A Composable Security Treatment of the Lightning Network

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

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

Abstract. The high latency and low throughput of blockchain protocols constitute one of the fundamental barriers for their wider adoption. Overlay protocols, notably the *lightning network*, have been touted as the most viable direction for rectifying this in practice. In this work we present for the first time a full formalisation and security analysis of the lightning network in the (global) universal composition setting that leverages a global ledger functionality, for which realisability by the Bitcoin blockchain protocol has been demonstrated in previous work [Badertscher et al., Crypto'17]. As a result, our treatment delineates exactly how the security guarantees of the protocol depend on the properties of the underlying ledger and the frequent availability of the protocol participants. Moreover, we provide a complete and modular description of the core of the lightning protocol that highlights precisely its dependency to underlying basic cryptographic primitives such as digital signatures, pseudorandom functions, identity-based signatures and a less common two-party primitive, which we term a combined digital signature, that were originally hidden within the lightning protocol's implementation.

1 Introduction

Improving the latency of blockchain protocols, in the sense of the time it takes for a transaction to be "finalised", as well as their throughput, in the sense of the number of transactions they can handle per unit of time, are perhaps the two most crucial open questions in the area of permissionless distributed ledgers and remain fundamental barriers for their wider adoption in applications that require large scale and reasonably expedient transaction processing, cf. [1]. The Bitcoin blockchain protocol, introduced by Nakamoto [2], provides settlement with probability of error that drops exponentially in the number of blocks k that accumulate over a transaction of interest. This has been informally argued in the original white paper, and further formally demonstrated in [3], from where it can be inferred that the total delay in actual time for a transaction to settle is linear in k in the worst case. These results were subsequently generalised to the setting of partial synchrony [4] and variable difficulty [5]. Interestingly, this latency "deficiency" is intrinsic to the blockchain approach (see below), i.e.,

latency's dependency on k is not a side-effect of the security analysis but rather a characteristic of the underlying protocol and the threat model it operates in.

Given the above state of affairs, one has to either change the underlying settlement protocol or devise some other mechanism that, in conjunction with the blockchain protocol, achieves high throughput and low latency. A number of works proceeded with the first direction, e.g., hybrid consensus [6], Algorand [7]. A downside of this approach is that the resulting protocols fundamentally change the threat model within which Bitcoin is supposed to operate, e.g., by reducing the threshold of corrupted players, strengthening the underlying cryptographic assumptions or complicating the setup assumption required (e.g., from a public to a private setup).

The alternative approach is to build an *overlay* protocol that utilises the blockchain protocol as a "fall back" layer and does not relax the threat model in any way while it facilitates fast "off-chain" settlement under certain additional assumptions. We note that in light of the impossibility result regarding protocol "responsiveness" from [6] that states that no protocol can provide settlement in time proportional to actual network delay (i.e., fast settlement) and provide a security threshold over 1/3, we know that maintaining Bitcoin's threat model will require some additional assumption for the overlay protocol to offer fast settlement.

The first instance of this approach and by far the most widely known and utilised to date, came with the *lightning network* [8][3] that functions over the Bitcoin blockchain and leverages the concept of a bilateral payment channel. The latency for a transaction becomes linear to actual network delay and another factor that equals the number of bilateral payment channel hops in the path that connects the two end-points of the transaction. Implicated parties are guaranteed that, if they wish so, *eventually* the ledger will record a "gross" settlement transaction that reflects the balance resulting from all in-channel payments. Deviations from this guarantee are cryptographically feasible but deincentivised: a malicious party trying to commit to an outdated state will lose funds to a peer that provides evidence of a subsequent state. Moreover, note that no record of a specific payment transaction need ever appear on-chain thus the number of lightning transactions that can be exchanged can reach the maximum capacity the network allows between the parties, without being impeded by any restrictions of the underlying blockchain protocol.

The lightning network has been very influential in the space and spun a number of follow up research and implementations (see below for references). We note that the lightning network is not the only option for building an overlay over a blockchain. See e.g., [9] for an alternative approach focusing on reducing latency, where it is shown that if the assumption is raised to a security threshold of 3/4 plus the honesty of an additional special player, it is possible to obtain optimal

[3] The specification available online is a more descriptive reference for the inner workings of the protocol, see `https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md`. See also the raiden network that implements Lightning over Ethereum, `https://raiden.network`.

latency. Nevertheless, this approach does not offer the throughput benefits that are intrinsic to the lightning network.

Despite the importance of the lightning network for blockchain scalability there is still no work so far providing a thorough formal security analysis. This is a dire state of affairs given the fact that the protocol is actually currently operational[4] and its complexity makes it difficult to extract precise statements regarding the level of security it offers.

1.1 Our Results

We present the first, to our knowledge, complete security analysis of the lightning network, which we carry out in the universal composition (UC) setting. We model the payment overlay that the lightning network provides as a local ideal functionality and we demonstrate how it can be implemented in a hybrid world which assumes a global ledger functionality. Our treatment is general and does not assume any specific implementation for the underlying ledger functionality. The "paynet" functionality that we introduce abstracts all the salient security features achieved by the lightning network. We subsequently describe the whole lightning protocol in this setting and we prove that it realises our paynet functionality under standard cryptographic assumptions; the security guarantees of the functionality reflect specific environmental conditions regarding the availability of the honest parties to poll the status of the network. In more details our results are as follows.

1. We present the $\mathcal{F}_{\text{PayNet}}$ functionality which abstracts the syntax and security properties provided by the lightning network. We describe our $\mathcal{F}_{\text{PayNet}}$ assuming a global ledger functionality $\mathcal{G}_{\text{Ledger}}$ as defined in [10], and further refined in [11], which we know that is realised by the Bitcoin blockchain. Our approach not only captures lightning, but it is also general as it can be applied to any payment network by finely tuning the following parts of the functionality: the exact channel opening message sequence, the details of the on-chain checks performed by $\mathcal{F}_{\text{PayNet}}$, the negligence time bounds and the penalty in case of a malicious closure being caught. Using $\mathcal{F}_{\text{PayNet}}$, parties can open and close channels, forward payments along channel paths in the network as well as poll its status. Importantly, the functionality keeps track of all the off-chain and on-chain balances and ensures that when a channel closes, the on-chain balances are in line with the off-chain ones. In order to handle realistic adversarial deviations, $\mathcal{F}_{\text{PayNet}}$ allows the adversary to choose one of the following outcomes for each multi-hop payment: (i) let it go through as requested, (ii) charge it to an adversarial party along the path, (iii) charge it to a *negligent* honest party along the path. This last outcome is a crucial security characteristic of the lightning network: honest parties are required to poll the functionality with a frequency that corresponds to

[4] For current deployment statistics see e.g., `https://1ml.com/statistics`.

their level of involvement in the network and the properties of the underlying ledger. If a party does not poll often enough, $\mathcal{F}_{\text{PayNet}}$ identifies it as negligent and it may lose funds.

2. We identify for the first time the exact polling requirements that are imposed by the lightning network to the honest participating parties, i.e. how often parties have to check the state of the ledger functionality $\mathcal{G}_{\text{Ledger}}$ (over which the lightning network is overlaid) and how to act depending on what they see. These requirements are a function of the parameters of $\mathcal{G}_{\text{Ledger}}$ and ensure that honest parties do not lose funds. $\mathcal{G}_{\text{Ledger}}$ provides explicit security guarantees with respect to consistency and liveness which in turn impact the guarantees provided by $\mathcal{F}_{\text{PayNet}}$. The polling requirements for each party are two-fold: (i) the first type refers to monitoring for closures of channels of which the party is a member, and is specified by the parameter "delay" (chosen by the party), (ii) the second type refers to monitoring for specific events related to receiving and relaying payments. In detail, let *Alice* be an intermediary of a multi-hop payment. When the payment starts, she specifies two blockheights h, h'. Also, let a be the upper bound to the number of blocks that may be finalised in the ledger from the time a certain transaction is emitted to the time it becomes finalised (i.e. it is included in a block in the "stable" part of the ledger). *Alice* should then poll twice while her local view of the chain advances from blockheight h to blockheight $h' - a$. Moreover, the two pollings should be separated by a time window so that the chain can grow by at least a blocks.
3. We provide a complete pseudocode description of the lightning network protocol Π_{LN} and prove that it realises $\mathcal{F}_{\text{PayNet}}$ in the Random Oracle model. We identify a number of underlying cryptographic primitives that have been used in the lightning network in a non-black-box fashion and without reference. Interestingly, while most of these primitives are quite standard (a PRF, a Digital Signature scheme and an Identity Based Signature scheme), there is also one that is less standard and requires a new definition. The *combined digital signature* – as we will call it – is a special case of an asymmetric two-party digital signature primitive (e.g., see [12] and references therein) with the following characteristic: One of the two parties, called the shareholder, generates and stores a share of the signing key. The public key of the combined signature can be determined non-interactively from public information produced by both parties. Issuing signatures requires the availability of the share, which is verifiable given the public information provided by the shareholder. We formalise the combined digital signature primitive and show that the construction lying within the lightning specification realises it under standard cryptographic assumptions. In summary, the realisation of $\mathcal{F}_{\text{PayNet}}$ is achieved assuming the security of the underlying primitives, which in turn can be based on EC-DLOG and the Random Oracle model.
4. We prove that a more idealized ledger functionality, i.e. a ledger with instant finality, is unrealisable even assuming a synchronous multicast network. This result supports our decision to use the more realistic ledger functionality of [10], since it establishes that if our analysis was based on such a perfect

ledger, it would not be relevant for any real world deployment of a payment network since such software would – necessarily – depend on a non-perfect ledger. This choice also distinguishes our work compared to previous attempts [13,14,15,16] to formalize payment networks, as well as highlights the considerable latency improvement that the protocol offers in comparison to directly using the ledger.

1.2 Related Work

A first suggestion for building a unidirectional payment channel appeared in [17]. Bidirectional payment channels were developed in [18] and of course as part of the lightning network [8]. Subsequent work on the topic dealt with either improving payment networks by utilising more expressive blockchains such as Ethereum [16], hardware assumptions, see e.g., [19], or extending its functionality beyond payments, to smart contracts, [15] or finally enhancing their privacy, see e.g., [14,20,21]. Additional work looked into developing supporting protocols for the payment networks such as rebalancing [22] or finding routes in a decentralised fashion [23,24]. With respect to idealising the payment network functionality in the UC setting, a number of previous papers [13,14,15,16] presented ideal functionalities abstracting the concept, but they did not prove that the lightning network realises them. The main advantage of our approach however here is that, for the first time, we present a payment network functionality that interoperates with a global ledger functionality for which we know, in light of the results of [10], that is realisable by the Bitcoin blockchain and hence also reflects the actual parameters that can be enforced by the implementation and the exact participation conditions needed for security. In contrast, previous work [13,14,16] utilized "too idealised" ledger functionalities for their analysis which offer instant finality; as we prove in Theorem 3, a representative variant of these functionalities (Fig. 7) is unrealisable even under strong network assumptions (cf. Section 8). It is worth noting here that, were such a ledger realizable, layer-2 payment networks would not be as useful in practice because one of their two main motivations is the high latency of real blockchains. On the other hand, in [15] the ledger is not explicitly specified as a functionality, but only informally described. Several smart contracts are formally defined instead as UC ITMs, which are the entities with which protocols ultimately interact. The execution model of these contracts and their interaction with the blockchain is explained in an intuitive way, but a complete formalization of the ledger is missing. Lastly, the ledger used in [13] cannot be used directly by protocol parties, only accessed via higher-level functionalities. This limitation is imposed because otherwise any party could arbitrarily change the balances of other parties, given the definition of the functionality. This ledger is therefore a useful abstraction for higher-level protocols, but not amenable to direct usage, let alone concrete realisation.

1.3 Organisation

In Section 2 we present preliminaries for the model we employ and the relevant cryptographic primitives. In Section 3 we present an overview of the lightning network, accompanied by figures of the relevant transactions. Our payment network functionality is given an overview description in Section 4. Our abstraction of the core lightning protocol is provided in Section 5. We give more details about the combined digital signature primitive in Section 6. In Section 7 we provide an overview of the security proof of the main simulation theorem. Finally, in Section 8 we formalise our claim that a ledger functionality with instant finality is unrealisable. We refer the reader to the appendix of the full version [25] for the formal definition of the paynet functionality $\mathcal{F}_{\text{PayNet}}$ and the protocol Π_{LN}, along with the proof that the latter UC-realises the former. Our claim on the unrealisability of a perfect ledger is also proven there.

2 Preliminaries

In this section we give a brief overview of the tools and frameworks used in this work.

2.1 Universal Composability framework

In simulation-based security, cryptographic tasks are defined via an ideal "functionality" $\mathcal{F}$, which can be thought of as an uncorruptible entity that gets the inputs of all parties and returns the expected outputs while also interacting with the adversary in a prescribed manner. In this way, the functionality expresses the essence of a cryptographic task and its security features. A protocol Π realises the functionality $\mathcal{F}$ if for any real world adversary we can define a "simulator" $\mathcal{S}$, acting as an ideal world adversary, such that any environment $\mathcal{E}$ cannot distinguish between the real world and the ideal world executions. Albeit a powerful tool, simulation-based security only works when a single instance of the protocol is run in isolation. However, real-world systems almost always run several programs concurrently, which furthermore may run different instances of the same protocol. To facilitate this, the Universal Composability [26] framework allows us to analyse a single instance of the protocol and then take advantage of a generic composition theorem to infer the security of the protocol more broadly. This is achieved by allowing arbitrary interactions between the environment and the real-world adversary.

As mentioned, lightning network members have to periodically check the blockchain to ensure the security of their funds. However, the execution model of the UC framework allows $\mathcal{E}$ to impose extended periods of inactivity to any party. We opted to avoid the complication of using the clock functionality to force regular activation. Restricting the analysis only to environments that always cater for the needed activations would preclude the composability of our model. We instead allow $\mathcal{E}$ to deny activation to players (therefore becoming negligent) and provide security guarantees conditional on $\mathcal{E}$ permitting the necessary monitoring of the blockchain.

2.2 Hybrid functionalities used

Both our main protocol and the corresponding functionality use $\mathcal{G}_{\text{Ledger}}$ [10,11] as a hybrid. $\mathcal{G}_{\text{Ledger}}$ formalizes an ideal distributed append-only data structure akin to a blockchain. Any participating party can read from $\mathcal{G}_{\text{Ledger}}$, which returns an ordered list of transactions. Furthermore parties can submit new transactions which, if valid, will be added to the ledger and made visible to all parties at the discretion of the adversary, but necessarily within a predefined time window. This property is called liveness. Once a transaction is added to the ledger, it becomes visible to all parties at the discretion of the adversary, but within another predefined time window, and cannot be removed or reordered. This is called persistence. The exact definition can be found in the full version [25]. The current work makes heavy use of these two security properties, as the security of the lightning network relies crucially on the security of the underlying ledger.

Furthermore, $\mathcal{G}_{\text{Ledger}}$ needs the $\mathcal{G}_{\text{CLOCK}}$ functionality, which models the notion of time. Every participating party can request to read the current time (which is initialized to 0) and inform $\mathcal{G}_{\text{CLOCK}}$ that her round is over. $\mathcal{G}_{\text{CLOCK}}$ increments the time by one once all parties have declared the end of their round.

As already mentioned, the protocol and functionality defined in the current work do not make direct use of $\mathcal{G}_{\text{CLOCK}}$. Indeed, the only notion of time both in the lightning protocol and in our work is provided by the height of the blockchain, as reported respectively by the underlying Bitcoin node and $\mathcal{G}_{\text{Ledger}}$. We therefore omit it in the statement of Theorem 2 for simplicity of notation; it should normally appear as hybrid along with $\mathcal{G}_{\text{Ledger}}$. Its exact definition can be found in the full version [25]. We also note that $\mathcal{G}_{\text{Ledger}}$ and $\mathcal{G}_{\text{CLOCK}}$ are global functionalities [27] and therefore can be accessed directly by the environment, whereas $\mathcal{F}_{\text{PayNet}}$ is not.

We next provide the complete description of the ledger functionality as well as the clock and network functionalities that are drawn from the UC formalisation of [10,11].

The key characteristics of the functionality are as follows. The variable `state` maintains the current immutable state of the ledger. An honest, synchronised party considers finalised a prefix of `state` (specified by a pointer position $\texttt{pt}_i$ for party U_i below). The functionality has a parameter `windowSize` such that no finalised prefix of any player will be shorter than $|\texttt{state}| - \texttt{windowSize}$. On any input originating from an honest party the functionality will run the ExtendPolicy function that ensures that a suitable sequence of transactions will be "blockified" and added to `state`. Honest parties may also find themselves in a desynchronised state: this is when honest parties lose access to some of their resources. The resources that are necessary for proper ledger maintenance and that the functionality keeps track of are the global random oracle $\mathcal{G}_{\text{RO}}$, the clock $\mathcal{G}_{\text{CLOCK}}$ and network $\mathcal{F}_{\text{N-MC}}$. 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 $\texttt{maxTime}_{\texttt{window}}$, a number of blocks at least `windowSize` are added to `state`.
2. in a period of time equal to $\texttt{minTime}_{\texttt{window}}$, no more blocks may be added to `state` if `windowSize` blocks have been already added.
3. each window of `windowSize` blocks has at most $\texttt{advBlcks}_{\texttt{window}}$ adversarial blocks included in it.
4. any transaction that (i) is submitted by an honest party earlier than $\frac{\texttt{Delay}}{2}$ 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.

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

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

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

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

General: The functionality is parameterized by four algorithms, Validate, ExtendPolicy, Blockify, and predict-time, along with three parameters: $\texttt{windowSize}, \texttt{Delay} \in \mathbb{N}$, and $\mathcal{S}_{\text{initStake}} := \{(U_1, s_1), \ldots, (U_n, s_n)\}$. The functionality manages variables `state` (the immutable state of the ledger), `NxtBC` (a list of transaction identifiers to be added to the ledger), `buffer` (the set of pending transactions), τ_L (the rules under which the state is extended), and $\vec{\tau}_{\texttt{state}}$ (the time sequence where all immutable blocks where added). The variables are initialized as follows: $\texttt{state} := \vec{\tau}_{\texttt{state}} := \texttt{NxtBC} := \varepsilon$, $\texttt{buffer} := \emptyset$, $\tau_L = 0$. For each

party $U_p \in \mathcal{P}$ the functionality maintains a pointer $\mathtt{pt}_i$ (initially set to 1) and a current-state view $\mathtt{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\vec{\mathcal{I}}_H^T$ (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 $\mathcal{P}_{DS} \subset \mathcal{H}$ (as discussed below). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a (currently unregistered) honest party is registered at the ledger, *if it is registered with the clock and the global RO already,* then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$.

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

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

1. Updating synchronized/desynchronized party set:
   (a) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) to the ledger, the clock, and the GRO since time $\tau' < \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
   (b) For any synchronized party $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if U_p is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{U_p\}$.
2. If I was received from an honest party $U_p \in \mathcal{P}$:
   (a) Set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T || (I, U_p, \tau_L)$;
   (b) Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \vec{\tau}_{\mathtt{state}})$ and if $\vec{N} \neq \varepsilon$ set $\mathtt{state} := \mathtt{state} || \mathsf{Blockify}(\vec{N}_1) || \ldots || \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathtt{state}} := \vec{\tau}_{\mathtt{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \ldots, || \tau_L$.
   (c) For each $\mathtt{BTX} \in \mathtt{buffer}$: if $\mathsf{Validate}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 0$ then delete $\mathtt{BTX}$ from $\mathtt{buffer}$. Also, reset $\mathtt{NxtBC} := \varepsilon$.
   (d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathtt{state}| - \mathtt{pt}_j > \mathtt{windowSize}$ or $\mathtt{pt}_j < |\mathtt{state}_j|$, then set $\mathtt{pt}_k := |\mathtt{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.
3. If the calling party U_p is *stalled or time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:
   - *Submitting a transaction:*
     If $I = (\text{SUBMIT}, \text{sid}, \mathtt{tx})$ and is received from a party $U_p \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party U_p) do the following
     (a) Choose a unique transaction ID txid and set $\mathtt{BTX} := (\mathtt{tx}, \text{txid}, \tau_L, U_p)$

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

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

- *Reading the state:*
  If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\mathtt{state}_p := \mathtt{state}|_{\min\{\mathtt{pt}_p, |\mathtt{state}|\}}$ and return $(\text{READ}, \text{sid}, \mathtt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\mathtt{state}, \mathtt{buffer}, \vec{\mathcal{I}}_H^T)$ 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) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

- *The adversary proposing the next block:*
  If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \ldots, \text{txid}_\ell))$ is sent from the adversary, update $\mathtt{NxtBC}$ as follows:
  (a) Set listOfTxid $\leftarrow \epsilon$
  (b) For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \mathtt{buffer}$ with ID $\text{txid} = \text{txid}_i$ then set $\text{listOfTxid} := \text{listOfTxid}||\text{txid}_i$.
  (c) Finally, set $\mathtt{NxtBC} := \mathtt{NxtBC}||(\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, \mathit{ok})$ to $\mathcal{A}$.

- *The adversary setting state-slackness:*
  If $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\mathtt{pt}}_{i_1}), \ldots, (U_{i_\ell}, \widehat{\mathtt{pt}}_{i_\ell}))$, with $\{U_{p_{i_1}}, \ldots, U_{p_{i_\ell}}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
  (a) If for all $j \in [\ell] : |\mathtt{state}| - \widehat{\mathtt{pt}}_{i_j} \leq \mathtt{windowSize}$ and $\widehat{\mathtt{pt}}_{i_j} \geq |\mathtt{state}_{i_j}|$, set $\mathtt{pt}_{i_1} := \widehat{\mathtt{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, \mathit{ok})$ to $\mathcal{A}$.
  (b) Otherwise set $\mathtt{pt}_{i_j} := |\mathtt{state}|$ for all $j \in [\ell]$.

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

Functionality $\mathcal{F}_{\text{N-MC}}^{\Delta}$

The functionality is parameterized with a set possible senders and receivers $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

– **Honest sender multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $U_p \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose n new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$, initialize $2n$ new variables $D_{\text{mid}_1} := D_{\text{mid}_1}^{MAX} \ldots := D_{\text{mid}_n} := D_{\text{mid}_n}^{MAX} := 1$, set $\vec{M} := \vec{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{MULTICAST}, \text{sid}, m, U_p, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
– **Adversarial sender (partial) multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell}))$ from the adversary with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}$, choose ℓ new unique message-IDs $\text{mid}_{i_1}, \ldots, \text{mid}_{i_\ell}$, initialize ℓ new variables $D_{\text{mid}_{i_1}} := D_{\text{mid}_{i_1}}^{MAX} := \ldots := D_{\text{mid}_{i_\ell}} := D_{\text{mid}_{i_\ell}}^{MAX} := 1$, set $\vec{M} := \vec{M}||(m_{i_1}, \text{mid}_{i_1}, D_{\text{mid}_{i_1}}, U_{i_1})|| \ldots ||(m_{i_\ell}, \text{mid}_{i_\ell}, D_{\text{mid}_{i_\ell}}, U_{i_\ell})$, and send $(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}, \text{mid}_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell}, \text{mid}_{i_\ell}))$ to the adversary.
– **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $U_p \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of U_p if U_p is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, U_p) \in \vec{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\vec{M}_0^{U_p}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, U_p)$ with $D_{\text{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0^{U_p}$ from $\vec{M}$, and send $\vec{M}_0^{U_p}$ to U_p.
– **Adding adversarial delays.** Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$:
  If $D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}} \leq \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\text{mid}_{i_j}} := D_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$ and set $D_{\text{mid}_{i_j}}^{MAX} := D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}}$; otherwise, ignore this pair.

– **Adversarially reordering messages.** Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from the adversary, if mid and mid′ are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

2.3 Transaction structure

$\mathcal{G}_{\text{Ledger}}$ does not define what is a valid transaction, but leaves it as a system parameter. Importantly, no notion of coins is built in $\mathcal{G}_{\text{Ledger}}$. We therefore specify a valid transaction, closely following concepts put forth in Bitcoin [2], but avoiding specifying the entire Bitcoin script.

At a high level, every transaction consists of inputs and outputs. Each output has an associated value in coins and a number of "spending methods". A spending method specifies the exact requirements for spending the output. Each input must be connected to exactly one output and satisfy one of its spending methods.

Transactions in $\mathcal{G}_{\text{Ledger}}$ form a DAG. A new transaction is valid only if each of its inputs correctly spends an output with no other connected input and the sum of the values of its outputs does not exceed the sum of the values of the outputs connected to its inputs. We refer the reader to the full version [25] for a complete overview.

2.4 Cryptographic Primitives

In the Lightning Network specification, a custom scheme for deriving keys is used. Its syntax and security aims closely match those of previously studied Identity Based Signature schemes [28,29], thus we use the latter to abstract away the complexity of the construction and highlight the security requirements it satisfies. We slightly modify previous IBS schemes by adding an algorithm that, on input of the public parameters mpk and a label l, returns the verification key pk_l. Such an IBS scheme provides 5 algorithms:

– $(mpk, msk) \leftarrow \text{SETUP}(1^k)$: master keypair generation
– $(pk_l, sk_l) \leftarrow \text{KEYDER}(mpk, msk, l)$: keypair derivation with label l
– $pk_l \leftarrow \text{PUBKEYDER}(mpk, l)$: verification key derivation with label l
– $\sigma \leftarrow \text{SIGNIBS}(m, sk_l)$: signature generation with signing key sk_l
– $\{0,1\} \leftarrow \text{VERIFYIBS}(\sigma, m, pk_l)$: signature verification

We refer the reader to [29] for more details. Other cryptographic primitives used are digital signatures and pseudorandom functions. Finally, a less common two-party cryptographic primitive is employed that we formalise as *combined digital signatures*, see Section 6.

3 Lightning Network overview

Two-party channels. The aim of LN is to enable fast, cheap, off-chain transactions, without compromising security. Specifically no trust between counterparties is needed. This is achieved as follows: Two parties that plan to have

recurring monetary exchanges lock up some funds with one special on-chain transaction. We say that they opened a new channel. They can then transact with the locked funds multiple times solely by interacting privately, without informing the blockchain. If they want to use their funds in the usual, on-chain way again, they have to close the channel and unlock the funds with one more on-chain transaction. Therefore the number of on-chain transactions implicated in a channel is constant in the number of off-chain payments. Furthermore, each party can unilaterally close the channel and retrieve the coins they are entitled to – according to the latest channel state – and thus neither party has to trust the other.

In more detail, to open a channel *Alice* and *Bob* first exchange a number of keys and desired timelocks $\text{relDel}_A, \text{relDel}_B$ (explained below) and then build locally some transactions; no transaction is published yet. The "funding transaction" F contains a 2-of-2 multisig output with one "funding" public key $pk_{F,A}, pk_{F,B}$ for each counterparty. This multisig output needs signatures for both designated public keys in order to be spent. F is funded with c_F coins that belong only to one of the two parties, say *Alice*.

Fig. 1. Funding TX (on-chain): Rules over, coins below output.

Each party also builds a slightly different version of the "commitment transaction", C_A (Figure 3) for *Alice*and C_B (Figure 3) for *Bob*. *Alice* uses her "delayed payment" key $pk_{\text{dcom},A}$ and *Bob*'s "revocation" key $pk_{\text{rev},B}$ (received before), whereas *Bob* uses *Alice*'s "payment" key $pk_{\text{com},A}$ (received before). Both C_A and C_B spend the funding output of F and allow *Alice* to retrieve her funds if she acts honestly, as we will explain shortly. *Alice* sends *Bob* the signature of C_B made with her $sk_{F,A}$ and vice-versa. Both parties verify that the received signature is valid.

Fig. 2. *Alice*'s initial commitment TX (off-chain): Required data over input, spent output below input.

Fig. 3. *Bob*'s initial commitment TX (off-chain): All coins belong to *Alice*, so she can immediately spend them if *Bob* closes.

Observe that *Alice* can now safely publish the funding transaction F without fear of losing her coins: the only possible ways for it to be spent are either via C_A

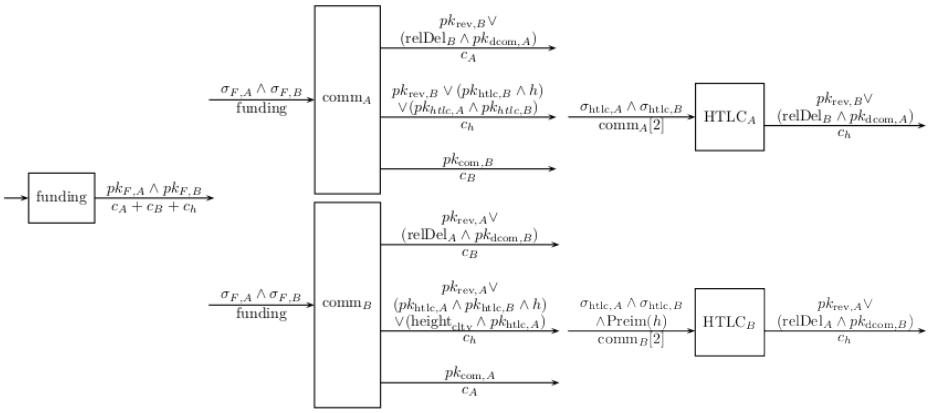

Fig. 4. All transactions of an open channel with an HTLC in flight. *Alice* owns c_A coins, *Bob* c_B coins, and c_h coins will be transferred to *Bob* if he discloses the preimage of h until the ledger has $\text{height}_{\text{cltv}}$ blocks, otherwise they will return to *Alice*. "funding" is in the ledger, *Alice* keeps locally "comm_A" and "HTLC_A", and *Bob* keeps "comm_B" and "HTLC_B".

or C_B, both of which transfer her funding coins back to a key she owns. She now broadcasts the funding transaction F; once both parties see that it is confirmed, they generate and exchange new "commitment" keys (used for updating the channel later) and the channel is open.

Every time they want to make a payment to each other, they exchange a series of messages that have two effects. First, a new pair of commitment transactions, along with their signatures by the funding keys, is created, one for each counterparty. The outputs of the new commitment transactions use new keys and reflect the new channel balance. Each of these transactions ensures that, if broadcast, each party will be able to spend the appropriate share from the coins contained in the funding output. Second, the two old commitment transactions are revoked. This ensures that no party can close a channel using an old commitment transaction, as such an old commitment transaction reflects a superseded channel state, which may be more beneficial than the latest state for one of the two parties.

Invalidating past commitments requires some care. The reason is that it is impossible to actually make past commitments invalid without spending the funding output on-chain; doing this for every update would however defeat the purpose of LN. The following idea is leveraged instead: If *Alice* broadcasts an old commitment and *Bob* sees it, he can punish *Alice* by taking all the money in the channel. Therefore *Alice* is technically able to broadcast an old commitment, but has no financial benefit in doing so. The same reasoning holds if *Bob* broadcasts an old commitment. On the donwside this imposes the requirement that parties must observe the blockchain periodically — see below the explanation

of timelocks and how they facilitate a time window within which parties should react.

The punishing mechanism operates as follows. Suppose *Alice* considers posting one of her old local commitments which has an output that carries her old share of the funds (c.f. Fig. 3). This output can be spent in two ways: either with a signature by *Alice*'s "delayed payment" secret key $sk_{\mathrm{dcom},A}$ which is a usual ECDSA key, or with a signature by *Bob*'s "revocation" secret key $sk_{\mathrm{rev},B}$, which is also an ECDSA key, but with an additional characteristic that we will explain soon. Thus, if *Alice* broadcasts an old commitment, *Bob* will be able to obtain her funds by spending her output using his "revocation" key. This privilege of course opens the possibility for *Bob* to abuse it (in particular, when a channel is closed — see below — *Bob* may steal *Alice*'s funds by using such a revocation key) and hence this side effect should also be carefully mitigated. The mitigation has the following form. At the time of creation of a new commitment, both parties will know *Bob*'s "revocation" public key $pk_{\mathrm{rev},B}$, but no party knows its corresponding secret – the key can only be computed by combining one secret value $sk_{\mathrm{com},n,A}$ that *Alice* knows and one secret value $sk_{\mathrm{com},n,B}$ that *Bob* knows. *Alice* therefore can prevent *Bob* from using his revocation key until she sends him $sk_{\mathrm{com},n,A}$. Therefore, *Alice* will send $sk_{\mathrm{com},n,A}$ to *Bob* only after the new commitment transaction is built and signed. As a result, all old commitment transactions are revoked, only the latest one is not.Thus *Bob* cannot abuse his revocation key on a commitment before this transaction is revoked. We note that the underlying cryptographic mechanism that enables such "revocation keys" is not straightforward and, as part of our contributions, we formalise it as a new two-party cryptographic primitive. We call it "combined signature" and we prove in Section 6 that the construction hidden in the LN implementation realises it in the random oracle model under the assumption that the underlying digital signature scheme is secure in the full version [25].

