),
127:         outputs:
128:           ( $c_{left} - c_{virt}$ ,  $left\_new\_fund$ ),
129:           ( $c_{virt}$ ,  $refund$ ),
130:           ( $c_{virt}$ ,
131:            (if ( $i - 1 > 1$ ) then  $all_{i-1,k}$  else
132:              False)
133:             $\vee$  (if ( $k + 1 < n$ ) then  $all_{k+1,i}$  else
134:              False)
135:             $\vee revocation_{2,k}$ 
136:             $\vee$  (
137:              if ( $i - 1 = 1 \wedge k + 1 = n$ ) then
138:                 $bridge_{2,k}$ 
139:              else if ( $i - 1 > 1 \wedge k + 1 = n$ )
140:                then  $bridge_{2,k} + t_{i-1}$ 
141:              else if ( $i - 1 = 1 \wedge k + 1 < n$ )
142:                then  $bridge_{2,k} + t_{k+1}$ 
143:              else /* $i - 1 > 1 \wedge k + 1 < n*/$ 
144:                 $bridge_{2,k} + max(t_{i-1}, t_{k+1})$ 
145:              )
146:            )
147:            $B_{2,k} \leftarrow TX:$ 
148:           input:
149:             ( $c_{virt}$ ,  $bridge_{2,k}$ )
150:           output:
151:             ( $c_{virt}$ ,  $\vee revocation \vee virt\_fund$ )
152:         end for
153:       for all  $(k_1, k_2) \in \{m, \dots, i - 1\} \times \{i + 1, \dots, l\}$  do
154:         //  $(i - m) \cdot (l - i)$  txs
155:          $TX_{3,k_1,k_2} \leftarrow TX:$ 
156:         inputs:
157:           ( $c_{virt}$ ,  $all_{i,k_1}$ ),
158:           ( $c_{virt}$ ,  $all_{i,k_2}$ )
159:         outputs:
160:           ( $c_{virt}$ ,  $refund$ ),
161:           ( $c_{virt}$ ,
162:            (if ( $k_1 - 1 > 1$ ) then
163:               $all_{k_1-1, min(k_2, n-1)}$  else False)
164:               $\vee$  (if ( $k_2 + 1 < n$ ) then
165:                 $all_{k_2+1, max(k_1, 2)}$  else False)
166:               $\vee revocation_{3,k_1,k_2}$ 
167:               $\vee$  (
168:                if ( $k_1 - 1 \leq 1 \wedge k_2 + 1 \geq n$ ) then
169:                   $bridge_{3,k_1,k_2}$ 
170:                else if ( $k_1 - 1 > 1 \wedge k_2 + 1 \geq n$ )
171:                  then  $bridge_{3,k_1,k_2} + t_{k_1-1}$ 
172:                else if ( $k_1 - 1 \leq 1 \wedge k_2 + 1 < n$ )
173:                  then  $bridge_{3,k_1,k_2} + t_{k_2+1}$ 
174:                )
175:              )
176:            )
177:            $B_{3,k_1,k_2} \leftarrow TX:$ 

```

```

163:           else /* $k_1 - 1 > 1 \wedge k_2 + 1 < n$ */
164:
165:           bridge3,k1,k2 + max (tk1-1, tk2+1)
166:           )
167:           B3,k1,k2 ← TX:
168:           input:
169:             (cvirt, bridge3,k1,k2)
170:           output:
171:             (cvirt, ∨ revocation ∨ virt_fund)
172:         end for
173:       return (
174:         TX1, B1,
175:         (TX2,k, B2,k)k ∈ {m,...,l} \ {i},
176:
177:         (TX3,k1,k2, B3,k1,k2)(k1,k2) ∈ {m,...,i-1} × {i+1,...,l}
      )

```

Figure 58

```

Process VIRT

1: // left and right refer to the two counterparties, with left
   being the one closer to the funder. Note difference with
   left/right meaning in VIRT.GETMIDTXS.
2: GETENDPOINTTX(i, n, cvirt, cleft, cright, pkleft,fund,old,
   pkright,fund,old, pkleft,fund,new, pkright,fund,new, pkleft,virt,
   pkright,virt, pkinterim,rev, pkendpoint,rev, (pkall,j)j ∈ [n], t):
3:   ensure i ∈ {1, n}
4:   ensure cleft ≥ cvirt // left party funds virtual channel
5:   ctot ← cleft + cright
6:   old_fund ← 2/{pkleft,fund,old, pkright,fund,old}
7:   new_fund ← 2/{pkleft,fund,new, pkright,fund,new} ∨
   2/{pkleft,rev, pkright,rev}
8:   virt_fund ← 2/{pkleft,virt, pkright,virt}
9:   revocation ← revocation1 ←
   2/{pkinterim,rev, pkendpoint,rev}
10:  if i = 1 then // funder's tx
11:    all ← n/{pkall,1, ..., pkall,n} ∧ "1"
12:    bridge ← 2/{pkall,2, pkall,n} ∧ "1" // We reuse
     the pkall,j keys to avoid new keys
13:  else // i = n, fundee's tx
14:    all ← n/{pkall,1, ..., pkall,n} ∧ "n"
15:    bridge ← 2/{pkall,1, pkall,n-1} ∧ "1"
16:  end if
17:  TX1 ← TX: // endpoints only have an “initiator” tx
18:  input:
19:    (ctot, old_fund)
20:  outputs:
21:    (ctot − cvirt, new_fund),
22:    (cvirt, all ∨ revocation1 ∨ (bridge + t))
23:  B1 ← TX:
24:  input:
25:    (cvirt, bridge)
26:  output:
27:    (cvirt, revocation ∨ virt_fund)

```

```

28:   return TX1, B1

```

Figure 59

```

Process VIRT.SIBLINGSIGS()

1: parse input as sigsbyLeft
2: if i = 2 then m ← 1 else m ← n − 1
3: if i = n − 1 then l ← n else l ← n − 1
4: (TXi,1, Bi,1, (TXi,2,k, Bi,2,k)k ∈ {m,...,l} \ {i},
   (TXi,3,k1,k2, Bi,3,k1,k2)(k1,k2) ∈ {m,...,i-1} × {i+1,...,l}) ←
   VIRT.GETMIDTXS(i, n, cvirt, ci-1,right, ci,left, ci,right,
   ci+1,left, pki-1,right,fund,old, pki,left,fund,old,
   pki,right,fund,old, pki+1,left,fund,old, pki-1,fund,new,
   pki,fund,new, pki,fund,new, pki+1,fund,new, pkleft,virt,
   pkright,virt, pki,out, pk1,rev, pkj,rev, pki+1,rev,
   pkn,rev, (pkh,j,k)h ∈ [n], j ∈ [n-1] \ {1}, k ∈ [n-1] \ {1,j},
   (pkh,2,1)h ∈ [n], (pkh,n-1,n)h ∈ [n], (th)h ∈ [n-1] \ {1})
5: // notation: sig(TX, pk) := sig with ANYPREVOUT flag
   such that VERIFY(TX, sig, pk) = True
6: ensure that the following signatures are present in
   sigsbyLeft and store them:
   • // (l − m) · (i − 1) signatures
7:   ∀k ∈ {m, ..., l} \ {i}, ∀j ∈ [i − 1] :
8:     sig(TXi,2,k, pkj,i,k)
   • // 2 · (l − m) · (l − i) · (i − 1) signatures
9:
   ∀k1 ∈ {m, ..., i − 1}, ∀k2 ∈ {i + 1, ..., l}, ∀j ∈ [i − 1] :
10:    sig(TXi,3,k1,k2, pkj,i,k1), sig(TXi,3,k1,k2, pkj,i,k2),
11:    sigstoRight ← sigsbyLeft
12: if i + 1 = n then // next hop is host_fundee
13:   TXn,1, Bn,1 ← VIRT.GETENDPOINTTX(n, n, cvirt,
   cn-1,right, cn,left, pkn-1,right,fund,old, pkn,left,fund,old,
   pkn-1,fund,new, pkn,fund,new, pkleft,virt, pkright,virt,
   pkn-1,rev, pkn,rev, (pkj,n-1,n)j ∈ [n], tn-1)
14:   add SIGN(Bn,1, ski,2,1, ANYPREVOUT) to sigstoRight
15: end if
16: for all j ∈ {2, ..., n − 1} \ {i} do
17:   if j = 2 then m' ← 1 else m' ← 2
18:   if j = n − 1 then l' ← n else l' ← n − 1
19:   (TXj,1, Bj,1, (TXj,2,k, Bj,2,k)k ∈ {m',...,l'} \ {i},
   (TXj,3,k1,k2, Bj,3,k1,k2)(k1,k2) ∈ {m',...,i-1} × {i+1,...,l'}) ←
   GETMIDTXS(j, n, cvirt, cj-1,right, cj,left, cj,right,
   cj+1,left, pkj-1,right,fund,old, pkj,left,fund,old,
   pkj,right,fund,old, pkj+1,left,fund,old, pkj-1,fund,new,
   pkj,fund,new, pkj,fund,new, pkj+1,fund,new, pkleft,virt,
   pkright,virt, pkj,out, pk1,rev, pkj-1,rev, pkj,rev, pkj+1,rev,
   pkn,rev, (pkk,p,s)k ∈ [n], p ∈ [n-1] \ {1}, s ∈ [n-1] \ {1,p},
   (pkk,2,1)k ∈ [n], (pkk,n-1,n)k ∈ [n], (tk)k ∈ [n-1] \ {1})
20:   if j = i − 1 then
21:     ensure that the following signatures are present in
     sigsbyLeft and store them:
     • // 2 signatures
22:     sig(Bi-1,1, pk1,2,1), sig(Bi-1,1, pki-1,2,1)
   • // 2(l' − m') signatures

```

```

23:            $\forall k \in \{m', \dots, l'\} \setminus \{i\} :$ 
24:                $\text{sig}(B_{i-1,2,k}, pk_{1,2,1}),$ 
25:                $\text{sig}(B_{i-1,2,k}, pk_{i-1,2,1})$ 
26:               • //  $2(i - m') \cdot (l' - i)$  signatures
27:           end if
28:           if  $j < i$  then  $\text{sigs} \leftarrow \text{sigs}_{\text{toLeft}}$  else  $\text{sigs} \leftarrow \text{sigs}_{\text{toRight}}$ 
29:           if  $j \in \{i - 1, i + 1\}$  then
30:               add  $\text{SIGN}(B_{j,1}, sk_{i,2,1}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
31:           end if
32:           for all  $k \in \{m', \dots, l'\} \setminus \{j\}$  do
33:               add  $\text{SIGN}(\text{TX}_{j,2,k}, sk_{i,j,k}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
34:               if  $j \in \{i - 1, i + 1\}$  then
35:                   add  $\text{SIGN}(B_{j,2,k}, sk_{i,2,1}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
36:               end if
37:           end for
38:           for all  $k_1 \in \{m', \dots, j - 1\}, k_2 \in \{j + 1, \dots, l'\}$  do
39:               add  $\text{SIGN}(\text{TX}_{j,3,k_1,k_2}, sk_{i,j,k_1}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
40:               add  $\text{SIGN}(\text{TX}_{j,3,k_1,k_2}, sk_{i,j,k_2}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
41:               if  $j \in \{i - 1, i + 1\}$  then
42:                   add  $\text{SIGN}(B_{j,3,k_1,k_2}, sk_{i,2,1}, \text{ANYPREVOUT})$  to  $\text{sigs}$ 
43:               end if
44:           end for
45:       end for

46: call  $\bar{P}.\text{CIRCULATEVIRTUALSIGS}(\text{sigs}_{\text{toRight}})$  and assign
   returned value to  $\text{sigs}_{\text{byRight}}$ 
47: output ( $\text{VIRTUALSIGSBACK}$ ,  $\text{sigs}_{\text{toLeft}}$ ,  $\text{sigs}_{\text{byRight}}$ )

```

Figure 60

```

 $c_{1,\text{right}}, c_{2,\text{left}}, pk_{1,\text{right,fund,old}}, pk_{2,\text{left,fund,old}},$ 
 $pk_{1,\text{fund,new}}, pk_{2,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}}, pk_{2,\text{rev}},$ 
 $pk_{1,\text{rev}}, (pk_{j,2,1})_{j \in [n]}, t_2)$ 
10: add  $\text{SIGN}(B_{1,1}, sk_{i,2,1}, \text{ANYPREVOUT})$  to  $\text{sigs}_{\text{toLeft}}$ 
11: end if
12: return  $\text{sigs}_{\text{toLeft}}$ 

```

Figure 61

Process VIRT.HOSTFUNDEESIGS()

