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 takes into account a global ledger functionality for which previous work [Badertscher et al., Crypto'17] has demonstrated its realisability by the Bitcoin blockchain protocol. As a result, our treatment delineates exactly how the security guarantees of the protocol depend on the properties of the underlying ledger. 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] or 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 changing the threshold required for security or the underlying cryptographic assumptions and setup that is needed. The additional side-effect of such solutions is that they are fundamentally incompatible with the Bitcoin blockchain, which is arguably the currently most successful deployed instance of the blockchain protocol.

The alternative approach is to build an *overlay* protocol that utilises the blockchain protocol as a "fall back" layer, while facilitating "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 and provide a security threshold over 1/3, we know that some additional assumption would be required for the overlay protocol to work.

The first instance of this approach and by far the most widely known and utilised so far, came with the *lightning network* [8][3] that provided an overlay mechanism over the Bitcoin blockchain that introduces and takes advantage of 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. If a payment transaction is confirmed by the parties implicated in a payment then, should the parties wish it, it is guaranteed that *eventually* the ledger will record "gross" settlement transactions between the parties in the path of the payment transaction that are consistent with it. Deviations from this guarantee are cryptographically feasible but deincentivised via on-chain penalties. 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 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 latency. Nevertheless, the lightning network is currently the only option that readily interoperates with the Bitcoin blockchain.

[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`.

Despite the importance of the lightning network for low latency payments over Bitcoin 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.

Our Results. We present the first, to our knowledge, complete security analysis of the lightning network in the universal composition (UC) setting. We model the payment overlay that the lightning network provides as an 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 that are provided by the lightning network. 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 of the parties registered and ensures that when a channel closes the on-chain balances are in line with the off-chain balances. In order to handle adversarial deviations in multi-hop payments, the functionality permits the adversary to determine the outcome of each payment by choosing either one of the following options (i) let it go through as requested, (ii) charge it to an adversarial party along the path, (iii) charge it to *negligent* honest party along the path. This last outcome is a crucial characteristic of the security that is provided by the lightning network to its participants: honest parties are expected to poll the functionality at a certain specific rhythm that corresponds to their level of involvement in the network and the properties of the underlying ledger. If a party misses that requirement it is identified as negligent by the functionality and may lose funds.
2. We identify for the first time the exact polling requirements that are imposed by the lightning network to the honest parties that are participating so that they do not lose funds as a function of the underlying parameters of the ledger functionality over which the lightning network is overlaid. We describe our $\mathcal{F}_{\text{PayNet}}$ given the global ledger functionality defined in [10], and further refined in [11], for which we know already that is realised by the Bitcoin blockchain. The functionality 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)

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

the first type of polling refers to monitoring for closures of channels that the party is one of the end-points and is specified by the user chosen parameter delay, (ii) the second type of polling refers to monitoring for specific events related to receiving and relaying payments; in particular for each payment that a party acts as an intermediary, polling should happen twice within the window of time when the chain in the view of the party advances from blockheight h to blockheight $h' - a$, where h, h' are two blockheight parameters specified in the particular payment path, and a is a (derived) ledger parameter which is 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). Moreover, the two pollings should be separated by a time window that allows the chain to grow by at least a blocks.

3. We provide a complete pseudocode description of the lightning network protocol Π_{LN} and we prove that indeed it realises $\mathcal{F}_{\text{PayNet}}$ under a specific set of cryptographic assumptions. In order to express Π_{LN} in a way that is succinct we identify a number of underlying cryptographic primitives that have been used in the specification of the lightning network in a non-black-box fashion and without reference. Interestingly, while some of these cryptographic primitives are standard (they include a PRF, a Digital Signature scheme and an Identity Based Signature scheme) there is one additional primitive that is slightly less standard and we call *combined digital signature*. A combined digital signature is a special case of an asymmetric two-party digital signature primitive (e.g., see [12] and references therein) with the characteristic that one of the two parties, the shareholder, generates and stores a share of the signing key, while the public-key of the combined signature is determined non-interactively based on public-key information produced by both two 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 specification of lightning is realising 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).

Related Work. A first suggestion for building a unidirectional payment channel appeared in [13]. Bidirectional payment channels were developed in [14] 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 [15], hardward assumptions, see e.g., [16], or extending its functionality beyond payments, to smart contracts, [17] or finally enhancing their privacy, see e.g., [18,19,20]. Additional work looked into developing supporting protocols for the payment networks such as rebalancing [21] or finding routes in a decentralised fashion [22]. With respect to idealising the payment network functionality in the UC setting, a number of previous papers [18,17,15] presented ideal functionalities abstracting the concept. The fundamental 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 (in contrast, previous work utilised "too idealised" ledger functionalities for which it is unknown whether they can be realised).

Organisation. In section 2 we present preliminaries for the model we employ and the relevant cryptographic primitives. In Section 5 we present an overview of the lightning network. 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. Finally, in Section 7 we provide an overview of the security proof. In the appendix, first we give more details on the preliminary primitives, specifically in Sections A,B,C we discuss digital signatures, identity based signatures and pseudorandom functions. Subsequently in Section D we go into more details about combined digital signatures. The transaction structure that is assumed to be provided by the underlying distributed ledger is discussed in Section E. The paynet functionality is presented in detail in Section F. The complete description of protocol Π_{LN} is presented in Section H. The ledger functionality is presented in Section I and finally, the security proof is presented in Section J.

2 Preliminaries

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

Universal Composability framework. In simulation-based security, cryptographic tasks are defined via an ideal "functionality" $\mathcal{F}$, which can be thought of as a trusted third party that gets the inputs of all parties and returns the outputs they expect while also interacting with the adversary. 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 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 in the Universal Composability [23] framework it is possible 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.

Hybrid functionalities used. Both our main protocol and the corresponding functionality use $\mathcal{G}_{\mathrm{Ledger}}$ [10,11] as a hybrid. $\mathcal{G}_{\mathrm{Ledger}}$ 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 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 Appendix I.

Furthermore, $\mathcal{G}_{\text{Ledger}}$ needs two more functionalities: $\mathcal{G}_{\text{CLOCK}}$ and $\mathcal{F}_{\text{N-MC}}^{\Delta}$. $\mathcal{G}_{\text{CLOCK}}$ 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. $\mathcal{F}_{\text{N-MC}}^{\Delta}$ provides an abstraction of the network. A party can send $\mathcal{F}_{\text{N-MC}}^{\Delta}$ a message to be multicast to all other participants of the network. A party can also fetch its new messages from $\mathcal{F}_{\text{N-MC}}^{\Delta}$. $\mathcal{F}_{\text{N-MC}}^{\Delta}$ makes a message available to be fetched immediately after its arrival, but the adversary may delay the arrival for up to Δ rounds. $\mathcal{A}$ can choose different delays for different players and reorder messages. Additionally, he has the power to send his own messages to selected subsets of players.

The protocol and functionality defined in the current work do not make direct use of $\mathcal{G}_{\text{CLOCK}}$ or $\mathcal{F}_{\text{N-MC}}^{\Delta}$. We therefore omit these two in the statements of Lemmas 4-8 and Theorem 1 for simplicity of notation; they should normally appear as hybrids along with $\mathcal{G}_{\text{Ledger}}$. Their exact definition can be found in Appendix I. We also note that $\mathcal{G}_{\text{Ledger}}$ and $\mathcal{G}_{\text{CLOCK}}$ are global functionalities [24], whereas $\mathcal{F}_{\text{PayNet}}$ and $\mathcal{F}_{\text{N-MC}}^{\Delta}$ are not.

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 Appendix E for a complete overview.

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 [25,26], 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 [26] for more details. Other cryptographic primitives used are digital signatures and pseudorandom functions, see Sections A, B, C. Finally, a less common two-party cryptographic primitive is employed that we formalise as *combined digital signatures*, see Section 6.

3 Lightning Network high level overview

Two-party channels. The aim of LN is to enable fast and cheap transactions that do not have to be added to the blockchain, without compromising security. Specifically no additional trust between counterparties is assumed. This is achieved in the following way: Two parties, *Alice* and *Bob*, that have recurring monetary exchanges create one on-chain transaction that locks up some funds, known as the "funding transaction". This transaction is funded by one of the two parties and has a 2-of-2 multisig output, which needs the signatures of both counterparties by their "funding" secret keys in order to be spent. Before actually submitting this transaction though, both parties individually ensure that they hold a transaction that spends the 2-of-2 funding output in a way that gives the funds to the funder, along with the signature of this transaction with the counterparty's funding key. These two transactions (one for each counterparty) are called "commitment transactions". Each party can broadcast her "local" commitment transaction and has signed the "remote" commitment transaction, which is the one held by the counterparty.

Every time they want to make a payment to each other, they exchange a sequence of messages (that include specially crafted signatures) that have two effects. Firstly, a new pair of commitment transactions, along with their signatures by the funding keys, is created. 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. Secondly, the two old commitment transactions are revoked. This ensures that no party can close a channel using an old commitment transaction which may be more beneficial to her than the latest one.

Invalidating past commitment transactions requires some care. The reason is that it is impossible to actually make past commitment transactions invalid without spending the funding output; however, spending it would need an on-chain transaction for each channel update, thus essentially defeating 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 transaction, but has no financial benefit in doing so. At the same time this imposes the requirement that parties are vigilant about observing the blockchain — see below

when we talk about time-locks and how they facilitate a time window within which parties should react.