The last element needed to make channel updates secure is the already mentioned "relative timelock". If *Alice* broadcasts a commitment transaction, she is not allowed to immediately spend her funds with her "delayed payment" key. Instead, she has to wait for the transaction to reach a pre-agreed block depth (the relative timelock, negotiated during the opening of the channel and hardcoded in the output script of the commitment transaction) in order to give some time to *Bob* to see the transaction and, if it does not correspond to the latest version of the channel, punish her with his "revocation" key. This avoids a scenario in which *Alice* broadcasts an old commitment transaction and immediately spends her output, which would prevent *Bob* from ever proving that this commitment was old.

Lastly, if *Alice* wants to unilaterally close a channel, all she has to do is broadcast her latest local commitment (the only one not revoked) and any outstanding HTLC transactions (explained below) and wait for the timelock to expire in order to spend her funds. The LN specification further allows for cooperative channel closure, achieved by negotiating and broadcasting the "closing transaction"

which is not encumbered with a timelock, providing immediate availability of funds.

As we mentioned previously, timelocks provide specific time windows within which both parties have to check the blockchain in order to punish a misbehaving counterparty who broadcasts an old commitment transaction. This means that parties have to be regularly online to safeguard against theft. Furthermore, LN makes it possible to trustlessly outsource this to so-called watchtowers, but this mechanism is not analyzed in the current work.

Multi-hop payments. Having funds locked down for exclusive use with a particular counterparty would be a serious limitation. LN goes beyond that by allowing multi-hop payments. In a situation where *Alice* has a channel with *Bob* and he has another channel with *Charlie*, it is possible for *Alice* to pay *Charlie* off-chain by leveraging *Bob*'s help. Remarkably, this can be achieved without any one party trusting any of the other two. One can think of *Alice* giving some "marked" money to *Bob*, who in turn either delivers it to *Charlie* or returns it to *Alice* – it is impossible for *Bob* to keep the money. It is also impossible for *Alice* and *Charlie* to make *Bob* pay for this transaction out of his pocket.

We will now give a brief overview of how this counterintuitive dynamic is made possible. *Alice* initiates the payment by asking *Charlie* to create a new hash for a payment of x coins. *Charlie* chooses a random secret, hashes it and sends the hash to *Alice*. *Alice* promises *Bob* to pay him x in their channel if he shows her the preimage of this particular hash within a specific time frame. *Bob* makes the same promise to *Charlie*: if *Charlie* tells *Bob* the preimage of the same hash within a specific time frame (shorter than the one *Bob* has agreed with *Alice*), *Bob* will pay him x in their common channel. *Charlie* then sends him the preimage (which is the secret he generated initially) and *Bob* agrees to update the channel to a new version where x is moved from him to *Charlie*. Similarly, *Bob* sends the preimage to *Alice* and once again *Alice* updates their channel to give *Bob* x coins. Therefore x coins were transmitted from *Alice* to *Charlie* and *Bob* did not gain or lose anything, he just increased his balance in the channel with *Alice* and reduced his balance by an equal amount in the channel with *Charlie*.

This type of promise where a preimage is exchanged for money is called Hashed Time Locked Contract (HTLC). It is enforceable on-chain in case the payer does not cooperatively update upon disclosure of the preimage, thus no trust is needed. It is realised as an additional output of the commitment transactions, which contains the specified hash and transfers its funds either to the party that should provide the preimage or to the other party after a timeout. A corresponding "HTLC transaction" that can spend this output is built by each party. In the previous example with *Alice*, *Bob* and *Charlie*, two HTLC transactions were signed and fulfilled in total for the payment to go through. Two updates happened in each channel: one to sign the HTLC and one to fulfill it. The time frames were chosen so that every intermediary has had the time to learn the preimage and give it to the previous party on the path. Figure 4 shows all transactions implicated in a channel that has an HTLC in flight.

In LN zero-hop payments are also carried out using HTLCs.

LN gives the possibility for intermediaries to charge a fee for their service, but such fees are not incorporated in the current analysis for the benefit of avoiding the added complexity and making it easier for the functionality to keep track of the correct balances. We note in passing that the "wormhole" attack described in [30] is captured by our model, as an adversary that controls two non-neighbouring nodes on a payment path can skip the intermediate nodes. Nevertheless, such an attack is inconsequential in our analysis given the lack of fees. Furthermore, LN leverages the Sphinx onion packet scheme [31] to increase the privacy of payments, but we do not formaly analyze the privacy of LN in this work – we just use it in our protocol description to syntactically match the message format used by LN.

4 Overview of $\mathcal{F}_{\text{PayNet}}$

One of our contributions is the specification of $\mathcal{F}_{\text{PayNet}}$, a local functionality that describes the functional and security guarantees given by an ideal payment network. Its definition can be found in the full version [25]. The central aim of $\mathcal{F}_{\text{PayNet}}$ is opening payment channels, keeping track of their state, updating them according to requested payments and closing them, as requested by honest players, all in a secure manner. In particular, the three main messages it can receive from *Alice* are (openChannel), (pay), (closeChannel) and (forceCloseChannel).

When $\mathcal{F}_{\text{PayNet}}$ receives (openChannel, *Alice*, *Bob*, x, *tid*) from *Alice*, it informs simulator $\mathcal{S}$ of the intention of environment $\mathcal{E}$ to create a channel between *Alice* and *Bob* where *Alice* owns x coins. When it receives (pay, *Bob*, $x, \overrightarrow{\texttt{path}}$, `receipt`) from *Alice*, it informs $\mathcal{S}$ that $\mathcal{E}$ asked to perform a multi-hop payment of x coins from *Alice* to *Bob* along the $\overrightarrow{\texttt{path}}$. In the same vein, when $\mathcal{F}_{\text{PayNet}}$ receives (closeChannel, `receipt`, *pchid*) or (forceCloseChannel, `receipt`, *pchid*) from *Alice* (for a cooperative or unilateral close respectively), it leaks to $\mathcal{S}$ the fact that $\mathcal{E}$ wants to close the relevant channel.

In order to provide security guarantees, there are various moments when $\mathcal{F}_{\text{PayNet}}$ verifies whether certain expected events have actually taken place and halts if these checks fail. Note that the protocol Π_{LN} (which realises $\mathcal{F}_{\text{PayNet}}$, c.f. Theorem 2) never halts, therefore all possible halts of $\mathcal{F}_{\text{PayNet}}$ correspond to specific security guarantees that Π_{LN} satisfies. A number of messages prompt $\mathcal{F}_{\text{PayNet}}$ to read from $\mathcal{G}_{\text{Ledger}}$ and perform these checks. In the actual implementations of LN these checks are done periodically by a polling daemon. Such checks are done by $\mathcal{F}_{\text{PayNet}}$ in the following cases:

– On receiving (poll) by *Alice*, $\mathcal{F}_{\text{PayNet}}$ asks $\mathcal{G}_{\text{Ledger}}$ for *Alice*'s latest state Σ_{Alice} and verifies that no bad events have happened. In particular, $\mathcal{F}_{\text{PayNet}}$ halts if any of *Alice*'s channels has been closed maliciously with a transaction at height h and, even though *Alice* has polled within $[h, h+\texttt{delay}(Alice)-1]$, she did not manage to punish the counterparty. If $\mathcal{F}_{\text{PayNet}}$ does not halt, it

leaks to $\mathcal{S}$ the polling details (including the identity of the poller and the state of the ledger in their view).

- $\mathcal{F}_{\text{PayNet}}$ expects $\mathcal{S}$ to send a (RESOLVEPAYS, `charged`) message that gives details on the outcome of one or more multi-hop payments that include the identity of the party that is charged. Moreover, for each resolved payment, the message includes two expiry values, expressed in absolute block height: `OutgoingCltvExpiry`, which is the highest block in which the charged party could claim money from the previous hop (closer to the payer) and `IncomingCltvExpiry`, which is the lowest block in which the charged party could claim money from the next hop (closer to the payee). $\mathcal{F}_{\text{PayNet}}$ checks that for each payment the charged party was one of the following: (a) the one that initiated the payment, (b) a malicious party or (c) an honest party that is negligent. The latter case happens when the honest party either:
  1. did not POLL in time to catch a malicious closure (similarly to the check performed when a POLL message is handled, as described above) or
  2. did not POLL twice while the block height in the view of the player was in $[\texttt{OutgoingCltvExpiry}, \texttt{IncomingCltvExpiry} - (2+r)\,\texttt{windowSize}]$ with a distance of at least $(2+r)\,\texttt{windowSize}$ between the two POLLs or
  3. did not enforce the retrieval of the funds lost as a result of this payment when the chain in her view had height $\texttt{IncomingCltvExpiry} - (2+r)\,\texttt{windowSize}$ with a FULFILLONCHAIN message, as discussed below.

  Note that $(2+r)\,\texttt{windowSize}$ is the maximum number of blocks an honest party needs to wait from the moment a valid transaction is submitted until it is added to the ledger state. $\mathcal{F}_{\text{PayNet}}$ also ensures that the two expiries (`OutgoingCltvExpiry` and `IncomingCltvExpiry`) have a distance of at least $\texttt{relayDelay}(\mathit{Alice}) + (2+r)\,\texttt{windowSize}$, otherwise it halts. In case the charged party was honest and non-negligent, $\mathcal{F}_{\text{PayNet}}$ halts. It also halts if a particular payment resulted in a channel update for which $\mathcal{S}$ did not inform $\mathcal{F}_{\text{PayNet}}$.
- $\mathcal{F}_{\text{PayNet}}$ executes the function $\texttt{checkClosed}(\Sigma_{\mathit{Alice}})$ every time it receives *Alice*'s ledger state. In this case, it checks that every channel that $\mathcal{E}$ has asked to be closed or $\mathcal{S}$ designated as closed indeed has a closing transaction that corresponds to its latest state in Σ_{Alice}. Enough time is given for that transaction to settle in Σ_{Alice}, but if that time passes and the channel is still open or it is closed to a wrong version and no opportunity for punishment was given, $\mathcal{F}_{\text{PayNet}}$ halts.

A number of messages that support the protocol progress are also handled:

- Every player has to send (REGISTER, delay, relayDelay) before participating in the network. This informs $\mathcal{F}_{\text{PayNet}}$ how often the player has to POLL. "delay" corresponds to the maximum time between POLLs so that malicious closures will be caught. "relayDelay" is useful when the player is an intermediary of a multi-hop payment. It roughly represents the size of the time window the player has to learn a preimage from the next and reveal it to the previous node. Subsequently $\mathcal{F}_{\text{PayNet}}$ asks $\mathcal{S}$ to create and send a public key

that will hold the player's funds. This public key is subsequently sent back to the player.
- To complete her registration, *Alice* has to send the (toppedUp) message. It lets $\mathcal{F}_{\text{PayNet}}$ know that the desired amount of initial funds have been transferred to *Alice*'s public key. $\mathcal{F}_{\text{PayNet}}$ reads *Alice*'s state on $\mathcal{G}_{\text{Ledger}}$ to retrieve this number and subsequently allows *Alice* to participate in the payment network after it updates her `onChainBalance`.
- When $\mathcal{F}_{\text{PayNet}}$ receives (checkForNew, *Alice*, *Bob*, *tid*) from *Alice*, it asks $\mathcal{G}_{\text{Ledger}}$ for *Alice*'s latest state Σ_{Alice} and looks for a funding transaction F in it. If one is found, $\mathcal{S}$ is asked to complete the opening procedure.
- (pushFulfill, *pchid*), (pushAdd, *pchid*) and (commit, *pchid*) all nudge $\mathcal{S}$ to carry on with the protocol that updates the state of a specific channel. $\mathcal{F}_{\text{PayNet}}$ simply forwards these messages to $\mathcal{S}$.
- (fulfillOnChain) prompts $\mathcal{S}$ to close channels in which the counterparty is not willing to pay, even though they have promised to do so upon disclosure of a particular preimage. This message is simply forwarded to $\mathcal{S}$, but $\mathcal{F}_{\text{PayNet}}$ takes a note that such a message was received and the current blockheight in the view of the calling party.

Last but not least, $\mathcal{E}$ sends (getNews) to obtain the latest changes regarding newly opened or closed channels, along with updates to the state of existing ones. Here we make an interesting observation: The most complex part of LN is arguably the negotiations that happen when a multi-hop payment takes place, due to the many channel updates needed; indeed, two complete channel updates for each hop are needed for a successful payment to go through. The fact that a proposal for an update can happen asynchronously with the commitment to this update, along with the fact that a single commitment may commit to many indiviual update proposals only adds to the complexity. It is therefore only natural to want $\mathcal{F}_{\text{PayNet}}$ to be unaware of these details. In order to disentangle the abstraction of $\mathcal{F}_{\text{PayNet}}$ from such minutiae, we allow the adversary full control of the updates that are reported back to $\mathcal{E}$ via $\mathcal{F}_{\text{PayNet}}$. Nevertheless, $\mathcal{F}_{\text{PayNet}}$ enforces that any reporting deviations induced by the adversary will be caught when a channel closes. This is quite intuitive: Consider a user of the payment network that does not understand its inner workings but can read $\mathcal{G}_{\text{Ledger}}$ and count her funds there. $\mathcal{F}_{\text{PayNet}}$ provides no guarantees regarding any specific interim reporting but the user is assured that in case she chooses to close the relevant channel, her state in $\mathcal{G}_{\text{Ledger}}$ will be consistent with all the payments that went through.

5 Lightning Protocol Π_{LN} Overview

In order to prove that software adhering to the LN specification fulfills the security guarantees given by $\mathcal{F}_{\text{PayNet}}$, a concrete protocol that implements LN in the UC model is needed. To that end we define the formal protocol Π_{LN}, an overview of which is given here.

For the rest of this section, we will assume that *Alice*, *Bob* and *Charlie* are interactive Turing machine instances (ITIs) [26] that honestly execute Π_{LN}. Similarly to the ideal world, the main functions of Π_{LN} are triggered when it receives (openChannel), (pay), (closeChannel) and (forceCloseChannel) from $\mathcal{E}$. These three messages along with (getNews) informally correspond to actions that a "human user" would instruct the system to perform. (register) and (toppedUp) are sent by $\mathcal{E}$ for player intialization. The rest of the messages sent from $\mathcal{E}$ prompt Π_{LN} to perform actions that a software implementation would spontaneously perform periodically. All messages sent between *Alice*, *Bob* and *Charlie* correspond to messages specified by LN. For clarity of exposition, we avoid mentioning the exact name and contents of every message. We refer the reader to the formal definition of Π_{LN} for further details [25].

Registration. Before *Alice* can use the network, $\mathcal{E}$ first has to send her a (register, delay, relayDelay) message. She generates her persistent key and sends it back to $\mathcal{E}$. The latter may choose to add some funds to this key and then send (toppedUp) to *Alice*, who checks her state in $\mathcal{G}_{\text{Ledger}}$ and records her on-chain balance.

Channel opening. When she receives (openChannel, *Alice*, *Bob*, x, *tid*) from $\mathcal{E}$, *Alice* initiates the message sequence needed to open a channel with *Bob*, funded by her with x coins. After following the steps described in Section 3, the funding transaction has been submitted to $\mathcal{G}_{\text{Ledger}}$. However the channel is not open yet.

At a later point $\mathcal{E}$ may send (checkForNew, *Alice*, *Bob*, *tid*) to *Alice*. She then checks if her state in $\mathcal{G}_{\text{Ledger}}$ contains the funding transaction with the temporary ID *tid* and in that case she exchanges new commitment keys with *Bob*, as per Section 3. The channel is now open. Both parties keep a note to give $\mathcal{E}$ a receipt of the new channel the next time they receive (getNews).

Channel closing. When sent by $\mathcal{E}$, the messages closeChannel and forceCloseChannel prompt *Alice* to close the channel cooperatively or unilaterally respectively, as explained in Section 3. In both cases she takes a note to notify $\mathcal{E}$ that the channel is closed when she receives (getNews).

Performing payments. We will now follow the exact steps needed for a multi-hop payment, filling in many details that we omitted from Section 3. When she receives (pay, *Charlie*, x, $\overrightarrow{\texttt{path}}$) from $\mathcal{E}$, *Alice* attempts to pay *Charlie* x coins, using the provided $\overrightarrow{\texttt{path}}$. Let us assume that the path is *Alice* – *Bob* – *Charlie*. *Alice* asks *Charlie* for an "invoice" with the HTLC hash, to which *Charlie* reacts by choosing a random preimage and sending back to *Alice* its hash. *Alice* then prepares a Sphinx [31] onion packet with the relevant information for each party on the $\overrightarrow{\texttt{path}}$ and sends it to *Bob*. *Bob* peels his layer of the onion and, after performing sanity checks and extracting the hash, he takes a note of this pending HTLC. He does not yet forward the onion to *Charlie*, because *Alice* is not yet committed to paying *Bob*. The latter happens if *Alice* subsequently receives (commit, $pchid_{AB}$) from $\mathcal{E}$, where $pchid_{AB}$ is the ID of the *Alice* – *Bob* channel. She then sends *Bob* all the signatures needed to make the new commitment transaction spendable, who replies with the secret commitment key of the old

commitment transaction (thus revoking it), along with the public commitment key of the future commitment transaction (to allow *Alice* to prepare the next update, when that happens). LN demands that before *Bob* forwards the onion to *Charlie*, he must commit as well to the HTLC to *Alice*. This happens if he receives the relevant COMMIT message from $\mathcal{E}$. Now that both parties have the HTLC in their commitment transaction and all past commitment transactions are revoked, they consider this HTLC "irrevocably committed".

Bob may then receive (PUSHADD, $pchid_{AB}$) from $\mathcal{E}$. *Bob* sends the onion to *Charlie*, who in turn peels it, recognizes that the payment is for him and that indeed he knows the preimage (since he generated it himself) and waits for the HTLC between him and *Bob* to be irrevocably committed. After both *Bob* and *Charlie* receive (COMMIT), *Charlie* awaits for a (PUSHFULFILL, $pchid_{BC}$) message from $\mathcal{E}$. If it arrives, *Charlie* sends the preimage to *Bob*, who sends it back to *Alice*. Once more every party has to receive a (COMMIT) message for each of the channels it participates in in order to remove the HTLC and update the definitive balance of each player to the appropriate value after the payment is complete. After this last update, each party keeps a note to inform $\mathcal{E}$ about the new balance when it receives (GETNEWS). *Alice* and *Charlie* also keep a note to inform $\mathcal{E}$ that the payment it had asked for succeeded.

Observe that while locked up in an HTLC, funds do not belong to either player; they are rather in a temporary, transitive state. If one party learns the preimage, the funds become theirs, whereas if it does not learn the preimage after some time, the other party is entitled to the funds. Also observe that within the UC framework the necessary messages COMMIT, PUSHFULFILL and PUSHADD may never arrive, but in a correct software implementation the corresponding actions happen automatically, without waiting for a prompt by the user.

Polling. Lastly, $\mathcal{E}$ may send (POLL) to *Alice*. She then reads her state in $\mathcal{G}_{\text{Ledger}}$ and checks for closed channels. If she finds maliciously closed channels (closed using old commitments), she punishes the counterparty and takes all the funds in the channel. If she finds in an honestly closed channel a preimage of an HTLC that she has previously signed and for which she is an intermediary, she records it and prepares to send it when she receives (PUSHFULFILL). Finally, if she finds an honestly closed channel with an HTLC output for which she knows the preimage, she spends it immediately. For every closed channel she finds, she keeps a note to report it to $\mathcal{E}$ the next time she receives (GETNEWS).

Remark 1 (Differences between LN and Π_{LN}). In LN, a custom construction for generating a new secret during each channel update is used. It reduces the space needed to maintain a channel from $O(n)$ to $O(\log n)$ in the number of updates. As far as we know, its security has not been formally analysed. In the current paper we use instead a PRF [32].

As mentioned earlier, LN uses a custom construction that takes advantage of elliptic curve homomorphic properties in order to derive any number of keypairs by combining a single "basepoint" with different "labels". We instead use Identity Based Signatures [28,29] (IBS) to abstract the properties provided by the

construction. We also prove in the full version [25] that it actually implements an IBS.

Additionally, we have chosen to simplify the protocol in a number of ways in order to keep the analysis tractable. In particular LN defines several additional messages that signal various types of errors in transmission. It also specifies exactly how message retransmission should happen upon reconnection, specifically for the case of connection failure while updating a channel. This allows for a more robust system by excluding many cases of accidental channel closures. What is more, an LN user can change their "delay" and "relayDelay" parameters even after registration, which is not the case in Π_{LN}.

Lastly, in order to incentivize users to act as intermediaries or check for channel closures on behalf of others, LN provides for fees for these two roles. Furthermore, in order to reduce transaction size and ensure that bitcoin nodes relay the transactions, it specifies exact rules for pruning outputs of too low value (known as "dust outputs"). In the current analysis we do not consider these features.

6 The Combined Signature primitive

As previously mentioned, we define a new primitive for combining keys and generating signatures, which is leveraged in the revocation and punishment mechanism of channel updates. Furthermore, we prove that the construction designed by the creators of LN realizes this primitive. We provide here the concrete syntax and correctness definitions, along with the intuition behind it, the exact security definitions, a concrete construction and proof of its security.

Previous work on the subject of multi-party signatures [12,33,34,35,36,37] focuses on use-cases where some parties desire to generate a signature without revealing their private information; the latter is created using an interactive protocol. The resulting signatures can be verified by a single verification key, which is also included in the output of the key generation protocol. As we will see however, the primitive defined here has different aims and limitations and, to our knowledge, has not been formalized yet.

A combined signature is a two-party primitive, say between *Alice* and *Bob*, with *Bob* being the signer and *Alice* the holder of a share of the secret key. This share is essential for issuing signatures, which in turn are verifiable with the "combined" verification key. The verification key is generated using public information drawn from *Alice* and *Bob* and is feasible without any party knowing the corresponding signing key. *Bob* will be able to construct the signing key only if *Alice* shares her secret information with him.

More specifically, the seven algorithms used by a Combined Signatures scheme are:

- $(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right)$
- $(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right)$
- $cpk_l \leftarrow \textsc{CombinePubKey}\left(mpk, pk\right)$

- $(cpk_l, csk_l) \leftarrow \textsc{CombineKey}\,(mpk, msk, pk, sk)$
- $\{0,1\} \leftarrow \textsc{TestKey}\,(pk, sk)$
- $\sigma \leftarrow \textsc{SignCS}\,(m, csk)$
- $\{0,1\} \leftarrow \textsc{VerifyCS}\,(\sigma, m, cpk)$

We demand that these three properties hold for a scheme to have correctness:

- $\forall k \in \mathbb{N}$,
  $\Pr[(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
  $\textsc{TestKey}(pk, sk) = 1] = 1$
  I.e. $\textsc{KeyShareGen}()$ must always generate a valid keypair.
- $\forall k \in \mathbb{N}$,
  $\Pr[(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right),$
  $(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
  $(cpk_1, csk_1) \leftarrow \textsc{CombineKey}\,(mpk, msk, pk, sk),$
  $cpk_2 \leftarrow \textsc{CombinePubKey}\,(mpk, pk),$
  $cpk_1 = cpk_2] = 1$
  I.e. for suitable input, $\textsc{CombinePubKey}()$ and $\textsc{CombineKey}()$ produce the same public key.
- $\forall k \in \mathbb{N}, m \in \mathcal{M}$,
  $\Pr[(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right),$
  $(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
  $(cpk, csk) \leftarrow \textsc{CombineKey}\,(mpk, msk, pk, sk),$
  $\textsc{VerifyCS}(\textsc{SignCS}(m, csk), m, cpk) = 1] = 1$
  I.e. for suitable input, honestly generated signatures always verify correctly.

Beyond correctness, combined signatures have two security properties expressed as follows. Share-EUF security expresses security from the point of view of *Alice*, and establishes that *Bob* cannot issue a valid combined signature if he does not possess *Alice*'s corresponding secret share. Formally:

Game share-EUF$^{\mathcal{A}}\left(1^k\right)$

1: $(\mathtt{aux}, mpk, n) \leftarrow \mathcal{A}\,(\textsc{init})$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad (pk_i, sk_i) \leftarrow \textsc{KeyShareGen}\left(1^k\right)$
4: **end for**
5: $(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{A}\,(\textsc{keys}, \mathtt{aux}, pk_1, \ldots, pk_n)$
6: **if** $pk^* \in \{pk_1, \ldots, pk_n\} \wedge cpk^* = \textsc{CombinePubKey}\,(mpk, pk^*) \wedge$
$\textsc{VerifyCS}\,(\sigma^*, m^*, cpk^*) = 1$ **then**
7: $\quad$ **return** 1
8: **else**
9: $\quad$ **return** 0
10: **end if**

Definition 1. *A Combined Signatures scheme is* share-EUF-secure *if*

$$\forall \mathcal{A} \in \mathsf{PPT}, \Pr\left[\mathsf{share\text{-}EUF}^{\mathcal{A}}\left(1^k\right) = 1\right] = \mathrm{negl}\left(k\right) \textit{ or equivalently}$$
$$\text{E-share}(k) = \mathrm{negl}\left(k\right) \quad ,$$
$$\textit{where } \text{E-share}(k) = \sup_{\mathcal{A} \in \mathsf{PPT}} \{\Pr[\mathsf{share\text{-}EUF}^{\mathcal{A}}\left(1^k\right) = 1]\}$$

On the other hand, master-EUF-CMA security is modeled very similarly to standard EUF-CMA security, with the difference that *Bob* (the signer) combines malicious shares into his public key and issues signatures with respect to such combined keys. The security property ensures that these signatures provide no advantage to the adversary in terms of producing a forged message for a combined key of its choice. Formally:

Game master-EUF-CMA$^{\mathcal{A}}\left(1^k\right)$

$(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right)$
$i \leftarrow 0$
$(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}\left(\textsc{init}, mpk\right)$
while response can be parsed as (pk, sk, m) **do**
    $i \leftarrow i + 1$
    store pk, sk, m as pk_i, sk_i, m_i
    $(cpk_i, csk_i) \leftarrow \textsc{CombineKey}\left(mpk, msk, pk_i, sk_i\right)$
    $\sigma_i \leftarrow \textsc{SignCS}\left(m_i, csk_i\right)$
    $(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}\left(\textsc{signature}, \mathtt{aux}_{i-1}, \sigma_i\right)$
end while
parse response as $(cpk^*, pk^*, m^*, \sigma^*)$
if $m^* \notin \{m_1, \ldots, m_i\} \wedge cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge$
$\textsc{VerifyCS}\left(\sigma^*, m^*, cpk^*\right) = 1$ **then**
    return 1
else
    return 0
end if

Definition 2. *A Combined Signatures scheme is* master-EUF-CMA-secure *if*

$$\forall \mathcal{A} \in \mathsf{PPT}, \Pr\left[\mathsf{master\text{-}EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = \mathrm{negl}\left(k\right) \textit{ or equivalently}$$
$$\text{E-master}(k) = \mathrm{negl}\left(k\right) \quad ,$$
$$\textit{where } \text{E-master}(k) = \sup_{\mathcal{A} \in \mathsf{PPT}} \{\Pr[\mathsf{master\text{-}EUF} - \mathsf{CMA}^{\mathcal{A}}\left(1^k\right) = 1]\}$$

Definition 3. *A Combined Signatures scheme is* combine-EUF-secure *if it is both share-EUF-secure and master-EUF-CMA-secure.*

In conclusion, a collection of algoritms is said to be a secure Combined Signatures scheme if it conforms to the syntax of the seven aforementioned algorithms, it satisfies the three correctness properties and provides existential unforgeability

against key-only attacks with respect to key shares and existential unforgeability against chosen message attacks with respect to master keys.

We here define the particular construction for Combined Signatures used in LN and prove its security.

Parameters: hash function $\mathcal{H}$, group generator G

MasterKeyGen(1^k):
  return KeyGen(1^k)

KeyShareGen(1^k, rand):
  return KeyGen(1^k)

CombinePubKey(mpk, pk):
  return $mpk \cdot \mathcal{H}(mpk \parallel pk) + pk \cdot \mathcal{H}(pk \parallel mpk)$

CombineKey(mpk, msk, pk, sk):
  return (CombinePubKey(mpk, pk), $msk \cdot \mathcal{H}(mpk \parallel pk) + sk \cdot \mathcal{H}(pk \parallel mpk)$)

TestKey(pk, sk):
  if $pk = textscPubKeyGen(sk)$ **then**
    return 1
  else
    return 0
  end if

SignCS(m, csk):
  return SignDS(m, csk)

VerifyCS(σ, m, cpk):
  return VerifyDS(σ, m, cpk)

One can check by inspection that the syntax above matches the one required by the Combined Signatures scheme definition. Furthermore, assuming that SignDS() and VerifyDS() are provided by a correct Digital Signature construction, it is straightforward to confirm that the construction here satisfies the Combined Signatures correctness properties.

We now move on to proving that the construction is also secure.

Lemma 1. *The construction defined above is share-EUF-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly EUF-CMA-secure and the range of the Random Oracle coincides with that of the underlying signature scheme signing keys.*

Proof. Let $k \in \mathbb{N}, \mathcal{B}$ PPT algorithm such that

$$\Pr\left[\mathsf{share\text{-}EUF}^{\mathcal{B}}\left(1^k\right) = 1\right] = a = \text{non} - \text{negl}(k) \quad .$$

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 5) such that

$$\Pr\left[\mathsf{EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = \text{non} - \text{negl}(k)$$

Algorithm $\mathcal{A}(vk)$

$j \xleftarrow{\$} U[1, T(\mathcal{B})]$ // $T(M)$ is the maximum running time of M
    Random Oracle: for every first-seen query q from $\mathcal{B}$ set $\mathcal{H}(q)$ to a random value
    return $\mathcal{H}(q)$ to $\mathcal{B}$
$(\mathsf{aux}, mpk, n) \leftarrow \mathcal{A}(\text{INIT})$
for $i \leftarrow 1$ to n **do**
    $(pk_i, sk_i) \leftarrow \text{KEYSHAREGEN}\left(1^k\right)$
end for
    Random Oracle: Let q be the jth first-seen query from $\mathcal{B}$:
    if $q = (mpk \,\|\, x)$ **then**
        if $\mathcal{H}(x \,\|\, mpk)$ unset **then**
            set $\mathcal{H}(x \,\|\, mpk)$ to a random value
        end if
        set $\mathcal{H}(mpk \,\|\, x)$ to $(vk - x \cdot \mathcal{H}(x \,\|\, mpk)) \cdot mpk^{-1}$
    else if $q = (x \,\|\, mpk)$ **then**
        if $\mathcal{H}(mpk \,\|\, x)$ unset **then**
            set $\mathcal{H}(mpk \,\|\, x)$ to a random value
        end if
        set $\mathcal{H}(x \,\|\, mpk)$ to $(vk - mpk \cdot \mathcal{H}(mpk \,\|\, x)) \cdot x^{-1}$
    else
        set $\mathcal{H}(q)$ to a random value
    end if
    return $\mathcal{H}(q)$ to $\mathcal{B}$
$(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\text{KEYS}, \mathsf{aux}, pk_1, \ldots, pk_n)$
if $vk = cpk^* \wedge \mathcal{B}$ wins the share-EUF game **then** // $\mathcal{A}$ won the EUF-CMA game
    return (m^*, σ^*)
else
    return FAIL
end if

Fig. 5.

that breaks the assumption, thus proving Lemma 1.

Let Y be the range of the random oracle. The modified random oracle used in Fig. 5 is indistinguishable from the standard random oracle by PPT algorithms since the statistical distance of the standard random oracle from the modified one is at most $\frac{1}{2|Y|} = \text{negl}(k)$ as they differ in at most one element.

Let E denote the event in which $\mathcal{B}$ does not invoke CombinePubKey to produce cpk^*. In that case the values $\mathcal{H}(pk^* \,\|\, mpk)$ and $\mathcal{H}(mpk \,\|\, pk^*)$ are decided after $\mathcal{B}$ terminates (Fig. 5, line 24) and thus

$$\begin{gathered}\Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) | E\right] = \frac{1}{|Y|} = \text{negl}(k) \Rightarrow \\ \Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge E\right] = \text{negl}(k) \quad .\end{gathered} \tag{1}$$

It is

$$\begin{gathered}(\mathcal{B} \text{ wins}) \rightarrow \left(cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right)\right) \Rightarrow \\ \Pr\left[\mathcal{B} \text{ wins}\right] \leq \Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right)\right] \Rightarrow \\ \Pr\left[\mathcal{B} \text{ wins} \wedge E\right] \leq \Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge E\right] \overset{(1)}{\Rightarrow} \\ \Pr\left[\mathcal{B} \text{ wins} \wedge E\right] = \text{negl}(k) \quad .\end{gathered}$$

But we know that $\Pr\left[\mathcal{B} \text{ wins}\right] = \Pr\left[\mathcal{B} \text{ wins} \wedge E\right] + \Pr\left[\mathcal{B} \text{ wins} \wedge \neg E\right]$ and $\Pr\left[\mathcal{B} \text{ wins}\right] = a$ by the assumption, thus

$$\Pr\left[\mathcal{B} \text{ wins} \wedge \neg E\right] > a - \text{negl}(k) \quad . \tag{2}$$

We now focus at the event $\neg E$. Let F the event in which the call of $\mathcal{B}$ to CombinePubKey to produce cpk^* results in the jth invocation of the Random Oracle. Since j is chosen uniformly at random and using Proposition 1 of [25], $\Pr\left[F|\neg E\right] = \frac{1}{T(\mathcal{B})}$. Observe that $\Pr\left[F|E\right] = 0 \Rightarrow \Pr\left[F\right] = \Pr\left[F|\neg E\right] = \frac{1}{T(\mathcal{B})}$.