```

1:  $TX_{n,1}, B_{n,1} \leftarrow \text{VIRT.GETENDPOINTTX}(n, n, c_{\text{virt}},$ 
 $c_{n-1,\text{right}}, c_{n,\text{left}}, pk_{n-1,\text{right,fund,old}}, pk_{n,\text{right,fund,old}},$ 
 $pk_{n-1,\text{fund,new}}, pk_{n,\text{fund,new}}, pk_{\text{left,virt}}, pk_{\text{right,virt}},$ 
 $pk_{n-1,\text{rev}}, pk_{n,\text{rev}}, (pk_{j,n-1,n})_{j \in [n]}, t_{n-1})$ 
2: ensure that  $\text{sig}(B_{n,1}, pk_{1,2,1}), \text{sig}(B_{n,1}, pk_{n-1,2,1})$  are
   present in  $\text{sigs}_{\text{byLeft}}$  and store them
3:  $\text{sigs}_{\text{toLeft}} \leftarrow \emptyset$ 
4: for all  $j \in [n - 1] \setminus \{1\}$  do
5:   if  $j = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
6:   if  $j = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
7:    $(TX_{j,1}, B_{j,1}, (TX_{j,2,k}, B_{j,2,k})_{k \in \{m, \dots, l\} \setminus \{j\}},$ 
 $(TX_{j,3,k_1,k_2}, B_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m, \dots, j-1\} \times \{j+1, \dots, l\}}) \leftarrow$ 
 $\text{VIRT.GETMIDTXS}(j, n, c_{\text{virt}}, c_{j-1,\text{right}}, c_{j,\text{left}}, c_{j,\text{right}},$ 
 $c_{j+1,\text{left}}, pk_{j-1,\text{right,fund,old}}, pk_{j,\text{left,fund,old}},$ 
 $pk_{j,\text{right,fund,old}}, pk_{j+1,\text{left,fund,old}}, pk_{j-1,\text{fund,new}},$ 
 $pk_{j,\text{fund,new}}, pk_{j,\text{fund,new}}, pk_{j+1,\text{fund,new}}, pk_{\text{left,virt}},$ 
 $pk_{\text{right,virt}}, pk_{j,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{j+1,\text{rev}},$ 
 $pk_{n,\text{rev}}, (pk_{h,s,k})_{h \in [n], s \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,s\}},$ 
 $(pk_{h,2,1})_{h \in [n]}, (pk_{h,n-1,n})_{h \in [n]}, (t_h)_{h \in [n-1] \setminus \{1\}})$ 
8:   for all  $k \in \{m, \dots, l\} \setminus \{j\}$  do
9:     ensure that  $\text{sig}(B_{j,2,k}, pk_{1,2,1}),$ 
 $\text{sig}(B_{j,2,k}, pk_{n-1,2,1})$  are present in  $\text{sigs}_{\text{byLeft}}$  and store
   them
10:    add  $\text{SIGN}(\text{TX}_{j,2,k}, sk_{n,j,k}, \text{ANYPREVOUT})$  to
     $\text{sigs}_{\text{toLeft}}$ 
11:    add  $\text{SIGN}(B_{j,2,k}, sk_{n,2,1}, \text{ANYPREVOUT})$  to
     $\text{sigs}_{\text{toLeft}}$ 
12:   end for
13:   for all  $k_1 \in \{m, \dots, j - 1\}, k_2 \in \{j + 1, \dots, l\}$  do
14:     ensure that  $\text{sig}(B_{j,3,k_1,k_2}, pk_{1,2,1}),$ 
 $\text{sig}(B_{j,3,k_1,k_2}, pk_{n-1,2,1})$  are present in  $\text{sigs}_{\text{byLeft}}$  and
   store them
15:     add  $\text{SIGN}(\text{TX}_{j,3,k_1,k_2}, sk_{n,j,k_1}, \text{ANYPREVOUT})$  to
     $\text{sigs}_{\text{toLeft}}$ 
16:     add  $\text{SIGN}(\text{TX}_{j,3,k_1,k_2}, sk_{n,j,k_2}, \text{ANYPREVOUT})$  to
     $\text{sigs}_{\text{toLeft}}$ 
17:     add  $\text{SIGN}(B_{j,3,k_1,k_2}, sk_{n,2,1}, \text{ANYPREVOUT})$  to
     $\text{sigs}_{\text{toLeft}}$ 
18:   end for
19: end for
20: return  $\text{sigs}_{\text{toLeft}}$ 

```

Figure 62

Process VIRT.INTERMEDIARYSIGS()

```

1: if  $i = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
2: if  $i = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
3:  $(TX_{i,1}, B_{i,1}, (TX_{i,2,k}, B_{i,2,k})_{k \in \{m, \dots, l\} \setminus \{i\}},$ 
 $(TX_{i,3,k_1,k_2}, B_{i,3,k_1,k_2})_{(k_1,k_2) \in \{m, \dots, i-1\} \times \{i+1, \dots, l\}}) \leftarrow$ 
 $\text{VIRT.GETMIDTXS}(i, n, c_{\text{virt}}, c_{i-1,\text{right}}, c_{i,\text{left}}, c_{i,\text{right}},$ 
 $c_{i+1,\text{left}}, pk_{i-1,\text{right,fund,old}}, pk_{i,\text{left,fund,old}},$ 
 $pk_{i,\text{right,fund,old}}, pk_{i+1,\text{left,fund,old}}, pk_{i-1,\text{fund,new}},$ 
 $pk_{i,\text{fund,new}}, pk_{i,\text{fund,new}}, pk_{i+1,\text{fund,new}}, pk_{\text{left,virt}},$ 
 $pk_{\text{right,virt}}, pk_{i,\text{out}}, pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{j+1,\text{rev}},$ 
 $pk_{n,\text{rev}}, (pk_{h,j,k})_{h \in [n], j \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,j\}},$ 
 $(pk_{h,2,1})_{h \in [n]}, (pk_{h,n-1,n})_{h \in [n]}, (t_h)_{h \in [n-1] \setminus \{1\}})$ 
4: // not verifying our signatures in  $\text{sigs}_{\text{byLeft}}$ , our (trusted)
   sibling will do that
5: input ( $\text{VIRTUAL SIGS FORWARD}$ ,  $\text{sigs}_{\text{byLeft}}$ ) to sibling
6: VIRT.SIBLINGSIGS()
7:  $\text{sigs}_{\text{toLeft}} \leftarrow \text{sigs}_{\text{byRight}} + \text{sigs}_{\text{toLeft}}$ 
8: if  $i = 2$  then // previous hop is host_funder
9:    $TX_{1,1}, B_{1,1} \leftarrow \text{VIRT.GETENDPOINTTX}(1, n, c_{\text{virt}},$ 

```

Process VIRT.HOSTFUNDERSIGS()

```

1: sigstoRight ← ∅
2: for all  $j \in [n - 1] \setminus \{1\}$  do
3:   if  $j = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
4:   if  $j = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
5:    $(TX_{j,1}, B_{j,1}, (TX_{j,2,k}, B_{j,2,k})_{k \in \{m, \dots, l\} \setminus \{j\}},$ 
 $(TX_{j,3,k_1,k_2}, B_{j,3,k_1,k_2})_{(k_1,k_2) \in \{m, \dots, j-1\} \times \{j+1, \dots, l\}}) \leftarrow$ 
 $VIRT.GETMIDTXS(j, n, cvirt, c_{j-1,right}, c_{j,left}, c_{j,right},$ 
 $c_{j+1,left}, pk_{j-1,right,fund,old}, pk_{j,left,fund,old},$ 
 $pk_{j,right,fund,old}, pk_{j+1,left,fund,old}, pk_{j-1,fund,new},$ 
 $pk_{j,fund,new}, pk_{j,fund,new}, pk_{j+1,fund,new}, pk_{left,virt},$ 
 $pk_{right,virt}, pk_{j,out}, pk_{1,rev}, pk_{j-1,rev}, pk_{j,rev}, pk_{j+1,rev},$ 
 $pk_{n,rev}, (pk_{h,s,k})_{h \in [n], s \in [n-1] \setminus \{1\}, k \in [n-1] \setminus \{1,s\}},$ 
 $(pk_{h,2,1})_{h \in [n]}, (pk_{h,n-1,n})_{h \in [n]}, (th)_{h \in [n-1] \setminus \{1\}})$ 
6:   for all  $k \in \{m, \dots, l\} \setminus \{j\}$  do
7:     add SIGN(TXj,2,k, sk1,j,k, ANYPREVOUT) to
      sigstoRight
8:     add SIGN(Bj,2,k, sk1,2,1, ANYPREVOUT) to
      sigstoRight
9:   end for
10:  for all  $k_1 \in \{m, \dots, j-1\}, k_2 \in \{j+1, \dots, l\}$  do
11:    add SIGN(TXj,3,k_1,k_2, sk1,j,k_1, ANYPREVOUT) to
      sigstoRight
12:    add SIGN(TXj,3,k_1,k_2, sk1,j,k_2, ANYPREVOUT) to
      sigstoRight
13:    add SIGN(Bj,3,k_1,k_2, sk1,2,1, ANYPREVOUT) to
      sigstoRight
14:  end for
15: end for
16: call VIRT.CIRCULATEVIRTUALSIGS(sigstoRight) of  $\bar{P}$  and
  assign output to sigsbyRight
17: TX1,1, B1,1 ← VIRT.GETENDPOINTTX(1, n, cvirt,
  c1,right, c2,left, pk1,right,fund,old, pk2,left,fund,old,
  pk1,fund,new, pk2,fund,new, pkleft,virt, pkright,virt, pk2,rev,
  pk1,rev, (pkj,2,1)j \in [n], t2)
18: ensure that sig(B1,1, pk2,2,1), sig(B1,1, pkn,2,1) are
  present in sigsbyRight and store them
19: for all  $j \in [n - 1] \setminus \{1\}$  do
20:   if  $j = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
21:   if  $j = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
22:   for all  $k \in \{m, \dots, l\} \setminus \{j\}$  do
23:     ensure that sig(Bj,2,k, pk2,2,1), sig(Bj,2,k, pkn,2,1)
      are present in sigsbyRight and store them
24:   end for
25:   for all  $k_1 \in \{m, \dots, j-1\}, k_2 \in \{j+1, \dots, l\}$  do
26:     ensure that sig(Bj,3,k_1,k_2, pk2,2,1),
      sig(Bj,3,k_1,k_2, pkn,2,1) are present in sigsbyRight and store
      them
27:   end for
28: end for
29: return (OK)

```

Figure 63

Process VIRT.CIRCULATEVIRTUALSIGS(sigs_{byLeft})

```

1: if  $1 < i < n$  then // we are not host_funder nor
  host_fundee
2: return VIRT.INTERMEDIARYSIGS()

```

```

3: else if  $i = 1$  then // we are host_funder
4:   return VIRT.HOSTFUNDERSIGS()
5: else if  $i = n$  then // we are host_fundee
6:   return VIRT.HOSTFUNDEESIGS()
7: end if // it is always  $1 \leq i \leq n - 1$  cf. Fig. 56, l. 12 and
  l. 37

```

Figure 64

Process VIRT.CIRCULATEFUNDINGSIGS(sigs_{byLeft})

```

1: if  $1 < i < n$  then // we are not endpoint
2:   if  $i = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
3:   if  $i = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
4:   ensure that the following signatures are present in
  sigsbyLeft and store them:
  • // 1 signature
5:   sig(TXi,1, pki-1,right,fund,old)
  • //  $n - 3 + \chi_{i=2} + \chi_{i=n-1}$  signatures
6:    $\forall k \in \{m, \dots, l\} \setminus \{i\}$ 
7:     sig(TXi,2,k, pki-1,right,fund,old)
8:   input (VIRTUAL BASE SIG FORWARD, sigsbyLeft) to
  sibling
9:   extract and store sig(TXi,1, pki-1,right,fund,old) and
   $\forall k \in \{m, \dots, l\} \setminus \{i\}$  sig(TXi,2,k, pki-1,right,fund,old)
  from sigsbyLeft // same signatures as sibling
10:  sigstoRight ←
    {SIGN(TXi+1,1, ski,right,fund,old, ANYPREVOUT)}
11:  if  $i + 1 < n$  then
12:    if  $i + 1 = n - 1$  then  $l' \leftarrow n$  else  $l' \leftarrow n - 1$ 
13:    for all  $k \in \{2, \dots, l'\}$  do
14:      add SIGN(TXi+1,2,k, ski,right,fund,old,
      ANYPREVOUT) to sigstoRight
15:    end for
16:  else //  $i + 1 = n$ 
17:    add SIGN(TXn,1, ski,right,fund,old, ANYPREVOUT)
  to sigstoRight
18:  end if
19:  call VIRT.CIRCULATEFUNDINGSIGS(sigstoRight) of  $\bar{P}$ 
  and assign returned values to sigsbyRight
20:  ensure that the following signatures are present in
  sigsbyRight and store them:
  • // 1 signature
21:  sig(TXi,1, pki+1,left,fund,old)
  • //  $n - 3 + \chi_{i=2} + \chi_{i=n-1}$  signatures
22:   $\forall k \in \{m, \dots, l\} \setminus \{i\}$ 
23:    sig(TXi,2,k, pki+1,right,fund,old)
24:  output (VIRTUAL BASE SIG BACK, sigsbyRight)
25:  extract and store sig(TXi,1, pki+1,right,fund,old) and
   $\forall k \in \{m, \dots, l\} \setminus \{i\}$  sig(TXi,2,k, pki+1,right,fund,old)
  from sigsbyRight // same signatures as sibling
26:  sigstoLeft ←
    {SIGN(TXi-1,1, ski,left,fund,old, ANYPREVOUT)}
27:  if  $i - 1 > 1$  then
28:    if  $i - 1 = 2$  then  $m' \leftarrow 1$  else  $m' \leftarrow 2$ 
29:    for all  $k \in \{m', \dots, n - 1\}$  do

```