The punishing mechanism operates as follows. Suppose *Alice* considers posting her old local commitment transaction which has an output that carries her old share of the funds. This output can be spent in two ways: either with a signature by *Alice*'s "delayed payment" secret key which is a usual ECDSA key, or with a signature by *Bob*'s "revocation" secret key, which is also an ECDSA key, but with an additional characteristic that we will explain soon. Thus, if *Alice* broadcasts an old commitment transaction, *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 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, but no party knows its corresponding secret – the key can only be computed by combining one secret value that *Alice* knows and one secret value that *Bob* knows. *Alice* therefore can prevent *Bob* from using his revocation key until she sends this secret value to him. As a result, *Alice* will send the secret value to *Bob* only after the new commitment transaction is built and signed. Thus *Bob* cannot abuse his ability to use the revocation key on a commitment transaction 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 the primitive "combined signature" and we prove that the construction hidden in the LN implementation realizes it in the random oracle model under the assumption that the underlying digital signature scheme is secure in Appendix D.

The last element needed to make updates secure is the so called "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, 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 transaction (the only one that is not revoked) and wait for the timelock to expire in order to spend her funds. The LN specification allows for cooperative channel closure which avoids the need to wait for the timelock, but in the current work this last type of closure is not considered.

As mentioned time locks provide specific time windows within which both parties have to be vigilant 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. LN makes it possible to trustlessly outsource this, 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 an informal 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 Hash TimeLocked 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. In the previous example with *Alice*, *Bob* and *Charlie*, two HTLCs were signed and fulfilled for the payment to go through and the whole interaction was completely off-chain. 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.

Direct 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. Furthermore, LN leverages the Sphinx onion packet scheme [27] 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 $\mathcal{F}_{\text{PayNet}}$ (Appendix F) a functionality that describes the functional and security guarantees given by an ideal payment network. 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) and (closeChannel).

When $\mathcal{F}_{\text{PayNet}}$ receives (openChannel, *Alice*, *Bob*, x, *tid*) from *Alice*, it informs $\mathcal{S}$ of $\mathcal{E}$'s intention 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 $\overrightarrow{\texttt{path}}$. As expected, when $\mathcal{F}_{\text{PayNet}}$ receives (closeChannel, `receipt`, *tid*) from *Alice*, 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. 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 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 and, even though *Alice* has been polling regularly, she did not manage to punish the counterparty. Refer to lines 5 and 10 of Fig. 16 for the exact halting conditions. If $\mathcal{F}_{\text{PayNet}}$ does not halt, it leaks $\mathcal{S}$ 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. $\mathcal{F}_{\text{PayNet}}$ checks that for each payment the charged party was (a) the one that initiated the payment, (b) a malicious party or (c) a honest party that is negligent. The latter case happens when the honest party either did not poll in time to catch a malicious closure (Fig. 12, line 14) or to learn the preimage from an honest closure, or did not enforce the retrieval of her funds by using the preimage to fulfill on chain (Fig. 12, line 23). It also halts if a particular payment resulted in a channel update for which $\mathcal{S}$ did not inform $\mathcal{F}_{\text{PayNet}}$ (Fig. 13, line 10).
- $\mathcal{F}_{\text{PayNet}}$ executes the function `checkClosed`(Σ_{Alice}) every time it receives Σ_{Alice} from $\mathcal{G}_{\text{Ledger}}$ (Fig. 15, lines 1-33). 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 Overview of the Lightning Protocol Π_{LN}

In order to prove that software adhering to the LN specification is secure with respect to the guarantees given by $\mathcal{F}_{\text{PayNet}}$, it is necessary to define a concrete protocol that implements LN in the UC model. 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* ITIs honestly execute Π_{LN}. Similarly to the ideal world, the main functions of Π_{LN} are triggered when it receives (openChannel), (pay) and (closeChannel) 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 (Appendix H).

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. She first generates and sends to *Bob* some keys and her timelock delay, who also generates some keys and sends them back along with his timelock delay. *Alice* then builds the funding transaction using *Bob*'s keys and sends its signature back to *Bob*. He again mirrors *Alice*'s steps, sending back his signature. Both parties can now unilaterally spend the funding transaction, so *Alice* submits it to $\mathcal{G}_{\text{Ledger}}$.

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 generates a new "commitment" key for the next update and sends it to *Bob*. *Bob* also confirms that his state contains the funding transaction, generates his next commitment key and sends it back to *Alice*. 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 she receives (closeChannel, `receipt`, *tid*) from $\mathcal{E}$, *Alice* checks that `receipt` corresponds to the latest state of the channel and submits to $\mathcal{G}_{\text{Ledger}}$ the latest commitment transaction, along with all the relevant HTLC transactions. It also takes a note to give $\mathcal{E}$ a receipt of the closed channel the next time she receives (getNews).

Performing payments. 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 [27] onion packet with the relevant information for each party on the $\overrightarrow{\texttt{path}}$ and sends it to *Bob* along with the hash. *Bob* peels his layer of the onion and, after performing sanity checks, 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, $pchid_1$) from $\mathcal{E}$, where $pchid_1$ 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, he also should commit to the new channel version (that includes the HTLC) to *Alice*. This happens if he receives a (COMMIT) message from $\mathcal{E}$, which causes a similar exchange as above, but with the roles swapped. 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, $pchid_1$) from $\mathcal{E}$. Given that the HTLC is irrevocably committed, *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*) 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 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).

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 amount of 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 analyzed. In the current paper we use instead a PRF [28].

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 [25,26] (IBS) to abstract the properties provided by the construction. We also prove that it actually implements an IBS, see Section B.

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}.

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, it specifies exact rules for prunning outputs of too low value (known as "dust outputs"). In the current analysis we do not consider these features.

Last but not least, LN makes it possible for parties to cooperatively close a channel, thus avoiding the need to wait for the expiry of timelocks and reducing the size of the transactions that are added to the blockchain. As we mentioned earlier, we do not analyze this part of the specification.

6 Overview of the Combined Signature primitive

As previously mentioned, we define in this work a new primitive for combining keys and generating signatures that 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 here provide the intuition behind it and refer the reader to Appendix D for the exact syntax, correctness and security definitions, concrete construction and proof of security.

A combined signature is a two-party primitive, say *Alice* and *Bob* with *Bob* playing the role of the signer and *Alice* the holder of a share of the secret-key that is essential for issuing signatures verifiable with the "combined" verification key. The derivation of the verification key is achieved using public information drawn from *Alice* and *Bob* and is feasible without any party knowing the corresponding signing key. Only if *Alice* shares her secret information with *Bob* will he be able to construct the signing key.

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* is incapable of issuing a valid combined signature if he does not possess the corresponding secret share. 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 that still provide no advantage to the adversary in terms of producing a forged message for a combined key of its choice.

7 Security proof overview

Theorem 1 (Lightning Payment Network Security). *The protocol* Π_{LN} *realises* $\mathcal{F}_{\text{PayNet}}$ *given a global functionality* $\mathcal{G}_{\text{Ledger}}$ *assuming the security of the underlying digital signature, identity-based signature, combined digital signature and PRF. Specifically,*

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

where n *is the max number of registered users,* m *is the max number of channels that a user is involved in,* p *is the max 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. In Lemma 4 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. Each step defines a different functionality that handles some additional messages from $\mathcal{E}$ exactly like $\mathcal{F}_{\text{PayNet}}$, until the last step (Lemma 8) 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.

Lemma 5 lets $\mathcal{F}$ handle registration messages, along with the corruption messages from $\mathcal{S}$. In Lemma 6 the functionality additionally handles messages related to channel opening. It behaves like $\mathcal{F}_{\text{PayNet}}$, but does not execute `checkClosed`(). Lemma 7 has the functionality handle all messages sent during channel updates. Lastly, Lemma 8 has the entire $\mathcal{F}_{\text{PayNet}}$ as its 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. □

For the formal proof, we refer the reader to Appendix J.

Appendices

A Digital Signatures

Digital signatures [28] enable a party to authenticate messages to other parties. A signature on a message is created by the signing party using the secret "signing key"; other parties can later verify that the signature was indeed made on the message using the public "verification key". Transactions in Bitcoin [2] are signed using digital signatures and are considered valid only if signatures verify correctly, thus ensuring that only parties entitled to particular funds can spend them. Bitcoin uses ECDSA signatures over the secp256k1 curve[5].

To ensure compatibility, LN uses ECDSA over the same curve as its basic signature scheme. In this work, we abstract the particular construction away and use instead the established primitive that a secure construction must realize.

The three algorithms used by a Digital Signatures scheme are:

- $(pk, sk) \leftarrow \textsc{KeyGen}(1^k)$
- $\sigma \leftarrow \textsc{SignDS}(m, sk)$
- $\{0,1\} \leftarrow \textsc{VerifyDS}(\sigma, m, pk)$

We demand that the following holds for a scheme to have correctness:
$\forall k \in \mathbb{N}, m \in \mathcal{M}$,
$\Pr[(pk, sk) \leftarrow \textsc{KeyGen}(1^k),$
$\textsc{VerifyDS}(\textsc{SignDS}(m, sk), m, pk) = 1] = 1$

Definition 1. *A Digital Signatures scheme is* strongly EUF-CMA-secure *if*

$$\forall k \in \mathbb{N}, \forall \mathcal{A} \in \mathsf{PPT}, \Pr\left[\mathsf{EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = negl\left(k\right) \quad .$$

Let $\text{E-ds}(k) = \sup_{\mathcal{A} \in \mathsf{PPT}} \{\Pr[\mathsf{EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1]\}$. Then Definition 1 is equivalent to the following:

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

$$\forall k \in \mathbb{N}, \text{E-ds}(k) = negl\left(k\right) \quad .$$

B Identity Based Signatures primitive

As we mentioned previously, LN uses a custom construction to derive three new keys on each update. We abstract this construction using a slight modification to the previously established Identity Based Signatures primitive [26,25]. Our version augments the scheme with explicit verification keys, which are generated together with the signing keys. Furthermore a new key derivation algorithm is

[5] https://en.bitcoin.it/wiki/Secp256k1

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

$(pk, sk) \leftarrow \textsc{KeyGen}(1^k)$
$i \leftarrow 0$
$(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}(\textsc{init}, pk)$
while response can be parsed as m **do**
    $i \leftarrow i + 1$
    store m as m_i
    $\sigma_i \leftarrow \textsc{SignDS}(m, sk)$
    $(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}(\textsc{signature}, \mathtt{aux}_{i-1}, \sigma_i)$
end while
parse response as (m^*, σ^*)
if $m^* \notin \{m_1, \ldots, m_i\} \wedge \textsc{VerifyDS}(\sigma^*, m^*, pk) = 1$ **then**
    return 1
else
    return 0
end if

Fig. 1.

introduced, which returns only the verification key of an identity, given its label. We furthermore prove that the custom construction used in LN realizes the primitive.