In the case where the event $(F \wedge \mathcal{B} \text{ wins} \wedge \neg E)$ holds, it is

$$\begin{gathered}cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) = \\ mpk \cdot \mathcal{H}\left(mpk \,\|\, pk^*\right) + pk^* \cdot \mathcal{H}\left(pk^* \,\|\, mpk\right)\end{gathered}$$

Since F holds, the jth invocation of the Random Oracle queried either the value $\mathcal{H}(mpk \,\|\, pk^*)$ or $\mathcal{H}(pk^* \,\|\, mpk)$. In either case (Fig. 5, lines 9-18), it is $cpk^* = vk$. This means that $\textsc{VerifyCS}\left(\sigma^*, m^*, vk\right) = 1$. We conclude that in the event $(F \wedge \mathcal{B} \text{ wins} \wedge \neg E)$, $\mathcal{A}$ wins the EUF-CMA game. A final observation is that the probability that the events $(\mathcal{B} \text{ wins} \wedge \neg E)$ and F are almost independent, thus

$$\begin{gathered}\Pr\left[F \wedge \mathcal{B} \text{ wins} \wedge \neg E\right] = \Pr\left[F\right] \Pr\left[\mathcal{B} \text{ wins} \wedge \neg E\right] \pm \text{negl}(k) \overset{(2)}{=} \\ \frac{a - \text{negl}(k)}{T(\mathcal{B})} \pm \text{negl}(k) = \text{non} - \text{negl}(k)\end{gathered}$$

□

Lemma 2. *The construction above is master-EUF-CMA-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly EUF-CMA-secure and the range of the Random Oracle coincides with that of the underlying signature scheme signing keys.*

Proof. Let $k \in \mathbb{N}, \mathcal{B}$ PPT algorithm such that

$$\Pr\left[\mathsf{master\text{-}EUF\text{-}CMA}^{\mathcal{B}}\left(1^k\right) = 1\right] = a = \text{non} - \text{negl}\left(k\right) \quad .$$

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 6) such that

$$\Pr\left[\mathsf{EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = \text{non} - \text{negl}\left(k\right)$$

that breaks the assumption, thus proving Lemma 2.

The modified random oracle used in Fig. 6 is indistinguishable from the standard random oracle for the same reasons as in the proof of Lemma 1.

Let E denote the event in which CombinePubKey is not invoked to produce cpk^*. In that case the values $\mathcal{H}\left(pk^* \parallel mpk\right)$ and $\mathcal{H}\left(mpk \parallel pk^*\right)$ are decided after $\mathcal{B}$ terminates (Fig. 6, line 30) and thus

$$\begin{aligned} &\Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) | E\right] = \text{negl}\left(k\right) \Rightarrow \\ &\Pr\left[cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge E\right] = \text{negl}\left(k\right) \quad . \end{aligned} \tag{3}$$

We can reason like in the proof of Lemma 1 to deduce that

$$\Pr\left[\mathcal{B} \text{ wins} \wedge \neg E\right] > a - \text{negl}\left(k\right) \quad . \tag{4}$$

We now focus at the event $\neg E$. Let F the event in which the call of to CombinePubKey that produces cpk^* results in the jth invocation of the Random Oracle. Since j is chosen uniformly at random and using Proposition 1 of [25], $\Pr\left[F|\neg E\right] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$. Observe that $\Pr\left[F|E\right] = 0 \Rightarrow \Pr\left[F\right] = \Pr\left[F|\neg E\right] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$.

Once more we can reason in the same fashion as in the proof of Lemma 1 to deduce that

$$\begin{aligned} \Pr\left[F \wedge \mathcal{B} \text{ wins} \wedge \neg E\right] &= \Pr\left[F\right] \Pr\left[\mathcal{B} \text{ wins} \wedge \neg E\right] \pm \text{negl}\left(k\right) \overset{(4)}{=} \\ &\frac{a - \text{negl}\left(k\right)}{T\left(\mathcal{B}\right) + T\left(\mathcal{A}\right)} \pm \text{negl}\left(k\right) = \text{non} - \text{negl}\left(k\right) \end{aligned}$$

□

The two results can then be combined to obtain the desired security property:

Theorem 1. *The construction above is combine-EUF-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly EUF-CMA-secure.*

Proof. The construction is combine-EUF-secure as a consequence of Lemma 1, Lemma 2 and the definition of combine-EUF-security. □

Algorithm $\mathcal{A}(vk)$

$j \xleftarrow{\$} U[1, T(\mathcal{B}) + T(\mathcal{A})]$ // $T(M)$ is the maximum running time of M
    Random Oracle: for every first-seen query q from $\mathcal{B}$ set $\mathcal{H}(q)$ to a random value
    return $\mathcal{H}(q)$ to $\mathcal{B}$
$(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right)$
    Random Oracle: Let q be the jth first-seen query from $\mathcal{B}$ or $\mathcal{A}$:
    if $q = (mpk \,\|\, x)$ **then**
        if $\mathcal{H}(x \,\|\, mpk)$ unset **then**
            set $\mathcal{H}(x \,\|\, mpk)$ to a random value
        end if
        set $\mathcal{H}(mpk \,\|\, x)$ to $(vk - x \cdot \mathcal{H}(x \,\|\, mpk)) \cdot mpk^{-1}$
    else if $q = (x \,\|\, mpk)$ **then**
        if $\mathcal{H}(mpk \,\|\, x)$ unset **then**
            set $\mathcal{H}(mpk \,\|\, x)$ to a random value
        end if
        set $\mathcal{H}(x \,\|\, mpk)$ to $(vk - mpk \cdot \mathcal{H}(mpk \,\|\, x)) \cdot x^{-1}$
    else
        set $\mathcal{H}(q)$ to a random value
    end if
    return $\mathcal{H}(q)$ to $\mathcal{B}$ or $\mathcal{A}$
$i \leftarrow 0$
$(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{B}(\textsc{init}, mpk)$
while response can be parsed as (pk, sk, m) **do**
    $i \leftarrow i + 1$
    store pk, sk, m as pk_i, sk_i, m_i
    $(cpk_i, csk_i) \leftarrow \textsc{CombineKey}(mpk, msk, pk_i, sk_i)$
    $\sigma_i \leftarrow \textsc{SignCS}(m_i, csk_i)$
    $(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{B}(\textsc{signature}, \mathtt{aux}_{i-1}, \sigma_i)$
end while
parse response as $(cpk^*, pk^*, m^*, \sigma^*)$
if $vk = cpk^* \wedge \mathcal{B}$ wins the master-EUF-CMA game **then** // $\mathcal{A}$ won the EUF-CMA game
    return (m^*, σ^*)
else
    return FAIL
end if

Fig. 6.

7 Security proof overview

Theorem 2 (Lightning Payment Network Security). *The protocol Π_{LN} realises $\mathcal{F}_{\text{PayNet}}$ given a global functionality $\mathcal{G}_{\text{Ledger}}$ and assuming the security of the underlying digital signature, identity-based signature, combined digital signature and PRF. Specifically,*

$$\forall k \in \mathbb{N},\ PPT\ \mathcal{E}, |\Pr[EXEC^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}} = 1] - \Pr[EXEC^{\mathcal{F}_{\text{PayNet}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}} = 1]| \leq$$
$$2nm\text{E-ds}(k) + 6np\text{E-ids}(k) + 2nmp\text{E-share}(k) + 2nm\text{E-master}(k) + 2\text{E-prf}(k)\ ,$$

where n is the maximum number of registered users, m is the maximum number of channels that a user is involved in, p is the maximum number of times that a channel is updated and the "E-" terms correspond to the insecurity bounds of the primitives.

Proof Sketch. The proof is done in 5 steps of successive game replacement. In the first lemma, we define a simulator $\mathcal{S}_{\text{LN}}$ that internally simulates a full execution of Π_{LN} for each player, and a "dummy" functionality that acts as a simple relay between $\mathcal{E}$ and $\mathcal{S}_{\text{LN}}$. We argue that this version of the ideal world trivially produces the exact same messages for $\mathcal{E}$ as the real world.

In each subsequent step, we incrementally move responsibilities from the simulator to the functionality, while ensuring the change is transparent to both $\mathcal{E}$ and $\mathcal{A}$. Each step defines a different functionality that handles some additional messages from $\mathcal{E}$ exactly like $\mathcal{F}_{\text{PayNet}}$, until the last step where we use $\mathcal{F}_{\text{PayNet}}$ itself. Correspondingly, the simulator of each step is adapted so that the new ideal execution is computationally indistinguishable from the previous one. For each step we exhaustively trace the differences from the previous step in order to prove that, given the same messages from $\mathcal{E}$ and $\mathcal{A}$, the resulting responses remain unchanged.

The second step, lets $\mathcal{F}$ handle registration messages, along with the corruption messages from $\mathcal{S}$. In the third step, the functionality additionally handles messages related to channel opening. It behaves like $\mathcal{F}_{\text{PayNet}}$, but does not execute `checkClosed`(). The fourth step, has the functionality handle all messages sent during channel updates. Lastly, the entire $\mathcal{F}_{\text{PayNet}}$ is used as functionality, by incorporating the message for closing a channel, executing `checkClosed`() normally and handing the message that returns to $\mathcal{E}$ the receipts for newly opened, updated and closed channels. The last two steps introduce a probability of failure in case the various types of signatures used in Π_{LN} are forged. We analyze these cases separately and argue that, if such forgeries do not happen, the emulation is perfect. Therefore we can calculate the concrete security bounds shown in the theorem. □

As a concrete example of the proof approach, the second step entails the following parts: First $\mathcal{F}_{\text{PayNet,Reg}}$ is defined, which is a functionality that behaves exactly like $\mathcal{F}_{\text{PayNet}}$ when receiving the messages REGISTER, REGISTERDONE, TOPPEDUP and CORRUPTED, but simply forwards all other messages along with the sender to $\mathcal{S}$. Then $\mathcal{S}_{\text{LN}-\text{Reg}}$ is defined, which simulates all protocol instances,

but in response to REGISTER messages from $\mathcal{F}_{\text{PayNet,Reg}}$, it provides the public key of the key it just generated (as $\mathcal{F}_{\text{PayNet,Reg}}$ expects). It also keeps track of corruptions and informs $\mathcal{F}_{\text{PayNet,Reg}}$ thereof. Lastly, we argue that the functionality and simulator that were used in the first step can be replaced by their newly defined counterparts without introducing any discernible difference to the transcript that any $\mathcal{E}$ sees. This is achieved by exhaustive enumeration of all possible messages and comparison of the behaviour of the ideal and the real world for each, to conclude that the change is transparent to $\mathcal{E}$. The formal proof can be found in the full version [25].

8 Instant finality ledgers are unrealisable

As already mentioned, previous attempts at formalising payment channels in UC [13,14,15,16] assume a variant of a ledger functionality with instant finality. We here define a representative variant of this approach $\mathcal{F}_{\text{PerfectL}}$ (defined below) where all submitted transactions are instantly added to the ledger and immediately available to be read by all players. Subsequently we argue that, albeit an attractive abstraction, such a functionality is unrealisable, even under strong network assumptions, i.e. a multicast synchronous network $\mathcal{F}^1_{\text{N-MC}}$. Such a network ensures that messages sent by honest parties will be instantly delivered to all other parties; no delays can be introduced by the adversary. The formal definition of $\mathcal{F}^1_{\text{N-MC}}$ can be found in Figure 8. The adversary however may choose to send its own messages only to specific parties. This allows the adversary to spread conflicting information or withhold data from some parties. This adversarial ability precludes the possibility of such a ledger to be realised.

Functionality $\mathcal{F}_{\text{PerfectL}}$

1: State: List of txs $\mathcal{L}$

2: Upon receiving (SUBMIT, m) from *Alice* or $\mathcal{A}$, append m to $\mathcal{L}$ and send (SUBMIT, *Alice* or $\mathcal{A}$, m) to $\mathcal{A}$

3: Upon receiving (READ) from *Alice*, send (READ, $\mathcal{L}$) to *Alice*

Fig. 7. $\mathcal{F}_{\text{PerfectL}}$functionality

Theorem 3 (Perfect Ledger is Unrealisable). *No ITM Π_{PerfectL} with hybrids $\mathcal{F}^1_{N\text{-}MC}$ and $\bar{\mathcal{G}}_{CLOCK}$ can realise $\mathcal{F}_{\text{PerfectL}}$. Put otherwise, for any ITM Π_{PerfectL} with hybrids $\mathcal{F}^1_{N\text{-}MC}$ and $\bar{\mathcal{G}}_{CLOCK}$, there exist ITMs $\mathcal{E}_{\text{PL}}$, $\mathcal{A}_{\text{PL}}$ such that for any ITM $\mathcal{S}$*

$$EXEC^{\mathcal{F}^1_{N\text{-}MC},\bar{\mathcal{G}}_{CLOCK}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} \not\approx EXEC^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{CLOCK}}_{\mathcal{S},\mathcal{E}_{\text{PL}}}$$

Functionality $\mathcal{F}_{\text{N-MC}}^{\Delta}$

The functionality is parameterised with a set of possible senders and receivers $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

- **Honest sender multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $U_p \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose n new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$, initialize $2n$ new variables $D_{\text{mid}_1} := D_{\text{mid}_1}^{MAX} \ldots := D_{\text{mid}_n} := D_{\text{mid}_n}^{MAX} := 1$, set $\vec{M} := \vec{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{MULTICAST}, \text{sid}, m, U_p, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
- **Adversarial sender (partial) multicast.** Upon receiving $(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell})$ from the adversary with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}$, choose ℓ new unique message-IDs $\text{mid}_{i_1}, \ldots, \text{mid}_{i_\ell}$, initialize ℓ new variables $D_{\text{mid}_{i_1}} := D_{\text{mid}_{i_1}}^{MAX} := \ldots := D_{\text{mid}_{i_\ell}} := D_{\text{mid}_{i_\ell}}^{MAX} := 1$, set $\vec{M} := \vec{M}||(m_{i_1}, \text{mid}_{i_1}, D_{\text{mid}_{i_1}}, U_{i_1})|| \ldots ||(m_{i_\ell}, \text{mid}_{i_\ell}, D_{\text{mid}_{i_\ell}}, U_{i_\ell})$, and send $(\text{MULTICAST}, \text{sid}, (m_{i_1}, U_{i_1}, \text{mid}_{i_1}), \ldots, (m_{i_\ell}, U_{i_\ell}, \text{mid}_{i_\ell})$ to the adversary.
- **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $U_p \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of U_p if U_p is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, U_p) \in \vec{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\vec{M}_0^{U_p}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, U_p)$ with $D_{\text{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0^{U_p}$ from $\vec{M}$, and send $\vec{M}_0^{U_p}$ to U_p.
- **Adding adversarial delays.** Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$:
  If $D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}} \leq \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\text{mid}_{i_j}} := D_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$ and set $D_{\text{mid}_{i_j}}^{MAX} := D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}}$; otherwise, ignore this pair.
- **Adversarially reordering messages.** Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

Fig. 8. $\mathcal{F}_{\text{N-MC}}^{\Delta}$ functionality

Proof Sketch. We take advantage of $\mathcal{A}_{\text{PL}}$'s ability to selectively send messages to specific players. In particular, $\mathcal{E}_{\text{PL}}$ starts an execution with two players and generates a random message m. In half of the executions (randomly selected), the adversary simulates a "broken" Π_{PerfectL} execution where the effects of submitting m are only shared with one of the two players, say *Alice* by $\mathcal{A}_{\text{PL}}$ (in the real world). The environment then sends (READ) to the other player, say *Bob*. If *Bob* returns a ledger containing m, then $\mathcal{E}_{\text{PL}}$ concludes that it is the ideal world, otherwise it sends (READ) to *Alice*. If she returns a ledger with m, then $\mathcal{E}_{\text{PL}}$ concludes it is in the real world, otherwise it concludes it is in the ideal world.

The above is not sufficient since a protocol may choose to return an empty ledger; to counter this, in the other half of the executions, $\mathcal{E}_{\text{PL}}$ sends (SUBMIT, m) to *Alice* and then (READ) to *Bob*. If, and only if, *Bob* knows m, then $\mathcal{E}_{\text{PL}}$ concludes this is the ideal world. This forces the Π_{PerfectL} protocol to achieve instant finality and will establish that a distinguishing advantage exists no matter how Π_{PerfectL} is implemented. □

Proof of Theorem 3. We first define the offending environment and adversary and subsequently show how they can distinguish the ideal from the real world.

Adversary $\mathcal{A}_{\text{PL}}$

Upon receiving (LEAK, m, *Alice*) from $\mathcal{E}_{\text{PL}}$, simulate Π_{PerfectL} reacting to (SUBMIT, m). If it attempts to send a message (MULTICAST, m') to $\mathcal{F}_{\text{N-MC}}^{1}$, send (MULTICAST, ($m'$, *Alice*)) to $\mathcal{F}_{\text{N-MC}}^{1}$.

Fig. 10. $\mathcal{A}_{\text{PL}}$adversary

Since we quantify over all possible $\mathcal{S}$ and Π_{PerfectL}, we have to refer to the probabilities of them taking specific actions of interest:

$$p_{\text{submits}}^{\Pi_{\text{PerfectL}}} = \Pr[\text{Upon receiving (SUBMIT, } m\text{) from } \mathcal{E},$$
$$\Pi_{\text{PerfectL}} \text{ sends (MULTICAST, } f(m)\text{) to } \mathcal{F}_{\text{N-MC}}^{1} \text{ for some function } f]$$
$$p_{\text{fetches}}^{\Pi_{\text{PerfectL}}} = \Pr[\text{Upon receiving (READ) from } \mathcal{E},$$
$$\Pi_{\text{PerfectL}} \text{ sends (FETCH) to } \mathcal{F}_{\text{N-MC}}^{1} \text{ for data } m'$$
$$\text{and sends back to } \mathcal{E} \text{ a (READ, } \mathcal{L}\text{) such that}$$
$$\text{if there is a unique element } m \text{ in } \mathcal{L}, \text{ it is } f(m) = m']$$

We first analyze the event in which the initial coin flip of $\mathcal{E}$ results in 0, Coin_0. In the ideal world, the submitted message m always ends up in the ledger right away and therefore when $\mathcal{E}$ has *Bob* READ, it always sees m in the answer,

Environment $\mathcal{E}_{\text{PL}}$

Spawn two players, *Alice* and *Bob*. Flip a coin. If it returns 0, execute writeWithPlayer, otherwise execute writeWithAdversary.

```
 1: procedure writeWithPlayer
 2:     First activation:
 3:         choose random number m ←$ {0,1}^k
 4:         assign at random names Alice, Bob to two players
 5:         send (SUBMIT, m) to Alice
 6:     Second activation:
 7:         send (READ) to Bob
 8:         if Bob does not give subroutine output then
 9:             return 0 // real world
10:         else if Bob's subroutine output L contains m then
11:             return 1 // players communicated
12:         else if L does not contain m then
13:             return 0 // players did not communicate
14:         end if
15: end procedure

16: procedure writeWithAdversary
17:     First activation:
18:         choose random number m ←$ {0,1}^k
19:         assign at random names Alice, Bob to two players
20:         send (LEAK, m, Alice) to A // in real world A will MULTICAST to Alice
21:     Second activation:
22:         send (READ) to Bob
23:     Third activation:
24:         if Bob does not give subroutine output then
25:             return 0 // real world
26:         else if Bob's subroutine output L_Bob contains m then
27:             return 1 // ideal world
28:         end if
29:         send (READ) to Alice
30:         if Alice does not give subroutine output then
31:             return 0 // real world
32:         else if Alice's subroutine output L_Alice contains m then
33:             return 0 // real world
34:         else if L_Alice does not contain m then
35:             return 1 // ideal world or real Alice misbehaving
36:         end if
37: end procedure
```

Fig. 9. $\mathcal{E}_{\text{PL}}$environment

therefore (Fig. 9, line 11)

$$\Pr[\text{EXEC}^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\mathcal{S},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_0] = 1 \ .$$

In the real world, in order for the submitted message m to be in *Bob*'s response to READ, he must have fetched from $\mathcal{F}^1_{\text{N-MC}}$ and considered this data as a new ledger entry, and *Alice* must have sent some function of m to $\mathcal{F}^1_{\text{N-MC}}$ when she received (SUBMIT, m), except if he could guess m, which can happen with negligible probability, therefore

$$\Pr[\text{EXEC}^{\mathcal{F}^1_{\text{N-MC}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_0] < p^{\Pi_{\text{PerfectL}}}_{\text{submits}} p^{\Pi_{\text{PerfectL}}}_{\text{fetches}} + \text{negl}(k) \ .$$

We now move on to the event in which the initial coin flip results in 1, Coin_1. In the ideal world, if $\mathcal{S}$ SUBMITs the received m to the ledger then $\mathcal{E}$'s READ request to *Bob* will be answered with an output that contains m and $\mathcal{E}$ will output 1 (Fig. 9, line 27). If on the other hand $\mathcal{S}$ does not SUBMIT it, then neither *Bob*'s nor *Alice*'s answer will contain m, so $\mathcal{E}$'s output will also be 1 (Fig. 9, line 35).

$$\Pr[\text{EXEC}^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\mathcal{S},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_1] = 1 \ .$$

Lastly, in the real world, *Bob*'s buffer in $\mathcal{F}^1_{\text{N-MC}}$ does not contain any information, so he may return a ledger containing m only with negligible probability. In case he returns a ledger without m, *Alice* will respond to $\mathcal{E}$'s READ query with a ledger containing m exactly in the case that the event that defines $p^{\Pi_{\text{PerfectL}}}_{\text{fetches}}$ is true, therefore

$$\Pr[\text{EXEC}^{\mathcal{F}^1_{\text{N-MC}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_1] < (1 - p^{\Pi_{\text{PerfectL}}}_{\text{fetches}}) + \text{negl}(k) \ .$$

Note that Π_{PerfectL} cannot leverage knowledge of its own pid in order to have *Alice* behave differently from *Bob* in a manner that tricks $\mathcal{E}_{\text{PL}}$ into believing that it interacts with the ideal world (i.e. make *Alice* also not return a ledger that contains m) because the roles of *Alice* and *Bob* are assigned and the coin is flipped secretly at random by $\mathcal{E}_{\text{PL}}$.

In aggregate,

$$\Pr[\text{EXEC}^{\mathcal{F}^1_{\text{N-MC}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} = 1] =$$

$$\frac{1}{2}(\Pr[\text{EXEC}^{\mathcal{F}^1_{\text{N-MC}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_0] + \Pr[\text{EXEC}^{\mathcal{F}^1_{\text{N-MC}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\Pi_{\text{PerfectL}},\mathcal{A}_{\text{PL}},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_1]) <$$

$$\frac{1}{2}(p^{\Pi_{\text{PerfectL}}}_{\text{submits}} p^{\Pi_{\text{PerfectL}}}_{\text{fetches}} + 1 - p^{\Pi_{\text{PerfectL}}}_{\text{fetches}}) + \text{negl}(k) =$$

$$\frac{1}{2} + p^{\Pi_{\text{PerfectL}}}_{\text{fetches}} \frac{p^{\Pi_{\text{PerfectL}}}_{\text{submits}} - 1}{2} + \text{negl}(k) \ ,$$

and

$$\Pr[\text{EXEC}^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\mathcal{S},\mathcal{E}_{\text{PL}}} = 1] =$$

$$\frac{1}{2}(\Pr[\text{EXEC}^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\mathcal{S},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_0] + \Pr[\text{EXEC}^{\mathcal{F}_{\text{PerfectL}},\bar{\mathcal{G}}_{\text{CLOCK}}}_{\mathcal{S},\mathcal{E}_{\text{PL}}} = 1|\text{Coin}_1]) = 1 \ .$$

For these two probabilities to be equal (which is necessary and sufficient for indistinguishability to hold), it would have to be $p_{\text{fetches}}^{\Pi_{\text{PerfectL}}}(p_{\text{submits}}^{\Pi_{\text{PerfectL}}} - 1) = 1$. One can verify that there is no assignment to the two probabilities that satisfies this equation and maintains both values within $[0, 1]$. Therefore, the real and the ideal world are distinguishable. □

9 Payment Network Functionality

Functionality $\mathcal{F}_{\text{PayNet}}$ – interface

- from $\mathcal{E}$:
  - (REGISTER, delay, relayDelay)
  - (TOPPEDUP)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *tid*)
  - (CHECKFORNEW, *Alice*, *Bob*, *tid*)
  - (PAY, *Bob*, $x, \overrightarrow{\texttt{path}}$, *expayid*)
  - (CLOSECHANNEL, `receipt`, *pchid*)
  - (FORCECLOSECHANNEL, `receipt`, *pchid*)
  - (POLL)
  - (PUSHFULFILL, *pchid*)
  - (PUSHADD, *pchid*)
  - (COMMIT, *pchid*)
  - (FULFILLONCHAIN)
  - (GETNEWS)
- to $\mathcal{E}$:
  - (REGISTER, *Alice*, `delay`(*Alice*), `relayDelay`(*Alice*), pubKey)
  - (REGISTERED)
  - (NEWS, `newChannels`, `closedChannels`, `updatesToReport`, `paymentsToReport`)
- from $\mathcal{S}$:
  - (REGISTERDONE, *Alice*, pubKey)
  - (CORRUPTED, *Alice*)
  - (CHANNELANNOUNCED, *Alice*, $p_{Alice,F}, p_{Bob,F}$, *fchid*, *pchid*, *tid*)
  - (NEWUPDATE, `receipt`, *Alice*)
  - (NEWPAYMENTS, `payments`, *Alice*)
  - (CLOSEDCHANNEL, `channel`, *Alice*)
  - (RESOLVEPAYS, `charged`)
  - (ADVERSARYOPENCHANNEL, x, `bobDelay`, *tid*, *intid*, *from Alice*, *to Bob*)
  - (ADVERSARYSENDINVOICE, x, *expayid*, *payid*, *invid*, *from Alice*, *to Bob*)
- to $\mathcal{S}$:
  - (REGISTER, *Alice*, delay, relayDelay)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*)
  - (CHANNELOPENED, *Alice*, *fchid*)
  - (PAY, *Alice*, *Bob*, $x, \overrightarrow{\texttt{path}}$, *expayid*, *payid*, STATE, Σ)
  - (CONTINUE)
  - (CLOSECHANNEL, *fchid*, *Alice*)
  - (FORCECLOSECHANNEL, *fchid*, *Alice*)
  - (POLL, Σ_{Alice}, *Alice*)
  - (PUSHFULFILL, *pchid*, *Alice*)
  - (PUSHADD, *pchid*, *Alice*)
  - (COMMIT, *pchid*, *Alice*)
  - (FULFILLONCHAIN, t, *Alice*)
  - (ADVERSARYOPENCHANNEL, *fchid*, *intid*)
  - (ADVERSARYSENDINVOICE, *invid*)

Fig. 11.

All players need to register in order to use channels. The registration of *Alice* works as follows: *Alice* inputs her desired delay and relayDelay that will be used for all her future channels. The first denotes how often she has to check the blockchain for revoked commitments and the second defines the minimum time distance between incoming and outgoing CLTV expiries. $\mathcal{F}_{\text{PayNet}}$ then informs $\mathcal{S}$, who sends back a long-lived public key for *Alice*. This key represents *Alice*'s account, from where $\mathcal{F}_{\text{PayNet}}$ can get coins to open new channels on her behalf and to place coins of closed channels. The key is sent to *Alice* who moves some initial funds to it and notifies $\mathcal{F}_{\text{PayNet}}$. She is now registered. The exact logic is found in Fig. 12, which also contains the actions of $\mathcal{F}_{\text{PayNet}}$ related to corruptions.

Additionally, the procedure `checkClosed`() is called after READing from $\mathcal{G}_{\text{Ledger}}$, with the received state Σ as input. This call happens every time $\mathcal{F}_{\text{PayNet}}$ READs from $\mathcal{G}_{\text{Ledger}}$. The formal definition of `checkClosed`() can be found in Fig. 21, along with a discussion of its purpose.

Functionality $\mathcal{F}_{\text{PayNet}}$ – registration and corruption

Initialisation:
    $\texttt{channels}, \texttt{pendingPay}, \texttt{pendingOpen} \leftarrow \emptyset$
    $\texttt{pendingDiffs}, \texttt{corrupted}, \Sigma \leftarrow \emptyset$

Upon receiving (REGISTER, delay, relayDelay) from *Alice*:
    $\texttt{delay}(\mathit{Alice}) \leftarrow$ delay // Must check chain at least once every $\texttt{delay}(\mathit{Alice})$ blocks
    $\texttt{relayDelay}(\mathit{Alice}) \leftarrow$ relayDelay
    $\texttt{updatesToReport}(\mathit{Alice}), \texttt{newChannels}(\mathit{Alice}) \leftarrow \emptyset$
    $\texttt{polls}(\mathit{Alice}) \leftarrow \emptyset$
    $\texttt{focs}(\mathit{Alice}) \leftarrow \emptyset$
    register *Alice* to $\mathcal{G}_{\text{Ledger}}$, send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, store reply to Σ_{Alice}, add Σ_{Alice} to Σ and add largest block number to $\texttt{polls}(\mathit{Alice})$
    $\texttt{checkClosed}(\Sigma_{\mathit{Alice}})$
    send (REGISTER, *Alice*, delay, relayDelay) to $\mathcal{S}$

Upon receiving (REGISTERDONE, *Alice*, pubKey) from $\mathcal{S}$:
    $\texttt{pubKey}(\mathit{Alice}) \leftarrow$ pubKey
    send (REGISTER, *Alice*, $\texttt{delay}(\mathit{Alice})$, $\texttt{relayDelay}(\mathit{Alice})$, pubKey) to *Alice*

Upon receiving (TOPPEDUP) from *Alice*:
    send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
    $\texttt{checkClosed}(\Sigma_{\mathit{Alice}})$
    assign the sum of all output values that are exclusively spendable by *Alice* to $\texttt{onChainBalance}$
    send (REGISTERED) to *Alice*

Upon receiving any message (M) except for (REGISTER) or (TOPPEDUP) from *Alice*:
    if if haven't received (REGISTER) and (TOPPEDUP) from *Alice* (in this order) **then**
        send (INVALID, M) to *Alice* and ignore message
    end if

Upon receiving (CORRUPTED, *Alice*) from $\mathcal{S}$:
    add *Alice* to $\texttt{corrupted}$
    for the rest of the execution, upon receiving any message for *Alice*, bypass normal execution and simply forward it to $\mathcal{S}$

Fig. 12.

The process of *Alice* opening a channel with *Bob* is as follows: First *Alice* asks $\mathcal{F}_{\text{PayNet}}$ to open and $\mathcal{F}_{\text{PayNet}}$ informs $\mathcal{S}$. $\mathcal{S}$ provides the necessary keys and IDs

for the new channel to $\mathcal{F}_{\text{PayNet}}$. *Alice* asks $\mathcal{F}_{\text{PayNet}}$ to check if $\mathcal{G}_{\text{Ledger}}$ contains the funding transaction from *Alice*'s point of view. If it does, $\mathcal{F}_{\text{PayNet}}$ activates $\mathcal{S}$, who in turn returns control to $\mathcal{F}_{\text{PayNet}}$. Now $\mathcal{F}_{\text{PayNet}}$ checks that the funding transaction is in the $\mathcal{G}_{\text{Ledger}}$ also from *Bob*'s point of view and in case it does, it notifies $\mathcal{S}$. $\mathcal{S}$ then confirms that to $\mathcal{F}_{\text{PayNet}}$ that the channel is open and $\mathcal{F}_{\text{PayNet}}$ finally stores the channel as open. This last exchange is needed to match the real-world interaction.

Functionality $\mathcal{F}_{\text{PayNet}}$ – initiate open

1: Upon receiving (openChannel, *Alice*, *Bob*, x, *tid*) from *Alice*:
2: ensure *tid* hasn't been used by *Alice* for opening another channel before
3: choose unique channel ID *fchid*
4: `pendingOpen` (*fchid*) $\leftarrow$ (*Alice*, *Bob*, x, *tid*)
5: send (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) to $\mathcal{S}$

6: Upon receiving (adversaryOpenChannel, x, `bobDelay`, *tid*, *intid*, *from Alice*, *to Bob*) from $\mathcal{S}$:
7: ensure *Alice* is `corrupted` and *Bob* is not
8: ensure *tid*, *intid* haven't been associated with *Alice* when opening another channel before
9: choose unique channel ID *fchid*
10: `pendingOpen` (*fchid*) $\leftarrow$ (*Alice*, *Bob*, x, *tid*)
11: send (adversaryOpenChannel, *fchid*, *intid*) to $\mathcal{S}$

Fig. 13.