```

30:      add SIGN(TXi-1,2,k, ski, left, fund, old,  

    ANYPREVOUT) to sigstoLeft  

31:   end for  

32: else // i - 1 = 1  

33:   add SIGN(TX1,1, ski, left, fund, old, ANYPREVOUT)  

    to sigstoLeft  

34: end if  

35: return sigstoLeft  

36: else if i = 1 then // we are host_funder  

37:   sigstoRight ←  

    {SIGN(TX2,1, sk1, right, fund, old, ANYPREVOUT)}  

38:   if 2 = n - 1 then l' ← n else l' ← n - 1  

39:   for all k ∈ {3, ..., l'} do  

40:     add SIGN(TX2,2,k, sk1, right, fund, old,  

    ANYPREVOUT) to sigstoRight  

41:   end for  

42:   call VIRT.CIRCULATEFUNDINGSIGS(sigstoRight) of  $\bar{P}$   

    and assign returned value to sigsbyRight  

43:   ensure that sig(TX1,1, pk2, left, fund, old) is present in  

    sigsbyRight and store it  

44:   return (OK)  

45: else if i = n then // we are host_fundee  

46:   ensure sig(TXn,1, pkn-1, right, fund, old) is present in  

    sigsbyLeft and store it  

47:   sigstoLeft ←  

    {SIGN(TXn-1,1, skn, left, fund, old, ANYPREVOUT)}  

48:   if n - 1 = 2 then m' ← 1 else m' ← 2  

49:   for all k ∈ {m', ..., n - 2} do  

50:     add SIGN(TXn-1,2,k, skn, left, fund, old,  

    ANYPREVOUT) to sigstoLeft  

51:   end for  

52:   return sigstoLeft  

53: end if // it is always 1 ≤ i ≤ n - cf. Fig. 56, l. 12 and  

    l. 37

```

Figure 65

Process VIRT.CIRCULATEREVOCATIONS(revoc_by_prev)

```

1: if revoc_by_prev is given as argument then // we are
not host_funder
2:   ensure
    guest.PROCESSREMOTEREVOCATION(revoc_by_prev)
    returns (OK)
3: else // we are host_funder
4:   revoc_for_next ← guest.REVOKEPREVIOUS()
5:   input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
6:   last_poll ← | $\Sigma$ |
7:   call
    VIRT.CIRCULATEREVOCATIONS(revoc_for_next) of
     $\bar{P}$  and assign returned value to revoc_by_next
8:   ensure
    guest.PROCESSREMOTEREVOCATION(revoc_by_next)
    returns (OK) // If the "ensure" fails, the opening process
    freezes, this is intentional. The channel can still close via
    (FORCECLOSE)
9:   return (OK)
10: end if
11: if we have a sibling then // we are not
    host_fundee nor host_funder

```

```

12:   input (VIRTUAL REVOCATION FORWARD) to
    sibling
13:   revoc_for_next ← guest.REVOKEPREVIOUS()
14:   input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
15:   last_poll ← | $\Sigma$ |
16:   call
    VIRT.CIRCULATEREVOCATIONS(revoc_for_next) of
     $\bar{P}$  and assign output to revoc_by_next
17:   ensure
    guest.PROCESSREMOTEREVOCATION(revoc_by_next)
    returns (OK)
18:   output (HOSTS READY,  $t_i$ ) to guest and expect reply
    (HOST ACK)
19:   output (VIRTUAL REVOCATION BACK)
20: end if
21: revoc_for_prev ← guest.REVOKEPREVIOUS()
22: if 1 < i < n then // we are intermediary
23:   output (HOSTS READY,  $t_i$ ) to guest and expect reply
    (HOST ACK) // p is every how many blocks we have to
    check the chain
24: else // we are host_fundee, case of host_funder
    covered earlier
25:   output (HOSTS READY,  $p + \sum_{j=2}^{n-1} (s - 1 + t_j)$ ) to
    guest and expect reply (HOST ACK)
26: end if
27: return revoc_for_prev

```

Figure 66

Process VIRT – poll

```

1: On input (CHECK FOR LATERAL CLOSE) by  $R \in$ 
{guest, funder, fundee}:
2:   input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
3:    $k_1 \leftarrow 0$ 
4:   if TXi-1,1 is defined and TXi-1,1 ∈  $\Sigma$  then
5:      $k_1 \leftarrow i - 1$ 
6:   end if
7:   for all k ∈ [i - 2] do
8:     if TXi-1,2,k is defined and TXi-1,2,k ∈  $\Sigma$  then
9:        $k_1 \leftarrow k$ 
10:    end if
11:   end for
12:    $k_2 \leftarrow 0$ 
13:   if TXi+1,1 is defined and TXi+1,1 ∈  $\Sigma$  then
14:      $k_2 \leftarrow i + 1$ 
15:   end if
16:   for all k ∈ {i + 2, ..., n} do
17:     if TXi+1,2,k is defined and TXi+1,2,k ∈  $\Sigma$  then
18:        $k_2 \leftarrow k$ 
19:     end if
20:   end for
21:   last_poll ← | $\Sigma$ |
22:   if  $k_1 > 0 \vee k_2 > 0$  then // at least one neighbour has
    published its TX
23:     ignore all messages except for (CHECK IF
    CLOSING) by  $R$ 
24:     State ← CLOSING
25:     sigs ← ∅

```

```

26:   end if
27:   if  $k_1 > 0 \wedge k_2 > 0$  then // both neighbours have
      published their TXs
28:     add  $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$  to sigs
29:     add  $(\text{sig}(\text{TX}_{i,3,k_1,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$  to sigs
30:     add  $\text{SIGN}(\text{TX}_{i,3,k_1,k_2}, sk_{i,i,k_1}, \text{ANYPREVOUT})$  to
      sigs
31:     add  $\text{SIGN}(\text{TX}_{i,3,k_1,k_2}, sk_{i,i,k_2}, \text{ANYPREVOUT})$  to
      sigs
32:     input (SUBMIT,  $\text{TX}_{i,3,k_1,k_2}$ , sigs) to  $\mathcal{G}_{\text{Ledger}}$ 
33:   else if  $k_1 > 0$  then // only left neighbour has
      published its TX
34:     add  $(\text{sig}(\text{TX}_{i,2,k_1}, pk_{p,i,k_1}))_{p \in [n] \setminus \{i\}}$  to sigs
35:     add  $\text{SIGN}(\text{TX}_{i,2,k_1}, sk_{i,i,k_1}, \text{ANYPREVOUT})$  to
      sigs
36:     add  $\text{SIGN}(\text{TX}_{i,2,k_1}, sk_{i,\text{left,fund,old}},$ 
       $\text{ANYPREVOUT})$  to sigs
37:     input (SUBMIT,  $\text{TX}_{i,2,k_1}$ , sigs) to  $\mathcal{G}_{\text{Ledger}}$ 
38:   else if  $k_2 > 0$  then // only right neighbour has
      published its TX
39:     add  $(\text{sig}(\text{TX}_{i,2,k_2}, pk_{p,i,k_2}))_{p \in [n] \setminus \{i\}}$  to sigs
40:     add  $\text{SIGN}(\text{TX}_{i,2,k_2}, sk_{i,i,k_2}, \text{ANYPREVOUT})$  to
      sigs
41:     add  $\text{SIGN}(\text{TX}_{i,2,k_2}, sk_{i,\text{right,fund,old}},$ 
       $\text{ANYPREVOUT})$  to sigs
42:     input (SUBMIT,  $\text{TX}_{i,2,k_2}$ , sigs) to  $\mathcal{G}_{\text{Ledger}}$ 
43:   end if

44: On input (CHECK FOR REVOKED) by
   $R \in \{\text{guest, funder, fundee}\}$ :
45:   input (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign output to  $\Sigma$ 
46:   if  $\text{TX}_{i-1,1} \in \Sigma \vee \exists k \in \mathbb{N} : \text{TX}_{i-1,2,k} \in \Sigma$  then // left
      counterparty maliciously published old virtual tx
47:     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
48:        $(R_a, sk_a, R_b, sk_b) \leftarrow$ 
         $(R_{i-1,2,k}, sk_{i,2,1}, R_{\text{loc, left, virt}}, sk_{i,\text{rev}})$ 
49:     else
50:
51:        $(R_a, sk_a, R_b, sk_b) \leftarrow (R_{i-1,1}, sk_{i,2,1}, R_{\text{loc, left, virt}}, sk_{i,\text{rev}})$ 
52:     end if
53:   input (SUBMIT,  $(R_a, R_b, R_{\text{loc, left, virt}},$ 
     $R_{\text{loc, left, fund}}), (\text{SIGN}(R_a, sk_a), (\text{SIGN}(R_b, sk_b),$ 
     $\text{SIGN}(R_{\text{loc, left, virt}}, sk_{i,\text{rev}}), \text{SIGN}(R_{\text{loc, left, fund}}, sk_{i,\text{rev}}))$ ) to
       $\mathcal{G}_{\text{Ledger}}$ 
54:   if  $\text{TX}_{i+1,1} \in \Sigma \vee \exists k \in \mathbb{N} : \text{TX}_{i+1,2,k} \in \Sigma$  then // right
      counterparty maliciously published old virtual tx
55:     input (SUBMIT,  $(R_{\text{loc, right, virt}}, R_{\text{loc, right, fund}},$ 
     $(\text{SIGN}(R_{\text{loc, right, virt}}, sk_{i,\text{rev}}), \text{SIGN}(R_{\text{loc, right, fund}},$ 
     $sk_{i,\text{rev}}))$ ) to  $\mathcal{G}_{\text{Ledger}}$ 
56:   end if
57:   output (NOTHING REVOKED) to  $R$ 

```

Figure 67

Process VIRT – On input (FORCECLOSE) by R :

```

1: // At most one of funder, fundee is defined
2: ensure  $R \in \{\text{guest, funder, fundee}\}$ 

```

```

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

```

Figure 68

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

```

1: ensure  $State = \text{CLOSING}$ 
2: ensure  $R \in \{\text{guest, funder, fundee}\}$ 
3: send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as  $R$  and assign reply to  $\Sigma$ 
4: if  $i = 1$  then // we are host_funder
5:   ensure that there exists an output with  $c_P + c_{\bar{P}} - c_{\text{virt}}$ 
  coins and a  $2/\{pk_{1,\text{fund,new}}, pk_{2,\text{fund,new}}\}$  spending
  method with expired/non-existent timelock in  $\Sigma$  // new
  base funding output
6:   ensure that there either exists an output with  $c_{\text{virt}}$ 
  coins and a  $2/\{pk_{\text{left,virt}}, pk_{\text{right,virt}}\}$  spending method
  with expired/non-existent timelock in  $\Sigma$  /*virtual funding
  output by a “bridge” tx*/ or a  $B_p$  output. In the
  latter case, collect all  $B_p$ ’s signatures in  $\text{sigs}$ , add
   $\text{SIGN}(B_p, sk_{1,2,1}, \text{ANYPREVOUT})$  (or, if  $p = n, 1$ ,
   $\text{SIGN}(B_p, sk_{1,n-1,n}, \text{ANYPREVOUT})$  instead) to  $\text{sigs}$ ,
  send (SUBMIT,  $B_p$ ,  $\text{sigs}$ ) to  $\mathcal{G}_{\text{Ledger}}$  and keep waiting
  here for (CHECK IF CLOSING) by  $R$  until  $B_p$  is in  $\Sigma$ 
  returned by sending (READ) to  $\mathcal{G}_{\text{Ledger}}$ .
7: else if  $i = n$  then // we are host_fundee
8:   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