The five algorithms used by an Identity Based Signatures scheme (with our modification) are:

- $(mpk, msk) \leftarrow \textsc{Setup}(1^k)$
- $(pk_l, sk_l) \leftarrow \textsc{KeyDer}(mpk, msk, l)$
- $pk_l \leftarrow \textsc{PubKeyDer}(mpk, l)$
- $\sigma \leftarrow \textsc{SignIBS}(m, sk_l)$
- $\{0, 1\} \leftarrow \textsc{VerifyIBS}(\sigma, m, pk_l)$

Observe that mpk is not part of the input to SignIBS and VerifyIBS. In our case, this input is not needed. In fact, because of the underlying similarity of these two algorithms with their counterparts from standard Digital Signatures, such an input would rather complicate the exposition.

We demand that the following holds for a scheme to have correctness:

- $\forall k \in \mathbb{N}, l \in \mathcal{L}$,
  $\Pr[(mpk, msk) \leftarrow \textsc{Setup}(1^k),$
  $(pk_1, sk_1) \leftarrow \textsc{KeyDer}\,(mpk, msk, l)\,,$
  $pk_2 \leftarrow \textsc{PubKeyDer}\,(mpk, l)\,,$
  $pk_1 = pk_2] = 1$
- $\forall k \in \mathbb{N}, m \in \mathcal{M}$,
  $\Pr[(mpk, msk) \leftarrow \textsc{Setup}(1^k),$
  $(pk, sk) \leftarrow \textsc{KeyDer}\,(mpk, msk, l)\,,$
  $\textsc{VerifyIBS}(\textsc{SignIBS}(m, sk), m, pk) = 1] = 1$

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

$(mpk, msk) \leftarrow \textsc{Setup}(1^k)$
$i, j \leftarrow 0$
$(\mathtt{aux}_0, \text{response}) \leftarrow \mathcal{A}(\textsc{init}, mpk)$
while response can be parsed as (m, l) or l **do**
    if response can be parsed as (m, l) **then**
        $i \leftarrow i + 1$
        store (m, l) as $(m, l)_i$
        $(pk, sk) \leftarrow \textsc{KeyDer}(mpk, msk, l)$
        $\sigma \leftarrow \textsc{SignIBS}(m, sk)$
        $(\mathtt{aux}_{i+j}, \text{response}) \leftarrow \mathcal{A}(\textsc{signature}, \mathtt{aux}_{i+j-1}, \sigma)$
    else // response can be parsed as l
        $j \leftarrow j + 1$
        store l as l_j
        $(pk, sk) \leftarrow \textsc{KeyDer}(mpk, msk, l)$
        $(\mathtt{aux}_{i+j}, \text{response}) \leftarrow \mathcal{A}(\textsc{keypair}, \mathtt{aux}_{i+j-1}, (pk, sk))$
    end if
end while
parse response as (m^*, l^*, σ^*)
if $(m^*, l^*) \notin \{(m, l)_1, \ldots, (m, l)_i\} \wedge l^* \notin \{l_1, \ldots, l_j\} \wedge \textsc{VerifyIBS}(\sigma^*, m^*, \textsc{PubKeyDer}(mpk, l^*)) = 1$ **then**
    return 1
else
    return 0
end if

Fig. 2.

Definition 3. *An Identity Based Signatures scheme is* IBS-EUF-CMA-secure *if*

$$\forall k \in \mathbb{N}, \forall \mathcal{A} \in \mathsf{PPT}, \Pr\left[\mathsf{IBS\text{-}EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = negl\left(k\right) \quad .$$

Let $\text{E-ibs}(k) = \sup_{\mathcal{A} \in \mathsf{PPT}} \{\Pr[\mathsf{IBS\text{-}EUF\text{-}CMA}^{\mathcal{A}}\left(1^k\right) = 1]\}$. Then Definition 3 is equivalent to the following:

Definition 4. *An Identity Based Signatures scheme is* IBS-EUF-CMA-secure *if*

$$\forall k \in \mathbb{N}, \text{E-ibs}(k) = negl\left(k\right) \quad .$$

B.1 Construction

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

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

SETUP(1^k, rand):
  return $(G \cdot \text{rand}, \text{rand})$

KEYDER(mpk, msk, l):
  $pk \leftarrow mpk + \mathcal{H}\left(l \parallel mpk\right) \cdot G$
  $sk \leftarrow msk + \mathcal{H}\left(l \parallel mpk\right)$
  return (pk, sk)

PUBKEYDER(mpk, l):
  return $mpk + \mathcal{H}\left(l \parallel mpk\right) \cdot G$

SIGNIBS(m, sk_l):
  return SIGNDS(m, sk_l)

VERIFYIBS(σ, m, pk_l):
  return VERIFYDS(σ, m, pk_l)

Lemma 1. *The construction above is IBS-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

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

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 3) such that

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

that breaks the assumption, thus proving Lemma 1.

Algorithm $\mathcal{A}(vk)$

$k \xleftarrow{\$} U[1, T(\mathcal{B}) + T(\mathcal{A})]$ // $T(M)$ 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 $\mathcal{B}$
$(mpk, msk) \leftarrow \textsc{Setup}\left(1^k\right)$
    Random Oracle: Let q be the kth first-seen query from $\mathcal{B}$ or $\mathcal{A}$:
    if $q = (l \,\|\, mpk)$ **then**
        set $\mathcal{H}(l \,\|\, mpk)$ to $(vk - mpk) \cdot G^{-1}$
    else
        set $\mathcal{H}(q)$ to a random value
    end if
    return $\mathcal{H}(q)$ to $\mathcal{B}$ or $\mathcal{A}$
$i \leftarrow 0$
$j \leftarrow 0$
$(\mathtt{aux}_0, \text{response}) \leftarrow \mathcal{B}(\textsc{init}, mpk)$
while response can be parsed as (m, l) or l **do**
    if response can be parsed as (m, l) **then**
        $i \leftarrow i + 1$
        store (m, l) as $(m, l)_i$
        $(pk, sk) \leftarrow \textsc{KeyDer}(mpk, msk, l)$
        $\sigma \leftarrow \textsc{SignIBS}(m, sk)$
        $(\mathtt{aux}_{i+j}, \text{response}) \leftarrow \mathcal{B}(\textsc{signature}, \mathtt{aux}_{i+j-1}, \sigma)$
    else // response can be parsed as l
        $j \leftarrow j + 1$
        store l as l_j
        $(pk, sk) \leftarrow \textsc{KeyDer}(mpk, msk, l)$
        $(\mathtt{aux}_{i+j}, \text{response}) \leftarrow \mathcal{B}(\textsc{keypair}, \mathtt{aux}_{i+j-1}, (pk, sk))$
    end if
end while
parse response as (m^*, l^*, σ^*)
if $vk = \textsc{PubKeyDer}(mpk, l^*) \wedge \mathcal{B}$ wins the IBS-EUF-CMA game **then** // $\mathcal{A}$ won the EUF-CMA game
    return (m^*, σ^*)
else
    return FAIL
end if

Fig. 3.

Let Y be the range of the random oracle. The modified random oracle used in Fig. 3 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 $\frac{1}{2|Y|} < negl(k)$ as they differ in at most one element.

Let E denote the event in which neither $\textsc{KeyDer}(mpk, msk, l^*)$ or $\textsc{PubKeyDer}(mpk, l^*)$ is invoked. In that case the value $\mathcal{H}(l \,\|\, mpk)$ is decided after $\mathcal{B}$ terminates (Fig. 3, line 30) and thus

$$\begin{gathered}\Pr[vk \in \textsc{KeyDer}(mpk, msk, l^*) \vee \\ vk = \textsc{PubKeyDer}(mpk, l^*) \,|E] < negl(k) \Rightarrow \\ \Pr[(vk \in \textsc{KeyDer}(mpk, msk, l^*) \vee \\ vk = \textsc{PubKeyDer}(mpk, l^*)) \wedge E] < negl(k) \Rightarrow \\ \Pr[vk = \textsc{PubKeyDer}(mpk, l^*) \wedge E] < negl(k) \quad .\end{gathered} \tag{1}$$

It is

$$\begin{gathered}(\mathcal{B} \text{ wins}) \rightarrow (vk = \textsc{PubKeyDer}(mpk, l^*)) \Rightarrow \\ \Pr[\mathcal{B} \text{ wins}] \leq \Pr[vk = \textsc{PubKeyDer}(mpk, l^*)] \Rightarrow \\ \Pr[\mathcal{B} \text{ wins} \wedge E] \leq \Pr[vk = \textsc{PubKeyDer}(mpk, l^*) \wedge E] \overset{(1)}{\Rightarrow} \\ \Pr[\mathcal{B} \text{ wins} \wedge E] < negl(k) \quad .\end{gathered}$$

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

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

We now focus at the event $\neg E$. Let F the event in which the call of to $\textsc{KeyDer}(mpk, msk, l^*)$ or $\textsc{PubKeyDer}(mpk, l^*)$ results in the kth invocation of the Random Oracle. Since k is chosen uniformly at random and using Proposition 2, $\Pr[F|\neg E] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$. Observe that $\Pr[F|E] = 0 \Rightarrow \Pr[F] = \Pr[F|\neg E] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$.