Functionality $\mathcal{F}_{\text{PayNet}}$ – negotiate open

1: Upon receiving (channelAnnounced, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*) from $\mathcal{S}$:
2: ensure that there is a `pendingOpen`(*fchid*) entry with temporary id *tid*
3: add $p_{Alice,F}, p_{Bob,F}$, *pchid* and mark "*Alice* announced" to `pendingOpen`(*fchid*)

4: Upon receiving (checkForNew, *Alice*, *Bob*, *tid*) from *Alice*:
5: ensure there is a matching `channel` in `pendingOpen`(*fchid*), marked with "*Alice* announced"
6: (funder, fundee, x, $p_{Alice,F}$, $p_{Bob,F}$) $\leftarrow$ `pendingOpen` (*fchid*)
7: send (read) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
8: `checkClosed`(Σ_{Alice})
9: ensure that there is a TX $F \in \Sigma_{Alice}$ with a $(x, (p_{\text{funder},F} \wedge p_{\text{fundee},F}))$ output
10: mark `channel` with "waiting for fundingLocked"
11: send (fundingLocked, *Alice*, Σ_{Alice}, *fchid*) to $\mathcal{S}$

12: Upon receiving (fundingLocked, *fchid*) from $\mathcal{S}$:
13: ensure a `channel` is in `pendingOpen`(*fchid*), marked with "waiting for fundingLocked" and replace mark with "waiting for channelOpened"
14: send (read) to $\mathcal{G}_{\text{Ledger}}$ as *Bob* and store reply to Σ_{Bob}
15: `checkClosed`(Σ_{Bob})
16: ensure that there is a TX $F \in \Sigma_{Bob}$ with a $(x, (p_{\text{funder},F} \wedge p_{\text{fundee},F}))$ output
17: add `receipt`(`channel`) to `newChannels`(*Bob*)
18: send (fundingLocked, *Bob*, Σ_{Bob}, *fchid*) to $\mathcal{S}$

19: Upon receiving (channelOpened, *fchid*) from $\mathcal{S}$:
20: ensure a `channel` is in `pendingOpen`(*fchid*), marked with "waiting for channelOpened" and remove mark
21: `offChainBalance` (funder) $\leftarrow$ `offChainBalance` (funder) $+ x$
22: `onChainBalance` (funder) $\leftarrow$ `onChainBalance` (funder) $- x$
23: `channel` $\leftarrow$ (funder, fundee, $x, 0, 0$, *fchid*, *pchid*)
24: add `channel` to `channels`
25: add `receipt`(`channel`) to `newChannels`(*Alice*)
26: clear `pendingOpen`(*fchid*) entry

Fig. 14.

When instructed to perform a payment, $\mathcal{F}_{\text{PayNet}}$ simply takes note of the message and forwards it to $\mathcal{S}$. It also remembers to inform the payer that the payment has been completed when $\mathcal{S}$ says so. Observe here that $\mathcal{F}_{\text{PayNet}}$ trusts $\mathcal{S}$ to correctly carry out channel updates. While counterintuitive, it allows $\mathcal{F}_{\text{PayNet}}$ to ignore the details of channel updates, signatures, key and transaction han-

dling. Nevertheless, as we will see $\mathcal{F}_{\text{PayNet}}$ keeps track of requested and ostensibly carried out updates and ensures that upon channel closure the balances are as expected, therefore ensuring funds security.

Functionality $\mathcal{F}_{\text{PayNet}}$ – pay

1: Upon receiving $\left(\text{PAY}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \mathit{expayid}\right)$ from *Alice*:
2: choose unique payment ID *payid*
3: add $\left(\mathit{Alice}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \mathit{expayid}, \mathit{payid}\right)$ to `pendingPay`
4: send (PAY, *Alice*, *Bob*, x, $\overrightarrow{\texttt{path}}$, *expayid*, *payid*, STATE, Σ) to $\mathcal{S}$

5: Upon receiving (ADVERSARYSENDINVOICE, x, *expayid*, *payid*, *invid*, *from Alice*, *to Bob*) from $\mathcal{S}$:
6: ensure *Alice* is `corrupted` and *Bob* is not
7: ensure *payid*, *invid* haven't been associated with *Alice* when starting another payment before
8: add (*Alice*, *Bob*, x, $\perp$, *expayid*, *payid*) to `pendingPay`
9: send (ADVERSARYSENDINVOICE, *invid*) to $\mathcal{S}$

10: Upon receiving (NEWUPDATE, `receipt`, *Alice*) from $\mathcal{S}$:
11: append `receipt` to `updatesToReport`(*Alice*) // trust $\mathcal{S}$ here, check on RESOLVEPAYS
12: send (CONTINUE) to $\mathcal{S}$

13: Upon receiving (NEWPAYMENTS, `payments`, *Alice*) from $\mathcal{S}$:
14: add `payments` to `pendingDiffs`
15: strip *payid* from each member of `payments` and add them to `paymentsToReport`(*Alice*)
16: send (CONTINUE) to $\mathcal{S}$

Fig. 15.

The message RESOLVEPAYS, sent by $\mathcal{S}$, is supposed to contain a list of resolved payments, along with who was charged for each payment after all. For each entry there are four "happy paths" that do not lead to $\mathcal{F}_{\text{PayNet}}$ halting ($\mathcal{F}_{\text{PayNet}}$ halts when it cannot uphold its security guarantees anymore): if the payment failed and no balance is changed, if the charged player is the one who initiated the payment, if the charged player is corrupted or if she has not checked the blockchain at the right times, i.e. was negligent (as discussed in Section 4 and formally defined in Figures 16 and 17). In case the payment was completed in a legal manner, the balance of all channels involved is updated accordingly (Fig. 19). Conversely, $\mathcal{F}_{\text{PayNet}}$ halts if the charged player was not on the payment path (Fig. 16, l. 14), if a signature forgery has taken place (Fig. 18, l. 4), if the

charged player has not been negligent (Fig. 18, ll. 7 and 15), or if any one of the individual channel updates needed to carry out the whole payment has not been previously reported with an UPDATE message by $\mathcal{S}$ (Fig. 19, l. 10).

Functionality $\mathcal{F}_{\text{PayNet}}$ – resolve payments

1: Upon receiving (RESOLVEPAYS, `charged`) from $\mathcal{S}$: // after first sending PAY, PUSHFULFILL, PUSHADD, COMMIT
2: **for all** *Alice* keys $\in$ `charged` **do**
3: **for all** $\left(Dave, payid, ?\overrightarrow{\texttt{path}}\right) \in$ `charged`$(Alice)$ **do**
4: retrieve $\left(Alice, Bob, x, \overrightarrow{\texttt{path}}, expayid, payid\right)$ and remove it from `pendingPay`
5: **if** there is a $\overrightarrow{\texttt{path}}$ in neither or both the `pendingPay` and `charged` entries **then**
6: halt
7: **end if**
8: **if** $Dave = \perp$ **then** // Payment failed
9: **if** there is an (*expayid*, *payid*, _) entry in `pendingDiffs`$(Alice)$ or `pendingDiffs`(Bob) AND *Alice* or *Bob* respectively is honest **then**
10: halt
11: **end if**
12: continue with next iteration of inner loop
13: **else if** $Dave \notin \overrightarrow{\texttt{path}}$ **then**
14: halt // Only players on path may be charged
15: **else if** $Dave \in$ `corrupted` **then**
16: run code of Fig. 19
17: increase `offChainBalance`(Bob) by x
18: **else** // *Dave* honest
19: run code of Fig. 18
20: **end if**
21: **end for**
22: **end for**

Fig. 16. r, `windowSize` as in Proposition 1

Absolute delay failure condition

$\texttt{IncomingCltvExpiry} - \texttt{OutgoingCltvExpiry} <$
$\texttt{relayDelay}(\mathit{Alice}) + (2 + r)\,\texttt{windowSize} \vee$
($\texttt{polls}(\mathit{Dave})$ contains two elements in
$[\texttt{OutgoingCltvExpiry}, \texttt{IncomingCltvExpiry} - (2 + r)\,\texttt{windowSize}]$ that have a difference of at least $(2 + r)\,\texttt{windowSize} \wedge$
$\texttt{focs}(\mathit{Dave})$ contains $\texttt{IncomingCltvExpiry} - (2 + r)\,\texttt{windowSize} \wedge$
the element in $\texttt{polls}(\mathit{Dave})$ was added before the element in $\texttt{focs}(\mathit{Dave})$)

Fig. 17.

Honest payer

send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Dave* and store reply to Σ_{Dave}
$\texttt{checkClosed}(\Sigma_{Dave})$
if Σ_{Dave} contains a tx that is not a $\texttt{localCom}_n$ or a $\texttt{remoteCom}_n$ and spends a funding tx for an open `channel` that contains *Dave* **then**
    halt // DS forgery
else if Σ_{Dave} contains in block $h_{\texttt{tx}}$ an old $\texttt{remoteCom}_m$ that does not contain the HTLC and a tx that spends the delayed output of $\texttt{remoteCom}_m$ **then**
    if $\texttt{polls}(\mathit{Dave})$ contains an element in $[h_{\texttt{tx}}, h_{\texttt{tx}} + \texttt{delay}\,(\mathit{Dave}) - 1]$ **then**
        halt // *Dave* POLLed, but successful malicious closure
    else
        $\texttt{negligent}(\mathit{Dave}) \leftarrow \texttt{true}$
    end if
else if *Dave* $\neq$ *Alice* **then**
    calculate `IncomingCltvExpiry`, `OutgoingCltvExpiry` of *Dave* (as in Fig. 35, l. 9)
    if Σ_{Dave} does not contain an old $\texttt{remoteCom}_m$ **then**
        if failure condition of Fig. 17 is true **then**
            halt // *Dave* POLLed and fulfilled, but charged
        else
            $\texttt{negligent}(\mathit{Dave}) \leftarrow \texttt{true}$
        end if
    end if
end if
run code of Fig. 19
decrease $\texttt{offChainBalance}(\mathit{Dave})$ by x
increase $\texttt{offChainBalance}(\mathit{Bob})$ by x

Fig. 18.

Loop over payment hops for update and check

1: **for all** open `channel`s $\in \overrightarrow{\texttt{path}}$ that are not in any `closedChannels`, starting from the one where *Dave* pays **do**
2: in the first iteration, `payer` is *Dave*. In subsequent iterations, `payer` is the unique player that has received but has not given. The other `channel` party is `payee`
3: **if** `payer` has x or more in `channel` **then**
4: update `channel` to the next version and transfer x from `payer` to `payee`
5: **else**
6: revert all updates done in this loop
7: **end if**
8: **end for**
9: **for all** updated `channel`s in the previous loop **do**
10: ensure that an element reporting the new balance exists either in the `updatesToReport` or the `updatesHistory` of each honest party and that element has not been checked by the current line in the past, otherwise halt
11: **end for**
12: **if** $Dave = Alice \wedge Alice \notin \texttt{corrupted} \wedge (expayid, payid, -x) \notin \texttt{pendingDiffs}(Alice)$ **then** // payer not informed
13: halt
14: **end if**
15: remove ($expayid$, $payid$, $-x$) from `pendingDiffs`($Alice$)
16: **if** $Bob \notin \texttt{corrupted} \wedge (expayid, payid, x) \notin \texttt{pendingDiffs}(Bob)$ **then** // payee not informed
17: halt
18: **end if**
19: remove ($expayid$, $payid$, x) from `pendingDiffs`(Bob)

Fig. 19.

Similarly to payment instructions, when $\mathcal{F}_{\text{PayNet}}$ receives a message instructing it to close a channel (Fig. 20), it takes a note of the pending closure, it stops serving any more requests for this channel and it forwards the request to $\mathcal{S}$. In turn $\mathcal{S}$ notifies $\mathcal{F}_{\text{PayNet}}$ of a closed channel with the corresponding message, upon which $\mathcal{F}_{\text{PayNet}}$ takes a note to inform the corresponding player. Depending on whether the message instructed for a unilateral or a cooperative close, $\mathcal{F}_{\text{PayNet}}$ will either put or not a time limit respectively to the service of the request. In particular, in case of cooperative close, the time limit is infinity (l. 4). As we will see, in case a unilateral close request was made and the time limit for servicing it is reached, $\mathcal{F}_{\text{PayNet}}$ halts (Fig. 21, l. 27). Once more $\mathcal{F}_{\text{PayNet}}$ trusts $\mathcal{S}$, but later checks that the chain contains the correct transactions with `checkClosed`() (Fig. 21).

Functionality $\mathcal{F}_{\text{PayNet}}$ – close

1: Upon receiving (CLOSECHANNEL, `receipt`, *pchid*) from *Alice*
2: ensure that there is a `channel` $\in$ `channels` : `receipt`(`channel`) = `receipt` with ID *pchid*
3: retrieve *fchid* from `channel`
4: add (*fchid*, `receipt`(`channel`), ∞) to `pendingClose`(*Alice*)
5: do not serve any other (PAY, CLOSECHANNEL) message from *Alice* for this channel
6: send (CLOSECHANNEL, `receipt`, *pchid*, *Alice*) to $\mathcal{S}$

7: Upon receiving (FORCECLOSECHANNEL, `receipt`, *pchid*) from *Alice*
8: retrieve *fchid* from `channel`
9: add (*fchid*, `receipt`(`channel`), $\perp$) to `pendingClose`(*Alice*)
10: do not serve any other (PAY, CLOSECHANNEL, FORCECLOSECHANNEL) message from *Alice* for this channel
11: send (FORCECLOSECHANNEL, `receipt`, *pchid*, *Alice*) to $\mathcal{S}$

12: Upon receiving (CLOSEDCHANNEL, `channel`, *Alice*) from $\mathcal{S}$:
13: remove any (*fchid* of channel, `receipt`(`channel`), ∞) from `pendingClose`(*Alice*)
14: add (*fchid* of channel, `receipt`(`channel`), $\perp$) to `closedChannels`(*Alice*) // trust $\mathcal{S}$ here, check on `checkClosed`()
15: send (CONTINUE) to $\mathcal{S}$

Fig. 20.

After every READ $\mathcal{F}_{\text{PayNet}}$ sends to $\mathcal{G}_{\text{Ledger}}$ and its response is received, `checkClosed`() (Fig. 21) is called. $\mathcal{F}_{\text{PayNet}}$ checks the input state Σ for transactions that close channels and, in case no security violation has taken place, it updates the on- and off-chain balances of the player accordingly (ll. 6-15). The possible security violations are: signature forgery (l. 17), malicious closure even though the player was not negligent (l. 20), no closing transaction in Σ even though the player asked for channel closure a substantial amount of time before (l. 27) and incorrect on- or off-chain balance after the closing of all of the player's channels (l. 31).

Functionality $\mathcal{F}_{\text{PayNet}}$ – `checkClosed`()

function `checkClosed`(Σ_{Alice}) // Called after every (READ), ensures requested closes eventually happen
  if there is any closing/commitment transaction in Σ_{Alice} with no corresponding entry in `pendingClose`($Alice$) $\cup$ `closedChannels`($Alice$) **then**
    add ($fchid$, `receipt`, $\perp$) to `closedChannels`($Alice$), where $fchid$ is the ID of the corresponding `channel`, `receipt` comes from the latest `channel` state
  end if
  for all entries ($fchid$, `receipt`, h) $\in$ `pendingClose`($Alice$) $\cup$ `closedChannels`($Alice$) **do**
    if there is a closing/commitment transaction in Σ_{Alice} for open `channel` with ID $fchid$ with a balance that corresponds to `receipt` **then**
      let x, y $Alice$'s and `channel` counterparty Bob's balances respectively
      reduce `offChainBalance`($Alice$) by x
      increase `onChainBalance`($Alice$) by x
      reduce `offChainBalance`(Bob) by y
      increase `onChainBalance`(Bob) by y
      remove `channel` from `channels` and entry from `pendingClose`($Alice$)
      if there is an ($fchid$, _, _) entry in `pendingClose`(Bob) **then**
        remove it from `pendingClose`(Bob)
      end if
    else if there is a tx in Σ_{Alice} that is not a closing/commitment tx and spends the funding tx of the `channel` with ID $fchid$ **then**
      halt // DS forgery
    else if there is a commitment transaction in block of height h in Σ_{Alice} for open `channel` with ID $fchid$ with a balance that does not correspond to the `receipt` and the delayed output has been spent by the counterparty **then**
      if `polls`($Alice$) contains an entry in $[h, h + \texttt{delay}(Alice) - 1]$ **then**
        halt
      else
        `negligent`($Alice$) $\leftarrow$ `true`
      end if
    else if there is no such closing/commitment transaction $\wedge$ $h = \perp$ **then**
      assign largest Σ_{Alice} block no. to entry's h
    else if there is no such closing/commitment transaction $\wedge$ $h \neq \perp$ $\wedge$ (largest block number of Σ_{Alice}) $\geq h + (2 + r)$ `windowSize` **then**
      halt
    end if
  end for
  if $Alice$ has no open channels in Σ_{Alice} $\wedge$ `negligent`($Alice$) = `false` $\wedge$ (`offChainBalance`($Alice$) $\neq 0 \vee$ `onChainBalance`($Alice$) is not equal to the total funds exclusively spendable by $Alice$ in Σ_{Alice}) **then**
    halt
  end if
end function

Fig. 21.

POLL is a request that every player has to make to $\mathcal{F}_{\text{PayNet}}$ periodically (once every `delay` blocks, as set on registration) in order to remain non-negligent. In a software implementation, such a request would be automatically sent at safe time intervals. When receiving POLL (Fig. 22), $\mathcal{F}_{\text{PayNet}}$ checks the ledger for maliciously closed channels and halts in case of a forgery (l. 6) or in case of a successful malicious closing of a channel whilst the offended player was non-negligent (l. 11). If on the other hand a channel has been closed maliciously but the offended player did not POLL in time, she is marked as negligent (l. 13).

Functionality $\mathcal{F}_{\text{PayNet}}$ – poll

1: Upon receiving (POLL) from *Alice*:
2: send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
3: add largest block number in Σ_{Alice} to `polls`(*Alice*)
4: `checkClosed`(Σ_{Alice})
5: **if** $\exists$`channel` $\in \Sigma_{Alice}$ that contains *Alice* and is closed by a tx that is not a commitment transaction **then**
6: halt // DS forgery
7: **end if**
8: **for all** `channel`s $\in \Sigma_{Alice}$ that contain *Alice* and are maliciously closed by a remote commitment `tx` (one with an older channel version than the irrevocably committed one) in block with height $h_{\texttt{tx}}$ **do**
9: **if** the delayed output (of the counterparty) has been spent **then**
10: **if** `polls`(*Alice*) has an element in $[h_{\texttt{tx}}, h_{\texttt{tx}} + \texttt{delay}\,(Alice) - 1]$ **then**
11: halt // *Alice* wasn't negligent but couldn't punish
12: **else**
13: `negligent`(*Alice*) $\leftarrow$ `true`
14: **end if**
15: **end if**
16: **end for**
17: send (POLL, Σ_{Alice}, *Alice*) to $\mathcal{S}$

Fig. 22.

The last part of $\mathcal{F}_{\text{PayNet}}$ (Fig. 23) contains some additional "daemon" messages that help various processes carry on. PUSHFULFILL, PUSHADD and COMMIT are simply forwarded to $\mathcal{S}$. They exist because the "token of execution" in the protocol does not follow the strict order required by UC, and thus some additional messages are needed for the protocol to carry on. In other words, they are needed due to the incompatibility of the serial execution of UC and the asynchronous nature of LN.

FULFILLONCHAIN has to be sent by a multi-hop payment intermediary that has not been paid by the previous player off-chain in order to close the chan-

nel. The request is noted and forwarded to $\mathcal{S}$. GETNEWS requests from $\mathcal{F}_{\text{PayNet}}$ information on newly opened, closed and updated channels.

Functionality $\mathcal{F}_{\text{PayNet}}$ – daemon messages

1: Upon receiving (PUSHFULFILL, *pchid*) from *Alice*:
2: send (PUSHFULFILL, *pchid*, *Alice*, STATE, Σ) to $\mathcal{S}$

3: Upon receiving (PUSHADD, *pchid*) from *Alice*:
4: send (PUSHADD, *pchid*, *Alice*, STATE, Σ) to $\mathcal{S}$

5: Upon receiving (COMMIT, *pchid*) from *Alice*:
6: send (COMMIT, *pchid*, *Alice*, STATE, Σ) to $\mathcal{S}$

7: Upon receiving (FULFILLONCHAIN) from *Alice*:
8: send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, store reply to Σ_{Alice} and assign largest block number to t
9: add t to `focs`(*Alice*)
10: `checkClosed`(Σ_{Alice})
11: send (FULFILLONCHAIN, t, *Alice*) to $\mathcal{S}$

12: Upon receiving (GETNEWS) from *Alice*:
13: copy members of `updatesToReport`(*Alice*) to `updatesHistory`(*Alice*)
14: clear `newChannels`(*Alice*), `closedChannels`(*Alice*), `updatesToReport`(*Alice*), `paymentsToReport`(*Alice*) and send them to *Alice* with message name NEWS, stripping *fchid* and h from `closedChannels`(*Alice*)

Fig. 23.

10 Lightning Protocol

Simiarly to Fig. 12 of $\mathcal{F}_{\text{PayNet}}$, the registration of a new player consists of the generation a new keypair, the topping up of the public key with some initial funds and the non-serving of any other messages until registration is complete. The main difference is that here the keypair is generated locally.

Protocol Π_{LN} (self is *Alice* always) – support

Initialisation:
    `channels`, `pendingOpen`, `pendingPay`, `pendingClose`, `paymentsToReport` $\leftarrow \emptyset$
    `newChannels`, `closedChannels`, `updatesToReport`, `gotPaid` $\leftarrow \emptyset$
    `unclaimedOfferedHTLCs`, `unclaimedReceivedHTLCs`, `pendingGetPaid` $\leftarrow \emptyset$

Upon receiving (REGISTER, delay, relayDelay) from $\mathcal{E}$:
    `delay` $\leftarrow$ delay // Must check chain at least once every `delay` blocks
    `relayDelay` $\leftarrow$ relayDelay
    send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
    $(pk_{Alice}, sk_{Alice}) \leftarrow$ KEYGEN ()
    send (REGISTER, *Alice*, `delay`, `relayDelay`, pk_{Alice}) to $\mathcal{E}$

Upon receiving (TOPPEDUP) from $\mathcal{E}$:
    send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
    assign the sum of all output values that are exclusively spendable by *Alice* to `onChainBalance`
    send (REGISTERED) to $\mathcal{E}$

Upon receiving any message (M) except for (REGISTER) or (TOPPEDUP):
    if if haven't received (REGISTER) and (TOPPEDUP) from $\mathcal{E}$ (in this order) **then**
        send (INVALID, M) to $\mathcal{E}$ and ignore message
    end if

function `GetKeys`
    $(p_F, s_F) \leftarrow$ KEYGEN () // For F output
    $(p_{\text{pay}}, s_{\text{pay}}) \leftarrow$ SETUP () // For com output to remote
    $(p_{\text{dpay}}, s_{\text{dpay}}) \leftarrow$ SETUP () // For com output to self
    $(p_{\text{htlc}}, s_{\text{htlc}}) \leftarrow$ SETUP () // For htlc output to self
    `seed` $\xleftarrow{\$} U(k)$ // For per com point
    $(p_{\text{rev}}, s_{\text{rev}}) \leftarrow$ MASTERKEYGEN () // For revocation in com
    return $((p_F, s_F), (p_{\text{pay}}, s_{\text{pay}}), (p_{\text{dpay}}, s_{\text{dpay}}),$
        $(p_{\text{htlc}}, s_{\text{htlc}}),$ `seed`, $(p_{\text{rev}}, s_{\text{rev}}))$
end function

Fig. 24.

When a player receives OPENCHANNEL from $\mathcal{E}$, she generates a number of base keypairs (funding, payment, delayed payment, HTLC), along with a seed for the per-update key shares, a master keypair and a share keypair used for revocation. She then sends the public keys along with the desired delay and initial funds to the counterparty as specified by $\mathcal{E}$ in a message labeled also OPENCHANNEL.

Protocol Π_{LN} – openChannel from $\mathcal{E}$

1: Upon receiving (openChannel, $Alice, Bob, x, tid$) from $\mathcal{E}$:
2: ensure tid hasn't been used for opening another channel before
3: // Keys marked with "b" are called basepoint keys. Their use reduces from three to one the public keys that have to be exchanged on each update.
4: $((ph_F, sh_F), (phb_{\text{pay}}, shb_{\text{pay}}), (phb_{\text{dpay}}, shb_{\text{dpay}}), (phb_{\text{htlc}}, shb_{\text{htlc}}), \texttt{seed}, (phb_{\text{rev}}, shb_{\text{rev}})) \leftarrow \texttt{GetKeys}()$
5: $\text{prand}_1 \leftarrow \text{PRF}(\texttt{seed}, 1)$
6: $(ph_{\text{com},1}, sh_{\text{com},1}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_1\right)$
7: associate keys with tid
8: add ($Alice$, Bob, x, $\texttt{tid}$, (ph_F, sh_F), $(phb_{\text{pay}}, shb_{\text{pay}})$, $(phb_{\text{dpay}}, shb_{\text{dpay}})$ $(phb_{\text{htlc}}, shb_{\text{htlc}})$, $(phb_{\text{com},1}, shb_{\text{com},1})$, $(phb_{\text{rev}}, shb_{\text{rev}})$, tid) to $\texttt{pendingOpen}$
9: send (openChannel, $x, \texttt{delay} + (2 + r)\,\texttt{windowSize}, ph_F, phb_{\text{pay}}, phb_{\text{dpay}}, phb_{\text{htlc}}, ph_{\text{com},1}, phb_{\text{rev}}, tid$) to Bob

Fig. 25.

When he receives openChannel by a funding party, the same steps are followed by the counterparty. The only difference is that he labels his reply with acceptChannel instead.

Protocol Π_{LN} – openChannel from Bob

1: Upon receiving (openChannel, x, $\texttt{remoteDelay}$, pt_F, ptb_{pay}, ptb_{dpay}, ptb_{htlc}, $pt_{\text{com},1}$, ptb_{rev}, tid) from Bob:
2: ensure tid has not been used yet with Bob
3: $((ph_F, sh_F), (phb_{\text{pay}}, shb_{\text{pay}}), (phb_{\text{dpay}}, shb_{\text{dpay}}), (phb_{\text{htlc}}, shb_{\text{htlc}}), \texttt{seed}$ $(phb_{\text{rev}}, shb_{\text{rev}})) \leftarrow \texttt{GetKeys}()$
4: $\text{prand}_1 \leftarrow \text{PRF}(\texttt{seed}, 1)$
5: $(ph_{\text{com},1}, sh_{\text{com},1}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_1\right)$
6: associate keys with tid and store in $\texttt{pendingOpen}$
7: send (acceptChannel, $\texttt{delay} + (2 + r)\,\texttt{windowSize}, ph_F, phb_{\text{pay}}, phb_{\text{dpay}}, phb_{\text{htlc}}, ph_{\text{com},1}, phb_{\text{rev}}, tid$) to Bob

Fig. 26.

When she receives acceptChannel by the counterparty, the funding party creates the funding transaction and the first commitment transaction, signs the funding transaction and sends this signature to the counterparty with a message labelled fundingCreated.

Protocol Π_{LN} – acceptChannel

1: Upon receiving (acceptChannel, `remoteDelay`, $pt_F, ptb_{\text{pay}}, ptb_{\text{dpay}}, ptb_{\text{htlc}}, pt_{\text{com},1}, ptb_{\text{rev}}$, *tid*) from *Bob*:
2: ensure there is a temporary ID *tid* with *Bob* in `pendingOpen` on which acceptChannel hasn't been received
3: associate received keys with *tid*
4: send (read) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
5: assign to `prevout` a transaction output found in Σ_{Alice} that is currently exclusively spendable by *Alice* and has value $y \geq x$
6: $F \leftarrow$ TX {input spends `prevout` with a SignDS(TX, sk_{Alice}), output 0 pays $y - x$ to pk_{Alice}, output 1 pays x to $tid.ph_F \wedge pt_F$}
7: $pchid \leftarrow \mathcal{H}(F)$
8: add *pchid* to `pendingOpen` entry with id *tid*
9: $pt_{\text{rev},1} \leftarrow \text{CombinePubKey}(ptb_{\text{rev}}, ph_{\text{com},1})$
10: $(ph_{\text{dpay},1}, sh_{\text{dpay},1}) \leftarrow \text{KeyDer}(phb_{\text{dpay}}, shb_{\text{dpay}}, ph_{\text{com},1})$
11: $(ph_{\text{pay},1}, sh_{\text{pay},1}) \leftarrow \text{KeyDer}(phb_{\text{pay}}, shb_{\text{pay}}, ph_{\text{com},1})$
12: $(ph_{\text{htlc},1}, sh_{\text{htlc},1}) \leftarrow \text{KeyDer}(phb_{\text{htlc}}, shb_{\text{htlc}}, ph_{\text{com},1})$
13: `remoteCom` $\leftarrow$ `remoteCom`$_1 \leftarrow$ TX {input: output 1 of F, outputs: $(x, ph_{\text{pay},1})$, $(0, ph_{\text{rev},1} \vee (pt_{\text{dpay},1}, \texttt{delay} + (2 + r)\,\texttt{windowSize}$ relative))}
14: `localCom` $\leftarrow$ TX {input: output 1 of F, outputs: $(x, pt_{\text{rev},1} \vee (ph_{\text{dpay},1}, \texttt{remoteDelay}$ relative)), $(0, pt_{\text{pay},1})$}
15: add `remoteCom` and `localCom` to channel entry in `pendingOpen`
16: sig $\leftarrow$ SignDS(`remoteCom`$_1, sh_F$)
17: `lastRemoteSigned` $\leftarrow 0$
18: send (fundingCreated, *tid*, *pchid*, sig) to *Bob*

Fig. 27.

When he receives fundingCreated by the funding party, the counterparty creates the funding transaction and then checks the validity of the received signature. If it is valid, he then creates the first commitment transaction as well, signs the funding transaction and sends this signature to the funding party with a message labelled fundingSigned.

Protocol Π_{LN} – fundingCreated

1: Upon receiving (fundingCreated, *tid*, *pchid*, $\texttt{BobSig}_1$) from *Bob*:
2: ensure there is a temporary ID *tid* with *Bob* in `pendingOpen` on which we have sent up to acceptChannel
3: $ph_{\text{rev},1} \leftarrow \text{CombinePubKey}\,(phb_{\text{rev}} pt_{\text{com},1})$
4: $pt_{\text{dpay},1} \leftarrow \text{PubKeyDer}\,(ptb_{\text{dpay}}, pt_{\text{com},1})$
5: $pt_{\text{pay},1} \leftarrow \text{PubKeyDer}\,(ptb_{\text{pay}}, pt_{\text{com},1})$
6: $pt_{\text{htlc},1} \leftarrow \text{PubKeyDer}\,(ptb_{\text{htlc}}, pt_{\text{com},1})$
7: $\texttt{localCom} \leftarrow \texttt{localCom}_1 \leftarrow$ TX {input: output 1 of F, outputs: $(x, pt_{\text{pay},1})$, $(0, pt_{\text{rev},1} \vee (ph_{\text{dpay},1}, \texttt{remoteDelay}\text{ relative})$}
8: ensure $\text{VerifyDS}\,(\texttt{BobSig}_1, \texttt{localCom}_1, pt_F) = \texttt{True}$
9: $\texttt{remoteCom} \leftarrow \texttt{remoteCom}_1 \leftarrow$ TX {input: output 1 of F, outputs: $(x, ph_{\text{rev},1} \vee (pt_{\text{dpay},1}, \texttt{delay} + (2 + r)\,\texttt{windowSize}\text{ relative}))$, $(0, ph_{\text{pay},1})$}
10: add $\texttt{BobSig}_1$, $\texttt{remoteCom}_1$ and $\texttt{localCom}_1$ to channel entry in `pendingOpen`
11: sig $\leftarrow \text{SignDS}\,(\text{remoteCom}_1, sh_F)$
12: mark channel as "broadcast, no fundingLocked"
13: $\texttt{lastRemoteSigned}, \texttt{lastLocalSigned} \leftarrow 0$
14: send (fundingSigned, *pchid*, sig) to *Bob*

Fig. 28.

When the funding party receives fundingSigned, she verifies the validity of the received signature. If it is valid, it broadcasts the funding transaction.