- 9: 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 bridge_p output. In the latter case, collect all B_p ’s signatures in sig_s , add $\text{SIGN}(B_p, sk_{1,2,1}, \text{ANYPREVOUT})$ (or, if $p = n, 1$, $\text{SIGN}(B_p, sk_{1,n-1,n}, \text{ANYPREVOUT})$ instead) to sig_s , send (SUBMIT, B_p , sig_s) 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 $\mathcal{G}_{\text{Ledger}}$.
- 10: **else** // we are intermediary
 - 11: **if** $\text{side} = \text{“left”}$ **then** $j \leftarrow i - 1$ **else** $j \leftarrow i + 1$ // side is defined for all intermediaries – cf. Fig. 56, l. 11
 - 12: 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
 - 13: 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 $\text{bridge}_{i-1,p}$ output. In the latter case, collect all $B_{i-1,p}$ ’s signatures in sig_s , add $\text{SIGN}(B_{i-1,p}, sk_{1,2,1}, \text{ANYPREVOUT})$ (or, if $i - 1, p = n, 1$, $\text{SIGN}(B_{i-1,p}, sk_{1,n-1,n}, \text{ANYPREVOUT})$ instead) to sig_s , send (SUBMIT, $B_{i-1,p}$, sig_s) 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 $\mathcal{G}_{\text{Ledger}}$.
- 14: **end if**
- 15: $\text{State} \leftarrow \text{CLOSED}$
- 16: output (CLOSED) to R

Figure 69

Process VIRT – cooperative closing

// we are left intermediary or host of fundee
 On (COOP CLOSE, sig_{bal} , $\text{left_comms_revkeys}$) by \bar{P} :

- 1: ensure $\text{State} = \text{OPEN}$
- 2: parse sig_{bal} as $(c'_1, c'_2), \text{sig}_1, \text{sig}_2$
- 3: ensure $c_{\text{virt}} = c'_1 + c'_2$
- 4: ensure $\text{VERIFY}((c'_1, c'_2), \text{sig}_1, pk_{\text{left,virt}}) = \text{True}$
- 5: ensure $\text{VERIFY}((c'_1, c'_2), \text{sig}_2, pk_{\text{right,virt}}) = \text{True}$
- 6: $\text{State} \leftarrow \text{COOP CLOSING}$
- 7: extract $\text{sig}_{i-1,\text{right,C}}, pk_{i-1,\text{right,R}}$ from $\text{left_comms_revkeys}$
- 8: **if** $i < n$ **then** $M \leftarrow \text{CHECK COOP CLOSE}$ **else** $M \leftarrow \text{CHECK COOP CLOSE FUNDEE}$
- 9: output $(M, (c'_1, c'_2), \text{sig}_{i-1,\text{right,C}}, pk_{i-1,\text{right,R}})$ to guest
- 10: ensure $\text{State} = \text{OPEN}$ // executed by guest
- 11: $\text{State} \leftarrow \text{COOP CLOSING}$
- 12: store received signature as $\text{sig}_{\bar{P},C,i+1}$ // in guests, i is the current state number
- 13: store received revocation key as $pk_{\bar{P},R,i+2}$
- 14: remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}, pk'_{P,F}$ and $pk'_{\bar{P},F}$
- 15: $C_{P,i+1} \leftarrow \text{TX}$ {input:

- $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs:
 $(c_P + c'_1, (pk_{P,\text{out}} + (p + s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\})$,
 $(c_{\bar{P}} + c'_2, pk_{\bar{P},\text{out}})\}$
- 16: ensure $\text{VERIFY}(C_{P,i+1}, \text{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = \text{True}$
- 17: input (COOP CLOSE CHECK OK) to $\text{host}_{\bar{P}}$
- 18: **if** $i < n$ **then** // we are intermediary
- 19: input (COOP CLOSE, left_comms_keys) to sibling
- 20: ensure $\text{State} = \text{OPEN}$ // executed by sibling
- 21: $\text{State} \leftarrow \text{COOP CLOSING}$
- 22: output (COOP CLOSE SIGN COMM, (c'_1, c'_2)) to guest
- 23: ensure $\text{State} = \text{OPEN}$ // executed by guest of sibling
- 24: $\text{State} \leftarrow \text{COOP CLOSING}$
- 25: remove most recent keys from list of old funding keys and assign them to $sk'_{P,F}, pk'_{P,F}$ and $pk'_{\bar{P},F}$
- 26: $C_{\bar{P},i+1} \leftarrow \text{TX}$ {input:
 $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs:
 $(c_P + c'_1, pk_{P,\text{out}}),$
 $(c_{\bar{P}} + c'_2, (pk_{\bar{P},\text{out}} + (p + s)) \vee 2/\{pk_{P,R,i+1}, pk_{\bar{P},R,i+1}\})\}$
- 27: $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk'_{P,F})$
- 28: $(sk_{P,R,i+2}, pk_{P,R,i+2}) \leftarrow \text{KEYGEN}()$
- 29: input (NEW COMM TX, $\text{sig}_{P,C,i+1}, pk_{P,R,i+2}$) to $\text{host}_{\bar{P}}$
- 30: rename received signature to $\text{sig}_{i,\text{right,C}}$ // executed by sibling
- 31: rename received public key to $pk_{i,\text{right,R}}$ // in hosts, i is our hop number
- 32: send (COOP CLOSE, sig_{bal} , $(\text{left_comms_keys}, \text{sig}_{i,\text{right,C}}, pk_{i,\text{right,R}})$ to \bar{P} and expect reply (COOP CLOSE BACK, $(\text{right_comms_revkeys}, \text{right_revocations})$)
- 33: $R_{\text{loc,right,virt}} \leftarrow \text{TX}$ {input:
 $(c_{\text{virt}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$, output: $(c_{\text{virt}}, pk_{i,\text{out}})\}$
- 34: extract $\text{sig}_{i+1,\text{right,rev,virt}}$ from right_revocations
- 35: ensure $\text{VERIFY}(R_{\text{loc,right,virt}}, \text{sig}_{i+1,\text{right,rev,virt}}, pk_{i+1,\text{rev}}) = \text{True}$
- 36: $R_{\text{loc,right,fund}} \leftarrow \text{TX}$ {input:
 $(c_P + c_{\bar{P}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$, output:
 $(c_P + c_{\bar{P}}, pk_{i,\text{out}})\}$
- 37: extract $\text{sig}_{i+1,\text{right,rev,fund}}$ from right_revocations
- 38: ensure $\text{VERIFY}(R_{\text{loc,right,fund}}, \text{sig}_{i+1,\text{right,rev,fund}}, pk_{i+1,\text{rev}}) = \text{True}$
- 39: extract $\text{sig}_{i+1,\text{left,C}}$ from $\text{right_comms_revkeys}$
- 40: extract $\text{sig}_{i+1,\text{left,R}}$ from right_revocations
- 41: extract $pk_{i+1,\text{left,R}}$ from $\text{right_comms_revkeys}$
- 42: output (VERIFY COMM REV, $\text{sig}_{i+1,\text{left,C}}, \text{sig}_{i+1,\text{left,R}}, pk_{i+1,\text{left,R}})$ to guest
- 43: store received public key as $pk_{\bar{P},R,i+2}$ // executed by guest of sibling
- 44: store $\text{sig}_{i+1,\text{left,C}}$ as $\text{sig}_{\bar{P},C,i+1}, pk_{\bar{P},R,i+2}$
- 45: $C_{P,i+1} \leftarrow \text{TX}$ {input:
 $(c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$, outputs:
 $(c_P + c_1, (pk_{P,\text{out}} + (p + s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\})$,
 $(c_{\bar{P}} + c'_2, pk_{\bar{P},\text{out}})\}$
- 46: ensure $\text{VERIFY}(C_{P,i+1}, \text{sig}_{\bar{P},C,i+1}, pk'_{\bar{P},F}) = \text{True}$
- 47: store $\text{sig}_{i+1,\text{left,R}}$ as $\text{sig}_{\bar{P},R,i}$
- 48: $R_{P,i} \leftarrow \text{TX}$ {input: $C_{\bar{P},i}.\text{outputs}.\bar{P}$, output: $(c_P + c_{\bar{P}},$

```

     $pk_{P,\text{out}}\})$ 
49:   ensure VERIFY( $R_{P,i}$ ,  $\text{sig}_{\bar{P},R,i}$ ,  $pk_{\bar{P},R,i}$ ) = True
50:   input (COMM REV VERIFIED) to hostP
51:   output (COOP CLOSE BACK,
      right_comms_revkeys, right_revocations) to
      sibling // executed by sibling
52:    $R_{\text{loc},\text{left},\text{virt}} \leftarrow \text{TX} \{\text{input}:$ 
      ( $c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{i-1,\text{rev}}, pk_{i,\text{rev}}, pk_{n,\text{rev}}\}$ ), 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  $\bar{P}$ 
53:   extract  $\text{sig}_{n,i,\text{left},\text{rev},\text{virt}}$  from
      right_revocations
54:   ensure VERIFY( $R_{\text{loc},\text{left}}$ ,  $\text{sig}_{n,\text{left},\text{rev}}$ ,  $pk_{n,\text{rev}}$ ) = True
55:   if  $i = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
56:   if  $i = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
57:   ensure that the following signatures are present in
      right_revocations and store them:
      • // 1 signature
58:      $\text{sig}(R_{i-1,1}, pk_{n,\text{rev}})$ 
      • //  $l - m$  signatures
59:      $\forall k \in \{m, \dots, l\} \setminus \{i\} :$ 
60:        $\text{sig}(R_{i-1,2,k}, pk_{n,\text{rev}})$ 
      • //  $(i - m) \cdot (l - i)$  signatures
61:        $\forall k_1 \in \{m, \dots, i - 1\}, \forall k_2 \in \{i + 1, \dots, l\} :$ 
62:          $\text{sig}(R_{i-1,3,k_1,k_2}, pk_{n,\text{rev}})$ 
63: else //  $i = n$ , we are host of fundee
64:   output (REVOKE) to fundee
65:    $R_{\bar{P},i} \leftarrow \text{TX} \{\text{input}: C_{P,i}.\text{outputs}.P, \text{output}:$ 
      ( $c_P, pk_{\bar{P},\text{out}}\}$ ) // executed by fundee
66:    $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$ 
67:   virtual_revocation_sigs  $\leftarrow \emptyset$ 
68:   for  $j \in [n - 1]$  do
69:      $R_{j,1} \leftarrow \text{TX} \{\text{input}: \text{TX}_{j,1}.\text{revocation}_1,$ 
      output: ( $c_{\text{virt}}, pk_{j+1,\text{out}}\}$ )
70:      $\text{sig}_{j,R,1,i} \leftarrow \text{SIGN}(R_{j,1}, sk_{i,\text{rev}});$ 
      virtual_revocation_sigs  $\leftarrow$ 
      virtual_revocation_sigs  $\cup \text{sig}_{j,R,1,i}$ 
71:     if  $j = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
72:     if  $j = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
73:     for  $k \in \{m, \dots, l\}$  do
74:        $R_{j,2,k} \leftarrow \text{TX} \{\text{input}:$ 
         $\text{TX}_{j,2,k}.\text{revocation}_{2,k}$ , output: ( $c_{\text{virt}}, pk_{j+1,\text{out}}\}$ )
75:        $\text{sig}_{j,R,2,k,i} \leftarrow \text{SIGN}(R_{j,2,k}, sk_{i,\text{rev}});$ 
      virtual_revocation_sigs  $\leftarrow$ 
      virtual_revocation_sigs  $\cup \text{sig}_{j,R,2,k,i}$ 
76:     end for
77:     for  $k_1 \in \{m, \dots, j - 1\}, k_2 \in \{j + 1, \dots, l\}$  do
78:        $R_{j,3,k_1,k_2} \leftarrow \text{TX} \{\text{input}:$ 
         $\text{TX}_{j,3,k_1,k_2}.\text{revocation}_{3,k_1,k_2}$ , output:
        ( $c_{\text{virt}}, pk_{j+1,\text{out}}\}$ )
79:        $\text{sig}_{j,R,3,k_1,k_2,i} \leftarrow \text{SIGN}(R_{j,3,k_1,k_2}, sk_{i,\text{rev}});$ 
      virtual_revocation_sigs  $\leftarrow$ 
      virtual_revocation_sigs  $\cup \text{sig}_{j,R,3,k_1,k_2,i}$ 
80:     end for
81:   end for
82:   input (REVOCATIONS,  $\text{sig}_{P,R,i}$ ,
      virtual_revocation_sigs) to hostP
83:   rename received signature  $\text{sig}_{P,R,i}$  to  $\text{sig}_{n,\text{right},R}$ 
84:   for all  $j \in \{2, \dots, n\}$  do