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

$$\textsc{PubKeyDer}(mpk, l^*) = mpk + \mathcal{H}(l^* \,\|\, mpk) \cdot G = mpk + (vk - mpk) \cdot G^{-1} \cdot G = vk$$

Since F holds, the kth invocation of the Random Oracle queried $\mathcal{H}(l^* \,\|\, mpk)$. Therefore it is $\textsc{PubKeyDer}(mpk, l^*) = vk$. This means that the verification is successful: $\textsc{VerifyIBS}(\sigma^*, m^*, vk) = 1$. We conclude that, if $(F \wedge \mathcal{B} \text{ wins} \wedge \neg E)$, $\mathcal{A}$ wins the $\mathsf{EUF\text{-}CMA}$ game. A final observation is that the probability that the events $(\mathcal{B} \text{ wins} \wedge \neg E)$ and F are almost independent, thus

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

□

C Pseudorandom Functions

A "pseudorandom function" [28] F is informally a function with two inputs: a secret seed and a bitstring. Given that the seed is randomly selected, no PPT algorithm can distinguish F from a randomly selected function.

In the current work a PRF is used in Π_{LN} to generate the randomness used for KeyShareGen(), which returns the so-called "per commitment" keypairs $(s_{Alice,\mathrm{com},n}, p_{Alice,\mathrm{com},n})$ (Fig. 20, line 4, Fig. 21, line 4, Fig. 25, line 5, Fig. 26, line 7 and Fig. 33, line 18).

Definition 5. *Let $k \in \mathbb{N}$. Let Func_k the set of all functions mapping k-bitstrings to k-bitstrings. We say that function $F : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ is pseudorandom if $\forall$ PPT $\mathcal{A}$,*

$$|\Pr_{\substack{s\in\{0,1\}^k \\ \mathcal{A}\text{'s coins}}}[\mathcal{A}^{F(s,\cdot)}(1^k)=1] - \Pr_{\substack{f\in\mathsf{Func}_k \\ \mathcal{A}\text{'s coins}}}[\mathcal{A}^{f(\cdot)}(1^k)=1]| = negl(k) \ ,$$

where $\mathcal{A}$ is given oracle access to $F(s,\cdot)$ and $f(\cdot)$ in each of the probability expressions above respectively.

$$\text{Let E-prf}(k) = \sup_{\mathcal{A}\in\mathsf{PPT}}\{|\Pr_{\substack{s\in\{0,1\}^k \\ \mathcal{A}\text{'s coins}}}[\mathcal{A}^{F(s,\cdot)}(1^k)=1] - \Pr_{\substack{f\in\mathsf{Func}_k \\ \mathcal{A}\text{'s coins}}}[\mathcal{A}^{f(\cdot)}(1^k)=1]|\}.$$

Then Definition 5 is equivalent to the following:

Definition 6. *$F : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ is pseudorandom if*

$$\forall k \in \mathbb{N}, \text{E-prf}(k) = negl\,(k) \ .$$

D Combined Signatures primitive

The seven algorithms used by a Combined Signatures scheme are:

- $(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right)$
- $(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right)$
- $(cpk_l, csk_l) \leftarrow \textsc{CombineKey}\,(mpk, msk, pk, sk)$
- $cpk_l \leftarrow \textsc{CombinePubKey}\,(mpk, pk)$
- $\{0,1\} \leftarrow \textsc{TestKey}\,(sk, pk)$
- $\sigma \leftarrow \textsc{SignCS}\,(m, csk)$
- $\{0,1\} \leftarrow \textsc{VerifyCS}\,(\sigma, m, cpk)$

We demand that the following holds for a scheme to have correctness:

- $\forall k \in \mathbb{N}$,
  $\Pr[(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
  $\textsc{TestKey}(pk, sk) = 1] = 1$

– $\forall k \in \mathbb{N}$,
$\Pr[(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right),$
$(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
$(cpk_1, csk_1) \leftarrow \textsc{CombineKey}\left(mpk, msk, pk, sk\right),$
$cpk_2 \leftarrow \textsc{CombinePubKey}\left(mpk, pk\right),$
$cpk_1 = cpk_2] = 1$
– $\forall k \in \mathbb{N}, m \in \mathcal{M}$,
$\Pr[(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right),$
$(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
$(cpk, csk) \leftarrow \textsc{CombineKey}\left(mpk, msk, pk, sk\right),$
$\textsc{VerifyCS}(\textsc{SignCS}(m, csk), m, cpk) = 1] = 1$

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

1: $(\mathtt{aux}, mpk, n) \leftarrow \mathcal{A}\left(\textsc{init}\right)$
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: $(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{A}\left(\textsc{keys}, \mathtt{aux}, pk_1, \ldots, pk_n\right)$
6: **if** $pk^* \in \{pk_1, \ldots, pk_n\} \wedge cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge$
$\textsc{VerifyCS}\left(\sigma^*, m^*, cpk^*\right) = 1$ **then**
7: $\quad$ **return** 1
8: **else**
9: $\quad$ **return** 0
10: **end if**

Fig. 4.

Definition 7. *A Combined Signatures scheme is* share-EUF-secure *if*

$$\forall k \in \mathbb{N}, \forall \mathcal{A} \in \mathtt{PPT}, \Pr\left[\textsf{share-EUF}^{\mathcal{A}}\left(1^k\right) = 1\right] = negl\left(k\right) \quad .$$

Let E-share$(k) = \sup_{\mathcal{A} \in \mathtt{PPT}} \{\Pr[\textsf{share-EUF}^{\mathcal{A}}\left(1^k\right) = 1]\}$. Then Definition 7 is equivalent to the following:

Definition 8. *A Combined Signatures scheme is* share-EUF-secure *if*

$$\forall k \in \mathbb{N}, \text{E-share}(k) = negl\left(k\right) \quad .$$

Definition 9. *A Combined Signatures scheme is* master-EUF-CMA-secure *if*

$$\forall k \in \mathbb{N}, \forall \mathcal{A} \in \mathtt{PPT}, \Pr\left[\textsf{master-EUF-CMA}^{\mathcal{A}}\left(1^k\right) = 1\right] = negl\left(k\right)$$

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

Fig. 5.

Let E-master$(k) = \sup_{\mathcal{A} \in \mathsf{PPT}} \{\Pr[\mathsf{master\text{-}EUF-CMA}^{\mathcal{A}}\left(1^k\right) = 1]\}$. Then Definition 9 is equivalent to the following:

Definition 10. *A Combined Signatures scheme is* master-EUF-CMA-secure *if*

$$\forall k \in \mathbb{N}, \text{E-master}(k) = negl\left(k\right) \quad .$$

Definition 11. *A Combined Signatures scheme is* combine-EUF-secure *if it is both share-EUF-secure and master-EUF-CMA-secure.*

D.1 Construction

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

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

$\textsc{MasterKeyGen}(1^k, \text{rand})$:
    return $(\text{rand}, G \cdot \text{rand})$

$\textsc{KeyShareGen}(1^k, \text{rand})$:
    return $(\text{rand}, G \cdot \text{rand})$

$\textsc{CombineKey}(mpk, msk, pk, sk)$:
    return $msk \cdot \mathcal{H}\left(mpk \,\|\, pk\right) + sk \cdot \mathcal{H}\left(pk \,\|\, mpk\right)$

$\textsc{CombinePubKey}(mpk, pk)$:

return $mpk \cdot \mathcal{H}(mpk \,\|\, pk) + pk \cdot \mathcal{H}(pk \,\|\, mpk)$

SIGNCS(m, csk):

return SIGNDS(m, csk)

VERIFYCS(σ, m, cpk):

return VERIFYDS(σ, m, cpk)

Lemma 2. *The construction 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

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

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

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

that breaks the assumption, thus proving Lemma 2.

Let Y be the range of the random oracle. The modified random oracle used in Fig. 6 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 $\frac{1}{2|Y|} < negl(k)$ 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. 6, line 24) and thus

$$\begin{gathered}\Pr\left[cpk^* = \textsc{CombinePubKey}(mpk, pk^*) \,|E\right] = \frac{1}{|Y|} < negl(k) \Rightarrow \\ \Pr\left[cpk^* = \textsc{CombinePubKey}(mpk, pk^*) \wedge E\right] < negl(k) \quad .\end{gathered} \tag{3}$$

It is

$$\begin{gathered}(\mathcal{B} \text{ wins}) \rightarrow (cpk^* = \textsc{CombinePubKey}(mpk, pk^*)) \Rightarrow \\ \Pr[\mathcal{B} \text{ wins}] \leq \Pr\left[cpk^* = \textsc{CombinePubKey}(mpk, pk^*)\right] \Rightarrow \\ \Pr[\mathcal{B} \text{ wins} \wedge E] \leq \Pr\left[cpk^* = \textsc{CombinePubKey}(mpk, pk^*) \wedge E\right] \overset{(3)}{\Rightarrow} \\ \Pr[\mathcal{B} \text{ wins} \wedge E] < negl(k) \quad .\end{gathered}$$

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

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

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. 6.

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 2, $\Pr[F|\neg E] = \frac{1}{T(\mathcal{B})}$. Observe that $\Pr[F|E] = 0 \Rightarrow \Pr[F] = \Pr[F|\neg E] = \frac{1}{T(\mathcal{B})}$.

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

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

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. 6, lines 9-18), it is $cpk^* = vk$. This means that $\textsc{VerifyCS}(\sigma^*, m^*, vk) = 1$. 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

$$\Pr[F \wedge \mathcal{B} \text{ wins} \wedge \neg E] = \Pr[F] \Pr[\mathcal{B} \text{ wins} \wedge \neg E] \pm negl(k) \overset{(4)}{=} \\ \frac{a - negl(k)}{T(\mathcal{B})} \pm negl(k) > negl(k)$$

□

Lemma 3. *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

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

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 7) such that

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

that breaks the assumption, thus proving Lemma 3.