Protocol Π_{LN} – fundingSigned

1: Upon receiving (fundingSigned, *pchid*, $\texttt{BobSig}_1$) from *Bob*:
2: ensure there is a channel ID *pchid* with *Bob* in `pendingOpen` on which we have sent up to fundingCreated
3: ensure $\text{VerifyDS}\,(\texttt{BobSig}_1, \texttt{localCom}, pt_F) = \texttt{True}$
4: $\texttt{localCom}_1 \leftarrow \texttt{localCom}$
5: $\texttt{lastLocalSigned} \leftarrow 0$
6: add $\texttt{BobSig}_1$ to channel entry in `pendingOpen`
7: sig $\leftarrow \text{SignDS}\,(F, sk_{Alice})$
8: mark *pchid* in `pendingOpen` as "broadcast, no fundingLocked"
9: send (submit, (sig, F)) to $\mathcal{G}_{\text{Ledger}}$

Fig. 29.

When either party (say *Alice*) receives checkForNew from $\mathcal{E}$, she checks if the funding transaction is in the ledger. If it is, it generates the keyshare for the next update and sends it to *Bob* in a message labelled fundingLocked. When *Alice* receives a similar message from *Bob*, she considers the channel open.

Protocol Π_{LN} – checkForNew

1: Upon receiving (checkForNew, *Alice*, *Bob*, *tid*) from $\mathcal{E}$: // lnd polling daemon
2: ensure there is a matching `channel` in `pendingOpen` with id *pchid*, with a "broadcast" and a "no fundingLocked" mark, funded with x coins
3: send (read) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
4: ensure $\exists$ unspent TX in Σ_{Alice} with ID *pchid* and a $(x, ph_F \wedge pt_F)$ output
5: $\text{prand}_2 \leftarrow \text{PRF}(\texttt{seed}, 2)$
6: $(ph_{\text{com},2}, sh_{\text{com},2}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_2\right)$
7: add TX to `channel` data
8: replace "broadcast" mark in `channel` with "fundingLocked sent"
9: send (fundingLocked, *pchid*, $ph_{\text{com},2}$) to *Bob*

Fig. 30.

Protocol Π_{LN} – fundingLocked

1: Upon receiving (fundingLocked, *pchid*, $pt_{\text{com},2}$) from *Bob*:
2: ensure there is a `channel` with ID *pchid* with *Bob* in `pendingOpen` with a "no fundingLocked" mark
3: **if** `channel` is not marked with "fundingLocked sent" **then** // i.e. marked with "broadcast"
4: send (read) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
5: ensure $\exists$ unspent TX in Σ_{Alice} with ID *pchid* and a $(x, ph_F \wedge pt_F)$ output
6: add TX to `channel` data
7: $\text{prand}_2 \leftarrow \text{PRF}(\texttt{seed}, 2)$
8: $(ph_{\text{com},2}, sh_{\text{com},2}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_2\right)$
9: generate 2nd remote delayed payment, htlc, payment keys
10: **end if**
11: replace "no fundingLocked" mark in `channel` with "fundingLocked received"
12: move channel data from `pendingOpen` to `channels`
13: add receipt of channel to `newChannels`, where $\text{receipt} \leftarrow (Alice : x, Bob : 0, pchid)$
14: **if** `channel` is not marked with "fundingLocked sent" **then**
15: replace "broadcast" mark in `channel` with "fundingLocked sent"
16: send (fundingLocked, *pchid*, $ph_{\text{com},2}$) to *Bob*
17: **end if**

Fig. 31.

When a player receives poll, she checks the ledger for closed channels and acts upon them. In particular, she retrieves funds from failed HTLC payments

and punishes counterparties that closed their channel maliciously. She also takes note of honestly closed channels. When she receives GETNEWS, she sends back a list of all unreported channels that opened or closed, along with payments that were carried out successfully.

Protocol Π_{LN} – poll

Upon receiving (POLL) from $\mathcal{E}$:
  send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
  `toSubmit` $\leftarrow \emptyset$
  for all $\tau \in$ `unclaimedOfferedHTLCs` **do**
    if input of τ has been spent **then** // by `remote HTLC-success`
      remove τ from `unclaimedOfferedHTLCs`
      if we are intermediary **then**
        retrieve preimage R, $pchid'$ of previous channel on the path of the HTLC, and `HTLCNo`$'$ of the corresponding HTLC$'$ in $pchid'$
        add (`HTLCNo`$'$, R) to $\texttt{pendingFulfills}_{pchid'}$
      end if
    else if input of τ has not been spent and timelock is over **then**
      remove τ from `unclaimedOfferedHTLCs`
      add τ to `toSubmit`
    end if
  end for
  run loop of Fig. 33
  for all honestly closed $\texttt{remoteCom}_n$ that were processed above, with channel id $pchid$ **do**
    for all received HTLC outputs i of $\texttt{remoteCom}_n$ **do**
      if there is an entry in $\texttt{pendingFulfills}_{pchid}$ with the same `HTLCNo` and R **then**
        TX $\leftarrow$ {input: i HTLC output of $\texttt{remoteCom}_n$ with $(ph_{\text{htlc},n}, R)$ as method, output: pk_{Alice}}
        sig $\leftarrow$ SIGNIBS $(TX, sh_{\text{htlc},n})$
        add (sig, TX) to `toSubmit`
        remove entry from $\texttt{pendingFulfills}_{pchid}$
      end if
    end for
  end for
  send (SUBMIT, `toSubmit`) to $\mathcal{G}_{\text{Ledger}}$

Upon receiving (GETNEWS) from $\mathcal{E}$:
  clear `newChannels`, `closedChannels`, `updatesToReport`, `paymentsToReport` and send them to $\mathcal{E}$ with message name NEWS

Fig. 32.

Loop over closed channels for poll

for all $\texttt{remoteCom}_n \in \Sigma_{Alice}$ that spend F of a `channel` $\in$ `channels` **do**
  if we do not have $sh_{\text{rev},n}$ **then** // Honest closure
    for all unspent offered HTLC outputs i of $\texttt{remoteCom}_n$ **do**
      TX $\leftarrow$ {input: i HTLC output of $\texttt{remoteCom}_n$ with $ph_{\text{htlc},n}$ as method, output: pk_{Alice}}
      sig $\leftarrow$ SIGNIBS $(TX, sh_{\text{htlc},n})$
      if timelock has not expired **then**
        add (sig, TX) to `unclaimedOfferedHTLCs`
      else if timelock has expired **then**
        add (sig, TX) to `toSubmit`
      end if
    end for
    for all spent offered HTLC output i of $\texttt{remoteCom}_n$ **do**
      if we are intermediary **then**
        retrieve preimage R, $pchid'$ of previous channel on the path of the HTLC, and $\texttt{HTLCNo}'$ of the corresponding HTLC$'$ in $pchid'$
        add $(\texttt{HTLCNo}', R)$ to $\texttt{pendingFulfills}_{pchid'}$
      else// we are the payer
        retrieve ($expayid$, x) from entry in `pendingPay` with h, `HTLCNo` of HTLC_i and remove the entry
        add ($expayid$, $-x$) to `paymentsToReport`
      end if
    end for
  else // malicious closure
    rev $\leftarrow$ TX {inputs: all $\texttt{remoteCom}_n$ outputs, choosing $ph_{\text{rev},n}$ method, output: pk_{Alice}}
    sig $\leftarrow$ SIGNCS $(\text{rev}, sh_{\text{rev},n})$
    add (sig, rev) to `toSubmit`
  end if
  add `receipt`(`channel`) to `closedChannels`
  remove `channel` from `channels`
end for

Fig. 33.

When a player (say *Alice*) receives PAY along with a payee (say *Bob*), a payment amount and a path, she informs *Bob* of the upcoming payment with a SENDINVOICE message. He then generates a secret preimage and sends back its hash and his desired minimum slack between the present and the moment he has to disclose the preimage (known as "relay delay") in a message labelled INVOICE. Subsequently *Alice* prepares a Sphinx onion packet [31] with one message for each path member, taking into account each hop's desired relay delay. Afterwards she creates an HTLC that transfers the payment amount and adds it to her channel with the first path member. She then sends the onion, the payment amount and the hash to the first hop in a message labelled UPDATEADDHTLC.

Protocol Π_{LN} – pay

1: Upon receiving (PAY, *Bob*, x, $\overrightarrow{\texttt{path}}$, *expayid*) from $\mathcal{E}$:
2: ensure that $\overrightarrow{\texttt{path}}$ consists of syntactically valid (*pchid*, `CltvExpiryDelta`) pair // Payment completes only if $\forall$ honest $i \in \overrightarrow{\texttt{path}}, \texttt{CltvExpiryDelta}_i \geq 3k + \texttt{RelayDelay}_i$
3: ensure that the first *pchid* $\in \overrightarrow{\texttt{path}}$ corresponds to an open `channel` $\in$ `channels` in which we own at least x in the irrevocably committed state.
4: choose unique payment ID *payid* // unique for *Alice* and *Bob*
5: add (*Bob*, x, $\overrightarrow{\texttt{path}}$, *expayid*, *payid*, "waiting for invoice") to `pendingPay`
6: send (SENDINVOICE, x, *expayid*, *payid*) to *Bob*

7: Upon receiving (SENDINVOICE, x, *expayid*, *payid*) from *Bob*:
8: ensure there is no (*Bob*, _, *expayid*, *payid*, _) entry in `pendingGetPaid`
9: choose random, unique preimage R
10: add (*Bob*, R, *expayid*, *payid*, x) to `pendingGetPaid`
11: send (INVOICE, $\mathcal{H}(R)$, `relayDelay` $+ (2 + r)$ `windowSize`, *payid*) to *Bob*

Fig. 34.

Protocol Π_{LN} – invoice

1: Upon receiving (INVOICE, h, `minFinalCltvExpiry`, *payid*) from *Bob*:
2: ensure there is a (*Bob*, x, $\overrightarrow{\texttt{path}}$, _, *payid*, "waiting for invoice") entry in `pendingPay`
3: ensure h is valid (in the range of $\mathcal{H}$)
4: remove mark from, add h and `HTLCNo` to entry in `pendingPay`
5: retrieve `CltvExpiryDelta`s from $\overrightarrow{\texttt{path}}$
6: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
7: $l \leftarrow |\left(\overrightarrow{\texttt{path}}\right)|$
8: $\texttt{CltvExpiry}_l \leftarrow t + \texttt{minFinalCltvExpiry}$
9: $\forall i \in \{1, \ldots, l-1\}, \texttt{CltvExpiry}_{l-i} \leftarrow \texttt{CltvExpiry}_{l-i+1} + \texttt{CltvExpiryDelta}_{l-i+1}$
10: ensure $\texttt{CltvExpiry}_1 \geq \texttt{CltvExpiry}_2 + \texttt{relayDelay} + (2+r)\,\texttt{windowSize}$
11: $m \leftarrow$ the concatenation of $l\,(x, \texttt{CltvExpiry})$
12: $(\mu_0, \delta_0) \leftarrow \texttt{SphinxCreate}\left(m,\ \text{public keys of } \overrightarrow{\texttt{path}} \text{ parties}\right)$
13: let $\texttt{remoteCom}_n$ the latest signed remote commitment tx with first $\overrightarrow{\texttt{path}}$ member
14: reduce simple payment output in `remoteCom` by x
15: add an additional (x, $ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1}$, on preimage of $h) \vee ph_{\text{htlc},n+1}$, $\texttt{CltvExpiry}_1$ absolute) output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`
16: reduce delayed payment output in `localCom` by x
17: add an additional (x, $pt_{\text{rev},n+1} \vee (pt_{\text{htlc},n+1}$, on preimage of $h) \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1}$, $\texttt{CltvExpiry}_1$ absolute)) output (all with $n+1$ keys) to `localCom`, marked with `HTLCNo`
18: increment $\texttt{HTLCNo}_{pchid}$ by one and associate $x, h, pchid$ with it
19: mark `HTLCNo` as "sender"
20: send (UPDATEADDHTLC, first *pchid* of $\overrightarrow{\texttt{path}}, \texttt{HTLCNo}_{pchid}, x, h, \texttt{CltvExpiry}_1, (\mu_0, \delta_0)$) to *pchid* channel counterparty

Fig. 35.

When a player receives UPDATEADDHTLC, she peels the outermost onion layer and extracts the CLTV expiry, the next channel ID (or $\perp$ if she is the payee) and the payment amount. She checks that the CLTV expiry is within her desired relay delay and adds to the incoming channel an HTLC that pays her the payment amount. If she is the payee, she prepares to disclose the preimage. Otherwise, she creates an HTLC that transfers the payment amount from her to the next party of the path, prepares the next onion and takes a note to send it to the next party.

Protocol Π_{LN} – UPDATEADDHTLC

1: Upon receiving (UPDATEADDHTLC, *pchid*, `HTLCNo`, x, h, `IncomingCltvExpiry`, M) from *Bob*:
2: run code of Fig. 37 – UPDATEADDHTLC checks
3: increment $\texttt{HTLCNo}_{pchid}$ by one
4: let $\texttt{remoteCom}_n$ the latest signed remote commitment tx
5: reduce delayed payment output in `remoteCom` by x
6: add an (x, $ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1}$, `IncomingCltvExpiry` absolute) $\vee\ ph_{\text{htlc},n+1}$, on preimage of h) htlc output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`
7: reduce simple payment output in `localCom` by x
8: add an (x, $pt_{\text{rev},n+1} \vee (pt_{\text{htlc},n+1}$, `IncomingCltvExpiry` absolute) $\vee$ ($pt_{\text{htlc},n+1} \wedge ph_{\text{htlc},n+1}$, on preimage of h)) htlc output (all with $n+1$ keys) to `localCom`, marked with `HTLCNo`
9: **if** $\delta = \texttt{receiver}$ **then**
10: retrieve $R : \mathcal{H}(R) = h$ from `pendingGetPaid` and move entry to `gotPaid`
11: add $(\texttt{HTLCNo}, R)$ to $\texttt{pendingFulfills}_{pchid}$
12: **else if** $\delta \neq \texttt{receiver}$ **then** // Send HTLC to next hop
13: retrieve $pchid'$ data
14: let $\texttt{remoteCom}'_n$ the latest signed remote commitment tx
15: reduce simple payment output in $\texttt{remoteCom}'$ by x
16: add an additional $\big(x, ph'_{\text{rev},n+1} \vee \big(ph'_{\text{htlc},n+1} \wedge pt'_{\text{htlc},n+1},$ on preimage of $h) \vee ph'_{\text{htlc},n+1}, \texttt{OutgoingCltvExpiry}$ absolute$\big)$ output (all with $n+1$ keys) to $\texttt{remoteCom}'$, marked with $\texttt{HTLCNo}'$
17: reduce delayed payment output in $\texttt{localCom}'$ by x
18: add an additional (x, $pt'_{\text{rev},n+1} \vee (pt'_{\text{htlc},n+1}$, on preimage of $h) \vee$ ($pt'_{\text{htlc},n+1} \wedge ph'_{\text{htlc},n+1}$, `OutgoingCltvExpiry` absolute)) output (all with $n+1$ keys) to $\texttt{localCom}'$, marked with $\texttt{HTLCNo}'$
19: increment $\texttt{HTLCNo}'$ by 1
20: $M' \leftarrow \texttt{SphinxPrepare}(M, \delta, sk_{Alice})$
21: add $(\texttt{HTLCNo}', x, h, \texttt{OutgoingCltvExpiry}, M')$ to $\texttt{pendingAdds}_{pchid'}$
22: **end if**

Fig. 36.

Protocol Π_{LN} – updateAddHtlc checks

1: ensure *pchid* corresponds to an open `channel` in `channels` where *Bob* has at least x
2: ensure $\texttt{HTLCNo} = \texttt{HTLCNo}_{pchid} + 1$
3: $(pchid', x', \texttt{OutgoingCltvExpiry}, \delta) \leftarrow \texttt{SphinxPeel}\,(sk_{Alice}, M)$
4: send (read) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
5: **if** $\delta = \texttt{receiver}$ **then**
6: ensure $pchid' = \bot, x = x', \texttt{IncomingCltvExpiry} \geq \texttt{OutgoingCltvExpiry} = \texttt{minFinalCltvExpiry}$
7: mark `HTLCNo` as "receiver"
8: **else** // We are an intermediary
9: ensure $x = x', \texttt{IncomingCltvExpiry} \geq \max\{\texttt{OutgoingCltvExpiry}, t\} + \texttt{relayDelay} + 2\,(2 + r)\,\texttt{windowSize}$
10: ensure $\texttt{pchid}'$ corresponds to an open channel in `channels` where we have at least x
11: mark `HTLCNo` as "intermediary"
12: **end if**

Fig. 37.

When a player receives a preimage for a particular HTLC in an updateFulfillHtlc message, she verifies the validity of the preimage, she removes the HTLC and incorporates the new balance to the commitment transactions and, if she is an intermediary, she sends the preimage to the previous party on the path if the channel is still open, or publishes the preimage on-chain otherwise.

Protocol Π_{LN} – updateFulfillHtlc

1: Upon receiving (updateFulfillHtlc, *pchid*, `HTLCNo`, R) from *Bob*:
2: **if** `HTLCNo` $>$ `lastRemoteSigned` $\vee$ `HTLCNo` $>$ `lastLocalSigned` $\vee\, \mathcal{H}(R) \neq h$, where h is the hash in the HTLC with number `HTLCNo` **then**
3: close channel (as in Fig. 44)
4: **return**
5: **end if**
6: ensure `HTLCNo` is an offered HTLC (`localCom` has h tied to a public key that we own)
7: add value of HTLC to delayed payment of `remoteCom`
8: remove HTLC output with number `HTLCNo` from `remoteCom`
9: add value of HTLC to simple payment of `localCom`
10: remove HTLC output with number `HTLCNo` from `localCom`
11: **if** we have a channel *phcid*′ that has a received HTLC with hash h with number `HTLCNo`′ **then** // We are an intermediary
12: send (read) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
13: **if** latest $\texttt{remoteCom}'_n \in \Sigma_{Alice}$ **then** // counterparty has gone on-chain
14: TX $\leftarrow$ {input: (`remoteCom`′ HTLC output with number `HTLCNo`′, R), output: pk_{Alice}}
15: sig $\leftarrow$ SignIBS(TX, $sh_{\text{htlc},n}$)
16: send (submit, (sig, TX)) to $\mathcal{G}_{\text{Ledger}}$ // shouldn't be already spent by remote `HTLCTimeout`
17: **else** // counterparty still off-chain
18: // Not having the HTLC irrevocably committed is impossible (Fig. 42, l. 22)
19: send (updateFulfillHtlc, *pchid*′, `HTLCNo`′, R) to counterparty
20: **end if**
21: **else**// We are the payer
22: retrieve (*expayid*, x) from entry in `pendingPay` with h, `HTLCNo` and remove the entry // entry is unique because of `HTLCNo`
23: add (*expayid*, $-x$) to `updateDiffs`(`channel`) of `channel` with ID *pchid*
24: **end if**

Fig. 38.

When a player receives commit from $\mathcal{E}$ accompanied by a channel ID, she generates and signs a new commitment tx and all new HTLC txs that correspond to unsigned added HTLCs. She then sends the signatures to the channel counterparty in a message labelled commitmentSigned.

Protocol Π_{LN} – COMMIT

1: Upon receiving (COMMIT, *pchid*) from $\mathcal{E}$:
2: ensure that there is a `channel` $\in$ `channels` with ID *pchid*
3: retrieve latest remote commitment tx $\texttt{remoteCom}_n$ in `channel`
4: ensure `remoteCom` $\neq \texttt{remoteCom}_n$ // there are uncommitted updates
5: ensure `channel` is not marked as "waiting for REVOKEANDACK"
6: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
7: undo adding all outgoing HTLCs in `remoteCom` for which we are intermediary and `IncomingCltvExpiry` $< t +$ `relayDelay` $+ (2 + r)$ `windowSize`
8: $\texttt{remoteCom}_{n+1} \leftarrow$ `remoteCom`
9: ComSig $\leftarrow$ SIGNDS $(\texttt{remoteCom}_{n+1}, sh_F)$
10: HTLCSigs $\leftarrow \emptyset$
11: **for** i from `lastRemoteSigned` + 1 to `HTLCNo` **do**
12: $\texttt{remoteHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of $\texttt{remoteCom}_{n+1}$, output: ($c_{\text{htlc,i}}$, $ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}$, `delay` $+ (2 + r)$ `windowSize` relative))}
13: add SIGNIBS $(\texttt{remoteHTLC}_{n+1,i}, sh_{\text{htlc},n+1})$ to HTLCSigs
14: **end for**
15: `lastRemoteSigned` $\leftarrow$ `HTLCNo`
16: mark `channel` as "waiting for REVOKEANDACK"
17: send (COMMITMENTSIGNED, *pchid*, ComSig, HTLCSigs) to *pchid* counterparty

Fig. 39.

When a player receives COMMITMENTSIGNED with some signatures for a particular channel, she verifies that the signatures correspond to the expected HTLCs (i.e. the added but unsigned ones) and that the signatures for the new commitment tx and said HTLCs are valid. She then generates the keyshare pair and sends its public part to the counterparty along with the secret part of the old keyshare in a message labelled REVOKEANDACK.

Protocol Π_{LN} – COMMITMENTSIGNED

Upon receiving (COMMITMENTSIGNED, *pchid*, $\texttt{comSig}_{n+1}$, $\texttt{HTLCSigs}_{n+1}$) from *Bob*:
  ensure that there is a $\texttt{channel} \in \texttt{channels}$ with ID *pchid* with *Bob*
  retrieve latest local commitment tx $\texttt{localCom}_n$ in `channel`
  ensure $\texttt{localCom} \neq \texttt{localCom}_n$ and $\texttt{localCom} \neq \texttt{pendingLocalCom}$ // there are uncommitted updates
  if VERIFYDS $\left(\texttt{comSig}_{n+1}, \texttt{localCom}, pt_F\right) = \texttt{false} \vee |\texttt{HTLCSigs}_{n+1}| \neq \texttt{HTLCNo} - \texttt{lastLocalSigned}$ **then**
    close channel (as in Fig. 44)
    return
  end if
  for i from `lastLocalSigned` + 1 to `HTLCNo` **do**
    $\texttt{localHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of `localCom`, output: ($c_{\text{htlc,i}}$, $ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}$, `remoteDelay` relative))}
    if VERIFYIBS($\texttt{HTLCSigs}_{n+1,i}$, $\texttt{localHTLC}_{n+1,i}$, $pt_{\text{htlc},n+1}$) = `false` **then**
      close channel (as in Fig. 44)
      return
    end if
  end for
  $\texttt{lastLocalSigned} \leftarrow \texttt{HTLCNo}$
  $\texttt{pendingLocalCom} \leftarrow \texttt{localCom}$
  mark `pendingLocalCom` as "irrevocably committed"
  $\text{prand}_{n+2} \leftarrow \text{PRF}\left(\texttt{seed}, n+2\right)$
  $(ph_{\text{com},n+2}, sh_{\text{com},n+2}) \leftarrow$ KEYSHAREGEN $\left(1^k; \text{prand}_{n+2}\right)$
  send (REVOKEANDACK, *pchid*, $\text{prand}_n, ph_{\text{com},n+2}$) to *Bob*

Fig. 40.

When a player receives REVOKEANDACK from a counterparty for a particular channel, she checks that the secret keyshare received corresponds to the public keyshare that had been received in the last REVOKEANDACK (or in the FUNDINGLOCKED if this is the first REVOKEANDACK received for this channel) and then generates all the necessary new keys for the next update. There are now no more outstanding updates. Furthermore, if a closing of the channel is pending, execution continues with the closing sequence.

Protocol Π_{LN} – REVOKEANDACK

1: Upon receiving (REVOKEANDACK, *pchid*, $st_{\text{com},n}, pt_{\text{com},n+2}$) from *Bob*:
2: ensure there is a `channel` ∈ `channels` with *Bob* with ID *pchid* marked as "waiting for REVOKEANDACK"
3: **if** TESTKEY $(pt_{\text{com},n}, st_{\text{com},n}) \neq 1$ **then** // wrong $st_{\text{com},n}$ - closing
4: close channel (as in Fig. 44)
5: **return**
6: **end if**
7: mark $\texttt{remoteCom}_{n+1}$ as "irrevocably committed"
8: $\texttt{localCom}_{n+1} \leftarrow \texttt{pendingLocalCom}$
9: unmark `channel`
10: append `receipt`(`channel`) to `updatesToReport`
11: **for all** (_, _, *payid*, `ref`) ∈ `updateDiffs`(`channel`) **do**
12: remove from `gotPaid` the entry referenced by `ref`
13: strip `ref` and *payid* from `updateDiffs`(`channel`) entry
14: **end for**
15: **if** `updateDiffs`(`channel`) $\neq \emptyset$ **then**
16: add `updateDiffs`(`channel`) members to `paymentsToReport`
17: **end if**
18: `updateDiffs`(`channel`) $\leftarrow \emptyset$
19: $sh_{\text{rev},n} \leftarrow$ COMBINEKEY $(phb_{\text{rev}}, shb_{\text{rev}}, pt_{\text{com}n}, st_{\text{com},n})$
20: $ph_{\text{rev},n+2} \leftarrow$ COMBINEPUBKEY $(phb_{\text{rev}}, pt_{\text{com},n+2})$
21: $pt_{\text{rev},n+2} \leftarrow$ COMBINEPUBKEY $(ptb_{\text{rev}}, ph_{\text{com},n+2})$
22: $(ph_{\text{dpay},n+2}, sh_{\text{dpay},n+2}) \leftarrow$ KEYDER $(phb_{\text{dpay}}, shb_{\text{dpay}}, ph_{\text{com},n+2})$
23: $pt_{\text{dpay},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{dpay}}, pt_{\text{com},n+2})$
24: $(ph_{\text{pay},n+2}, sh_{\text{pay},n+2}) \leftarrow$ KEYDER $(phb_{\text{pay}}, shb_{\text{pay}}, ph_{\text{com},n+2})$
25: $pt_{\text{pay},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{pay}}, pt_{\text{com},n+2})$
26: $(ph_{\text{htlc},n+2}, sh_{\text{htlc},n+2}) \leftarrow$ KEYDER $(phb_{\text{htlc}}, shb_{\text{htlc}}, ph_{\text{com},n+2})$
27: $pt_{\text{htlc},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{htlc}}, pt_{\text{com},n+2})$
28: **if** no outstanding HTLCs remain for this channel and the sequence for CLOSECHANNEL or SHUTDOWN (Fig. 45) has been initiated **then**
29: continue execution at Fig. 45, l. 7 or l. 16 respectively
30: **end if**

Fig. 41.

The following code defines a player's actions when she receives from $\mathcal{E}$ one of the three messages needed to "nudge" her to carry on with some particular part of the protocol. PUSHFULFILL makes the player read a preimage of a hash from $\mathcal{G}_{\text{Ledger}}$ and use it to fulfill the relevant HTLC that pays her, PUSHADD has the player send an UPDATEADDHTLC message that is ready but has not yet been sent and FULFILLONCHAIN requests that the player publish all HTLC txs that are about to expire on-chain. A player that does not receive such messages in time may end up losing funds albeit being honest; such a player is dubbed "negligent".

Protocol Π_{LN} – PUSH

Upon receiving (pushFulfill, *pchid*) from $\mathcal{E}$:
  ensure that there is a `channel` $\in$ `channels` with ID *pchid*
  choose a member (`HTLCNo`, R) of $\texttt{pendingFulfills}_{pchid}$ that is both in an "irrevocably committed" $\texttt{remoteCom}_n$ and $\texttt{localCom}_n$
  send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
  remove (`HTLCNo`, R) from $\texttt{pendingFulfills}_{pchid}$
  if $\texttt{remoteCom}_n \notin \Sigma_{Alice}$ **then** // counterparty cooperative
    if $\exists(\_, R, expayid, payid, x) \in$ `gotPaid` **then** // receiver
      add ($expayid, x$, $payid$, reference to `gotPaid` entry) to `updateDiffs`(`channel`)
    end if
    send (updateFulfillHtlc, *pchid*, `HTLCNo`, R) to *pchid* counterparty
  else // counterparty gone on-chain
    if $\exists(\_, R, expayid, payid, x) \in$ `gotPaid` **then**
      remove entry from `gotPaid`
      add ($expayid, x$) to `paymentsToReport`
    end if
    TX $\leftarrow$ {input: ($\texttt{remoteCom}_n$ HTLC output with number `HTLCNo`, R), output: pk_{Alice}}
    sig $\leftarrow$ SignIBS (TX, $sh_{\text{htlc},n}$)
    send (submit, (sig, TX)) to $\mathcal{G}_{\text{Ledger}}$ // shouldn't be already spent by remote `HTLCTimeout`
  end if

Upon receiving (pushAdd, *pchid*) from $\mathcal{E}$:
  ensure that there is a `channel` $\in$ `channels` with ID *pchid*
  choose a member (`HTLCNo`, $x, h,$ `CltvExpiry`, M) of $\texttt{pendingAdds}_{pchid}$ that is both in an "irrevocably committed" $\texttt{remoteCom}_n$ and $\texttt{localCom}_n$
  remove chosen entry from $\texttt{pendingAdds}_{pchid}$
  send (updateAddHtlc, *pchid*, `HTLCNo`, $x, h,$ `CltvExpiry`, M) to *pchid* counterparty

Fig. 42.

Protocol Π_{LN} – FULFILLONCHAIN

1: Upon receiving (FULFILLONCHAIN) from $\mathcal{E}$:
2: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
3: `toSubmit` $\leftarrow \emptyset$
4: **for all** channels **do**
5: **if** there exist HTLCs with hashes $h_1, \ldots, h_k$ in latest $\texttt{localCom}_n$ for which we have sent both UPDATEFULFILLHTLC and COMMITMENTSIGNED to a transaction without those HTLCs to counterparty, but have not received the corresponding REVOKEANDACKS AND some of the HTLCs expire within $[t, t + (2 + r)\,\texttt{windowSize}]$ **then**
6: add $\texttt{localCom}_n$ of the channel and all corresponding valid `HTLC-success`es and `HTLC-timeout`s (for both $\texttt{localCom}_n$ and $\texttt{remoteCom}_n$[a]), along with their signatures to `toSubmit`
7: **for all** such HTLCs for which we are the payee (i.e., we have received a (SENDINVOICE, x, *expayid*, *payid*) message in response to which we generated and sent h_i) **do**
8: add (*expayid*, x) to `paymentsToReport`
9: **end for**
10: **end if**
11: **end for**
12: send (SUBMIT, `toSubmit`) to $\mathcal{G}_{\text{Ledger}}$

[a] Ensures funds retrieval if counterparty has gone on-chain

Fig. 43.

When a player receives FORCECLOSECHANNEL for a particular channel, she signs and publishes the latest commitment and HTLC txs for this particular channel, in effect unilateraly closing the channel.

Protocol Π_{LN} – close unilaterally

Upon receiving (FORCECLOSECHANNEL, `receipt`, *pchid*) from $\mathcal{E}$:
  ensure `receipt` corresponds to an open `channel` $\in$ `channels` with ID *pchid*
  if the sequence for CLOSECHANNEL has been initiated and is pending on clearing all outstanding HTLCs **then**
    forget this "hook"
  end if
  assign latest `channel` sequence number to n
  HTLCs $\leftarrow \emptyset$
  for every HTLC output $\in$ $\texttt{localCom}_n$ with number i **do**
    if there is a (_, R, *expayid*, *payid*, x) entry in `gotPaid` for R the preimage of HTLC_i **then**
      remove entry from `gotPaid`
      add (*expayid*, x) to `paymentsToReport`
    end if
    sig $\leftarrow$ SIGNIBS $(\texttt{localHTLC}_{n,i}, sh_{\text{htlc},n})$
    add $\left(\text{sig}, \texttt{HTLCSigs}_{n,i}, \texttt{localHTLC}_{n,i}\right)$ to HTLCs
  end for
  sig $\leftarrow$ SIGNDS $(\texttt{localCom}_n, sh_F)$
  add `receipt`(`channel`) to `closedChannels`
  remove `channel` from `channels`
  send (SUBMIT, $(\text{sig}, \texttt{remoteSig}_n, \texttt{localCom}_n)$, HTLCs) to $\mathcal{G}_{\text{Ledger}}$

Fig. 44.

When a player receives CLOSECHANNEL for a particular channel, she initiates the cooperative channel close sequence with the counterparty. The sequence includes completing all outstanding updates and creating, signing and publishing the "closing" transaction that spends the funding transaction and distributes the funds to the counterparties without timelocks.