```

```

85:    $R_{j,\text{left}} \leftarrow \text{TX} \{\text{input}:$ 
      ( $c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{j-1,\text{rev}}, pk_{j,\text{rev}}, pk_{n,\text{rev}}\}$ ), output:
      ( $c_{\text{virt}}, pk_{j,\text{out}}\}$ )
86:    $\text{sig}_{n,j,\text{left},\text{rev}} \leftarrow \text{SIGN}(R_{j,\text{left}}, sk_{n,\text{rev}})$ 
87: end for
88: end if
89: output (NEW COMM REV) to guest
90:  $C_{\bar{P},i+1} \leftarrow \text{TX} \{\text{input}:$ 
      ( $c_P + c_{\bar{P}} + c'_1 + c'_2, 2/\{pk'_{\bar{P},F}, pk'_{P,F}\}$ ), outputs:
      ( $c_P + c'_1, (pk_{\bar{P},\text{out}} + (p + s)) \vee 2/\{pk_{\bar{P},R,i+1}, pk_{P,R,i+1}\}$ ),
      ( $c_P + c'_2, pk_{P,\text{out}}\}$ ) // executed by guest
91:  $\text{sig}_{P,C,i+1} \leftarrow \text{SIGN}(C_{\bar{P},i+1}, sk'_{P,F})$ 
92:  $R_{\bar{P},i} \leftarrow \text{TX} \{\text{input}: C_{P,i}.\text{outputs}.P, \text{output}: (c_P + c_{\bar{P}},$ 
       $pk_{\bar{P},\text{out}}\}$ )
93:  $\text{sig}_{P,R,i} \leftarrow \text{SIGN}(R_{\bar{P},i}, sk_{P,R,i})$ 
94: ( $sk_{P,R,i+2}, pk_{P,R,i+2}\} \leftarrow \text{KEYGEN}()$ 
95: input (NEW COMM REV,  $\text{sig}_{P,C,i+1}$ ,  $\text{sig}_{P,R,i}$ ,  $pk_{P,R,i+2}$ )
      to hostP
96: rename  $\text{sig}_{P,C,i+1}$  to  $\text{sig}_{i,\text{left},C}$ 
97: rename  $\text{sig}_{P,R,i}$  to  $\text{sig}_{i,\text{left},R}$ 
98: rename received public key to  $pk_{i,\text{left},R}$ 
99:  $R_{\text{rem},\text{left},\text{virt}} \leftarrow \text{TX} \{\text{input}: (c_{\text{virt}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\}),$ 
      output: ( $c_{\text{virt}}, pk_{i-1,\text{out}}\}$ )
100:  $\text{sig}_{i,\text{left},\text{rev},\text{virt}} \leftarrow \text{SIGN}(R_{\text{rem},\text{left},\text{virt}}, sk_{i,\text{rev}})$ 
101:  $R_{\text{rem},\text{left},\text{fund}} \leftarrow \text{TX} \{\text{input}:$ 
      ( $c_P + c_{\bar{P}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\}$ ), output:
      ( $c_P + c_{\bar{P}}, pk_{i-1,\text{out}}\}$ )
102:  $\text{sig}_{i,\text{left},\text{rev},\text{fund}} \leftarrow \text{SIGN}(R_{\text{rem},\text{left},\text{fund}}, sk_{i,\text{rev}})$ 
103: if  $i < n$  then // we are intermediary
104:    $M \leftarrow (\text{COOP CLOSE BACK},$ 
      ((right_comms_revkeys,  $\text{sig}_{i,\text{left},C}, pk_{i,\text{left},R}$ ),
      (right_revocations,  $\text{sig}_{i,\text{left},\text{rev},\text{virt}}$ ,
       $\text{sig}_{i,\text{left},\text{rev},\text{fund}}, \text{sig}_{i,\text{left},R}))$ 
105: else //  $i = n$ , we are host of fundee
106:    $M \leftarrow (\text{COOP CLOSE BACK}, (\text{sig}_{i,\text{left},C}, pk_{i,\text{left},R},$ 
       $\text{sig}_{n,\text{left},R}), (\text{sig}_{n,\text{left},\text{rev},\text{virt}}, \text{sig}_{n,\text{left},\text{rev},\text{fund}},$ 
       $(\text{sig}_{n,j,\text{left},\text{rev}})_{j \in \{2, \dots, n\}}$ ), virtual_rev_sigs)
107: end if
108: send  $M$  to  $\bar{P}$  and expect reply (COOP CLOSE
      REVOCATIONS, left_revocations)
109: extract  $\text{sig}_{i-1,\text{right},R}$ ,  $\text{sig}_{1,i,\text{right},\text{rev}}$ ,  $\text{sig}_{i-1,\text{right},\text{rev}}$  from
      left_revocations
110: ensure VERIFY( $R_{\text{loc},\text{left},\text{virt}}$ ,  $\text{sig}_{1,\text{right},\text{rev}}$ ,  $pk_{1,\text{rev}}\}$  =
      True
111: ensure VERIFY( $R_{\text{loc},\text{left},\text{virt}}$ ,  $\text{sig}_{i-1,\text{right},\text{rev}}$ ,  $pk_{i-1,\text{rev}}\}$  =
      True
112:  $R_{\text{loc},\text{left},\text{fund}} \leftarrow \text{TX} \{\text{input}:$ 
      ( $c_P + c_{\bar{P}}, 2/\{pk_{i-1,\text{rev}}, pk_{i,\text{rev}}\}$ ), output:
      ( $c_P + c_{\bar{P}}, pk_{i,\text{out}}\}$ ) // the input corresponds to the
      revocation path of the right funding output of all virtual
      txs owned by  $\bar{P}$ 
113: extract  $\text{sig}_{i-1,\text{left},\text{rev},\text{fund}}$  from left_revocations
114: ensure VERIFY( $R_{\text{loc},\text{left},\text{fund}}$ ,  $\text{sig}_{i-1,\text{left},\text{rev},\text{fund}}$ ,
       $pk_{i-1,\text{rev}}\} = \text{True}$ 
115: if  $i = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$ 
116: if  $i = n - 1$  then  $l \leftarrow n$  else  $l \leftarrow n - 1$ 
117: ensure that the following signatures are present in
      left_revocations and store them:
      • // 2 signatures
118:    $\text{sig}(R_{i-1,1}, pk_{1,\text{rev}}), \text{sig}(R_{i-1,1}, pk_{i-1,\text{rev}})$ 
      • //  $2(l - m)$  signatures

```

```

119:    $\forall k \in \{m, \dots, l\} \setminus \{i\}$  :  

120:      $\text{sig}(R_{i-1,2,k}, pk_{1,\text{rev}}), \text{sig}(R_{i-1,2,k}, pk_{i-1,\text{rev}})$   

121:     • //  $2(i-m) \cdot (l-i)$  signatures  

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

123:        $\text{sig}(R_{i-1,3,k_1,k_2}, pk_{1,\text{rev}}),$   

124:        $\text{sig}(R_{i-1,3,k_1,k_2}, pk_{i-1,\text{rev}})$   

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

126:     store received signature as  $\text{sig}_{\bar{P},R,i}$  // executed by guest  

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

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

129:     add  $\text{host}_P$  to list of old hosts  

130:     assign received host to  $\text{host}_P$   

131:      $i \leftarrow i+1; c_P \leftarrow c_P + c'_2; c_{\bar{P}} \leftarrow c_{\bar{P}} + c'_1$   

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

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

134:     layer  $\leftarrow$  layer - 1  

135:     locked $_P \leftarrow$  locked $_P - c_{\text{virt}}$   

136:     State  $\leftarrow$  OPEN  

137:     hosting  $\leftarrow$  False  

138:     input (REV VERIFIED) to last old host  

139:     State  $\leftarrow$  COOP CLOSED  

140:     if  $i < n$  then // we are intermediary  

141:       send (COOP CLOSE REVOCATIONS, left_revocations) to sibling  

142:       output (COOP CLOSE REVOCATIONS,  $\text{host}_P$ ) to guest // executed by sibling  

143:        $R_{\bar{P},i} \leftarrow \text{TX}$  {input:  $C_{\bar{P},i}.\text{outputs}.P$ , output:  $(c_P, pk_{\bar{P},\text{out}})$ } // executed by guest of sibling  

144:       add  $\text{host}_P$  to list of old hosts  

145:       assign received host to  $\text{host}_P$   

146:        $i \leftarrow i+1; c_P \leftarrow c_P + c'_1; c_{\bar{P}} \leftarrow c_{\bar{P}} + c'_2$   

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

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

149:       layer  $\leftarrow$  layer - 1  

150:       locked $_P \leftarrow$  locked $_P - c_{\text{virt}}$   

151:       State  $\leftarrow$  OPEN  

152:       hosting  $\leftarrow$  False  

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

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

155:        $R_{\text{rem,right,virt}} \leftarrow \text{TX}$  {input:  

156:          $(c_{\text{virt}}, 4/\{pk_{1,\text{rev}}, pk_{i,\text{rev}}, pk_{i+1,\text{rev}}, pk_{n,\text{rev}}\})$ , output:  

157:          $(c_{\text{virt}}, pk_{i+1,\text{out}})}$   

158:       sig $_{i,\text{right,rev,virt}} \leftarrow \text{SIGN}(R_{\text{rem,right,virt}}, sk_{i,\text{rev}})$   

159:        $R_{\text{rem,right,fund}} \leftarrow \text{TX}$  {input:  

160:          $(c_P + c_{\bar{P}}, 2/\{pk_{i,\text{rev}}, pk_{i+1,\text{rev}}\})$ , output:  

161:          $(c_P + c_{\bar{P}}, pk_{i+1,\text{out}})}$   

162:       sig $_{i,\text{right,rev,fund}} \leftarrow \text{SIGN}(R_{\text{rem,right,fund}}, sk_{i,\text{rev}})$   

163:        $R_{i,1} \leftarrow \text{TX}$  {input:  $\text{TX}_{i,1}.\text{revocation}_1$ , output:  

164:          $(c_{\text{virt}}, pk_{i+1,\text{out}})}$   

165:       sig $_{i,R,1,i} \leftarrow \text{SIGN}(R_{i,1}, sk_{i,\text{rev}});$   

166:       left_revocations  $\leftarrow$   

167:       left_revocations  $\cup$  sig $_{i,R,1,i}$   

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

```

```

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

162:   for  $k \in \{m, \dots, l\}$  do  

163:      $R_{i,2,k} \leftarrow \text{TX}$  {input:  $\text{TX}_{i,2,k}.\text{revocation}_{2,k}$ ,  

164:       output:  $(c_{\text{virt}}, pk_{i+1,\text{out}})}$   

165:     sig $_{i,R,2,k,i} \leftarrow \text{SIGN}(R_{i,2,k}, sk_{i,\text{rev}});$   

166:     left_revocations  $\leftarrow$   

167:     left_revocations  $\cup$  sig $_{i,R,2,k,i}$   

168:   end for  

169:   for  $k_1 \in \{m, \dots, i-1\}, k_2 \in \{i+1, \dots, l\}$  do  

170:      $R_{i,3,k_1,k_2} \leftarrow \text{TX}$  {input:  

171:        $(c_{\text{virt}}, pk_{i+1,\text{out}})}$   

172:     sig $_{i,R,3,k_1,k_2,i} \leftarrow \text{SIGN}(R_{i,3,k_1,k_2}, sk_{i,\text{rev}});$   

173:     left_revocations  $\leftarrow$   

174:     left_revocations  $\cup$  sig $_{i,R,3,k_1,k_2,i}$   

175:   end for  

176:   send (COOP CLOSE REVOCATIONS,  

177:     (left_revocations,  $\text{sig}_{i,\text{right},R}, \text{sig}_{i,\text{right,rev,virt}},$   

178:        $\text{sig}_{i,\text{right,rev,fund}})$  to  $\bar{P}$ )  

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

180:   extract  $\text{sig}_{1,\text{right},R}$  from left_revocations  

181:   output (VERIFY REVOCATION,  $\text{sig}_{1,\text{right},R}$ ) to fundee  

182:   store received signature as  $\text{sig}_{\bar{P},R,i}$  // executed by fundee  

183:    $R_{P,i} \leftarrow \text{TX}$  {input:  $C_{\bar{P},i}.\text{outputs}.\bar{P}$ , output:  

184:      $(c_P, pk_{P,\text{out}})}$   

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

186:   for  $j \in [n-1]$  do  

187:     if  $j = 2$  then  $m \leftarrow 1$  else  $m \leftarrow 2$   

188:     if  $j = n-1$  then  $l \leftarrow n$  else  $l \leftarrow n-1$   

189:     ensure that the following signatures are present in  

190:     left_revocations and store them: // exclude  

191:     signatures by  $j+1$  if  $j = n-1$   

192:     • // 3 signatures  

193:     sig $_{(j,1),pk_{1,\text{rev}}}, \text{sig}(R_{j,1}, pk_{j,\text{rev}}),$   

194:     sig $_{(j,1),pk_{j+1,\text{rev}}}$   

195:     • //  $3(l-m) \cdot (l-i)$  signatures  

196:      $\forall k \in \{m, \dots, l\} \setminus \{i\}$  :  

197:       sig $_{(j,2,k),pk_{1,\text{rev}}}, \text{sig}(R_{j,2,k}, pk_{j,\text{rev}}),$   

198:       sig $_{(j,2,k),pk_{j+1,\text{rev}}}$   

199:       • //  $3(i-m) \cdot (l-i)$  signatures  

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

201:       sig $_{(j,3,k_1,k_2),pk_{1,\text{rev}}},$   

202:       sig $_{(j,3,k_1,k_2),pk_{j+1,\text{rev}}}$   

203:     end for  

204:     State  $\leftarrow$  COOP CLOSED  

205:     if close_initiator =  $P$  then //  $\mathcal{E}$  instructed us  

206:       to close the channel  

207:       execute code of Fig. 52  

208:     else //  $\mathcal{E}$  instructed another party to close the channel  

209:       send (COOPCLOSED) to close_initiator  

210:     end if  

211:   end if

```

Figure 70

Process VIRT – punishment handling

```

1: On input (USED REVOCATION) by guest: // (USED
   REVOCATION) by funder/fundee is ignored
2:   State ← GUEST PUNISHED
3:   input (USED REVOCATION) to hostP, expect reply
   (USED REVOCATION OK)
4:   if funder or fundee is defined then
5:     output (ENABLER USED REVOCATION) to it
6:   else // sibling is defined
7:     output (ENABLER USED REVOCATION) to
   sibling
8:   end if

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

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

```

Figure 71

Appendix M. Omitted Theorems and Proofs

Lemma 3 (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 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 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 host_P ≠ “ledger” the output of host_P is (CLOSED), then eventually the state obtained when P inputs (READ) to

$\mathcal{G}_{\text{Ledger}}$ will contain h outputs each of value c_i and that has been spent or is exclusively spendable by $pk_{R,\text{out}}$ such that

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

with overwhelming probability in the security parameter, where R is a local, kindred LN machine (i.e., either P , the guest of 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) host_P ≠ “ledger”,
- 2) P has State OPEN.