The modified random oracle used in Fig. 7 is indistinguishable from the standard random oracle for the same reasons as in the proof of Lemma 2.

Let E denote the event in which CombinePubKey is not invoked to produce cpk^*. In that case the values $\mathcal{H}(pk^* \,\|\, mpk)$ and $\mathcal{H}(mpk \,\|\, pk^*)$ are decided after $\mathcal{B}$ terminates (Fig. 7, line 30) and thus

$$\Pr[cpk^* = \textsc{CombinePubKey}(mpk, pk^*) \,|E] < negl(k) \Rightarrow \\ \Pr[cpk^* = \textsc{CombinePubKey}(mpk, pk^*) \wedge E] < negl(k) \quad . \tag{5}$$

We can reason like in the proof of Lemma 2 to deduce that

$$\Pr[\mathcal{B} \text{ wins} \wedge \neg E] > a - negl(k) \quad . \tag{6}$$

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. 7.

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 2, $\Pr[F|\neg E] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$. Observe that $\Pr[F|E] = 0 \Rightarrow \Pr[F] = \Pr[F|\neg E] = \frac{1}{T(\mathcal{B})+T(\mathcal{A})}$.

Once more we can reason in the same fashion as in the proof of Lemma 2 to deduce that

$$\Pr[F \wedge \mathcal{B} \text{ wins} \wedge \neg E] = \Pr[F] \Pr[\mathcal{B} \text{ wins} \wedge \neg E] \pm negl(k) \overset{(6)}{=}$$
$$\frac{a - negl(k)}{T(\mathcal{B}) + T(\mathcal{A})} \pm negl(k) > negl(k)$$

□

Theorem 2. *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 2, Lemma 3 and the definition of combine-EUF-security. □

E Transaction Structure

A well-formed transaction consists of a list of inputs and a list of outputs. For the transaction to be valid, each input must be connected to a single valid, previously unconnected (unspent) output of another transaction in $\mathcal{G}_{\text{Ledger}}$.

A well-formed output consists of a value in coins and a list of "spending methods". A well-formed input consists of a reference to a previously unconnected output and a reference to a single of the latter's spending methods, along with the data needed to satisfy that method. A well-formed spending method contains any combination of the following:

- Public keys in disjunctive normal form. An input that spends using this spending method must contain signatures valid by the public keys of one of the conjunctions. If no public keys are specified in the output, no signatures are needed in the input.
- Absolute locktime d_a, in block height or time. An input that spends this output can only be in block of height at least d_a if d_a is a block height, or enter the ledger on or after time d_a otherwise. Zero means no absolute locktime.
- Relative locktime d_r, in block height or time. The distance of an input that spends this output must be at least d_r, counted in block height or time. Zero means no relative locktime.
- Hashlock value. The output can be spent by an input that contains a preimage that hashes to the hashlock value. If no hashlock value is specified in the output, no preimage is needed in the spending input.

Lastly, the sum of coins of the outputs referenced by the inputs of the transaction (to-be-spent outputs) should be greater than or equal to the sum of coins of the outputs of the transaction.

We say that an unspent output is currently exclusively spendable by a player *Alice* with a public key pk and a hash list hl if for each spending method one of the following two holds:

- It still has a locktime that has not expired and thus is currently unspendable, or
- The only specified public key is pk and if there is a hashlock, its hash is contained in hl.

If an output is exclusively spendable, we say that its coins are exclusively spendable.

F Payment Network Functionality

Functionality $\mathcal{F}_{\text{PayNet}}$ - preamble

Interface:

- from $\mathcal{E}$:
  - (REGISTER, delay, relayDelay)
  - (TOPPEDUP)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *tid*)
  - (CHECKFORNEW, *Alice*, *Bob*, *tid*)
  - (PAY, *Bob*, $x, \overrightarrow{\texttt{path}}$, `receipt`)
  - (CLOSECHANNEL, `receipt`, *tid*)
  - (POLL)
  - (PUSHFULFILL, *pchid*)
  - (PUSHADD, *pchid*)
  - (COMMIT, *pchid*)
  - (FULFILLONCHAIN)
  - (GETNEWS)
- to $\mathcal{E}$:
  - (REGISTER, *Alice*, `delay`(*Alice*), `relayDelay`(*Alice*), pubKey)
  - (REGISTERED)
  - (CHANNELCLOSED, `receipt`)
  - (NEWS, newChannels, closedChannels, updatesToReport)
- from $\mathcal{S}$:
  - (REGISTERDONE, *Alice*, pubKey)
  - (CORRUPTED, *Alice*)
  - (CHANNELANNOUNCED, *Alice*, $p_{Alice,F}, p_{Bob,F}$, *fchid*, *pchid*, *tid*)
  - (UPDATE, `receipt`, *Alice*)
  - (RESOLVEPAYS, *payid*, `charged`)
- to $\mathcal{S}$:
  - (REGISTER, *Alice*, delay, relayDelay, lastPoll)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*)
  - (CHANNELOPENED, *Alice*, *fchid*)
  - (PAY, *Alice*, *Bob*, $x, \overrightarrow{\texttt{path}}$, `receipt`, *payid*)
  - (CONTINUE)
  - (CLOSECHANNEL, *fchid*, *Alice*)
  - (POLL, Σ_{Alice}, *Alice*)
  - (PUSHFULFILL, *pchid*, *Alice*)
  - (PUSHADD, *pchid*, *Alice*)
  - (COMMIT, *pchid*, *Alice*)
  - (FULFILLONCHAIN, t, *Alice*)

Fig. 8.

Functionality $\mathcal{F}_{\text{PayNet}}$ - support

Initialisation:
    `channels`, `pendingPay`, `pendingOpen`, `corrupted`, $\Sigma \leftarrow \emptyset$

Upon receiving (REGISTER, delay, relayDelay) from *Alice*:
    `delay` (*Alice*) $\leftarrow$ delay // Must check chain at least once every `delay`(*Alice*) blocks
    `relayDelay` (*Alice*) $\leftarrow$ relayDelay
    `updatesToReport` (*Alice*), `newChannels` (*Alice*) $\leftarrow \emptyset$
    `polls` (*Alice*) $\leftarrow \emptyset$
    `focs` (*Alice*) $\leftarrow \emptyset$
    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 `polls`(*Alice*)
    `checkClosed`(Σ_{Alice})
    send (REGISTER, *Alice*, delay, relayDelay, lastPoll) to $\mathcal{S}$

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

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

Upon receiving any message except for (REGISTER) from *Alice*:
    ignore message if *Alice* has not registered

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

Fig. 9.

Functionality $\mathcal{F}_{\text{PayNet}}$ - open

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

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

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

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

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

Fig. 10.

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

1: Upon receiving $\left(\text{PAY}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}\right)$ from *Alice*:

2: choose unique payment ID *payid*

3: add $\left(\mathit{Alice}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \mathit{payid}\right)$ to `pendingPay`

4: send $\left(\text{PAY}, \mathit{Alice}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \mathit{payid}, \text{STATE}, \Sigma\right)$ to $\mathcal{S}$

5: Upon receiving (UPDATE, `receipt`, *Alice*) from $\mathcal{S}$:

6: add `receipt` to `updatesToReport`(*Alice*) // trust $\mathcal{S}$ here, check on RESOLVEPAYS

7: send (CONTINUE) to $\mathcal{S}$

Fig. 11.

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 ∈ `charged` **do**
3: **for all** (*Dave*, *payid*) ∈ `charged` (*Alice*) **do**
4: retrieve $\left(Alice, Bob, x, \overrightarrow{\texttt{path}}\right)$ with ID *payid* and remove it from `pendingPay`
5: **if** *Dave* = ⊥ **then**
6: continue with next iteration of inner loop
7: **else if** *Dave* ∈ `corrupted` **then**
8: run code of Fig. 13
9: `offChainBalance` (*Bob*) ← `offChainBalance` (*Bob*) + x
10: **else** // *Dave* honest
11: **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**
12: halt // DS forgery
13: **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**
14: **if** `polls`(*Dave*) contains an element in $[h_{\texttt{tx}}, h_{\texttt{tx}} + \texttt{delay}\,(Dave) - 1]$ **then**
15: halt // *Dave* POLLed, but malicious closure
16: **else**
17: `negligent`(*Alice*) ← `true`
18: **end if**
19: **else if** *Dave* ≠ *Alice* **then**
20: calculate `IncomingCltvExpiry`, `OutgoingCltvExpiry` of *Dave* (as in Fig. 29, l. 19)
21: **if** Σ_{Dave} does not contain an old $\texttt{remoteCom}_m$ **then**
22: **if** `polls`(*Dave*) contains two elements in [`OutgoingCltvExpiry`, `IncomingCltvExpiry` − (2 + r) `windowSize`] that have a difference of at least (2 + r) `windowSize` ∧ `focs`(*Dave*) contains `IncomingCltvExpiry` − (2 + r) `windowSize` ∧
23: the element in `polls`(*Dave*) was added before the element in `focs`(*Dave*) **then**
24: halt // *Dave* POLLed and fulfilled, but charged
25: **else**
26: `negligent`(*Alice*) ← `true`
27: **end if**
28: **end if**
29: **end if**
30: run code of Fig. 13
31: `offChainBalance` (*Dave*) ← `offChainBalance` (*Dave*) − x
32: `offChainBalance` (*Bob*) ← `offChainBalance` (*Bob*) + x
33: **end if**
34: **end for**
35: **end for**

Fig. 12. r, `windowSize` as in Proposition 1

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 a corresponding element has been added to the `updatesToReport` of each honest counterparty, otherwise halt
11: **end for**

Fig. 13.

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

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

Fig. 14.