Protocol Π_{LN} – close cooperatively

Upon receiving (closeChannel, `receipt`, *pchid*) from $\mathcal{E}$:
  ensure `receipt` corresponds to an open `channel` $\in$ `channels` with ID *pchid*
  stop serving any (pay, closeChannel) message from $\mathcal{E}$ and any (UpdateAddHtlc, sendInvoice) message from counterparty for this channel.
  mark `channel` as "coop closing"
  if there are outstanding HTLC outputs in the latest $\texttt{localCom}_n$ **then**
    continue from here when there are none left
  end if
  send (shutdown, pk_{Alice}, *pchid*) to *Bob*

Upon receiving (shutdown, pk_{Bob}, *pchid*) from *Bob*:
  ensure there is an open `channel` $\in$ `channels` with *Bob* with ID *pchid*
  if `channel` is not marked "coop closing" **then**
    stop serving any (pay, closeChannel) message from $\mathcal{E}$ and any (UpdateAddHtlc, sendInvoice) message from counterparty for this channel.
    mark channel as "coop closing"
    if there are outstanding HTLC outputs in the latest $\texttt{localCom}_n$ **then**
      continue from here when there are none left
    end if
    send (shutdown, pk_{Alice}, *pchid*) to *Bob*
  else
    $Cl \leftarrow$ TX {input spends `channel` funding TX output, outputs pay x, y to pk_{Alice}, pk_{Bob} respectively, alphabetically ordered by some fixed encoding of the keys, where x is *Alice*'s and y is *Bob*'s balance in the latest `channel` state
    sig $\leftarrow$ SignDS(Cl, sk_{Alice})
    send (closingSigned, sig, *pchid*) to *Bob*
  end if

Upon receiving (closingSigned, bobSig, *pchid*) from *Bob*:
  ensure there is an open `channel` $\in$ `channels` with *Bob*with ID *pchid*
  ensure `channel` is marked as "coop closing"
  add `receipt`(`channel`) to `closedChannels`
  remove `channel` from `channels`
  $Cl \leftarrow$ TX {input spends `channel` funding TX output, outputs pay x, y to pk_{Alice}, pk_{Bob} respectively, alphabetically ordered by some fixed encoding of the keys, where x is *Alice*'s and y is *Bob*'s balance in the latest `channel` state
  ensure VerifyDS(bobSig, Cl, pk_{Bob}) = `True`
  aliceSig $\leftarrow$ SignDS(Cl, sk_{Alice})
  sort aliceSig, bobSig according to the ordering of the respective keys to produce sig1, sig2
  send (submit, ((sig1, sig2), Cl)) to $\mathcal{G}_{\text{Ledger}}$

Fig. 45.

11 Security Proof

Functionality $\mathcal{F}_{\text{PayNet,dummy}}$

1: Upon receiving any message M from *Alice*:
2: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from a player **then**
3: send $(M, \textit{Alice})$ to $\mathcal{S}$
4: **end if**

5: Upon receiving any message $(M, \textit{Alice})$ from $\mathcal{S}$:
6: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from $\mathcal{S}$ **then**
7: send M to *Alice*
8: **end if**

Fig. 46.

Simulator $\mathcal{S}_{\text{LN}}$

Expects the same messages as the protocol, but messages that the protocol expects to receive from $\mathcal{E}$, the simulator expects to receive from $\mathcal{F}_{\text{PayNet,dummy}}$ with the name of the player appended. The simulator internally executes one copy of the protocol per player. Upon receiving any message, the simulator runs the relevant code of the protocol copy tied to the appended player name. Mimicking the real-world case, if a protocol copy sends a message to another player, that message is passed to $\mathcal{A}$ as if sent by the player and if $\mathcal{A}$ allows the message to reach the receiver, then the simulator reacts by acting upon the message with the protocol copy corresponding to the recipient player. A message sent by a protocol copy to $\mathcal{E}$ will be routed by $\mathcal{S}$ to $\mathcal{F}_{\text{PayNet,dummy}}$ instead. To distinguish which player it comes from, $\mathcal{S}$ also appends the player name to the message. Corruption messages in the backdoor tapes of simulated parties are also forwarded to $\mathcal{F}_{\text{PayNet,dummy}}$.

Fig. 47.

Lemma 3. $\textit{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}} = \textit{EXEC}^{\mathcal{F}_{\text{PayNet,dummy}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}},\mathcal{E}}$

Proof. Consider a message that $\mathcal{E}$ sends. In the real world, the protocol ITIs produce an output. In the ideal world, the message is given to $\mathcal{S}_{\text{LN}}$ through $\mathcal{F}_{\text{PayNet,dummy}}$. The former simulates the protocol ITIs of the real world (along with their coin flips) and so produces an output from the exact same distribution, which is given to $\mathcal{E}$ through $\mathcal{F}_{\text{PayNet,dummy}}$. Thus the two outputs are indistinguishable. □

Functionality $\mathcal{F}_{\text{PayNet,Reg}}$

1: For messages REGISTER, REGISTERDONE, TOPPEDUP and CORRUPTED, act like $\mathcal{F}_{\text{PayNet}}$, but skip lines that call `checkClosed`().

2: Upon receiving any other message M from *Alice*:
3: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from a player **then**
4: send $(M, \textit{Alice})$ to $\mathcal{S}$
5: **end if**

6: Upon receiving any other message $(M, \textit{Alice})$ from $\mathcal{S}$:
7: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from $\mathcal{S}$ **then**
8: send M to *Alice*
9: **end if**

Fig. 48.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}}$

Like $\mathcal{S}_{\text{LN}}$, but it does not accept (TOPPEDUP) from $\mathcal{F}_{\text{PayNet,Reg}}$. Additional differences:

1: Upon receiving (REGISTER, *Alice*, delay, relayDelay) from $\mathcal{F}_{\text{PayNet,Reg}}$:
2: `delay` of *Alice* ITI $\leftarrow$ delay
3: `relayDelay` of *Alice* ITI $\leftarrow$ relayDelay
4: (pk_{Alice}, sk_{Alice}) of *Alice* ITI $\leftarrow$ `KeyGen`()
5: send (REGISTERDONE, *Alice*, pk_{Alice}) to $\mathcal{F}_{\text{PayNet,Reg}}$

6: Upon receiving (CORRUPT) on the backdoor tape of *Alice*'s simulated ITI:
7: add *Alice* to `corrupted`
8: for the rest of the execution, upon receiving any message for *Alice*, bypass normal execution and simply forward it to *Alice*
9: send (CORRUPTED, *Alice*) to $\mathcal{F}_{\text{PayNet,Reg}}$

Fig. 49.

Lemma 4. $EXEC^{\mathcal{F}_{\text{PayNet}},\text{dummy},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}},\mathcal{E}} = EXEC^{\mathcal{F}_{\text{PayNet}},\text{Reg},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}},\mathcal{E}}$

Proof. When $\mathcal{E}$ sends (REGISTER, delay, relayDelay) to *Alice*, it receives as a response (REGISTER, *Alice*, delay, relayDelay, pk_{Alice}) where pk_{Alice} is a public key generated by `KeyGen`() both in the real (c.f. Fig. 24, line 9) and in the ideal world (c.f. Fig. 49, line 4).

Furthermore, one (READ) is sent to $\mathcal{G}_{\text{Ledger}}$ from *Alice* in both cases (Fig. 24, line 8 and Fig. 12, line 10).

Additionally, $\mathcal{S}_{\text{LN}-\text{Reg}}$ ensures that the state of *Alice* ITI is exactly the same as what would have been in the case of $\mathcal{S}_{\text{LN}}$, as lines 6-9 of Fig. 24 change the state of *Alice* ITI in the same way as lines 2-4 of Fig. 49.

Lastly, the fact that the state of the *Alice* ITIs are changed in the same way in both worlds, along with the same argument as in the proof of Lemma 3 ensures that the rest of the messages are responded in an indistinguishable way in both worlds. □

Functionality $\mathcal{F}_{\text{PayNet},\text{Open}}$

1: For messages REGISTER, REGISTERDONE, TOPPEDUP, OPENCHANNEL, ADVERSARYOPENCHANNEL, CHANNELANNOUNCED and CHECKFORNEW, act like $\mathcal{F}_{\text{PayNet}}$, but skip lines that call `checkClosed`().

2: Upon receiving any other message M from *Alice*:
3: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from a player **then**
4: send $(M, Alice)$ to $\mathcal{S}$
5: **end if**

6: Upon receiving any other message $(M, Alice)$ from $\mathcal{S}$:
7: **if** M is a valid $\mathcal{F}_{\text{PayNet}}$ message from $\mathcal{S}$ **then**
8: send M to *Alice*
9: **end if**

Fig. 50.

Lemma 5. $EXEC^{\mathcal{F}_{\text{PayNet}},\text{Reg},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}},\mathcal{E}} = EXEC^{\mathcal{F}_{\text{PayNet}},\text{Open},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}},\mathcal{E}}$

Proof. When $\mathcal{E}$ sends (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*) to *Alice*, the interaction of Figures 25-29 will be executed in both the real and the ideal world. In more detail, in the ideal world the execution of the honest parties will be simulated by the respective ITIs run by $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$, so their state will be identical to that of the parties in the real execution. Furthermore, since $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ executes faithfully the protocol code, it generates the same messages as would be generated by the parties themselves in the real-world setting.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ – open

Like $\mathcal{S}_{\text{LN}-\text{Reg}}$. Differences:

Upon receiving (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) from $\mathcal{F}_{\text{PayNet,Open}}$:
  if both *Alice* and *Bob* are honest **then**
    Simulate the interaction between *Alice* and *Bob* in their respective ITI, as defined in Figures 25-29. All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After sending (fundingSigned) as *Bob* to *Alice*, send (channelAnnounced, *Bob*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*) to $\mathcal{F}_{\text{PayNet,Open}}$.
    After submitting F to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, send (channelAnnounced, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*) to $\mathcal{F}_{\text{PayNet,Open}}$.
  else if *Alice* is honest, *Bob* is corrupted **then**
    Simulate *Alice*'s part of the interaction between *Alice* and *Bob* in *Alice*'s ITI, as defined in Figures 25, 27, and 29.All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After submitting F to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, send (channelAnnounced, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*) to $\mathcal{F}_{\text{PayNet,Open}}$.
  else if *Alice* is corrupted, *Bob* is honest **then**
    send (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) to simulated (corrupted) *Alice*
    Simulate *Bob*'s part of the interaction between *Alice* and *Bob* in *Bob*'s ITI, as defined in Figures 26 and 28. All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After sending (fundingSigned) as *Bob* to *Alice*, send (channelAnnounced, *Bob*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*) to $\mathcal{F}_{\text{PayNet,Open}}$.
  else if both *Alice* and *Bob* are corrupted **then**
    forward message to $\mathcal{A}$ // $\mathcal{A}$ may open the channel or not
  end if

Upon receiving (openChannel, x, `remoteDelay`, pt_F, ptb_{pay}, ptb_{dpay}, ptb_{htlc}, $pt_{\text{com},1}$, ptb_{rev}, *tid*, *from Alice*, *to Bob*) from $\mathcal{A}$:
  ensure *Alice* is `corrupted` and that *Bob* is not
  choose unique number and assign it to *intid*
  store received message along with *intid*
  send (adversaryOpenChannel, x, `remoteDelay`, *tid*, *intid*, *from Alice*, *to Bob*) to $\mathcal{F}_{\text{PayNet,Open}}$

Upon receiving (adversaryOpenChannel, *fchid*, *intid*) from $\mathcal{F}_{\text{PayNet,Open}}$:
  retrieve message associated with *intid*
  Continue simulation with *Alice* receiving the message from *Bob*

Continued in Fig. 52

Fig. 51.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ – funding locked

Continuing from Fig. 51

1: Upon receiving (FUNDINGLOCKED, *Alice*, Σ_{Alice}, *fchid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
2: execute lines 5-9 of Fig. 30 with *Alice*'s ITI, using Σ_{Alice} from message
3: **if** *Bob* is honest **then**
4: expect the delivery of *Alice*'s (FUNDINGLOCKED) message from $\mathcal{A}$
5: send (FUNDINGLOCKED, *fchid*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$
6: upon receiving (FUNDINGLOCKED, *Bob*, Σ_{Bob}, *fchid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
7: simulate Fig. 31 with message from *Alice* in *Bob*'s ITI, using Σ_{Bob} from $\mathcal{F}_{\text{PayNet},\text{Open}}$'s message
8: **end if**

9: Upon receiving the (FUNDINGLOCKED) message with the simulated *Alice* ITI:
10: simulate Fig. 31 receiving the message with *Alice*'s ITI
11: send (CHANNELOPENED, *fchid*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$

Fig. 52.

We observe that the input validity check executed by $\mathcal{F}_{\text{PayNet},\text{Open}}$ (Fig. 13, line 2) filters only messages that would be ignored by the real protocol as well and would not change its state either (Fig. 25, line 2).

We also observe that, upon receiving the message OPENCHANNEL or CHANNELANNOUNCED, $\mathcal{F}_{\text{PayNet},\text{Open}}$ does not send any messages to parties other than $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$, so we don't have to simulate those.

When $\mathcal{E}$ sends (CHECKFORNEW, *Alice*, *Bob*, *tid*) to *Alice* in the real world, line 2 of Fig. 30 will allow execution to continue if there exists an entry with temporary id *tid* in `pendingOpen` marked as "broadcast". Such an entry can be added either in Fig. 25, line 8 or in Fig. 26, line 6. The former event can happen only in case *Alice* received a valid OPENCHANNEL message from *Bob* with temporary id *tid*, which in turn can be triggered only by a valid OPENCHANNEL message with the same temporary id from $\mathcal{E}$ to *Bob*, whereas the latter only in case *Alice* received a valid OPENCHANNEL message from $\mathcal{E}$ with the same temporary id. Furthermore, in the first case the "broadcast" mark can be added only before *Alice* sends (FUNDINGSIGNED, *pchid*, sig) to *Bob* (Fig. 28, line 12) (which needs a valid *Alice*-*Bob* interaction up to that point), and in the second case the "broadcast" mark can be added only before *Alice* sends (SUBMIT, (sig, F)) to $\mathcal{G}_{\text{Ledger}}$ (Fig. 29, line 8) (which also needs a valid *Alice*-*Bob* interaction up to that point)

In the ideal world, when the simulated $\mathcal{A}$ attempts to open a new channel by instructing corrupted *Alice* to send (OPENCHANNEL, x, `bobDelay`, _, _, _, _, _, _, *tid*) to honest *Bob*, $\mathcal{S}$ sends (ADVERSARYOPENCHANNEL, x, `bobDelay`, *tid*, *intid*, *from Alice*, *to Bob*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$ (Fig. 51, lines 16-20). In this case, $\mathcal{F}_{\text{PayNet},\text{Open}}$ stores the intention and prompts $\mathcal{S}$ to continue the channel negotiation (Fig. 13, lines 6-11), who in turn simulates the receipt of OPENCHANNEL by

Bob (Fig. 51, lines 21-23). In the real world, such a message by the adversarially controlled *Alice* would result in the honest player *Bob* starting the same opening negotiation as the one that was just described in the ideal world, therefore the two behaviours are indistinguishable.

When $\mathcal{E}$ sends (CHECKFORNEW, *Alice*, *Bob*, *tid*) to *Alice* in the ideal world, line 5 of Fig. 14 will allow execution to continue if there exists an entry with temporary ID *tid* and one member *Alice*, marked as "*Alice* announced" in `pendingOpen`(*fchid*) for some *fchid*. This can only happen if line 3 of Fig. 14 is executed, where `pendingOpen`(*fchid*) contains *tid* as the temporary ID. This line in turn can only be executed if $\mathcal{F}_{\text{PayNet,Open}}$ received (CHANNELANNOUNCED, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*) from $\mathcal{S}_{\text{LN-Reg-Open}}$ such that the entry `pendingOpen`(*fchid*) exists and has temporary ID *tid*, as mandated by line 2 of Fig. 14. Such a message is sent by $\mathcal{S}_{\text{LN-Reg-Open}}$ of Fig. 51 either in lines 5/8, or in lines 4/12. One of the first pair of lines is executed only if $\mathcal{S}_{\text{LN-Reg-Open}}$ receives (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*) from $\mathcal{F}_{\text{PayNet,Open}}$ and the simulated $\mathcal{A}$ allows a valid *Alice-Bob* interaction up to the point where *Alice* sends (SUBMIT) to $\mathcal{G}_{\text{Ledger}}$, whereas one of the second pair of lines is executed only if $\mathcal{S}_{\text{LN-Reg-Open}}$ receives (OPENCHANNEL, *Bob*, *Alice*, x, *fchid*, *tid*) from $\mathcal{F}_{\text{PayNet,Open}}$ and the simulated $\mathcal{A}$ allows a valid *Alice-Bob* interaction up to the point where *Alice* sends (FUNDINGSIGNED) to *Bob*.

The last two points lead us to deduce that line 5 of Fig. 14 in the ideal and line 2 of Fig. 30 in the real world will allow execution to continue in the exact same cases with respect to the messages that $\mathcal{E}$ and $\mathcal{A}$ send. Given that execution continues, *Alice* subsequently sends (READ) to $\mathcal{G}_{\text{Ledger}}$ and performs identical checks in both the ideal (Fig. 14, lines 8-9) and the real world (Fig. 30, lines 3-4).

Moving on, in the real world lines 5-9 of Fig. 30 are executed by *Alice* and, given that $\mathcal{A}$ allows it, the code of Fig. 31 is executed by *Bob*. Likewise, in the ideal world, the functionality executes lines 10-11 of Fig. 30 and as a result it (always) sends (FUNDINGLOCKED, *Alice*, Σ_{Alice}, *fchid*) to $\mathcal{S}_{\text{LN-Reg-Open}}$. In turn $\mathcal{S}_{\text{LN-Reg-Open}}$ simulates lines 5-9 of Fig. 30 with *Alice*'s ITI and, if $\mathcal{A}$ allows it, $\mathcal{S}_{\text{LN-Reg-Open}}$ simulates the code of Fig. 31 with *Bob*'s ITI. Once more we conclude that both worlds appear to behave identically to both $\mathcal{E}$ and $\mathcal{A}$ under the same inputs from them. □

Functionality $\mathcal{F}_{\text{PayNet,Pay}}$

For messages REGISTER, REGISTERDONE, TOPPEDUP, OPENCHANNEL, CHANNELANNOUNCED, CHECKFORNEW, POLL, PAY, ADVERSARYSENDINVOICE, PUSHADD, PUSHFULFILL, FULFILLONCHAIN and COMMIT, act like $\mathcal{F}_{\text{PayNet}}$, but skip lines that call `checkClosed`().

Upon receiving any other message M from *Alice*:
  if M is a valid $\mathcal{F}_{\text{PayNet}}$ message from a player **then**
    send (M, Alice) to $\mathcal{S}$
  end if

Upon receiving any other message (M, Alice) from $\mathcal{S}$:
  if M is a valid $\mathcal{F}_{\text{PayNet}}$ message from $\mathcal{S}$ **then**
    send M to *Alice*
  end if

Fig. 53.

Simulator $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$ – resolve payments

Upon receiving any message with a concatenated (STATE, Σ) part from $\mathcal{F}_{\text{PayNet,Pay}}$: // PAY, PUSHFULFILL, PUSHADD, COMMIT
  handle first part of the message normally
  if at the end of the simulation above, control is still held by $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$ **then**
    for all $\Sigma_{\mathit{Alice}} \in \Sigma$ **do**
      for all $(\overrightarrow{\texttt{path}}, \_, \mathit{payid}) \in \texttt{payids} : \mathit{Alice} \in \overrightarrow{\texttt{path}}$ **do**
        if condition in Fig. 58 is met **then** // no or bad communication with *Bob*'s previous player
          execute effect in Fig. 58
        else if condition in Fig. 59 is met **then** // *Alice* did not fulfill in time
          execute effect in Fig. 59
        else if condition in Fig. 60 is met **then** // honest payment completed
          execute effect in Fig. 60
        end if
      end for
    end for
    clear `charged` and send (RESOLVEPAYS, `charged`) to $\mathcal{F}_{\text{PayNet,Pay}}$
  end if

Fig. 57.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – pay

Like $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$. Differences:

1: Upon simulating line 1 of Fig. 35:
2: **if** $\exists(\bot, \bot, payid) \in$ `payids` where *payid* is the one in the INVOICE message **then**
3: replace second $\bot$ with the h found in the INVOICE message
4: **end if**

5: Upon simulating line 1 of Fig. 36 with honest *Alice*:
6: **if** $\exists payid : (\overrightarrow{\texttt{path}}, h, payid) \in$ `payids` where h is the one in the UPDATEADDHTLC message **then**
7: append *Alice* to $\overrightarrow{\texttt{path}}$
8: **end if**

9: Upon receiving (FULFILLONCHAIN, t, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
10: execute lines 3-12 of Fig. 43 as *Alice*, using t from message
11: **if** during the simulation above, line 8 is simulated in *Alice*'s ITI **then**
12: send (NEWPAYMENTS, `payment`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `payment` consists of the tuple just added to the simulated `paymentsToReport`, appended with the *payid* mentioned in line 7
13: upon receiving (CONTINUE) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
14: **end if**

15: Upon receiving $\big($PAY, *Alice*, *Bob*, $x, \overrightarrow{\texttt{path}}$, *expayid*, *payid*$\big)$ from $\mathcal{F}_{\text{PayNet,Pay}}$:
16: add ($\overrightarrow{\texttt{path}}$, $\top$, *payid*) to `payids`
17: strip *payid* and *Alice*, simulate receiving the message with *Alice* ITI and further execute the parts of Π_{LN} that correspond to honest parties (Fig. 34-Fig. 38)
18: **if** any "ensure" in Π_{LN} fails until *Bob* processes UPDATEADDHTLC **then** // payment failed
19: add ($\bot$, *payid*) to `charged`(*Alice*)
20: remove ($\overrightarrow{\texttt{path}}$, _, *payid*) from `payids`
21: **end if**

22: Upon receiving (POLL, Σ_{Alice}, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
23: simulate Fig. 32, lines 3-27 receiving (POLL), using Σ_{Alice} from the message, with *Alice*'s ITI
24: **if** during the simulation above, line 18 of Fig. 33 is simulated in *Alice*'s ITI **then**
25: send (NEWPAYMENTS, `payment`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `payment` consists of the tuple just added to the simulated `paymentsToReport`, appended with the *payid* stored the entry of `pendingPay` referred to in line 17 (Fig. 33, line 18)
26: upon receiving (CONTINUE) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
27: **end if**

Fig. 54.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – adversarial payment

Upon receiving (sendInvoice, x, *expayid*, *payid*, *from Alice*, *to Bob*) from $\mathcal{A}$:
    ensure *Alice* is `corrupted` and that *Bob* is not
    ensure no other payment with the same *payid* exists
    choose unique number adn assign it to *invid*
    store received message along with *invid*
    add ($\perp$, $\perp$, *payid*) to `payids`
    send (adversarySendInvoice, x, *expayid*, *payid*, *intid*, *from Alice*, *to Bob*) to $\mathcal{F}_{\text{PayNet},\text{Pay}}$

Upon receiving (adversarySendInvoice, *invid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
    retrieve message associated with *invid*
    Continue simulation with *Alice* receiving the message from *Bob*

Fig. 55.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – corrupted charged

Condition:
    Alice sent updateFulfillHtlc to a corrupted player and either (got the fulfillment of the HTLC irrevocably committed OR fulfilled the HTLC on-chain (i.e. `HTLC-success` is in Σ_{Alice})), AND the next honest player *Bob* down the line successfully timed out the HTLC on-chain (i.e. `HTLC-timeout` is in Σ_{Bob})

Effect:
    add (`corrupted`, *payid*) to `charged`(payer) where `corrupted` is set to one of the corrupted parties between *Alice* and *Bob*
    if there is an $h \neq \top$ member in `payids` entry **then** // adversarially initiated payment, functionality doesn't know path
        append $\overrightarrow{\texttt{path}}$ to `charged`(*Alice*) entry
    end if
    remove ($\overrightarrow{\texttt{path}}$, _, *payid*) from `payids`

Fig. 58.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – push

1: Upon receiving (pushFulfill, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
2: simulate Fig. 42, lines 1-19 on input (pushFulfill, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players
3: **if** during the simulation above, line 14 of Fig. 42 is simulated in *Alice*'s ITI **then**
4: send (newPayments, `payment`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `payment` consists of the tuple just added to the simulated `paymentsToReport`, appended with the *payid* of line 12 (Fig. 42, line 14)
5: upon receiving (continue) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
6: **end if**

7: Upon receiving (pushAdd, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
8: simulate Fig. 42, lines 20-24 on input (pushAdd, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players

9: Upon receiving (commit, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
10: simulate Fig. 39 on input (commit, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players
11: **if** during the simulation above, line 10 of Fig. 41 is simulated in *Alice*'s ITI **then**
12: send (newUpdate, `receipt`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `receipt` is the field just added to the simulated `updatesToReport` (Fig. 41, line 10)
13: upon receiving (continue) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
14: **end if**
15: **if** during the simulation above, line 16 of Fig. 41 is simulated in *Alice*'s ITI **then**
16: send (newPayments, `payments`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `payments` consists of the fields just added to the simulated `paymentsToReport`, each field appended with the respective *payid* of line 11 (Fig. 41, line 16)
17: upon receiving (continue) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
18: **end if**

Fig. 56.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – no timely fulfill

Condition:

1: Σ_{Alice} contains an old $\texttt{remoteCom}_m$ of the channel before *Alice* (closer to payer) on the $\overrightarrow{\texttt{path}}$ that does not contain the relevant HTLC anymore and a tx that spends the delayed output of $\texttt{remoteCom}_m \vee ((\Sigma_{Alice}$ contains the most recent $\texttt{remoteCom}_n$ or $\texttt{localCom}_n$ of the channel before *Alice* and the `HTLC-timeout` of the relevant HTLC $\vee$ *Alice*'s latest irrevocably committed $\texttt{remoteCom}_n$ for the channel before *Alice* does not contain the HTLC) $\wedge \Sigma_{Alice}$ contains the most recent $\texttt{remoteCom}_l$ or $\texttt{localCom}_l$ and the `HTLC-success` that pays the counterparty for HTLC of the channel after *Alice*)

Effect:

2: add (*Alice*, *payid*) to `charged`(payer)
3: **if** there is an $h \neq \top$ member in `payids` entry **then** // adversarially initiated payment, functionality doesn't know path
4: append $\overrightarrow{\texttt{path}}$ to `charged`(*Alice*) entry
5: **end if**
6: remove ($\overrightarrow{\texttt{path}}$, _, *payid*) from `payids`

Fig. 59.

Simulator $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ – honest payment

Condition:

1: *Alice* is the payer in $\overrightarrow{\texttt{path}}$ AND ((she has received UPDATEFULFILLHTLC AND has subsequently sent COMMIT and REVOKEANDACK) OR (player after *Alice* has irrevocably fulfilled the HTLC on-chain (i.e. his `HTLC-success` is in Σ_{Alice}) AND *Alice* has received POLL to observe it))

Effect:

2: add (*Alice*, *payid*) to `charged`(*Alice*)
3: **if** there is an $h \neq \top$ member in `payids` entry **then** // adversarially initiated payment, functionality doesn't know path
4: append $\overrightarrow{\texttt{path}}$ to `charged`(*Alice*) entry
5: **end if**
6: remove ($\overrightarrow{\texttt{path}}$, _, *payid*) from `payids`

Fig. 60.

Lemma 6.

$$\forall k \in \mathbb{N},\ PPT\ \mathcal{E},$$
$$|\Pr[EXEC^{\mathcal{F}_{\text{PayNet}},\text{Open},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}},\mathcal{E}} = 1] - \Pr[EXEC^{\mathcal{F}_{\text{PayNet}},\text{Pay},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}},\mathcal{E}} = 1]| \leq$$
$$nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) +$$
$$nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)\ .$$

Proof. Before focusing on individual messages sent by $\mathcal{E}$ or $\mathcal{A}$, we will first prove that four particular forgery events happen with negligible probability. Let P be the event in which at some point during the execution a transaction that has the following two characteristics appears in Σ_{Alice}, for some honest player *Alice*: it spends a funding transaction of a channel that contains *Alice* (and thus has a $p_{Alice,F}$ public key) and it was never signed by *Alice*. Suppose that n is the number of players, m is the maximum number of channels that a player can open and p is the maximum number of opens and updates a player can perform in all channels, and $\exists$ PPT $\mathcal{E}_P : \Pr[P] = a$. We show in Proposition 3 that $\forall \mathcal{E}, \Pr[P] \leq nm \cdot \text{E-ds}(k)$.

Let Q be the event in which at some point during the execution a transaction that has the following two characteristics appears in Σ_{Alice}, for some honest player *Alice*: it spends a simple output, delayed output or htlc output tied with a public key that was created by *Alice* ($p_{Alice,\text{pay},n}, p_{Alice,\text{dpay},n}, p_{Alice,\text{htlc},n}$ respectively) and it was never signed by *Alice*. Suppose that p is the maximum total number of opens and updates that a player can perform across all channels and that $\exists$ PPT $\mathcal{E}_Q : \Pr[Q] = b$. We show in Proposition 4 that $\forall \mathcal{E}, \Pr[Q] \leq 3np \cdot \text{E-ibs}(k)$.

Let R be the event in which at some point during the execution a transaction that has the following characteristic appears in Σ_{Alice}, for some honest player *Alice*: it spends the revocation output of a local (for *Alice*) commitment transaction for a channel that contains *Alice* and *Bob* (and thus has a $p_{Bob,\text{rev},n}$ key). Observe that, since *Alice* is honest and according to both the real and the ideal execution, if *Alice* submits her local commitment transaction $\texttt{localCom}_n$ to the ledger, under no circumstances does she subsequently go on to send $s_{Alice,\text{com},n}$ to any party. (This secret information could be used by *Bob* to efficiently compute $s_{Bob,\text{rev},n}$ with $\textsc{CombineKey}(pb_{Bob,\text{rev}}, sb_{Bob,\text{rev}}, p_{Alice,\text{com},n}, s_{Alice,\text{com},n})$.) Suppose that p is the maximum total number of opens and updates that a player can perform across all channels, m is the maximum number of channels a player can open and $\exists$ PPT $\mathcal{E}_R : Pr[R] = c$. We show in Proposition 5 that $\forall \mathcal{E}, \Pr[R] \leq nmp \cdot \text{E-share}(k) + \text{E-prf}(k)$.

Lastly, let S be the event in which at some point during the execution a transaction that has the following two characteristics appears in Σ_{Alice}, for some honest player *Alice*: (a) it spends the revocation output of a remote (for *Alice*) commitment transaction for a channel that contains *Alice* (and thus has a $p_{Alice,\text{rev},n}$ key) and (b) it was never signed by *Alice*. Observe that, since *Alice* is honest, she has never sent $s_{Alice,\text{rev},n}$ to any party. Suppose that m is the maximum total

number of opens and updates that a player can perform and that $\exists$ PPT $\mathcal{E}_S$: $\Pr[S] = d$. We show in Proposition 6 that $\forall \mathcal{E}, \Pr[S] \leq nm \cdot \text{E-master}(k)$.

In the ideal world, when the simulated $\mathcal{A}$ attempts to send a payment by instructing corrupted *Alice* to send (sendInvoice, x, *expayid*, *payid*) to the honest payee *Bob*, $\mathcal{S}$ sends (adversarySendInvoice, x, *expayid*, *payid*, *intid*, *from Alice*, *to Bob*) to $\mathcal{F}_{\text{PayNet,Pay}}$ (Fig. 55, lines 1-7). In this case, $\mathcal{F}_{\text{PayNet,Pay}}$ stores the intention and prompts $\mathcal{S}$ to continue the payment (Fig. 15, lines 5-9), who in turn simulates the receipt of sendInvoice by *Bob* (Fig. 55, lines 8-10). In the real world, such a message by the adversarially controlled *Alice* would result in the honest *Bob* sending the same invoice, therefore the two behaviours are indistinguishable.

In the ideal world, when the simulated $\mathcal{A}$ attempts to add an HTLC by instructing corrupted *Alice* to send (updateAddHtlc, *pchid*, `HTLCNo`, x, h, `IncomingCltvExpiry`, M) to honest *Bob*, $\mathcal{S}$ simply simulates the receiving of the message by *Bob* and subsequently carries out the rest of the simulation. In the real world, such a message by the adversarially controlled *Alice* would result in *Bob* and the rest of the ITIs following the same steps as in the simulated case, therefore the two behaviours are indistinguishable.

We can now move on to treating individual messages sent by $\mathcal{E}$ during the execution. When $\mathcal{E}$ sends $\left(\text{PAY}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \mathit{expayid}\right)$ to *Alice* in the ideal world, $\mathcal{S}_{\text{LN-Reg-Open}}$ is always notified (Fig. 15, line 4) and simulates the relevant execution of the real world (Fig. 54, line 17). No messages to $\mathcal{G}_{\text{Ledger}}$ or $\mathcal{E}$ that differ from the real world are generated in the process. At the end of this simulation, no further messages are sent (and the control returns to $\mathcal{E}$). Therefore, when $\mathcal{E}$ sends PAY, no opportunity for distinguishability arises.