Consider two alternative series of subsequent execution steps:

- 1) The guest of 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

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

with overwhelming probability in the security parameter.

Proof: [Proof of Lemma 3] 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 $(\text{OPEN}, c, \text{hops}, \dots)$ 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, \dots) to P , as they were at the moment the latter message was sent. After initialised, F remains immutable. Observe that, if $\text{hops} \neq \text{“ledger”}$, both aforementioned messages must have been received before P transitions to the OPEN state.

The list C of party P is initialised to $[[g]]$ when P 's State transitions for the first time to OPEN, where $g = c$ if $P = \text{Alice}$, or $g = 0$ if $P = \text{Bob}$; 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. 43, l. 21) or sent (Fig. 43, l. 6) respectively by P . Moving on to the funding of new virtual channels, whenever P funds a new virtual channel (Fig. 40, 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. 40, l. 24), $[0]$ is appended to C . In case of cooperatively closing a channel (Figs. 51–54 & 70) to which P 's channel is base, if this channel was initially funded by P , when the closing procedure completes (Fig. 54, l. 53) $[c'_1]$ is appended to C . Likewise, if in the closed virtual channel P was the base of the fundee (Fig. 70, l. 171), then $[c'_2]$ (Fig. 70, l. 9) is appended to C . In case P was a left intermediary for the closed virtual channel (Fig. 70, l. 10), then $[c'_2]$ is appended to C . Lastly, in case P was a right intermediary for the closed virtual channel (Fig. 70, 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 $\sum_{s \in C} \sum_{x \in s} x$, 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 $\text{hops} = \text{“ledger”}$ and is closed right away, therefore $H = (\[], [g])$, where $g = c$ if $P = \text{Alice}$ and $g = 0$ if $P = \text{Bob}$. 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, \dots) while in the INIT State. In case $P = \text{Alice}$, this message must have been received as input by \mathcal{E} (Fig. 38, l. 1), or in case $P = \text{Bob}$, this message must have been received via the network by \bar{P} (Fig. 33, l. 3).
- It has received $pk_{\bar{P}, F}$. In case $P = \text{Bob}$, $pk_{\bar{P}, F}$ must have been contained in the (OPEN, \dots) message by \bar{P} (Fig. 33, l. 3), otherwise if $P = \text{Alice}$ $pk_{\bar{P}, F}$ must have been contained in the $(\text{ACCEPT CHANNEL}, \dots)$ message by \bar{P} (Fig. 33, 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. 35, ll. 12 and 23).

- It has the transaction F in the $\mathcal{G}_{\text{Ledger}}$ state (Fig. 36, l. 3 or Fig. 37, 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. 35, ll. 4 or 14), and sent it to \bar{P} (Fig. 35, 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. 35, 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. 35, l. 2 or 3). Additionally, if P receives (FORCECLOSE) by \mathcal{E} while $H = (\[], [g])$, it attempts to publish $C_{P,0}$ (Fig. 49, 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, \dots) by a (local, kindred) LN ITI R , subsequently transitions back to the OPEN state (therefore moving to history (F, C') where $C' = C + [-f]$) and finally receives (FORCECLOSE) by \mathcal{E} and (CLOSED) by host_P before any further change to its history, then eventually P 's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by $pk_{P,\text{out}}$ that are descendants of an output with spending method $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ such that $\sum_{i=1}^h c_i \geq \sum_{s \in C'} \sum_{x \in s} x$. Furthermore, given that P moves to the OPEN state after the (FUND ME, \dots) message, it also sends (FUNDED, \dots) to R (Fig. 40, 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 host_R ($\text{host}_R = \text{host}_P - \text{Fig. 37, l. 3}$) before any further change to its history, then eventually R 's $\mathcal{G}_{\text{Ledger}}$ state will contain k transaction outputs each of value c_i^R exclusively spendable or already spent by $pk_{R,\text{out}}$ that are descendants of an output with spending method $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ such that $\sum_{i=1}^k c_i^R \geq \sum_{s \in C_R} \sum_{x \in s} x$.

- If P receives (VIRTUALISING, ...) by \bar{P} 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 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with spending method $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ such that $\sum_{i=1}^h c_i \geq \sum_{s \in C} \sum_{x \in s} x$. Furthermore, given that P moves to the OPEN state after the (VIRTUALISING, ...) message and in case it sends (FUNDED, ...) to some party R (Fig. 40, 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 host_R ($\text{host}_R = \text{host}_P - \text{Fig. 37, l. 3}$) before any further change to its history, then eventually R 's $\mathcal{G}_{\text{Ledger}}$ state will contain an output with a $2/\{\text{pk}_{R,F}, \text{pk}_{\bar{R},F}\}$ spending method.
- If P receives (CHECK COOP CLOSE, ...) by 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 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with spending method $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ 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 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 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with spending method $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ 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 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 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with spending method $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ 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 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 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with spending method $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ 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 host_P (the latter only if $\text{host}_P \neq \text{"ledger"}$ or equivalently $F \neq []$) before any further change to its history, then eventually P 's $\mathcal{G}_{\text{Ledger}}$ state will contain h transaction outputs each of value c_i exclusively spendable or already spent by $\text{pk}_{P,\text{out}}$ that are descendants of an output with a $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ spending method such that $\sum_{i=1}^h c_i \geq \sum_{s \in C'} \sum_{x \in s} x$.
- If P receives (GET PAID, e) by \mathcal{E} , subsequently transitions back to OPEN (therefore moving to history (F, C') where C' is C with e appended to the last list of C) and finally receives (FORCECLOSE) by \mathcal{E} and (CLOSED) by host_P (the latter only if $\text{host}_P \neq \text{"ledger"}$ 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 $\text{pk}_{P,\text{out}}$ that are descendants of an output with a $2/\{\text{pk}_{P,F}, \text{pk}_{\bar{P},F}\}$ 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 $\text{pk}_{P,\text{out}}$ with a sum value of c_P . When P is in the OPEN state and receives (FUND ME, f , ...), it can only move again to the OPEN state after doing the following state transitions: OPEN \rightarrow VIRTUALISING \rightarrow WAITING FOR REVOCATION \rightarrow WAITING FOR INBOUND REVOCATION \rightarrow WAITING FOR HOSTS READY \rightarrow OPEN. During this sequence of events, a new host_P is defined (Fig. 40, l. 6), new commitment transactions are negotiated with \bar{P} (Fig. 40, l. 9), control of the old funding output is handed over to host_P (Fig. 40, l. 11), host_P negotiates with its counterparty a new set of transactions and signatures that spend the aforementioned funding output and make available a new funding output with the keys $\text{pk}'_{P,F}, \text{pk}'_{\bar{P},F}$

as P instructed (Fig. 63 and 65) and the previous valid commitment transactions of both P and \bar{P} are invalidated (Fig. 32, l. 1 and l. 14 respectively). We note that the use of the ANYPREVOUT flag in all signatures that correspond to transaction inputs that may spend various different transaction outputs ensures that this is possible, as it avoids tying each input to a specific, predefined output. When P receives (FORCECLOSE) by \mathcal{E} , it inputs (FORCECLOSE) to host_P (Fig. 49, l. 4). As per the Lemma conditions, host_P will output (CLOSED). This can happen only when $\mathcal{G}_{\text{Ledger}}$ contains a suitable output for both P 's and R 's channel (Fig. 69, l. 5 and l. 6 respectively).

If the host of host_P is “ledger”, then the funding output $o_{1,2} = (c_P + c_{\bar{P}}, 2/\{pk_{P,F}, pk_{\bar{P},F}\})$ for the P, \bar{P} channel is already on-chain. Regarding the case in which $\text{host}_P \neq \text{“ledger”}$, after the funding procedure is complete, the new host_P will have as its host the old host_P of P . If the (FORCECLOSE) sequence is initiated, the new host_P will follow the same steps that will be described below once the old host_P succeeds in closing the lower layer (Fig. 68, l. 6). The old host_P however will see no difference in its interface compared to what would happen if P had received (FORCECLOSE) before the funding procedure, therefore it will successfully close by the induction hypothesis. Thereafter the process is identical to the one when the old $\text{host}_P = \text{“ledger”}$.

Moving on, 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. 65, l. 43), $\text{sig}(B_{1,1}, pk_{2,2,1})$ and $\text{sig}(B_{1,1}, pk_{n,n-1,n})$ (Fig. 63, 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 host_P has signed with $sk_{P,F}$ (Fig. 65, ll. 37-41). The output can be also spent by old, revoked commitment transactions, but in that case host_P would not have output (CLOSED); P would have instead detected this triggered by a (CHECK CHAIN FOR CLOSED) message by \mathcal{E} (Fig. 47) 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 $\text{TX}_{1,1}$, $\text{TX}_{2,1}$, $(\text{TX}_{2,2,k})_{k>2}$ has a $(c_P + c_{\bar{P}} - f, 2/\{pk'_{P,F}, pk'_{\bar{P},F}\})$ output (Fig. 59, l. 21 and Fig. 58, ll. 41 and 128) which will end up in $\mathcal{G}_{\text{Ledger}}$ – call this output o_P . We will prove that at most $\sum_{i=2}^{n-1} (t_i + p + s - 1)$ blocks after (FORCECLOSE) is received by P , an output o_R with c_{virt} coins and a $2/\{pk_{R,F}, pk_{\bar{R},F}\}$ spending condition without or with an expired timelock will be included in $\mathcal{G}_{\text{Ledger}}$. In case party \bar{P} is idle, then $o_{1,2}$ is consumed by $\text{TX}_{1,1}$, 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 $\text{TX}_{2,1}$, $(\text{TX}_{2,2,k})_{k>2}$ or $(\text{TX}_{2,3,1,k})_{k>2}$ is a descendant of $o_{1,2}$; if the transaction

belongs to one of the two last transaction groups (with subscript g) then necessarily $\text{TX}_{1,1}$ is on-chain in some block height h and given the timelock on the virtual output of $\text{TX}_{1,1}$, \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. 69, l. 6). The rest of the cases are covered by the following sequence of events:

```

  Closing sequence
  -----
  1: maxDel ←  $t_2 + p + s - 1$  //  $A_2$  is active and the virtual
     output of  $\text{TX}_{1,1}$  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 ← maxDel +  $t_i + p + s - 1$ 
    8:     The published transaction can be of the form
       $\text{TX}_{i,2,2}$  or  $(\text{TX}_{i,3,2,k})_{k>i}$  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. 69, 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  $\text{TX}_{i,3,2,k}$  then  $i \leftarrow k$  else
       $i \leftarrow i + 1$ 
    13:      end if
    14:    end if
    15:  end loop
  16: //  $\text{maxDel} \leq \sum_{i=2}^{n-1} (t_i + p + s - 1)$ 

```

Figure 72

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

channel for a $(c_P - f, pk_{P,\text{out}})$ output that is a descendant of an output with spending method $2/\{pk_{P,F}, pk_{\bar{P},F}\}$ and that lower bound of value holds for the duration of the funding procedure, i.e., we have proven the first claim of the first bullet.