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

function `checkClosed`(Σ_{Alice}) // Called after every (READ), ensures requested closes eventually happen
  for all entries $(fchid, \texttt{receipt}, h) \in \texttt{pendingClose}(Alice) \cup \texttt{closedChannels}(Alice)$ **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 the balances of *Alice* and the `channel` counterparty *Bob* respectively
      offChainBalance $(Alice) \leftarrow$ offChainBalance $(Alice) - x$
      onChainBalance $(Alice) \leftarrow$ onChainBalance $(Alice) + x$
      offChainBalance $(Bob) \leftarrow$ offChainBalance $(Bob) - y$
      onChainBalance $(Bob) \leftarrow$ onChainBalance $(Bob) + y$
      remove `channel` from `channels`
      remove 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 commitment tx and spends the funding tx of the `channel` with ID *fchid* **then**
      halt // DS forgery
    else if there is a closing 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 transaction $\wedge\ h = \perp$ **then**
      assign largest block number of Σ_{Alice} to h of entry
    else if there is no such closing 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} AND `negligent`(*Alice*) = `false` **then**
    if offChainBalance(*Alice*) $\neq 0$ OR onChainBalance(*Alice*) is not equal to the total funds exclusively spendable by *Alice* in Σ_{Alice} **then**
      halt
    end if
  end if
end function

Fig. 15.

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

Upon receiving (POLL) from *Alice*:
  send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
  add largest block number in Σ_{Alice} to $\texttt{polls}(Alice)$
  $\texttt{checkClosed}(\Sigma_{Alice})$
  if $\exists \texttt{channel} \in \Sigma_{Alice}$ that contains *Alice* and is closed by a tx that is not a commitment transaction **then**
    halt // DS forgery
  end if
  for all $\texttt{channel}s \in \Sigma_{Alice}$ 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**
    if the delayed output (of the counterparty) has been spent **then**
      if $\texttt{polls}(Alice)$ has an element in $[h_{\texttt{tx}}, h_{\texttt{tx}} + \texttt{delay}\,(Alice) - 1]$ **then**
        halt // *Alice* wasn't negligent but couldn't punish
      else
        $\texttt{negligent}(Alice) \leftarrow \texttt{true}$
      end if
    end if
  end for
  send (POLL, Σ_{Alice}, *Alice*) to $\mathcal{S}$

Fig. 16.

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 (closedChannel, `channel`, *Alice*) from $\mathcal{S}$:
13: add (*fchid* of channel, `receipt`(`channel`), $\perp$) to `closedChannels`(*Alice*) // trust $\mathcal{S}$ here, check on `checkClosed`()
14: send (continue) to $\mathcal{S}$

15: Upon receiving (getNews) from *Alice*:
16: clear `newChannels`(*Alice*), `closedChannels`(*Alice*), `updatesToReport`(*Alice*) and send them to *Alice* with message name news, stripping *fchid* and h from `closedChannels`(*Alice*)

Fig. 17.

G Channel data

- *Alice*'s secret keys:
  - s_{Alice}: key for on-chain funds (DS)[a]
  - $s_{Alice,F}$: funding (DS)
  - $sb_{Alice,\mathrm{pay}}$: payment basepoint (IBS)[b]
  - $sb_{Alice,\mathrm{dpay}}$: delayed payment basepoint (IBS)
  - $sb_{Alice,\mathrm{htlc}}$: htlc basepoint (IBS)
  - $sb_{Alice,\mathrm{rev}}$: revocation basepoint (CS[c] – master)
- *Bob*'s public keys: public counterparts to 5 keys above
- `seed`: for deriving *Alice*'s per commitment keys $s_{Alice,\mathrm{com},i}$ with PRF
- *Bob*'s per commitment points:
  - $\forall i \in [1,\ldots,n], s_{Bob,\mathrm{com},i}$: one secret per revokeAndAck received (CS – share)
  - $p_{Bob,\mathrm{com},n+1}$ and $p_{Bob,\mathrm{com},n+2}$: current and next points (CS – share)
- *Alice*'s coins
- *Bob*'s coins
- for every HTLC that is included in the latest irrevocably committed (local or remote) commitment:
  - direction (*Alice* $\rightarrow$ *Bob* or *Bob* $\rightarrow$ *Alice*)
  - hash
  - preimage (if known)
  - coins
  - Is it included in latest $\texttt{localCom}_n$? (boolean)
  - HTLC number
- signatures:
  - signature resulting from SignDS($\texttt{localCom}_{n+1}, s_{Bob,F}$) (DS)
  - for every HTLC included in $\texttt{localCom}_{n+1}$, if HTLC is outgoing, a signature for `HTLC-timeout`, else a signature for `HTLC-success` with $s_{Bob,\mathrm{htlc},n+1}$ (IBS)

[a] basic Digital Signature
[b] Identity Based Signature
[c] Combined Signature

Fig. 18. Data *Alice* holds in an *Alice* – *Bob* channel with n updates

H Lightning Protocol

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

Initialisation:
    `channels`, `pendingOpen`, `pendingPay`, `pendingClose` $\leftarrow \emptyset$
    `newChannels`, `closedChannels`, `updatesToReport` $\leftarrow \emptyset$
    `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 largest block number to `lastPoll`
    $(pk_{Alice}, sk_{Alice}) \leftarrow$ 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):
    if if haven't received (REGISTER) from $\mathcal{E}$ **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. 19.

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: $((ph_F, sh_F), (phb_{\text{pay}}, shb_{\text{pay}}), (phb_{\text{dpay}}, shb_{\text{dpay}}), (phb_{\text{htlc}}, shb_{\text{htlc}}), \texttt{seed}, (phb_{\text{rev}}, shb_{\text{rev}})) \leftarrow \texttt{GetKeys}()$
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
7: 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 $\texttt{pendingOpen}$
8: 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. 20.

Protocol Π_{LN} - openChannel from Bob

1: Upon receiving (openChannel, $x, \texttt{remoteDelay}, pt_F, ptb_{\text{pay}}, ptb_{\text{dpay}}, ptb_{\text{htlc}}, pt_{\text{com},1}, ptb_{\text{rev}}, tid$) 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. 21.

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. 22.

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: $ph_{\text{rev},1} \leftarrow \text{CombinePubKey}\left(phb_{\text{rev}} pt_{\text{com},1}\right)$
4: $pt_{\text{dpay},1} \leftarrow \text{PubKeyDer}\left(ptb_{\text{dpay}}, pt_{\text{com},1}\right)$
5: $pt_{\text{pay},1} \leftarrow \text{PubKeyDer}\left(ptb_{\text{pay}}, pt_{\text{com},1}\right)$
6: $pt_{\text{htlc},1} \leftarrow \text{PubKeyDer}\left(ptb_{\text{htlc}}, pt_{\text{com},1}\right)$
7: $\texttt{localCom} \leftarrow \texttt{localCom}_1 \leftarrow$ TX {input: output 1 of F, outputs: $\left(x, pt_{\text{pay},1}\right), \left(0, pt_{\text{rev},1} \vee \left(ph_{\text{dpay},1}, \texttt{remoteDelay} \text{ relative}\right)\right)$}
8: ensure $\text{VerifyDS}\left(\texttt{BobSig}_1, \texttt{localCom}_1, pt_F\right) = \texttt{True}$
9: $\texttt{remoteCom} \leftarrow \texttt{remoteCom}_1 \leftarrow$ TX {input: output 1 of F, outputs: $\left(x, ph_{\text{rev},1} \vee \left(pt_{\text{dpay},1}, \texttt{delay} + (2 + r)\, \texttt{windowSize} \text{ relative}\right)\right), \left(0, ph_{\text{pay},1}\right)$}
10: add $\texttt{BobSig}_1$, $\texttt{remoteCom}_1$ and $\texttt{localCom}_1$ to channel entry in `pendingOpen`
11: sig $\leftarrow \text{SignDS}\left(\text{remoteCom}_1, sh_F\right)$
12: mark channel as "broadcast, no fundingLocked"
13: $\texttt{lastRemoteSigned}, \texttt{lastLocalSigned} \leftarrow 0$
14: send (fundingSigned, *pchid*, sig) to *Bob*

Fig. 23.

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 $\text{VerifyDS}\left(\texttt{BobSig}_1, \texttt{localCom}, pb_F\right) = \texttt{True}$
4: $\texttt{localCom}_1 \leftarrow \texttt{localCom}$
5: $\texttt{lastLocalSigned} \leftarrow 0$
6: add $\texttt{BobSig}_1$ to channel entry in `pendingOpen`
7: sig $\leftarrow \text{SignDS}\left(F, sk_{Alice}\right)$
8: mark *pchid* in `pendingOpen` as "broadcast, no fundingLocked"
9: send (submit, (sig, F)) to $\mathcal{G}_{\text{Ledger}}$

Fig. 24.

Protocol Π_{LN} - checkForNew

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

Fig. 25.

Protocol Π_{LN} - fundingLocked

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

Fig. 26.

Protocol Π_{LN} - poll

Upon receiving (POLL) from $\mathcal{E}$:
  send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
  assign largest block number in Σ_{Alice} to `lastPoll`
  `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. 28
  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 *Alice*:
  clear `newChannels`, `closedChannels`, `updatesToReport` and send them to *Alice* with message name NEWS

Fig. 27.

Loop over closed channels for poll

for all $\texttt{remoteCom}_n \in \Sigma_{Alice}$ that spend F of a $\texttt{channel} \in \texttt{channels}$ **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 $\texttt{pendingFulfills}_{pchid'}$
      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 (rev, $sh_{\text{rev},n}$)
    add (sig, rev) to `toSubmit`
  end if
  add `receipt(channel)` to `closedChannels`
  remove `channel` from `channels`
end for

Fig. 28.