When $\mathcal{E}$ sends any message of (pushAdd, *pchid*), (pushFulfill, *pchid*), (commit, *pchid*) to *Alice* in the ideal world, it is forwarded to $\mathcal{S}_{\text{LN-Reg-Open}}$ (Fig. 23, lines 2, 4, 6 respectively), who in turn simulates *Alice*'s real-world execution with her simulated ITI and the handling of any subsequent messages sent by *Alice*'s ITI (Fig. 56, lines 2, 8, 10). Neither $\mathcal{F}_{\text{PayNet,Pay}}$ nor $\mathcal{S}_{\text{LN-Reg-Open}}$ alter their state as a result of these messages, apart from the state of *Alice*'s simulated ITI and the state of other simulated ITIs that receive and handle messages that were sent as a result of *Alice*'s ITI simulation. The states of these ITIs are modified in the exact same way as they would in the real world. We deduce that these three messages do not introduce any opportunity for $\mathcal{E}$ to distinguish the real and the ideal world.

When $\mathcal{E}$ sends (fulfillOnChain) to *Alice* in the real world, lines 1-12 of Fig. 43 are executed by *Alice*. In the ideal world on the other hand, $\mathcal{F}_{\text{PayNet,Pay}}$ sends (read) to $\mathcal{G}_{\text{Ledger}}$ (Fig. 23, line 8) as *Alice* and subsequently instructs $\mathcal{S}_{\text{LN-Reg-Open}}$ to simulate the receiving of (fulfillOnChain) with *Alice*'s ITI (Fig. 54, lines 9-10). Observe that during this simulation a second (read) message to $\mathcal{G}_{\text{Ledger}}$ (that would not match any message in the real world) is avoided because $\mathcal{S}_{\text{LN-Reg-Open}}$ skips line 2 of Fig. 43, using as t the one received from $\mathcal{F}_{\text{PayNet,Pay}}$ in the message (fulfillOnChain, t, *Alice*). Since $\mathcal{F}_{\text{PayNet,Pay}}$ sends (read) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and given that after $\mathcal{G}_{\text{Ledger}}$ replies, control is given

directly to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$, the t used during the simulation of *Alice*'s ITI is identical to the one that *Alice* would obtain in the real-world execution. The rest of the simulation is thus identical with the real-world execution, therefore FULFILLONCHAIN does not introduce any opportunity for distinghuishability.

When $\mathcal{E}$ sends (POLL) to *Alice*, the first action is sending (READ) as *Alice* to $\mathcal{G}_{\text{Ledger}}$ both in the ideal (Fig. 22, line 4) and the real (Fig. 32, line 2) worlds. Subsequently, in the real world lines 3-27 of Fig. 32 are executed by *Alice*, whereas in the ideal world, given that the checks of lines 10 and 5 do not lead to a bad event (and thus given that the functionality does not halt in lines 11 or 6), a (POLL) message is sent to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$. We will prove later that $\mathcal{F}_{\text{PayNet},\text{Pay}}$ does not halt here. Upon receiving (POLL), $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ simulates receiving (POLL) with *Alice*'s ITI (Fig. 54, line 23), but does not READ from $\mathcal{G}_{\text{Ledger}}$ and uses instead the Σ_{Alice} provided along with the message. A reasoning identical to that found in the previous paragraph shows that this Σ_{Alice} is exactly the same as that which *Alice*'s ITI would obtain had it executed line 2 of Fig. 32 and thus the simulation of *Alice*'s ITI is identical to what would happen in the same case in the real world, up to and including line 27 of Fig. 32.

The event E in which $\mathcal{F}_{\text{PayNet},\text{Pay}}$ executes line 6 of Fig. 22 and halts can only happen if there is a non-commitment transaction that contains a valid signature by the $p_{Alice,F}$ key that is needed to spend the funding transaction of an open channel. According to Π_{LN}, *Alice* signs with her $s_{Alice,F}$ key only commitment transactions. Therefore $E \subset P \Rightarrow \Pr[E|\neg P] = 0$.

Let E' the "bad" event in which $\mathcal{F}_{\text{PayNet},\text{Pay}}$ executes line 11 of Fig. 22 and halts. We will now prove that, during $\text{EXEC}^{\mathcal{F}_{\text{PayNet},\text{Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}},\mathcal{E}}$, it is $\Pr[E|\neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$. The condition of Fig. 22, line 10 is triggered if the delayed output (that of the malicious party) of $\texttt{tx}_1$ has been spent by the transaction $\texttt{tx}_2$ in Σ_{Alice} (event E'_1) and $\texttt{polls}(Alice)$ contains an element in $[h_1, h_1 + \texttt{delay}\,(Alice) - 1]$, where h_1 is the block height where $\texttt{tx}_1$ is (event E'_2). Observe that $E' = E'_1 \wedge E'_2$. We note that the elements in $\texttt{polls}(Alice)$ correspond to the block heights of Σ_{Alice} at the moments when *Alice* POLLs (Fig. 22, line 3). Consider the following two events: $E'_{1,1}$: $\texttt{tx}_2$ spends the delayed output with a signature valid by the delayed payment public key after the locktime expires. $E'_{1,2}$: $\texttt{tx}_2$ spends the delayed output with a signature valid by the revocation public key $p_{Alice,\text{rev}}$. Note that $E'_1 = E'_{1,1} \vee E'_{1,2}$ and $E'_{1,1}, E'_{1,2}$ are mutually exclusive (since the same output cannot be spent twice). Observe that $E'_{1,2} \subset S$, thus $\Pr\left[E'_{1,2}|\neg S\right] = 0$. We now concetrate on the event $E'_{1,1}$. Due to the fact that $\texttt{tx}_2$ spends an output locked with a relative timelock of length $\texttt{delay}\,(Alice) + (2 + r)\,\texttt{windowSize}$, the commitment transaction $\texttt{tx}_1$ can reside in a block of maximum height $h_1 \leq h_2 - \texttt{delay}\,(Alice) - (2 + r)\,\texttt{windowSize}$, where h_2 is the block height where $\texttt{tx}_2$ is. If *Alice* POLLs on a moment when $|\Sigma_{Alice}| \geq h_1, \Sigma_{Alice}$ necessarily contains $\texttt{tx}_1$. Furthermore, if *Alice* POLLs on a moment when $|\Sigma_{Alice}| \leq h_1 + \texttt{delay}\,(Alice) - 1 \leq h_2 - (2 + r)\,\texttt{windowSize} - 1$, she sees $\texttt{tx}_1$ and directly submits the punishment transaction $\texttt{tx}_3$ (which she has, given that a maliciously closed channel is defined as one where the non-closing party has the punishment transaction) (Fig. 33, lines 22-24). Given

that $\mathtt{tx}_3$ is broadcast when $|\Sigma_{Alice}| \leq h_2 - (2+r)\,\mathtt{windowSize}$, it is guaranteed to be on-chain in a block $h_3 \leq h_2$ (according to Proposition 1). Since $\mathtt{tx}_3$ spends the same funds as $\mathtt{tx}_2$, the two cannot be part of the chain simultaneously. Since $E'_{1,1} \Rightarrow \Sigma_{Alice}$ contains $\mathtt{tx}_2$ and $E'_2 \Rightarrow \Sigma_{Alice}$ contains $\mathtt{tx}_3$, $E'_{1,1}$ and E'_2 are mutually exclusive. Therefore, assuming $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$, it is $\Pr[E'] = \Pr\left[\left(E'_{1,1} \vee E'_{1,2}\right) \wedge E'_2\right] = \Pr\left[\left(E'_{1,1} \wedge E'_2\right) \vee \left(E'_{1,2} \wedge E'_2\right)\right] \leq \Pr\left[E'_{1,1} \wedge E'_2\right] + \Pr\left[E'_{1,2} \wedge E'_2\right] = \Pr\left[E'_{1,2} \wedge E'_2\right] \leq \Pr\left[E'_{1,2}\right] = 0$. We conclude that, given $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$ POLL introduces no opportunity for distinghuishability.

We now treat the effects of the (STATE, Σ) message that $\mathcal{F}_{\text{PayNet,Pay}}$ sends to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$, appended to PAY, PUSHFULFILL, PUSHADD and COMMIT messages. We first observe that the (STATE) message is handled after handling the first message (which is of one of the four aforementioned types) (Fig. 57, line 2). It may be the case that at the end of the handling of line 2, $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ does not have control of the execution. That can happen if a simulated ITI sends a message to a corrupted player and that player does not respond (e.g. in Fig. 34, line 6, when the first message is $\left(\text{PAY}, Bob, x, \overrightarrow{\mathtt{path}}, \_\right)$ and Bob is corrupted), or if the handling of the message results in sending (SUBMIT) to $\mathcal{G}_{\text{Ledger}}$ (e.g. in Fig. 42, line 18 when the first message is (PUSHFULFILL, *pchid*) and counterparty has gone on-chain). In that case, the (STATE) message is simply ignored (Fig. 57, line 3) and does not influence execution in any way.

In the case when (STATE, Σ) is handled, $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ attempts to specify who was charged for each pending payment, based on the information that the potentially paying party sees in its view of the $\mathcal{G}_{\text{Ledger}}$ state (Fig. 57, lines 4-14). The resolution is then sent to $\mathcal{F}_{\text{PayNet,Pay}}$ with the message (RESOLVEPAYS, `charged`). $\mathcal{F}_{\text{PayNet,Pay}}$ handles this message in Fig. 16 and 19, where, if it does not halt (Fig. 18, lines 4, 7 and 15 and Fig. 19, lines 10, 13 and 17), it updates the state of each affected channel (Fig. 19, line 4) and does not send any message, thus control returns to $\mathcal{E}$. We will prove that, under $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$, $\mathcal{F}_{\text{PayNet,Pay}}$ does not halt and thus conclude that the handling of a (STATE) message does not introduce opportunity for distinguishability.

$\mathcal{F}_{\text{PayNet,Pay}}$ halts in line 4 of Fig. 18 if the honest player *Dave* was charged for a payment over a channel that was closed without using a commitment transaction. Like E, this event is a subset of P, thus cannot happen given $\neg P$.

$\mathcal{F}_{\text{PayNet,Pay}}$ halts in line 7 of Fig. 18 if the player *Dave* charged is an honest member of the payment path, has POLLed in time to catch a malicious closure (event A) but a malicious closure succeeded (event B). $\mathcal{F}_{\text{PayNet,Pay}}$ halts in line 15 of Fig. 18 if *Dave* is not the payer, no malicious closure succeeded ($\neg B$) and *Dave* has POLLed in time twice to learn the preimage of the HTLC early enough (event C) and has attempted to fulfill on chain at the right moment (event D). $\mathcal{F}_{\text{PayNet,Pay}}$ also halts if the two expiries do not have the expected distance (event F) – i.e. halts in the event $(A \wedge B) \vee (\neg B \wedge (F \vee (C \wedge D)))$. $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ decides that *Dave* is charged if his previous counterparty did a malicious closure to a channel version without the HTLC and spent their (delayed) output (B), or if his next counterparty fulfilled (event G) and his previous counterparty timed

out the HTLC (event H) (Fig. 57, line 8), – i.e. *Dave* is charged in the event $B \vee (G \wedge H)$.

We will now show that $\Pr[A \wedge B | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0 \wedge \Pr[(C \wedge D) \wedge (G \wedge H) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0 \wedge \Pr[F \wedge (G \wedge H) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$, from which we can deduce that $\Pr[(A \wedge B) \vee ((F \vee (C \wedge D)) \wedge (G \wedge H)) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$ and thus $\Pr[((A \wedge B) \vee (\neg B \wedge (F \vee (C \wedge D)))) \wedge (B \vee (G \wedge H)) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$. This last step holds because $(A \wedge B) \vee ((F \vee (C \wedge D)) \wedge (G \wedge H)) = (A \wedge B) \vee ((F \vee (C \wedge D) \wedge G \wedge H)$ and $((A \wedge B) \vee (\neg B \wedge (F \vee (C \wedge D)))) \wedge (B \vee (G \wedge H)) = (A \wedge B) \vee (\neg B \wedge (F \vee (C \wedge D)) \wedge G \wedge H)$ and the latter is a subset of the former.

The analysis of the event $A \wedge B$ is identical to the one we did previously for the events E'_1, E'_2, with A corresponding to E'_2 and B to E'_1. We thus deduce that $\Pr[A \wedge B | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$.

Event F is true only if $\texttt{IncomingCltvExpiry} - \texttt{OutgoingCltvExpiry} < \texttt{relayDelay}(\textit{Alice}) + (2 + r)\,\texttt{windowSize}$. This cannot happen however for any honest *Alice*, since $\mathcal{S}$ will simulate line 9 of Fig. 37 with *Alice*'s ITI before having her agree to participate as an intermediary in the multi-hop payment. Therefore $\Pr[F \wedge (G \wedge H) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$.

The only way for event C to be true is if $\mathcal{E}$ sends (POLL) to *Dave* during the prescribed time period (Fig. 22, line 3) – note that the addition to $\texttt{polls}(\textit{Dave})$ during registration (Fig. 12, line 10) cannot be within the desired range due to the fact that `OutgoingCltvExpiry` is not smaller than the chain height when the corresponding (INVOICE) was received (Fig. 35, line 9), registration happens necessarily before handling (INVOICE) (Fig. 12, line 24) and the element added to $\texttt{polls}(\textit{Dave})$ at registration is the chain height at that time (Fig. 12, line 10). When *Dave* receives (POLL), $\mathcal{F}_{\text{PayNet,Pay}}$ always sends (GETCLOSEDFUNDS) to $\mathcal{S}_{\text{LN-Reg-Open}}$ (Fig. 22, line 17) (since, as we saw earlier, $\mathcal{F}_{\text{PayNet,Pay}}$ never halts).

Event H happens only when the previous counterparty successfully appends `HTLC-timeout` to Σ_{Dave}, which is a valid transaction only from the block of height `IncomingCltvExpiry`+1 and on, or if the previous counterparty learns the preimage of the HTLC and forges a signature valid by *Dave*'s public HTLC key, or if the previous counterparty forges a signature valid by *Dave*'s public revocation key; the two latter scenarios can never happen. Thus, given that G happens until a moment when $|\Sigma_{Dave}| \leq \texttt{IncomingCltvExpiry} - (2 + r)\,\texttt{windowSize}$, *Dave* has the time to successfully fulfill the HTLC. Given C, *Dave* has POLLed at two moments $h_1, h_2 \in [\texttt{OutgoingCltvExpiry}, \texttt{IncomingCltvExpiry} - (2 + r)\texttt{windowSize}]$, such that $h_2 \geq h_1 + (2 + r)\,\texttt{windowSize}$. If Σ_{Dave} contains the preimage at moment h_1 or h_2, then *Dave* may try to update the previous channel off-chain if he receives a (PUSHFULFILL) for that channel (Fig. 42, lines 1-18), and if the off-chain update is never attempted (because (PUSHFULFILL) and (COMMIT) are not received) or fails (because the previous counterparty does not send (REVOKEANDACK)), then the (FULFILLONCHAIN) that he receives according to D will make him submit `HTLC-success` (Fig. 43, lines 1-12) and have it on-chain by block of height `IncomingCltvExpiry` (Proposition 1). Furthermore, in the case that the `HTLC-success` is not found at the (POLL) of h_1, *Dave* immediately submits `HTLC-timeout` (Fig. 33, line 9), which either ends up in

Σ_{Dave} by block height $h_1 + (2 + r)$ `windowSize` (Proposition 1) or is rejected because the counterparty managed to append `HTLC-success` before it. In the first case, *Dave* is not charged for the payment. In the second case, the second (POLL) (at block height h_2) necessarily reveals the `HTLC-success` to *Dave* and subsequently the (FULFILLONCHAIN) causes *Dave* to fulfill the HTLC with the previous counterparty, as argued above. Therefore in no case *Dave* is charged for the payment, i.e. $\Pr[(C \wedge D) \wedge (G \wedge H) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$.

We will now show that the halt of line 10 in Fig. 16 does not occur with non-negligible probability. If the check of line 8 succeeds, this means that line 19 of Fig. 54 was executed by $\mathcal{S}$, therefore the payment referred to by *payid* failed, therefore neither *Alice* nor *Bob* saw the relevant HTLC being fulfilled; call this event T. If now the check of line 9 succeeds, this means that $\mathcal{F}_{\text{PayNet}}$ has received the relevant entry from $\mathcal{S}$ in a NEWPAYMENTS message (Fig. 13, line 14). This in turn happens in one of the following cases: line 12 of Fig. 54, line 25 of Fig. 54, line 4 of Fig. 56, and line 16 of Fig. 56. Line 12 of Fig. 54 can only be executed if the simulated *Alice* or *Bob* has successfully fulfilled on chain the relevant HTLC (Fig. 43, line 7), which however is incompatible with T. Line 25 of Fig. 54 is executed if the simulated *Alice* has executed line 18 of Fig. 33, which means that the relevant (offered) HTLC has been spent (i.e. fulfilled) (line 12, Fig. 33), which is incompatible with T. Line 4 of Fig. 56 is executed if the simulated *Bob* executes line 14 of Fig. 42, which happens only if a relevant entry exists in `gotPaid` (line 12, Fig. 42), which in turn can only be added in line 10 of Fig. 36. In order to reach that point of the execution however, all UPDATEADDHTLC checks must have passed, which is incompatible with T. The last possible case in which the entry is added to `pendingDiffs` is line 16 of Fig. 56. This is executed if line 16 of Fig. 41 is executed either in the simulated *Alice* or in the simulated *Bob*. The field `updateDiffs` can be populated only in line 8 of Fig. 42 for *Alice* or line 23 of Fig. 38 for *Bob*. The former case can only happen if there was a corresponding entry in `gotPaid`, which, as we saw before, is incompatible with T. Likewise, the latter can only happen if *Alice* receives an UPDATEFULFILLHTLC message for the HTLC corresponding to the payment, which can only happen if no UPDATEADDHTLC fails, which is contrary to T. Therefore the halt of Fig. 16, line 10 cannot happen.

Moving on, we have to prove that the halt of line 10 in Fig. 19 does not occur with non-negligible probability. Indeed, $\mathcal{S}$ only reports the payment as resolved in RESOLVEPAYS if a party has been irrevocably charged for it (Fig. 57, lines 6, 8, or 10). In all three cases, all channels that follow the `charged` party on the $\overrightarrow{\texttt{path}}$ have either been closed or irrevocably updated to a newer version that includes the new balance. Since $\mathcal{F}_{\text{PayNet}}$ may only halt for a `channel` that has not been declared or confirmed as closed (Fig. 19, lines 1 and 9), all channels that can cause a halt are channels that have the update of this payment irrevocably committed. This only happens when both sides send a REVOKEANDACK that updates the channel from a version that contains the relevant HTLC to a version that doesn't; and when an honest party receives such a REVOKEANDACK message, it logs the new receipt in `updatesToReport` (Fig. 41, line 10) which causes $\mathcal{S}$ to report the

update to $\mathcal{F}_{\text{PayNet}}$ (Fig. 56, line 12). We therefore conclude that $\mathcal{F}_{\text{PayNet}}$ never halts on line 10 of Fig. 19.

It remains to be proven that $\mathcal{F}_{\text{PayNet}}$ does not halt with non-negligible probability in lines 13 and 17 of Fig. 19. For the halt of line 13 to occur, the player charged must be the payer (as intended by the original PAY messsage) and honest, but its `pendingDiffs` must not contain a corresponding entry for that payment. This however cannot happen for the following reason. As execution is in the loop of Fig. 16, line 3, it is (*Dave*, *payid*) $\in$ `charged`(*Alice*). As the code of Fig. 19 is being run, either line 16 of Fig. 16 or 21 of Fig. 18 has been executed, so line 12 has not been executed in the same iteration of the loop, therefore *Dave* $\neq \perp$. Additionally, since $\mathcal{F}_{\text{PayNet}}$ received a RESOLVEPAYS message, $\mathcal{S}$ has executed Fig. 57 and added (*Dave*, *payid*) to `charged`(*Alice*) in exactly one of lines 7, 9, or 11. Furthermore, since *Alice* $\notin$ `corrupted` (Fig. 19, line 12), the condition of Fig. 58 does not hold and since *Alice* is the payer (*Dave* = *Alice*), there is no channel "before" her, therefore the condition of Fig. 59 does not hold either. Therefore the only way for `charged`(*Alice*) to contain (*Alice*, *payid*) is if the condition of Fig 60 is true. The condition holds either when *Alice* has received UPDATEFULFILLHTLC on the fulfilment of the payment, followed by a REVOKEANDACK, or if her counterparty has fulfilled on-chain and she has checked the blockchain since. In the first case, the tuple (*expayid*, *payid*, _) would have been added to `pendingDiffs`(*Alice*) as $\mathcal{S}$ would have simulated line 23 of Fig. 38 on receiving UPDATEFULFILLHTLC, thus adding the tuple to `updateDiffs`, and line 16 of Fig. 41, thus prompting $\mathcal{S}$ to send the tuple to $\mathcal{F}_{\text{PayNet}}$ (Fig. 56, line 16), which would subsequently add it to `pendingDiffs` (Fig. 15, line 14). Therefore in this case the halt is impossible. In the alternative case, since *Alice* must have POLLed after her counterparty's `HTLC-success` was settled on-chain (Fig. 60, line 1) and therefore the loop of line 1 in Fig. 33 must have been executed by $\mathcal{S}$ while simulating *Alice*, as well as the loop of line 12 of the same figure and eventually line 18 in that figure, which would have prompted in turn $\mathcal{S}$ to send the relevant NEWPAYMENTS message to $\mathcal{F}_{\text{PayNet}}$ (Fig. 54, line 25). In that case however the tuple (`expayid`, `payid`, _) would be in `pendingDiffs`(*Alice*) and the check of line 12, Fig. 19 would fail, therefore preventing the functionality from halting.

We will now prove that the halt of line 17 in Fig. 19 does not occur. This halt would take place if the payee *Bob* is honest but was not previously informed of the payment, i.e. the tuple (*expayid*, *payid*, _) is not in `pendingDiffs`(*Bob*). Similarly to the argument for line 13, $\mathcal{S}$ added (*Dave*, *payid*) to `charged`(*Bob*) in exactly one of lines 7, 9, or 11 of Fig. 57. In the first case, there was a player on the path that fulfilled the payment. This can only happen if first *Bob* has fulfilled the payment and irrevocably committed to a channel version that resolves the HTLC to his favour, therefore *Bob* has either sent a relevant UPDATEFULFILLHTLC and REVOKEANDACK or has fulfilled on-chain. As we saw previously, both these scenarios would lead to `pendingDiffs`(*Bob*) containing the (`expayid`, `payid`, _) tuple. In the second and third cases (line 1, Fig. 59 and line 1, Fig. 60) once again the conditions describe situations which are possible only if *Bob* has

already successfully fulfilled, which means that he has necessarily caused $\mathcal{S}$ to send the payment to $\mathcal{F}_{\text{PayNet}}$, which has added it to `pendingDiffs`(Bob). We conclude that the halt of line 17, Fig. 19 cannot occur.

To conclude, given $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$, it is $\text{EXEC}^{\mathcal{F}_{\text{PayNet,Open}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open}},\mathcal{E}} = \text{EXEC}^{\mathcal{F}_{\text{PayNet,Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}},\mathcal{E}}$. If we allow for forgeries again, i.e. if we allow the event $P \vee Q \vee R \vee S$, we observe that $\Pr[P \vee Q \vee R \vee S] \leq nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) + nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)$, where n is the number of players, m is the maximum channels a player can open and p is the maximum number of updates a player can perform. We thus deduce that

$$\forall k \in \mathbb{N},\ \text{PPT}\ \mathcal{E},$$
$$|\Pr[\text{EXEC}^{\mathcal{F}_{\text{PayNet,Open}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open}},\mathcal{E}} = 1] - \Pr[\text{EXEC}^{\mathcal{F}_{\text{PayNet,Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}},\mathcal{E}} = 1]| \leq$$
$$nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) +$$
$$nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)\ .$$

□

Simulator $\mathcal{S}$

Like $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$. Differences:

1: Upon receiving (FORCECLOSECHANNEL, `receipt`, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet}}$:
2: simulate Fig. 44 receiving (FORCECLOSECHANNEL, `receipt`, *pchid*) with *Alice*'s ITI
3: **if** during the simulation above, line 11 of Fig. 44 is simulated in *Alice*'s ITI **then**
4: send (NEWPAYMENTS, `payment`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `payment` consists of the tuple just added to the simulated `paymentsToReport`, appended with the *payid* of line 9 (Fig. 44, line 11)
5: upon receiving (CONTINUE) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
6: **end if**

7: Upon receiving (CLOSECHANNEL, `receipt`, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet}}$:
8: simulate the interaction between *Alice* and *Bob* in Fig. 45, starting with *Alice* receiving (CLOSECHANNEL, `receipt`, *pchid*) with *Alice*'s ITI. If *Bob* is honest, simulate his part as well; if not, let $\mathcal{A}$ handle his part.

9: every time `closedChannels` of *Alice* is updated with data from a `channel` (Fig. 44, line 17, Fig 45, line 26 and Fig. 33, line 26), send (CLOSEDCHANNEL, `channel`, *Alice*) to $\mathcal{F}_{\text{PayNet}}$ and expect (CONTINUE) from $\mathcal{F}_{\text{PayNet}}$ to resume simulation

Fig. 61.

Lemma 7.

$$\forall k \in \mathbb{N},\ \ PPT\ \mathcal{E},$$
$$|\Pr[EXEC^{\mathcal{F}_{\text{PayNet}},\text{Pay},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}},\mathcal{E}} = 1] - \Pr[EXEC^{\mathcal{F}_{\text{PayNet}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}} = 1]| \leq$$
$$nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) +$$
$$nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)\ \ .$$

Proof. Like in the previous proof, we here also assume that $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$ holds.

When $\mathcal{E}$ sends (FORCECLOSECHANNEL, `receipt`, *pchid*) to *Alice*, in the ideal world, if it is not the first such message to *Alice* for this channel, the message is ignored (Fig. 20, line 10). Similarly in the real world, if there has been another such message, *Alice* ignores it (Fig. 44, lines 18 and 2).

In the case that it is indeed the first FORCECLOSECHANNEL message for this channel, in the ideal world $\mathcal{F}_{\text{PayNet}}$ takes note that this close is pending (Fig. 20, lines 8-9) and stops serving more requests for this channel (line 10), before asking $\mathcal{S}$ to carry out channel closing (Fig. 20, line 11). $\mathcal{S}$ then simulates the response to the original message from $\mathcal{E}$ with *Alice*'s ITI (Fig. 61). Observe that, since $\mathcal{F}_{\text{PayNet}}$ has ensured that this is the first request for closing this particular channel, the simulated check of line 2 in Fig. 44 always passes and the rest of Fig. 44 is executed. In the real world, the check also passes (since we are in the case where this is the first closing message) and Fig. 44 is executed by the real *Alice* in its entirety. Therefore, when $\mathcal{E}$ sends FORCECLOSECHANNEL, no opportunity for distinguishability arises.

In the ideal world, when the simulated $\mathcal{A}$ attempts to cooperatively close a channel by instructing corrupted *Alice* to send a SHUTDOWN message to honest *Bob*, $\mathcal{S}$ simply simulates the receiving of the message by *Bob* and subsequently carries out the rest of the simulation. In the real world, such a message by the adversarially controlled *Alice* would result in *Bob* and the rest of the ITIs following the same steps as in the simulated case, therefore the two behaviours are indistinguishable. Note that $\mathcal{F}_{\text{PayNet}}$ handles gracefully the fact that it is not notified of the closing negotiation until it sees the closing transaction on-chain, due to the check and addition of line 2 and 3 respectively in Fig. 21.

When $\mathcal{E}$ sends (CLOSECHANNEL, `receipt`, *pchid*) to *Alice*, in both the ideal and the real world the request will be further processed only if the identical checks of Fig. 20, line 2 and Fig. 45, line 2 respectively pass. Further, in the ideal world, if *Alice* has already received a closing message for this channel (either CLOSECHANNEL or FORCECLOSECHANNEL), the message is ignored (Fig. 20, lines 5 and 10). Likewise in the real world, if *Alice* has received another such message, she ignores it (Fig. 45, line 3 and Fig. 44, line 18).

In the case that this is the first closing message for this channel, in the ideal world $\mathcal{F}_{\text{PayNet}}$ takes note that this close is pending (Fig. 20, lines 3-4) and stops rerving more CLOSECHANNEL requests for this channel (Fig. 20, line 5), before asking $\mathcal{S}$ to carry out the cooperative channel closing negotiation (Fig. 20, line 6). $\mathcal{S}$ then simulates the negotiation, respecting the fact that the counter-

party may be corrupted (Fig. 61, line 8 and Fig. 45). Observe that all checks of Fig. 45 (lines 2, 7, 16, 24, 25 and 29) will either succeed or fail in both the real and the ideal world in the same manner, since the same code is run in both cases (simulated in the ideal, directly in the real) and the inputs and state are, as we have seen, indistinguishable up to that point. Furthermore, for the same reason, if *Bob* is corrupted in the ideal world at the moment when *Alice* sends him SHUTDOWN, he would also be corrupted in the real world at the corresponding moment. Therefore, when $\mathcal{E}$ sends CLOSECHANNEL, no opportunity for distinguishability arises.

When $\mathcal{E}$ sends (GETNEWS) to *Alice*, in the ideal world $\mathcal{F}_{\text{PayNet}}$ sends (NEWS, `newChannels`(*Alice*), `closedChannels`(*Alice*), `updatesToReport`(*Alice*), `paymentsToReport`(*Alice*)) to $\mathcal{E}$ and empties these fields (Fig. 23, lines 12-14). In the real world, *Alice* sends (NEWS, `newChannels`, `closedChannels`, `updatesToReport`, `paymentsToReport`) to $\mathcal{E}$ and empties these fields as well (Fig. 32, lines 28-29). `newChannels`(*Alice*) in the ideal world is populated in two cases: First, when $\mathcal{F}_{\text{PayNet}}$ receives (CHANNELOPENED) after *Alice* has previously received (CHECKFORNEW) (Fig. 14, line 25). This happens when the simulated *Alice* ITI handles a FUNDINGLOCKED message from *Bob* (Fig. 52, line 11). In the real world *Alice* would have modified her `new-Channels` while handling *Bob*'s FUNDINGLOCKED (Fig. 31, line 13), thus as far as this case is concerned, `newChannels` has the same contents in the real world as does `newChannels`(*Alice*) in the ideal. The other case when `newChannels`(*Alice*) is populated is when $\mathcal{F}_{\text{PayNet}}$ receives (FUNDINGLOCKED) after *Bob* has previously received (CHECKFORNEW) (Fig. 14, line 17). This (FUNDINGLOCKED) can only be sent by $\mathcal{S}$ if *Alice* is honest and right before the receiving of (FUNDINGLOCKED) is simulated with her ITI (Fig. 52, lines 2-7). In the real world, *Alice*'s `newChannels` would be populated upon handling the same (FUNDINGLOCKED). Therefore the `newChannels` part of the message is identical in the real and the ideal world at every moment when $\mathcal{E}$ can send (GETNEWS).

Moving on to `closedChannels`(*Alice*), we observe that $\mathcal{F}_{\text{PayNet}}$ adds `channel` information when it receives (CLOSEDCHANNEL, `channel`, *Alice*) from $\mathcal{S}$ (Fig. 20, line 14), which in turn happens exactly when the simulated *Alice* ITI adds the `channel` to her `closedChannels` (Fig. 61, line 9). Therefore the real and ideal `closedChannels` are always synchronized.

Regarding `updatesToReport`, in the real world it is populated exclusively in line 10 of Fig. 41. In the ideal world on the other hand, it is updated in line 11 of Fig. 15, which is triggered only by a (NEWUPDATE) message by $\mathcal{S}$. This message is sent only when line 10 of Fig. 41 is simulated by $\mathcal{S}$ (Fig. 56, line 12). In the real world, this happens only after receiving a valid (REVOKEANDACK) message from the channel counterparty and after first having sent a corresponding (COMMITMENTSIGNED) message (Fig. 41, line 2 and Fig. 40, lines 5 and 16), which happens only after receiving (COMMIT) from $\mathcal{E}$. In the ideal world a simulation of the same events can only happen in the exact same case, i.e. when $\mathcal{E}$ sends an identical (COMMIT) to the same player. Indeed, $\mathcal{F}_{\text{PayNet}}$ simply forwards this message to $\mathcal{S}$ (Fig. 23, line 6), who in turn simply simulates the response to the message with the simulated ITI that corresponds to the player

that would receive the message in the real world (Fig. 56, line 10). We conclude that the `updatesToReport` sent to $\mathcal{E}$ in either the real or the ideal world are always identical.