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

We now move on to the second bullet. In case P is the 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 $o_{n-1,n}$, $\text{TX}_{1,1}$ with $\text{TX}_{n,1}$, $B_{1,1}$ with $B_{n,1}$, $\text{TX}_{2,1}$ with $\text{TX}_{n-1,1}$, $B_{2,1}$ with $B_{n-1,1}$, $(\text{TX}_{2,2,k})_{k>2}$ with $(\text{TX}_{n-1,2,k})_{k<n-1}$, $(B_{2,2,k})_{k>2}$ with $(B_{n-1,2,k})_{k<n-1}$, $(\text{TX}_{2,3,1,k})_{k>2}$ with $(\text{TX}_{n-1,3,n,k})_{k<n-1}$, $(B_{2,3,1,k})_{k>2}$ with $(B_{n-1,3,n,k})_{k<n-1}$, t_2 with t_{n-1} , $\text{TX}_{i,3,2,k}$ with $\text{TX}_{i,3,n-1,k}$, $B_{i,3,2,k}$ with $B_{i,3,n-1,k}$, i is initialized to $n-2$ in l. 2 of Fig. 72, 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 $c_P = \sum_{s \in C} \sum_{x \in s} x$ coins. If P is on the “left” of its sibling (i.e., there is an untrusted party that sent the (VIRTUALISING, ...) message to P which triggered the latter to move to the VIRTUALISING state and to send a (VIRTUALISING, ...) message to its own sibling), the “left” funding output o_{left} (the one held with the untrusted party to the left) can be spent by one of $\text{TX}_{i,1}$, $(\text{TX}_{i,2,k})_{k>i}$, $\text{TX}_{i-1,1}$, 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 and a $pk_{P,\text{out}}$ spending method and no other spending method can be used (as P has not signed the “revocation” spending method of $C_{P,0}$).

In the case that P is to the right of its sibling (i.e., P receives by sibling the (VIRTUALISING, ...) message that causes P 's transition to the 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 host_P has not signed the required revocation transaction). If instead $\text{TX}_{i+1,1}$ or one of $(\text{TX}_{i+1,2,k})_{k>i+1}$ 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 method cannot be used since 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 $c_P = \sum_{s \in C} \sum_{x \in s} x$. 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. 70, l. 16) which spends the latest old funding output (Fig. 70, l. 14), effectively removing one virtualisation layer. In $C_{P,i+1}$ P owns c'_2 more coins than before that moment (Fig. 70, l. 15). It then signs the corresponding commitment transaction for the counterparty (Fig. 70, l. 91) and expects a valid signature for the revocation transaction of the old commitment transaction of the counterparty (Fig. 70, 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 host_P will punish the publisher of such transaction with the corresponding virtual revocation transaction (Fig. 70, l. 35, l. 38, l. 62, l. 110, l. 111 and l. 114) at the latest when P receives (CHECK CHAIN FOR CLOSED) (Fig. 47, 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. 70, 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 $c_P = \sum_{s \in C} \sum_{x \in s} x$. 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. 70, l. 27) which spends the latest old funding output (Fig. 70, 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. 70, 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. 70, l. 46) and on the revocation transaction of the old commitment transaction of the counterparty (Fig. 70, 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 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. 70, 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. 54.

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 $c_P = \sum_{s \in C} \sum_{x \in s} x$. (Note that $\sum_{s \in C'} \sum_{x \in s} x = d + \sum_{s \in C} \sum_{x \in s} x$.) When P receives (PAY, d) while in the OPEN state, it moves to the WAITING FOR COMMITMENT SIGNED state before returning to the OPEN state. It signs (Fig. 42, l. 2) the new commitment transaction $C_{\bar{P},i+1}$ in which the counterparty owns d more coins than before that moment (Fig. 42, l. 1), sends the signature to the counterparty (Fig. 42, l. 5) and expects valid signatures on its own updated commitment transaction (Fig. 43, l. 1) and the revocation transaction for the old commitment transaction of the counterparty (Fig. 43, 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. 43, l. 5) and send (Fig. 43, l. 17) the counterparty's revocation transaction for P 's previous commitment transaction.

Similarly to previous bullets, if $\text{host}_P \neq \text{"ledger"}$ the funding output can be put on-chain, otherwise the funding output is already on-chain. In both cases, since the closing procedure continues, one of $C_{P,i+1}, (C_{\bar{P},j})_{0 \leq j \leq i+1}$ will end up on-chain. If $C_{\bar{P},j}$ for some $j < i + 1$ is on-chain, then P submits $R_{P,j}$ (we discussed how P obtained $R_{P,i}$ and the rest of the cases are covered by induction) and takes the entire value of the channel which is at least $c_P - d$. If $C_{P,i+1}$ 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 $c_P = \sum_{s \in C} \sum_{x \in s} x$. (Note that $e + \sum_{s \in C'} \sum_{x \in s} x = \sum_{s \in C} \sum_{x \in s}$ 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. 45, l. 6). If subsequently it receives a valid signature for $C_{P,i+1}$ (Fig. 42, l. 9) which is a commitment transaction that can spend the o_F output and gives to P an additional e coins compared to $C_{P,i}$. Subsequently P 's state transitions to WAITING FOR PAY REVOCATION and sends signatures for $C_{\bar{P},i+1}$ and $R_{\bar{P},i}$ to \bar{P} . If the o_F output is spent while P is in the latter state, it can be spent by one of $C_{P,i+1}$ or $(C_{\bar{P},j})_{0 \leq j \leq i+1}$. If it is spent by $C_{P,i+1}$ or $C_{\bar{P},i+1}$, then these two transactions have a $(c_P + e, pk_{P,\text{out}})$ output. (Note that the former is encumbered with a timelock, but the alternative spending method cannot be used as P has not signed $R_{\bar{P},i+1}$.) If it is spent by $C_{\bar{P},i}$ then a $(c_P, pk_{P,\text{out}})$ output becomes available instead, therefore P can still get the c_P coins that correspond to the previous state. If any of $(C_{\bar{P},j})_{0 \leq j < i}$ spends o_F then it makes available a $pk_{P,\text{out}}$ output with the coins that P had at state j and additionally P can publish $R_{P,j}$ that spends \bar{P} 's output of $C_{\bar{P},j}$ and obtain the entirety of \bar{P} 's coins at state j for a total of $c_P + c_{\bar{P}}$ coins. Therefore in every case

P can claim at least c_P coins. In the case that P instead subsequently receives a valid signature to $R_{P,i}$ (Fig. 43, 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. 54, l. 4). If P is the fundee, it requests that the virtual channel be closed with the current honest coin balance (Fig. 53, 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. \square

Lemma 4 (Ideal world balance). Consider an ideal world execution with functionality $\mathcal{G}_{\text{Chan}}$ and simulator \mathcal{S} . Let $P \in \{\text{Alice}, \text{Bob}\}$ one of the two parties of $\mathcal{G}_{\text{Chan}}$. Assume that all of the following are true:

- $\text{State}_P \neq \text{IGNORED}$,
- P has transitioned to the OPEN State at least once. Additionally, if $P = \text{Alice}$, it has received (OPEN, c, \dots) by \mathcal{E} prior to transitioning to the OPEN State,
- P [has received (FUND ME, f, \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 $P = \text{Alice}$, or $\phi = 0$ if $P = \text{Bob}$. If $\mathcal{G}_{\text{Chan}}$ receives (FORCECLOSE, P) by \mathcal{S} , then the following

holds with overwhelming probability on the security parameter:

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

Proof: [Proof of Lemma 4] We will prove the Lemma by following the evolution of the balance_P variable.

- When $\mathcal{G}_{\text{Chan}}$ is activated for the first time, it sets $\text{balance}_P \leftarrow 0$ (Fig. 16, l. 1).
- If $P = \text{Alice}$ and it receives (OPEN, c, \dots) by \mathcal{E} , it stores c (Fig. 16, l. 11). If later State_P becomes OPEN, $\mathcal{G}_{\text{Chan}}$ sets $\text{balance}_P \leftarrow c$ (Fig. 16, ll. 14 or 34). In contrast, if $P = \text{Bob}$, it is $\text{balance}_P = 0$ until at least the first transition of State_P to OPEN (Fig. 16).
- Every time that P receives input $(\text{FUND ME}, f_i, \dots)$ by another party while $\text{State}_P = \text{OPEN}$, P stores f_i (Fig. 18, l. 1). The next time State_P transitions to OPEN (if such a transition happens), balance_P is decremented by f_i (Fig. 18, l. 27). Therefore, if this cycle happens $n \geq 0$ times, balance_P will be decremented by $\sum_{i=1}^n f_i$ in total.
- Every time $\mathcal{G}_{\text{Ledger}}$ receives $(\text{COOP CLOSING}, P, r_i)$ by \mathcal{S} while State_P is OPEN, r_i is stored (Fig. 20, l. 1). The next time State_P transitions to OPEN (if such a transition happens), balance_P is incremented by r_i (Fig. 20, l. 9). Therefore, if this cycle happens $k \geq 0$ times, 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 $\text{State}_P = \text{OPEN}$, d_i is stored (Fig. 17, l. 2). The next time State_P transitions to OPEN (if such a transition happens), balance_P is decremented by d_i (Fig. 17, l. 13). Therefore, if this cycle happens $m \geq 0$ times, balance_P will be decremented by $\sum_{i=1}^m d_i$ in total.
- Every time P receives input $(\text{GET PAID}, e_i)$ by \mathcal{E} while $\text{State}_P = \text{OPEN}$, e_i is stored (Fig. 17, l. 7). The next time State_P transitions to OPEN (if such a transition happens) balance_P is incremented by e_i (Fig. 17, l. 19). Therefore, if this cycle happens $l \geq 0$ times, 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 $(\text{FORCECLOSE}, P)$ by \mathcal{S} , it is $\text{balance}_P = c - \sum_{i=1}^n f_i - \sum_{i=1}^m d_i + \sum_{i=1}^l e_i + \sum_{i=1}^k r_i$ if $P = \text{Alice}$, or else if $P = \text{Bob}$, $\text{balance}_P = - \sum_{i=1}^n f_i - \sum_{i=1}^m d_i + \sum_{i=1}^l e_i + \sum_{i=1}^k r_i$. \square

Lemma 5 (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. 19 is executed negligibly often).

Proof: [Proof of Lemma 5] We prove the Lemma in two steps. We first show that if the conditions of Lemma 4 hold, then the conditions of Lemma 3 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 host_P must output (ENABLER USED REVOCATION) to $\mathcal{G}_{\text{Chan}}$ (Fig. 16, l. 5). The first case only happens when either P receives (CORRUPT) by \mathcal{A} (Fig. 30, l. 1), which means that the simulated P is not honest anymore, or when P becomes negligent (Fig. 30, l. 4), which means that the first condition of Lemma 3 is violated. In the second case, it is $\text{host}_P \neq \mathcal{G}_{\text{Ledger}}$ and the state of host_P is GUEST PUNISHED (Fig. 71, ll. 1 or 12), so in case P receives (FORCECLOSE) by \mathcal{E} the output of host_P will be (GUEST PUNISHED) (Fig. 68, l. 4). In all cases, some condition of Lemma 3 is violated.

For State_P to become OPEN at least once, the following sequence of events must take place (Fig. 16): If $P = \text{Alice}$, it must receive (INIT, pk) by \mathcal{E} when $\text{State}_P = \text{UNINIT}$, then either receive ($\text{OPEN}, c, \mathcal{G}_{\text{Ledger}}, \dots$) by \mathcal{E} and (BASE OPEN) by \mathcal{S} or ($\text{OPEN}, c, \text{hops} (\neq \mathcal{G}_{\text{Ledger}}), \dots$) by \mathcal{E} , (FUNDED, HOST, \dots) by $\text{hops}[0].\text{left}$ 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. 30, l. 10), therefore if the second condition of Lemma 4 holds and $P = \text{Alice}$, then the second condition of Lemma 3 holds as well. The same line of reasoning can be used to deduce that if $P = \text{Bob}$, 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 4 holds only if the second condition of Lemma 3 holds as well. We also observe that, if both parties are honest, they will transition to the OPEN state simultaneously.

Regarding the third Lemma 4 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 $(\text{FUND ME}, f, \dots)$ 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 $\text{State}_{\bar{P}} = \text{IGNORED}$, then State_P transitions directly back to OPEN. If on the other hand $\text{State}_{\bar{P}} = \text{OPEN}$ and $\mathcal{G}_{\text{Chan}}$ intercepts a similar VIRT ITI definition command through \bar{P} , $\text{State}_{\bar{P}}$ transitions to TENTATIVE HELP FUND. On receiving the aforementioned (FUND) message by \mathcal{S} and given that $\text{State}_{\bar{P}} = \text{TENTATIVE HELP FUND}$, $\mathcal{G}_{\text{Chan}}$ also sets $\text{State}_{\bar{P}}$ to SYNC HELP FUND. Then both State_P and $\text{State}_{\bar{P}}$ transition simultaneously to OPEN (Fig. 18). 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. 30, l. 18 and Fig. 40, l. 12), so the third condition of Lemma 3 holds with the same n as that of Lemma 4.

Moving on to the fourth Lemma 4 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 $\text{layer}_P = 0$ 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 3) plus k (from the fifth condition of Lemma 3) is equal to the k of Lemma 4.