Protocol Π_{LN} - invoice

1: Upon receiving (PAY, *Bob*, x, $\overrightarrow{\texttt{path}}$) 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, $\overrightarrow{\texttt{path}}$, *payid*, "waiting for invoice") to `pendingPay`
6: send (SENDINVOICE, *payid*) to *Bob*

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

12: Upon receiving (INVOICE, h, `minFinalCltvExpiry`, *payid*) from *Bob*:
13: ensure there is a (*Bob*, x, $\overrightarrow{\texttt{path}}$, *payid*, "waiting for invoice") entry in `pendingPay`
14: ensure h is valid (in the range of $\mathcal{H}$)
15: retrieve `CltvExpiryDelta`s from $\overrightarrow{\texttt{path}}$ and remove entry from `pendingPay`
16: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
17: $l \leftarrow |\left(\overrightarrow{\texttt{path}}\right)|$
18: $\texttt{CltvExpiry}_l \leftarrow t + \texttt{minFinalCltvExpiry}$
19: $\forall i \in \{1, \ldots, l-1\}, \texttt{CltvExpiry}_{l-i} \leftarrow \texttt{CltvExpiry}_{l-i+1} + \texttt{CltvExpiryDelta}_{l-i+1}$
20: ensure $\texttt{CltvExpiry}_1 \geq \texttt{CltvExpiry}_2 + \texttt{relayDelay} + (2+r)\,\texttt{windowSize}$
21: $m \leftarrow$ the concatenation of $l\,(x, \texttt{CltvExpiry})$
22: $(\mu_0, \delta_0) \leftarrow \texttt{SphinxCreate}\left(m,\ \text{public keys of } \overrightarrow{\texttt{path}} \text{ parties}\right)$
23: let $\texttt{remoteCom}_n$ the latest signed remote commitment tx
24: reduce simple payment output in `remoteCom` by x
25: 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`
26: reduce delayed payment output in `localCom` by x
27: 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`
28: increment $\texttt{HTLCNo}_{pchid}$ by one and associate $x, h,$ *pchid* with it
29: mark `HTLCNo` as "sender"
30: 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. 29.

Protocol Π_{LN} - updateAddHtlc

1: Upon receiving (updateAddHtlc, *pchid*, $\texttt{HTLCNo}, x, h, \texttt{IncomingCltvExpiry}, M$) from *Bob*:
2: ensure *pchid* corresponds to an open `channel` in `channels` where *Bob* has at least x
3: ensure $\texttt{HTLCNo} = \texttt{HTLCNo}_{pchid} + 1$
4: $(pchid', x', \texttt{OutgoingCltvExpiry}, \delta) \leftarrow \texttt{SphinxPeel}\,(sk_{Alice}, M)$
5: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
6: **if** $\delta = \texttt{receiver}$ **then**
7: ensure $pchid' = \bot, x = x', \texttt{IncomingCltvExpiry} \geq \texttt{OutgoingCltvExpiry} = \texttt{minFinalCltvExpiry}$
8: mark `HTLCNo` as "receiver"
9: **else** // We are an intermediary
10: ensure $x = x', \texttt{IncomingCltvExpiry} \geq \max\{\texttt{OutgoingCltvExpiry}, t\} + \texttt{relayDelay} + 2\,(2 + r)\,\texttt{windowSize}$
11: ensure $\texttt{pchid}'$ corresponds to an open channel in `channels` where we have at least x
12: mark `HTLCNo` as "intermediary"
13: **end if**
14: increment $\texttt{HTLCNo}_{pchid}$ by one
15: let $\texttt{remoteCom}_n$ the latest signed remote commitment tx
16: reduce delayed payment output in `remoteCom` by x
17: add an $(x, ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1}, \texttt{IncomingCltvExpiry}\text{ absolute}) \vee ph_{\text{htlc},n+1},$ on preimage of h) htlc output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`
18: reduce simple payment output in `localCom` by x
19: add an $(x, pt_{\text{rev},n+1} \vee pt_{\text{htlc},n+1}, \texttt{IncomingCltvExpiry}\text{ absolute}) \vee ((pt_{\text{htlc},n+1} \wedge ph_{\text{htlc},n+1},$ on preimage of $h))$ htlc output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`
20: **if** $\delta = \texttt{receiver}$ **then**
21: retrieve $R : \mathcal{H}(R) = h$ from `pendingGetPaid` and clear entry
22: add $(\texttt{HTLCNo}, R)$ to $\texttt{pendingFulfills}_{pchid}$
23: **else if** $\delta \neq \texttt{receiver}$ **then** // Send HTLC to next hop
24: retrieve $pchid'$ data
25: let $\texttt{remoteCom}'_n$ the latest signed remote commitment tx
26: reduce simple payment output in $\texttt{remoteCom}'$ by x
27: add an additional $(x, ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1},$ on preimage of $h) \vee ph_{\text{htlc},n+1}\texttt{OutgoingCltvExpiry}$ absolute) output (all with $n+1$ keys) to $\texttt{remoteCom}'$, marked with $\texttt{HTLCNo}'$
28: reduce delayed payment output in $\texttt{localCom}'$ by x
29: add an additional $(x, pt_{\text{rev},n+1} \vee (pt_{\text{htlc},n+1},$ on preimage of $h) \vee (pt_{\text{htlc},n+1} \wedge ph_{\text{htlc},n+1}\texttt{OutgoingCltvExpiry}$ absolute)) output (all with $n+1$ keys) to $\texttt{remoteCom}'$, marked with $\texttt{HTLCNo}'$
30: increment $\texttt{HTLCNo}'$ by 1
31: $M' \leftarrow \texttt{SphinxPrepare}\,(M, \delta, sk_{Alice})$
32: add $(\texttt{HTLCNo}', x, h, \texttt{OutgoingCltvExpiry}, M')$ to $\texttt{pendingAdds}_{pchid'}$
33: **end if**

Fig. 30.

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. 36)
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 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. 35, l. 15)
19: send (updateFulfillHtlc, *pchid*$'$, `HTLCNo`$'$, R) to counterparty
20: **end if**
21: **end if**

Fig. 31.

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 $\texttt{IncomingCltvExpiry} < t + \texttt{relayDelay} + (2 + r)\,\texttt{windowSize}$
8: $\texttt{remoteCom}_{n+1} \leftarrow \texttt{remoteCom}$
9: ComSig $\leftarrow$ SIGNDS $(\texttt{remoteCom}_{n+1}, sh_F)$
10: HTLCSigs $\leftarrow \emptyset$
11: **for** i from `lastRemoteSigned` to `HTLCNo` **do**
12: $\texttt{remoteHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of $\texttt{remoteCom}_{n+1}$, output: $(c_{\text{htlc,i}}, ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}, \texttt{delay} + (2 + r)\,\texttt{windowSize}$ relative))}
13: add SIGNIBS $(\texttt{remoteHTLC}_{n+1,i}, sh_{\text{htlc},n+1})$ to HTLCSigs
14: **end for**
15: add SIGNIBS $(\texttt{remoteHTLC}_{n+1,m+1}, sh_{\text{htlc},n+1})$ to HTLCSigs
16: `lastRemoteSigned` $\leftarrow$ `HTLCNo`
17: mark `channel` as "waiting for REVOKEANDACK"
18: send (COMMITMENTSIGNED, *pchid*, ComSig, HTLCSigs) to *pchid* counterparty

Fig. 32.

Protocol Π_{LN} - commitmentSigned

1: Upon receiving (commitmentSigned, *pchid*, $\texttt{comSig}_{n+1}$, $\texttt{HTLCSigs}_{n+1}$) from *Bob*:
2: ensure that there is a `channel` $\in$ `channels` with ID *pchid* with *Bob*
3: retrieve latest local commitment tx $\texttt{localCom}_n$ in `channel`
4: ensure `localCom` $\neq \texttt{localCom}_n$ and `localCom` $\neq$ `pendingLocalCom` // there are uncommitted updates
5: **if** VerifyDS $\left(\texttt{comSig}_{n+1}, \texttt{localCom}, pt_F\right) = \texttt{false} \vee |\texttt{HTLCSigs}_{n+1}| \neq \texttt{HTLCNo} - \texttt{lastLocalSigned} + 1$ **then**
6: close channel (as in Fig. 36)
7: **return**
8: **end if**
9: **for** i from `lastLocalSigned` to `HTLCNo` **do**
10: $\texttt{localHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of `localCom`, output: $(c_{\text{htlc,i}}, ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}, \texttt{remoteDelay}$ relative))}
11: **if** VerifyIBS $\left(\texttt{HTLCSigs}_{n+1,i}, \texttt{localHTLC}_{n+1,i}, pt_{\text{htlc},n+1}\right) = \texttt{false}$ **then**
12: close channel (as in Fig. 36)
13: **return**
14: **end if**
15: **end for**
16: `pendingLocalCom` $\leftarrow$ `localCom`
17: mark `pendingLocalCom` as "irrevocably committed"
18: $\text{prand}_{n+2} \leftarrow \text{PRF}\left(\texttt{seed}, n+2\right)$
19: $(ph_{\text{com},n+2}, sh_{\text{com},n+2}) \leftarrow$ KeyShareGen $\left(1^k; \text{prand}_{n+2}\right)$
20: send (revokeAndAck, *pchid*, $\text{prand}_n, ph_{\text{com},n+2}$) to *Bob*

Fig. 33.