As far as `paymentsToReport` is concerned, in the real world it is populated in one of following cases: line 8 of Fig. 43, line 18 of Fig. 33, line 14 of Fig. 42, and line 16 of Fig. 41, line 11 of Fig. 44. All of these cases can occur in the simulation of the ideal world in the same way, which would result in $\mathcal{S}$ informing $\mathcal{F}_{\text{PayNet}}$ (lines 11-12 of Fig. 54, lines 24-25 of Fig. 54, lines 3-4 of Fig. 56, lines 15-16 of Fig. 56, and lines 3-4 of Fig. 61 respectively) of the exact same payments. In turn, $\mathcal{F}_{\text{PayNet}}$ would store the new payments (Fig. 15, line 14) and report them back to $\mathcal{E}$ upon receiving GETNEWS. We therefore conclude that the `paymentsToReport` that are sent to $\mathcal{E}$ are exactly the same in the ideal and the real world.

Because $\mathcal{S}$ now sends an additional NEWPAYMENTS message (line 4, Fig. 61) which prompts $\mathcal{F}_{\text{PayNet}}$ to populate `paymentsToReport` (line 14, Fig. 15), we have to revisit the halt of line 10 in Fig. 16 to ensure that it still cannot occur with non-negligible probability. Indeed, the halt happens if either a payment has been reported as successful and is found in the `pendingDiffs` of one of the two endpoints, but the payment is now found by $\mathcal{F}_{\text{PayNet}}$ to have failed. This particular payment is considered as failed by $\mathcal{F}_{\text{PayNet}}$ only if $\mathcal{S}$ has included it in `charged` (line 3, Fig. 16) and the first element is $\perp$ (line 8, Fig. 16), which can happen only in case line 19 of Fig. 54 was previously executed by $\mathcal{S}$. As we saw in detail in the proof of Lemma 6 when focussing on the fact that the same halt could never occur however, in the case of line 19 of Fig. 54 none of the events that trigger $\mathcal{S}$ to send a report of the success of this particular payment to $\mathcal{F}_{\text{PayNet}}$ via a NEWPAYMENTS message could have occurred, therefore the halt in question can still never happen.

Lastly, in the ideal world, whenever (READ) is sent to $\mathcal{G}_{\text{Ledger}}$ and a reply is received, the function `checkClosed` (Fig. 21) is called with the reply of the $\mathcal{G}_{\text{Ledger}}$ as argument. This function does not generate new messages, but may cause the $\mathcal{F}_{\text{PayNet}}$ to halt. We will now prove that this never happens.

$\mathcal{F}_{\text{PayNet}}$ halts in line 17 of Fig. 21 in case a channel is closed without using a commitment transaction. Similarly to event E in the proof of Lemma 6, this event is a subset of P and thus is impossible to happen given that we assume $\neg P$.

$\mathcal{F}_{\text{PayNet}}$ halts in line 20 of Fig. 21 in case a malicious closure by the counterparty was successful, in spite of the fact that *Alice* polled in time to apply the punishment. A (POLL) message to *Alice* within the prescribed time frame (line 19) would cause $\mathcal{F}_{\text{PayNet}}$ to alert $\mathcal{S}$ (Fig. 22, line 17), who in turn would submit the punishment transaction in time to prevent the counterparty from spending the delayed payment (Fig. 33, lines 22-24). Therefore the only way for a malicious counterparty to spend the delayed output before *Alice* has the time to punish is by spending the punishment output themself. This however can never happen, since this event would be a subset of either R, if $\texttt{remoteCom}_n$ (i.e. the counterparty closed the channel) is in Σ_{Alice}, or Q, if $\texttt{localCom}_n$ is in Σ_{Alice} (i.e. *Alice* closed the channel).

$\mathcal{F}_{\text{PayNet}}$ halts in line 27 of Fig. 21 in case $\mathcal{E}$ has asked for the channel to close, but too much time has passed since. This event cannot happen for two reasons. First, regarding elements in `pendingClose`(*Alice*), because $\mathcal{F}_{\text{PayNet}}$ forwards a (FORCECLOSECHANNEL) message to $\mathcal{S}$ (Fig. 20, line 11) for every element that it adds to `pendingClose` (Fig 20, line 9) and this causes $\mathcal{S}$ to submit the commitment transaction to $\mathcal{G}_{\text{Ledger}}$ (Fig. 44, line 19). This transaction is necessarily valid, because there is no other transaction that spends the funding transaction of the channel, according to the first check of line 26 of Fig. 21. $\mathcal{F}_{\text{PayNet}}$ halts in this case only if it is sure that the chain has grown by $(2 + r)$ `windowSize` blocks, and thus if the closing transaction had been submitted when (FORCECLOSECHANNEL) was received, it should have been necessarily included (Proposition 1). Second, elements added to `pendingClose`(*Alice*) as a reaction to a (CLOSECHANNEL) message are built with an infinite waiting time (Fig. 20, line 4). Third, every element added to `closedChannels` (Fig. 44, line 17 and Fig. 33, line 26) corresponds to a submission of a closing transaction for the same channel (Fig. 44, line 19), or to a channel for which the closing transaction is already in the ledger state (Fig. 33, line 1). In both cases, the transaction has been submitted at least $(2 + r)$ `windowSize` blocks earlier, thus again by Proposition 1 it is impossible for the transaction not to be in the ledger state. Therefore $\mathcal{F}_{\text{PayNet}}$ cannot halt in line 27 of Fig. 21. We deduce that, given $\neg P \wedge \neg Q \wedge \neg R$, the execution of `checkClosed` by $\mathcal{F}_{\text{PayNet}}$ does not contribute any increase to the probability of distinguishability. Put otherwise, given $\neg P \wedge \neg Q \wedge \neg R$, it is $\text{EXEC}^{\mathcal{F}_{\text{PayNet}},\text{Pay},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}},\mathcal{E}} = \text{EXEC}^{\mathcal{F}_{\text{PayNet}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}}$.

$\mathcal{F}_{\text{PayNet}}$ halts in line 31 of Fig. 21 in case all *Alice*'s channels are closed on-chain and either *Alice*'s off-chain balance is not equal to zero, or if her on-chain balance is not the expected one, as reported by $\mathcal{S}$. This event can never happen for the following reasons. Firstly, as we have seen, $\mathcal{S}$ reports all updates with a (NEWUPDATE) message (Fig. 56, line 12) and a (RESOLVEPAYS) message; upon receiving the latter and given that it doesn't halt, $\mathcal{F}_{\text{PayNet}}$ updates `offChainBalance`(*Alice*) if she is the payer or payee of one of the resolved payments (Fig. 18, lines 17, 22 and 23). Secondly, upon closure of each channel, $\mathcal{F}_{\text{PayNet}}$ would have halted if the closing balance were not the expected one (Fig. 21, line 19), an event that cannot happen as we have already proven. Lastly, upon each channel opening and closing, $\mathcal{F}_{\text{PayNet}}$ updates `offChainBalance`(*Alice*) and `onChainBalance`(*Alice*) to reflect the event (Fig. 14, lines 21 and 22 and Fig. 21, lines 9 or 11 respectively). Therefore, it is impossible for $\mathcal{F}_{\text{PayNet}}$ to halt here.

Similarly to the previous proof, if we allow for forgeries again, i.e. if we allow the event $P \vee Q \vee R \vee S$, we observe that $\Pr[P \vee Q \vee R \vee S] \leq nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) + nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)$, where n is the number of players, m is the maximum channels a player can open and p is the

maximum number of updates a player can perform. We thus deduce that

$$\forall k \in \mathbb{N},\ \text{PPT}\ \mathcal{E},$$
$$|\Pr[\text{EXEC}^{\mathcal{F}_{\text{PayNet}},\text{Pay},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}},\mathcal{E}} = 1] - \Pr[\text{EXEC}^{\mathcal{F}_{\text{PayNet}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}} = 1]| \leq$$
$$nm \cdot \text{E-ds}(k) + 3np \cdot \text{E-ibs}(k) +$$
$$nmp \cdot \text{E-share}(k) + \text{E-prf}(k) + nm \cdot \text{E-master}(k)\ .$$

□

Proof of Theorem 2. The theorem is a direct result of Lemmas 3-7. □

11.1 Forgery algorithms

Proposition 2. *Let $k \in \mathbb{N}$, p a polynomial an arbitrary distribution T and the uniform distribution U over a set A of size $p(k)$. It is*

$$\Pr[T = U] = \frac{1}{p(k)}$$

Proof.

$$\Pr[T = U] = \sum_{a \in A} \Pr[T = a \wedge U = a] = \sum_{a \in A} \frac{1}{p(k)} \Pr[U = a] =$$
$$= \frac{1}{p(k)} \sum_{a \in A} \Pr[U = a] = \frac{1}{p(k)}$$

□

Proposition 3. $\forall \mathcal{E}, \Pr[P] \leq nm \cdot \text{E-ds}(k)$

Proof. Let $\Pr[P] = a$ for an unmodified execution. $\mathcal{A}_{\text{ds}}$ simulates faithfully $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$, since it does the following two changes. The first is to replace one p_F public key with the public key pk given by the challenger. Both keys are generated by KEYGEN(), thus their distribution is identical. The second is to replace signatures done by s_F with signatures done by the challenger with sk. Both signatures are generated with SIGNDS() and thus their distribution is identical. We deduce that, within the simulated execution, $\Pr[P] = a$.

At the beginning of an execution, *Alice* and i are chosen uniformly at random, therefore given P, by Proposition 2 we have that

$$\Pr[\mathcal{A}_{\text{ds}} \text{ chooses correct keypair}] = \frac{1}{nm}\ .$$

Since the selection happens independently from the forgery, we deduce that

$$\Pr[\mathcal{A}_{\text{ds}} \text{ wins } \mathsf{EUF\text{-}CMA}] = \frac{a}{nm}$$

Algorithm EUF-CMA forgery

$\mathcal{A}_{\text{ds}}(\text{INIT}, pk)$:

- Choose uniformly at random *Alice* from the set of players $\mathcal{P}$ of an execution $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$
- Choose uniformly at random i from $\{1,\ldots,m\}$
- Simulate internally $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$ with $\mathcal{E}_P$
- When *Alice* opens her i-th channel, replace p_F of KEYGEN() in Fig. 24, line 20 with pk
- Whenever $\text{SIGNDS}(M, s_F)$ is called, ask challenger for the signature σ with (unknown) sk on M and use that instead
- If event P takes place and the forged signature is valid by pk, retrieve forged signature σ^* and the corresponding transaction m^* and output (m^*, σ^*)
- If the simulated execution completes and *Alice* has opened less than i channels, or if no forgery happened, or if a forgery for another player/channel happened, return FAIL

Fig. 62. wins EUF-CMA game

Since the Digital Signatures scheme used during the execution is assumed to be EUF-CMA-secure, it is

$$\Pr[\mathcal{A}_{\text{ds}} \text{ wins EUF-CMA}] \leq \text{E-ds}(k) \Rightarrow \forall \mathcal{E}, a \leq nm \cdot \text{E-ds}(k) \ .$$

□

Proposition 4. *Let Q, n, p be as defined in the proof of Lemma 6. It is*

$$\forall \mathcal{E}, \Pr[Q] \leq 3np \cdot \text{E-ibs}(k) \ .$$

Proof. Let $\Pr[Q] = b$ for an unmodified execution. $\mathcal{A}_{\text{ibs}}$ simulates faithfully $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$, since it does the following two changes. The first is to replace one $ph_{j,n}$ public key with $pk \leftarrow \text{PUBKEYDER}(mpk, ph_{\text{com},n})$, where mpk is given by the challenger. Both mpk and the normally used phb_j are generated by KEYDER(), thus their distribution is identical. The second is to replace signatures done by $sh_{j,n}$ with signatures done by the challenger with $sk \leftarrow \text{KEYDER}(mpk, msk, ph_{j,n})$. Both signatures are generated with SIGNIBS() and thus their distribution is identical. We deduce that, within the simulated execution, $\Pr[Q] = b$.

At the beginning of an execution, *Alice*, i and j are chosen uniformly at random, therefore given Q, by Proposition 2 we have that

$$\Pr[\mathcal{A}_{\text{ibs}} \text{ chooses correct keypair}] = \frac{1}{3np} \ .$$

Since the selection happens independently from the forgery, we deduce that

$$\Pr[\mathcal{A}_{\text{ibs}} \text{ wins IBS-EUF-CMA}] = \frac{b}{3np}$$

Algorithm IBS-EUF-CMA forgery

$\mathcal{A}_{\text{ibs}}(\text{INIT}, mpk)$:

- Choose uniformly at random *Alice* from the set of players $\mathcal{P}$ of an execution $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$
- Choose uniformly at random i from $\{1,\ldots,p\}$
- Choose uniformly at random j from $\{\text{pay}, \text{dpay}, \text{htlc}\}$
- Simulate internally $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$ with $\mathcal{E}_Q$
- When *Alice* performs her i-th opening or update, replace the $ph_{j,n}$ output of $\text{KEYDER}(phb_j, shb_j, ph_{\text{com},n})$ with $pk \leftarrow \text{PUBKEYDER}(mpk, ph_{\text{com},n})$
- Whenever $\text{SIGNIBS}(M, sh_{j,n})$ is called, ask challenger for the signature σ with (unknown) $sk \leftarrow \text{KEYDER}(mpk, msk, ph_{\text{com},n})$ on M and use that instead
- If event Q takes place and the forged signature is valid by pk, retrieve forged signature σ^* and the corresponding transaction m^* and output $(m^*, ph_{\text{com},n}, \sigma^*)$
- If the simulated execution completes and *Alice* has updated or opened a channel less than i times, or if no forgery happened, or if a forgery for another player/opening/update happened, return FAIL

Fig. 63. wins IBS-EUF-CMA game

Since the Identity Based Signatures scheme used during the execution is assumed to be IBS-EUF-CMA-secure, it is

$$\Pr[\mathcal{A}_{\text{ibs}} \text{ wins IBS-EUF-CMA}] \leq \text{E-ibs}(k) \Rightarrow$$
$$\forall \mathcal{E}, b \leq 3np \cdot \text{E-ibs}(k) \ .$$

□

Proposition 5. $\forall \mathcal{E}, \Pr[R] \leq nmp \cdot \text{E-share}(k) + \text{E-prf}(k)$

Proof. First we observe that the halting of the simulation on an additional update does not interfere with the probability of the desired forgery taking place because such a forgery can only occur if *Alice* has broadcast `localCom`, which prevents her from further updating the channel. Therefore such halts happen only after an event that extinguishes the hope for a successful forgery.

Let $\Pr[R] = c$ for the unmodified execution. While doing the simulation of $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$, $\mathcal{A}_{\text{share}}$ does the following change to the execution. It replaces a single $ph_{\text{com},j}$ public key with the public key pk which is given by the challenger. pk is generated by KEYSHAREGEN() with fresh randomness, whereas in an unmodified execution $ph_{\text{com},j}$ is generated by KEYSHAREGEN(), using as its randomness prand $\leftarrow \text{PRF}(\texttt{seed}, j)$. Given though that prand is not used anywhere else and the fact that the computational distance of an output of a PRF from true randomness is at most E-prf(k), we deduce that the computational distance of an unmodified and the modified executions are at most E-prf(k), therefore for the modified execution it is $\Pr[R] \in [c - \text{E-prf}(k), c + \text{E-prf}(k)]$.

Algorithm share-EUF forgery

$\mathcal{A}_{\text{share}}(\text{INIT})$:

- Choose uniformly at random *Alice* from the set of players $\mathcal{P}$ of an execution $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$
- Choose uniformly at random i from $\{1,\ldots,m\}$
- Choose uniformly at random j from $\{1,\ldots,p\}$
- Simulate internally $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$ with $\mathcal{E}_R$
- When *Alice* opens a channel for the i-th time, save $(phb_{\text{rev}}, shb_{\text{rev}})$ (generated from MASTERKEYGEN() in Fig. 24, line 25) as (mpk, msk) and send $(mpk, 1)$ to challenger, to receive key pk
- The j-th time *Alice* calls KEYSHAREGEN() to produce a per commitment pair $(ph_{\text{com},j}, sh_{\text{com},j})$ for the chosen channel (either during opening or during an update), replace its output with the next unused pk
- If *Alice* attempts to update the chosen channel once more and has to send $sh_{\text{com},j}$ to the counterparty, stop simulation and return FAIL
- If event R takes place and the forged signature is valid by pk, retrieve forged signature σ^* and the corresponding transaction m^* and output (m^*, σ^*)
- If the simulated execution completes and *Alice* has opened less than i channels, or if no forgery happened, or if a forgery for another player/channel happened, return FAIL

Fig. 64. wins share-EUF game

At the beginning of an execution, *Alice*, i and j are chosen uniformly at random, therefore given R, by Proposition 2 we have that

$$\Pr[\mathcal{A}_{\text{share}} \text{ chooses correct keypair}] = \frac{1}{nmp} \ .$$

Since the selection happens independently from the forgery, we deduce that

$$\Pr[\mathcal{A}_{\text{share}} \text{ wins share-EUF}] \in \left[\frac{c - \text{E-prf}(k)}{nmp}, \frac{c + \text{E-prf}(k)}{nmp}\right] \ .$$

Since the Combined Signatures scheme used is assumed to be share-EUF-secure, it is

$$\Pr[\mathcal{A}_{\text{share}} \text{ wins share-EUF}] \leq \text{E-share}(k) \Rightarrow$$
$$\forall \mathcal{E}, c \leq nmp \cdot \text{E-share}(k) + \text{E-prf}(k) \ .$$

□

Proposition 6. *Let S, n, m be as defined in the proof of Lemma 6. It is*

$$\forall \mathcal{E}, \Pr[S] \leq nm \cdot \text{E-master}(k) \ .$$

Algorithm master-EUF-CMA forgery

$\mathcal{A}_{\text{master}}(\text{INIT}, mpk)$:

- Choose uniformly at random *Alice* from the set of players $\mathcal{P}$ of an execution $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$
- Choose uniformly at random i from $\{1,\ldots,m\}$
- Simulate internally $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$ with $\mathcal{E}_S$
- When *Alice* opens a channel for the i-th time, replace phb_{rev} (generated from MASTERKEYGEN() in Fig. 24, line 25) with mpk
- Ignore calls to COMBINEKEY() that need the missing msk and assume that the resulting combined secret key is known (to satisfy line 21 of Fig. 33 if needed).
- Whenever $\text{SIGNCS}(M, sh_{\text{rev},n})$ is called within this channel, ask challenger for the signature σ with signing key $csk \leftarrow \text{COMBINEKEY}(mpk, msk, pt_{\text{com},n}, st_{\text{com},n})$ on M by sending them $(pt_{\text{com},n}, st_{\text{com},n}, M)$ and use that instead
- If event S takes place and the forged signature is valid by $cpk \leftarrow \text{COMBINEPUBKEY}(mpk, pt_{\text{com},n})$ for some $pt_{\text{com},n}$ of the channel, retrieve forged signature σ^* and the corresponding transaction m^* and output (m^*, σ^*)
- If the simulated execution completes and *Alice* has opened less than i channels, or if no forgery happened, or if a forgery for another player/channel happened, return FAIL

Fig. 65. wins master-EUF-CMA game

Proof. Let $\Pr[S] = d$ hold for the unmodified execution. When it is simulating $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$, $\mathcal{A}_{\text{master}}$ does the following two changes to the execution. Firstly, it replaces a single phb_{rev} public master key with mpk which is given by the challenger. Both mpk and phb_{rev} are generated by MASTERKEYGEN() with fresh randomness, thus their distribution is identical. Secondly, it replaces signatures done by the secret key $sh_{\text{rev},n} \leftarrow \text{COMBINEKEY}(phb_{\text{rev}}, shb_{\text{rev}}, pt_{\text{com}n}, st_{\text{com},n})$ with signatures created by the challenger with the secret key resulting from executing $\text{COMBINEKEY}(mpk, msk, pt_{\text{com}n}, st_{\text{com},n})$, thus the distribution of the two signatures is identical. We deduce that for the modified execution it is $\Pr[S] = d$.

At the beginning of an execution, *Alice*and i are chosen uniformly at random, therefore given S, by Proposition 2 we have that

$$\Pr[\mathcal{A}_{\text{master}} \text{ chooses correct keypair}] = \frac{1}{nm} \ .$$

Since the selection happens independently from the forgery, we deduce that

$$\Pr[\mathcal{A}_{\text{master}} \text{ wins master-EUF-CMA}] \geq \frac{d}{nm}$$

Since the Combined Signatures scheme used during the execution is assumed to be master-EUF-CMA-secure, it is

$$\Pr[\mathcal{A}_{\text{master}} \text{ wins master-EUF-CMA}] \leq \text{E-master}(k) \Rightarrow$$
$$\forall \mathcal{E}, d \leq nm \cdot \text{E-master}(k) \ .$$

□

12 Future Work and Conclusion

In order to remain tractable, the current analysis omits some parts of the lightning specification. In particular, the specification defines how intermediaries of multi-hop payments can charge a fee for their service. Furthermore, the per-update secret generation is not done with a PRF according to the specification: an optimisation that reduces the storage overhead for the counterparty is used instead. The security of this optimisation however has not been yet formally inspected. Additionally, the specification provisions for a number of failure messages that help in keeping counterparties informed of issues with requested payments and in alleviating the problem of unneeded precautionary channel closures. Transactions that are added on-chain offer a fee to the blockchain miners (unrelated to the fee of the off-chain multi-hop payments). When closing a channel cooperatively, this fee is contributed by both counterparties, therefore the closing sequence of the specification includes an iterative negotiation of said fee where the two parties repeatedly propose a value based on their settings until they converge to a compromise or fail to agree. Lastly, most Bitcoin nodes do not relay transactions that include outputs with tiny amounts of coins, a.k.a "dust" outputs, to avoid bloating the blockchain. The lightning specification provides extensive instructions as to how to prune such outputs.

All aforementioned parts of the protocol were not analysed so that the security of the core parts of the lightning protocol could be discussed without distractions. In order for the analysis to cover the entirety of the current version of the lightning specification however, the aforementioned features should be incorporated and their security should be proven. This expansion of the analysis is left as future work.

In a different direction, big parts of our main security proof consist of an exhaustive enumeration of the possible messages that $\mathcal{E}$ and $\mathcal{A}$ can send to the protocol, the simulator or the functionality and tracking how such messages would change the flow of the execution of the ideal and the real world. It is then argued that in all cases the messages that would be sent to $\mathcal{E}$ and $\mathcal{A}$ are indistinguishable. These parts of the proof are good candidates for rewriting in the environment of an automated proof assistant [38] to instill additional certainty that all possible execution paths are indeed checked and do not contain subtle sources of distinguishability. Combining our results with the recent mechanization of UC via Easycrypt [39] would be a natural and interesting direction for future work.

The lightning specification is not static, but it is continuously undergoing a number of improvements. The most noteworthy upcoming change is the introduction of Pointlocked TimeLocked Contracts (PTLCs). This mechanism replaces HTLCs and promises to combat the "wormhole" attack [30], while increasing privacy. Our work can be modified to cover the case of PTLCs with relative ease. It also provides a suitable framework for future work that aims to shed light on the exact privacy benefit that PTLCs offer as opposed to HTLCs.

The present analysis constitutes the first comprehensive treatment in the Universal Composability framework of a deployed layer-2 protocol on top of a functional ledger. It can be extended and adapted to analyze other similar protocols that achieve different security goals or use another ledger as base layer.

Conclusion. The present work constitutes a fortunate result, since it conclusively proves that software that adhere to the lightning specification cannot lose funds accidentally to a malicious protocol player. Indeed, such a result reinforces trust to the lightning network and acts as a guarantee to the almost 900 bitcoins currently in circulation in the layer-2 protocol.

By leveraging the guarantees provided by the Universal Composability framework, we further assert that the lightning protocol is freely composable with other protocols. As such it can run side by side with arbitrary protocols, or be used as a subroutine to higher-level protocols without needing to prove its security anew.

By separating particular subroutines of the protocol as distinct cryptographic primitives and analysing them individually, we have contributed to its cryptorgaphic agility. Lastly, by keeping it as protocol-agnostic as possible, our payment network functionality can be adapted to express the functional and security requirements of other layer-2 protocols with relative ease.

References

1. Croman K., Decker C., Eyal I., Gencer A. E., Juels A., Kosba A., Miller A., Saxena P., Shi E., Sirer E. G., et al.: On scaling decentralized blockchains. In International Conference on Financial Cryptography and Data Security: pp. 106–125: Springer (2016)
2. Nakamoto S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
3. Garay J., Kiayias A., Leonardos N.: The Bitcoin Backbone Protocol: Analysis and Applications. Cryptology ePrint Archive, Report 2014/765: https://eprint.iacr.org/2014/765 (2014)
4. Pass R., Seeman L., Shelat A.: Analysis of the Blockchain Protocol in Asynchronous Networks. IACR Cryptology ePrint Archive: vol. 2016, p. 454: URL http://eprint.iacr.org/2016/454 (2016)
5. Garay J. A., Kiayias A., Leonardos N.: The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In J. Katz, H. Shacham (editors), Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I: vol. 10401 of *Lecture Notes in Computer Science*: pp. 291–323: Springer: ISBN 978-3-319-63687-0: doi:10.1007/978-3-319-63688-7_10: URL https://doi.org/10.1007/978-3-319-63688-7_10 (2017)

6. Pass R., Shi E.: Hybrid Consensus: Efficient Consensus in the Permissionless Model. In A.W. Richa (editor), 31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria: vol. 91 of *LIPIcs*: pp. 39:1–39:16: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik: ISBN 978-3-95977-053-8: doi:10.4230/LIPIcs.DISC.2017.39: URL https://doi.org/10.4230/LIPIcs.DISC.2017.39 (2017)
7. Micali S.: ALGORAND: The Efficient and Democratic Ledger. CoRR: vol. abs/1607.01341: URL http://arxiv.org/abs/1607.01341 (2016)
8. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf (2016)
9. Pass R., Shi E.: Thunderella: Blockchains with Optimistic Instant Confirmation. In J.B. Nielsen, V. Rijmen (editors), Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II: vol. 10821 of *Lecture Notes in Computer Science*: pp. 3–33: Springer: ISBN 978-3-319-78374-1: doi:10.1007/978-3-319-78375-8_1: URL https://doi.org/10.1007/978-3-319-78375-8_1 (2018)
10. Badertscher C., Maurer U., Tschudi D., Zikas V.: Bitcoin as a transaction ledger: A composable treatment. In Annual International Cryptology Conference: pp. 324–356: Springer (2017)
11. Badertscher C., Gaži P., Kiayias A., Russell A., Zikas V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security: pp. 913–930: ACM (2018)
12. Nicolosi A., Krohn M. N., Dodis Y., Mazières D.: Proactive Two-Party Signatures for User Authentication. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA: The Internet Society: ISBN 1-891562-16-9: URL http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/15.pdf (2003)
13. Dziembowski S., Faust S., Hostáková K.: General State Channel Networks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018: pp. 949–966: doi:10.1145/3243734.3243856: URL https://doi.org/10.1145/3243734.3243856 (2018)
14. Malavolta G., Moreno-Sanchez P., Kate A., Maffei M., Ravi S.: Concurrency and Privacy with Payment-Channel Networks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security: CCS '17: pp. 455–471: ACM, New York, NY, USA: ISBN 978-1-4503-4946-8: doi:10.1145/3133956.3134096: URL http://doi.acm.org/10.1145/3133956.3134096 (2017)
15. Miller A., Bentov I., Kumaresan R., Cordi C., McCorry P.: Sprites and State Channels: Payment Networks that Go Faster than Lightning. ArXiv preprint arXiv:1702.05812 (2017)
16. Dziembowski S., Eckey L., Faust S., Malinowski D.: Perun: Virtual Payment Hubs over Cryptocurrencies. In 2019 2019 IEEE Symposium on Security and Privacy (SP): pp. 344–361: IEEE Computer Society, Los Alamitos, CA, USA: ISSN 2375–1207: doi:10.1109/SP.2019.00020: URL https://doi.ieeecomputersociety.org/10.1109/SP.2019.00020 (2019)
17. Spilman J.: Anti dos for tx replacement. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html (2013)

18. Decker C., Wattenhofer R.: A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In A. Pelc, A.A. Schwarzmann (editors), Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings: vol. 9212 of *Lecture Notes in Computer Science*: pp. 3–18: Springer: ISBN 978-3-319-21740-6: doi:10.1007/978-3-319-21741-3_1: URL https://doi.org/10.1007/978-3-319-21741-3_1 (2015)
19. Lind J., Naor O., Eyal I., Kelbert F., Pietzuch P. R., Sirer E. G.: Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels. In Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018, HAIFA, Israel, June 04-07, 2018: p. 125: ACM: doi:10.1145/3211890.3211904: URL https://doi.org/10.1145/3211890.3211904 (2018)
20. Green M., Miers I.: Bolt: Anonymous Payment Channels for Decentralized Currencies. In Thuraisingham et al. [40]: pp. 473–489: doi:10.1145/3133956.3134093: URL https://doi.org/10.1145/3133956.3134093 (2017)
21. Heilman E., Alshenibr L., Baldimtsi F., Scafuro A., Goldberg S.: TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017: The Internet Society: URL https://www.ndss-symposium.org/ndss2017/ (2017)
22. Khalil R., Gervais A.: Revive: Rebalancing Off-Blockchain Payment Networks. In Thuraisingham et al. [40]: pp. 439–453: doi:10.1145/3133956.3134033: URL https://doi.org/10.1145/3133956.3134033 (2017)
23. Prihodko P., Zhigulin S., Sahno M., Ostrovskiy A.: Flare: An Approach to Routing in Lightning Network: White Paper. https://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf (2016)
24. Sivaraman V., Venkatakrishnan S. B., Alizadeh M., Fanti G. C., Viswanath P.: Routing Cryptocurrency with the Spider Network. CoRR: vol. abs/1809.05088: URL http://arxiv.org/abs/1809.05088 (2018)
25. Kiayias A., Litos O. S. T.: A Composable Security Treatment of the Lightning Network. https://eprint.iacr.org/2019/778 (2019)
26. Canetti R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA: pp. 136–145: doi:10.1109/SFCS.2001.959888: URL https://eprint.iacr.org/2000/067.pdf (2001)
27. Canetti R., Dodis Y., Pass R., Walfish S.: Universally Composable Security with Global Setup. In Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings: pp. 61–85: doi:10.1007/978-3-540-70936-7_4: URL https://doi.org/10.1007/978-3-540-70936-7_4 (2007)
28. Shamir A.: Identity-Based Cryptosystems and Signature Schemes. In Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings: pp. 47–53: doi:10.1007/3-540-39568-7_5: URL https://doi.org/10.1007/3-540-39568-7_5 (1984)
29. Paterson K. G., Schuldt J. C. N.: Efficient Identity-Based Signatures Secure in the Standard Model. In Information Security and Privacy, 11th Australasian Conference, ACISP 2006, Melbourne, Australia, July 3-5, 2006, Proceedings: pp. 207–222: doi:10.1007/11780656_18: URL https://doi.org/10.1007/11780656_18 (2006)

30. Malavolta G., Moreno-Sanchez P., Schneidewind C., Kate A., Maffei M.: Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019 (2019)
31. Danezis G., Goldberg I.: Sphinx: A compact and provably secure mix format. In Security and Privacy, 2009 30th IEEE Symposium on: pp. 269–282: IEEE (2009)
32. Katz J., Lindell Y.: Introduction to Modern Cryptography, Second Edition. CRC Press: ISBN 9781466570269 (2014)
33. Bellare M., Sandhu R. S.: The Security of Practical Two-Party RSA Signature Schemes. IACR Cryptology ePrint Archive: vol. 2001, p. 60: URL http://eprint.iacr.org/2001/060 (2001)
34. Boyd C.: Digital Multisignatures. Cryptography and Coding: pp. 241–246: URL https://ci.nii.ac.jp/naid/10013157942/en/ (1986)
35. Ganesan R.: Yaksha: augmenting Kerberos with public key cryptography. In 1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, USA, February 16-17, 1995: pp. 132–143: doi:10.1109/NDSS.1995.390639: URL https://doi.org/10.1109/NDSS.1995.390639 (1995)
36. MacKenzie P. D., Reiter M. K.: Two-Party Generation of DSA Signatures. In Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings: pp. 137–154: doi:10.1007/3-540-44647-8_8: URL https://doi.org/10.1007/3-540-44647-8_8 (2001)
37. Ganesan R., Yacobi Y.: A secure joint signature and key exchange system. Bellcore TM: vol. 24531 (1994)
38. McCarthy J.: Computer Programs for Checking Mathematical Proofs. Proceedings of the Fifth Symposium in Pure Mathematics of the American Mathematical Society: pp. 219–227: american Mathematical Society (1961)
39. Canetti R., Stoughton A., Varia M.: EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In 32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019: pp. 167–183: IEEE: ISBN 978-1-7281-1407-1: doi:10.1109/CSF.2019.00019: URL https://doi.org/10.1109/CSF.2019.00019 (2019)
40. Thuraisingham B. M., Evans D., Malkin T., Xu D. (editors): Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017: ACM: ISBN 978-1-4503-4946-8: doi:10.1145/3133956: URL https://doi.org/10.1145/3133956 (2017)