Regarding the sixth Lemma 4 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 transitions 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 $\text{State}_{\bar{P}} = \text{IGNORED}$, then State_P transitions directly back to OPEN. If on the other hand $\text{State}_{\bar{P}} = \text{OPEN}$ and $\mathcal{G}_{\text{Chan}}$ receives (GET PAID, d) by \mathcal{E} addressed to \bar{P} , $\text{State}_{\bar{P}}$ transitions to TENTATIVE GET PAID. On receiving the aforementioned (PAY) message by \mathcal{S} and given that $\text{State}_{\bar{P}} = \text{TENTATIVE GET PAID}$, $\mathcal{G}_{\text{Chan}}$ also sets $\text{State}_{\bar{P}}$ to SYNC GET PAID. Then both State_P and $\text{State}_{\bar{P}}$ transition simultaneously to OPEN (Fig. 17). 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. 30, l. 16), so the sixth condition of Lemma 3 holds with the same m as that of Lemma 4. As far as the seventh condition of Lemma 4 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 4 holds with some l , then Lemma 3 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. 19, l. 18), we deduce that if the conditions of Lemma 4 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 4 do not hold, then the check of Fig. 19, l. 18 never takes place. We first discuss the $\text{State}_P = \text{IGNORED}$ 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 $\text{State}_P \neq \text{IGNORED}$ (Fig. 19, l. 9). We deduce that, once $\text{State}_P = \text{IGNORED}$, 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. 19 without first having been in the OPEN state. Moreover if $P = \text{Alice}$, it is impossible to reach the OPEN state without receiving input (OPEN, c , ...) by \mathcal{E} . Lastly, as we have observed already, the three last conditions of Lemma 4 are always satisfied. We conclude that if the conditions to Lemma 4 do not hold, then the check of Fig. 19, 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. \square

Theorem 6 (Simple Payment Channel Security). The protocol Π_{Chan}^1 UC-realises $\mathcal{G}_{\text{Chan}}^1$ in the presence of a global functionality $\mathcal{G}_{\text{Ledger}}$ and assuming the security of the underlying digital signature:

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

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

The corresponding proof stems from Lemma 5, the fact that $\mathcal{G}_{\text{Chan}}$ is a simple relay and that \mathcal{S} faithfully simulates Π_{Chan}^1 . Lastly we prove that $\forall n \geq 2$, Π_{Chan}^n UC-realises $\mathcal{G}_{\text{Chan}}^n$ in the presence of $\mathcal{G}_{\text{Chan}}^1, \dots, \mathcal{G}_{\text{Chan}}^{n-1}$ and $\mathcal{G}_{\text{Ledger}}$, assuming the security of the underlying digital signature (leveraging the relevant definition from [8]).

Theorem 7 (Recursive Virtual Payment Channel Security).

$\forall n \in \mathbb{N}^* \setminus \{1\}$, the protocol Π_{Chan}^n UC-realises $\mathcal{G}_{\text{Chan}}^n$ in the presence of $\mathcal{G}_{\text{Chan}}^1, \dots, \mathcal{G}_{\text{Chan}}^{n-1}$ and $\mathcal{G}_{\text{Ledger}}$, assuming the security of the underlying digital signature. Specifically,

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

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

Proof: [Proof of Theorem 6] By inspection of Figures 15 and 29 we can deduce that for a particular \mathcal{E} , in the ideal world execution $\text{EXEC}_{\mathcal{S}_A, \mathcal{E}}^{\mathcal{G}_{\text{Chan}}^1, \mathcal{G}_{\text{Ledger}}}$, \mathcal{S}_A simulates internally the two Π_{Chan}^1 parties exactly as they would execute in $\text{EXEC}_{\Pi_{\text{Chan}}^1, \mathcal{A}, \mathcal{E}}^{\mathcal{G}_{\text{Ledger}}}$, the real world execution, in case $\mathcal{G}_{\text{Chan}}^1$ does

not halt. Indeed, $\mathcal{G}_{\text{Chan}}^1$ only halts with negligible probability according to Lemma 5, therefore the two executions are computationally indistinguishable. \square

Proof: [Proof of Theorem 7] The proof is exactly the same as that of Theorem 6, replacing superscripts 1 for n . \square

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 $\mathcal{G}_{\text{Ledger}}, \mathcal{G}_{\text{Chan}}^1, \dots, \mathcal{G}_{\text{Chan}}^i$ 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 $\Pi_{\text{Chan}}^1, \dots, \Pi_{\text{Chan}}^i$ can be merged into Π^i .

Lemma 8. For all $i \in [n]$, Π^i UC-emulates \mathcal{G}^i .

Proof: [Proof of Lemma 8] 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). $\mathcal{G}_{\text{Chan}}^1$ and Π_{Chan}^1 are $\mathcal{G}_{\text{Ledger}}$ - and Π_{Ledger} -subroutine respecting respectively (Def. 2.3 [10]), as 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 $j \in [i] \setminus \{1\}$, $\mathcal{G}_{\text{Chan}}^j$ and Π_{Chan}^j are \mathcal{G}^{j-1} - and Π^{j-1} -subroutine respecting respectively, as 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. \square

Theorem 9 (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. 29–30) 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. 29. Thus Theorem 3.5 of [10], full replacement, applies inductively to \mathcal{G}^i and Π^i . \square

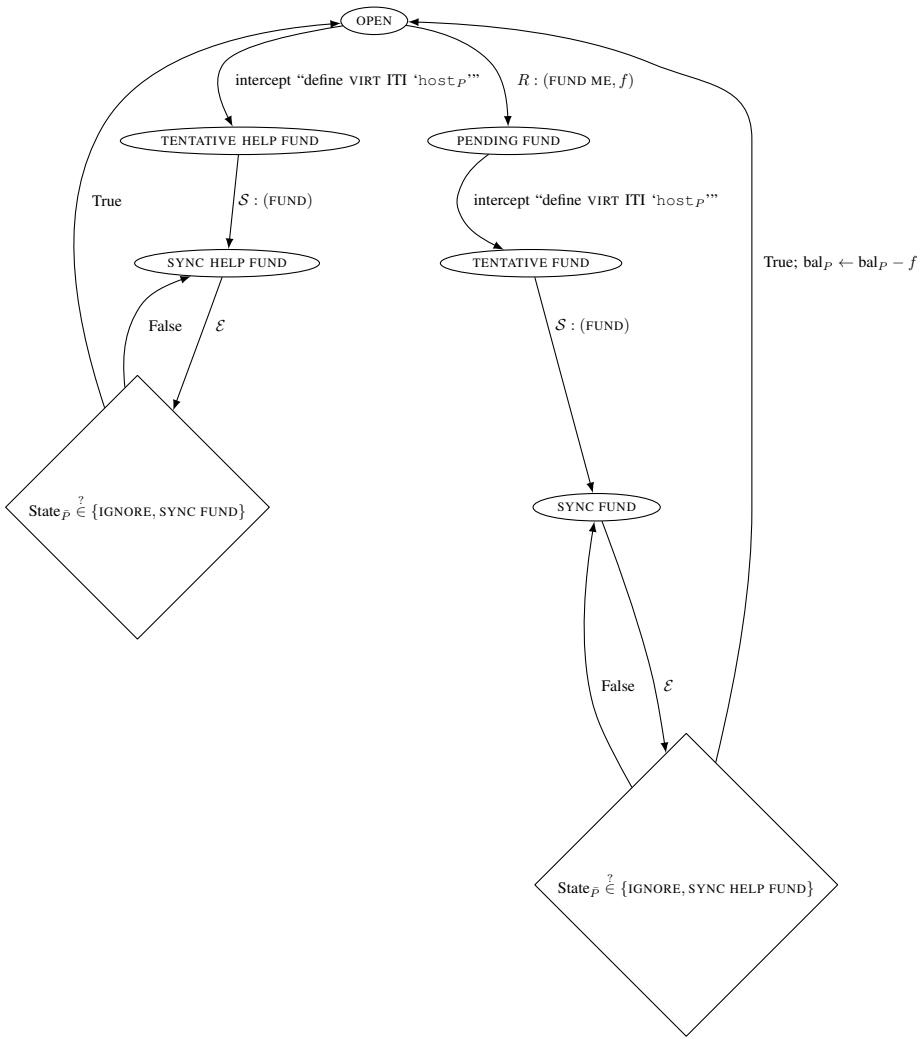


Figure 25: $\mathcal{G}_{\text{Chan}}$ state machine for funding new virtuals (both parties)

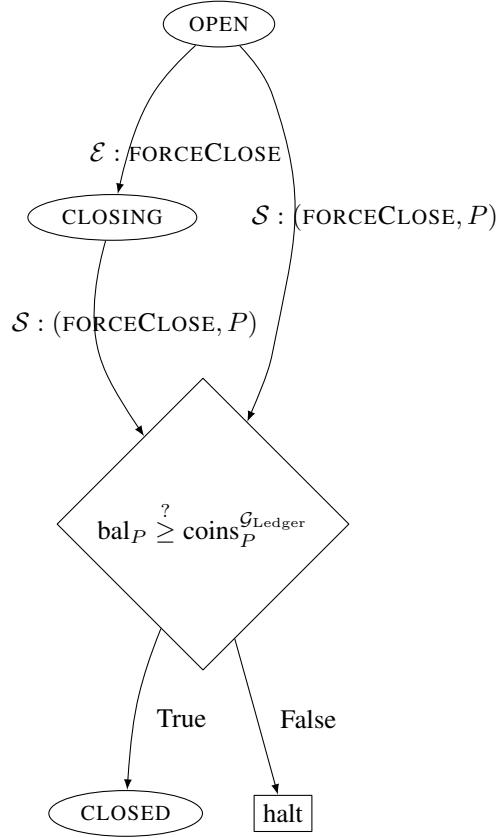


Figure 26: $\mathcal{G}_{\text{Chan}}$ state machine for channel closure (both parties)

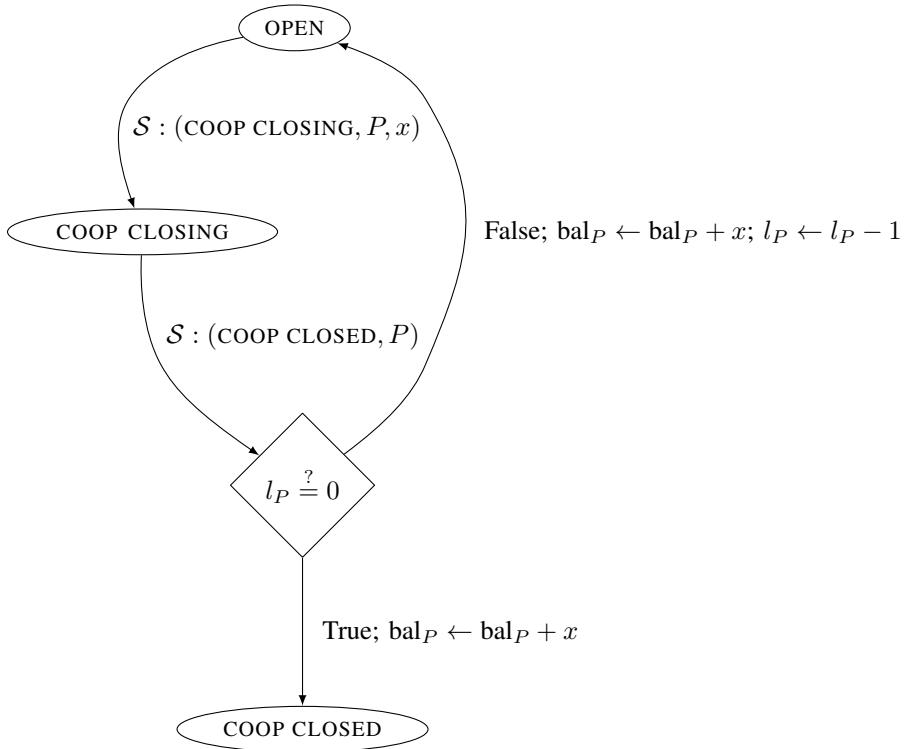


Figure 27: $\mathcal{G}_{\text{Chan}}$ state machine for cooperative channel closure (all parties)

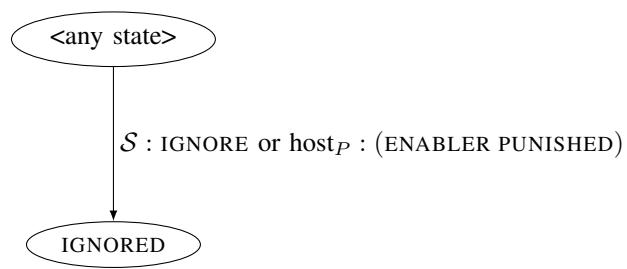


Figure 28: $\mathcal{G}_{\text{Chan}}$ state machine for corruption, negligence or punishment of the counterparty of a lower layer (both parties)