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` $\in$ `channels` with *Bob* with ID *pchid* marked as "waiting for REVOKEANDACK"
3: **if** TESTKEY $(st_{\text{com},n}, pt_{\text{com},n}) \neq 1$ **then** // wrong $st_{\text{com},n}$ - closing
4: close channel (as in Fig. 36)
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: add `receipt`(`channel`) to `updatesToReport`
11: $sh_{\text{rev},n} \leftarrow$ COMBINEKEY $(phb_{\text{rev}}, shb_{\text{rev}}, pt_{\text{com}n}, st_{\text{com},n})$
12: $ph_{\text{rev},n+2} \leftarrow$ COMBINEPUBKEY $(phb_{\text{rev}}, pt_{\text{com},n+2})$
13: $pt_{\text{rev},n+2} \leftarrow$ COMBINEPUBKEY $(ptb_{\text{rev}}, ph_{\text{com},n+2})$
14: $(ph_{\text{dpay},n+2}, sh_{\text{dpay},n+2}) \leftarrow$ KEYDER $(phb_{\text{dpay}}, shb_{\text{dpay}}, ph_{\text{com},n+2})$
15: $pt_{\text{dpay},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{dpay}}, pt_{\text{com},n+2})$
16: $(ph_{\text{pay},n+2}, sh_{\text{pay},n+2}) \leftarrow$ KEYDER $(phb_{\text{pay}}, shb_{\text{pay}}, ph_{\text{com},n+2})$
17: $pt_{\text{pay},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{pay}}, pt_{\text{com},n+2})$
18: $(ph_{\text{htlc},n+2}, sh_{\text{htlc},n+2}) \leftarrow$ KEYDER $(phb_{\text{htlc}}, shb_{\text{htlc}}, ph_{\text{com},n+2})$
19: $pt_{\text{htlc},n+2} \leftarrow$ PUBKEYDER $(ptb_{\text{htlc}}, pt_{\text{com},n+2})$

Fig. 34.

Protocol Π_{LN} - PUSH

1: Upon receiving (pushFulfill, *pchid*) from $\mathcal{E}$:
2: ensure that there is a `channel` $\in$ `channels` with ID *pchid*
3: choose a member (`HTLCNo`, R) of $\texttt{pendingFulfills}_{pchid}$ that is both in an "irrevocably committed" $\texttt{remoteCom}_n$ and $\texttt{localCom}_n$
4: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
5: remove (`HTLCNo`, R) from $\texttt{pendingFulfills}_{pchid}$
6: **if** $\texttt{remoteCom}_n \notin \Sigma_{Alice}$ **then** // counterparty cooperative
7: send (updateFulfillHtlc, *pchid*, `HTLCNo`, R) to *pchid* counterparty
8: **else** // counterparty gone on-chain
9: TX $\leftarrow$ {input: ($\texttt{remoteCom}_n$ HTLC output with number `HTLCNo`, R), output: pk_{Alice}}
10: sig $\leftarrow$ SignIBS (TX, $sh_{\text{htlc},n}$)
11: send (SUBMIT, (sig, TX)) to $\mathcal{G}_{\text{Ledger}}$ // shouldn't be already spent by remote `HTLCTimeout`
12: **end if**

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

18: Upon receiving (fulfillOnChain) from $\mathcal{E}$:
19: send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign largest block number to t
20: `toSubmit` $\leftarrow \emptyset$
21: **for all** channels **do**
22: **if** there exists an HTLC in latest $\texttt{localCom}_n$ for which we have sent both updateFulfillHtlc and commitmentSigned to a transaction without that HTLC to counterparty, but have not received the corresponding revokeAndAck AND the HTLC expires within $[t, t + (2 + r)\,\texttt{windowSize}]$ **then**
23: 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`
24: **end if**
25: **end for**
26: send (SUBMIT, `toSubmit`) to $\mathcal{G}_{\text{Ledger}}$

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

Fig. 35.

Protocol Π_{LN} - close

1: Upon receiving (CLOSECHANNEL, `receipt`, *tid*) from $\mathcal{E}$:
2: ensure `receipt` corresponds to an open `channel` $\in$ `channels` with ID *tid*
3: assign latest `channel` sequence number to n
4: HTLCs $\leftarrow \emptyset$
5: **for** every HTLC output $\in \texttt{localCom}_n$ with number i **do**
6: sig $\leftarrow$ SIGNIBS $(\texttt{localHTLC}_{n,i}, sh_{\text{htlc},n})$
7: add $\left(\text{sig}, \texttt{HTLCSigs}_{n,i}, \texttt{localHTLC}_{n,i}\right)$ to HTLCs
8: **end for**
9: sig $\leftarrow$ SIGNDS $(\texttt{localCom}_n, sh_F)$
10: add `receipt`(`channel`) to `closedChannels`
11: remove `channel` from `channels`
12: send (SUBMIT, $(\text{sig}, \texttt{remoteSig}_n, \texttt{localCom}_n)$, HTLCs) to $\mathcal{G}_{\text{Ledger}}$

Fig. 36.

I The Ledger, Clock and Network Functionality

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 $\texttt{windowSize}$ 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 $\texttt{windowSize}$ positions before the head of the $\texttt{state}$ was included, and (ii) is valid with respect to an honest block that extends $\texttt{state}$, then it must be included in such block.

Given a synchronised honest party, we say that a transaction $\texttt{tx}$ is finalised when it becomes a part of $\texttt{state}$ in its view.

Proposition 1. *Consider any synchronised honest party that wishes to place a transaction $\texttt{tx}$ in some specific block height $[h+1, h+t-1]$ where t is a parameter and h an arbitrary positive integer. Then, as long as $t \geq 1+(2+r)\texttt{windowSize}$, where $r = \lceil(\texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2})/\texttt{minTime}_{\texttt{window}}\rceil$, $\texttt{tx}$ is guaranteed to be included in the intended block range as long as the party submits $\texttt{tx}$ to the ledger functionality by the time the block indexed by $h+t-1-(2+r)\texttt{windowSize}$ is added to $\texttt{state}$ in its view.*

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

Suppose U transmits the transaction $\texttt{tx}$ to the ledger at time τ^U_{h+x}. Observe that as long as $\tau_{h+x} + \texttt{maxTime}_{\texttt{window}}$ is $\texttt{Delay}/2$ before the time that block with index $h+t-1-2\texttt{windowSize}$ enters $\texttt{state}$, then $\texttt{tx}$ is guaranteed to enter the $\texttt{state}$ in a block with index up to $h+t-1$ since $\texttt{advBlcks}_{\texttt{window}} < \texttt{windowSize}$. It follows we need $\tau_{h+x} + \texttt{maxTime}_{\texttt{window}} < \tau_{h+t-1-2\texttt{windowSize}} - \frac{\texttt{Delay}}{2}$. 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 $\texttt{windowSize}$ blocks enter $\texttt{state}$. As a result $r\texttt{windowSize}$ blocks require at least $r\texttt{minTime}_{\texttt{window}} \geq \texttt{maxTime}_{\texttt{window}} + \frac{\texttt{Delay}}{2}$ rounds. It follows that if $t \geq 1+(2+r)\texttt{windowSize}$ and $x = t-1-(2+r)\texttt{windowSize}$ 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 $\texttt{state}$ (the immutable state of the ledger), $\texttt{NxtBC}$ (a list of transaction identifiers to be added to the ledger), $\texttt{buffer}$ (the set of pending transactions), τ_L (the rules under which the state is extended), and $\boldsymbol{\tau}_{\texttt{state}}$ (the time sequence where all immutable blocks where added). The variables are initialized as follows: $\texttt{state} := \boldsymbol{\tau}_{\texttt{state}} := \texttt{NxtBC} := \varepsilon$, $\texttt{buffer} := \emptyset$, $\tau_L = 0$. For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer $\texttt{pt}_i$ (initially set to 1) and a

current-state view $\texttt{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\boldsymbol{\mathcal{I}}_H^T$ (initially $\boldsymbol{\mathcal{I}}_H^T := \varepsilon$).

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 $\tau' < \tau_L - \texttt{Delay}$. 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 $\boldsymbol{\mathcal{I}}_H^T := \boldsymbol{\mathcal{I}}_H^T || (I, U_p, \tau_L)$;
   (b) Compute $\boldsymbol{N} = (\boldsymbol{N}_1, \ldots, \boldsymbol{N}_\ell) := \mathsf{ExtendPolicy}(\boldsymbol{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \boldsymbol{\tau}_{\texttt{state}})$ and if $\boldsymbol{N} \neq \varepsilon$ set $\texttt{state} := \texttt{state} || \mathsf{Blockify}(\boldsymbol{N}_1) || \ldots || \mathsf{Blockify}(\boldsymbol{N}_\ell)$ and $\boldsymbol{\tau}_{\texttt{state}} := \boldsymbol{\tau}_{\texttt{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \ldots, || \tau_L$.
   (c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then delete BTX from buffer. Also, reset $\texttt{NxtBC} := \varepsilon$.
   (d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k := |\texttt{state}|$ 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 $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ 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 $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau_L, U_p)$

(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 $(\mathtt{state}, \mathtt{buffer}, \boldsymbol{\mathcal{I}}_H^T)$ 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 $\boldsymbol{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\boldsymbol{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ 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 listOfTxid := listOfTxid$||\text{txid}_i$.
  (c) Finally, set $\mathtt{NxtBC} := \mathtt{NxtBC}||(\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.

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

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

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

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

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

Synchronization:

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

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

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 $\boldsymbol{M} := \boldsymbol{M}||(m, \text{mid}_1, D_{\text{mid}_1}, U_1)|| \ldots ||(m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, 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 $\boldsymbol{M} := \boldsymbol{M}||(m_{i_1}, \text{mid}_{i_1}, D_{\text{mid}_{i_1}}, U_{i_1})|| \ldots ||(m_{i_\ell}, \text{mid}_{i_\ell}, D_{\text{mid}_{i_\ell}}, U_{i_\ell})$, 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 $(m, \text{mid}, D_{\text{mid}}, U_p) \in \boldsymbol{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\boldsymbol{M}_0^{U_p}$ denote the subvector $\boldsymbol{M}$ 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 $\boldsymbol{M}$). Delete all entries in $\boldsymbol{M}_0^{U_p}$ from $\boldsymbol{M}$, and send $\boldsymbol{M}_0^{U_p}$ 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 $\boldsymbol{M}$, 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 $\boldsymbol{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\boldsymbol{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

J 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. 37.

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. 38.

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

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. 39.

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

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. 19, line 9) and in the ideal world (c.f. Fig. 40, line 5).

Furthermore, one (READ) is sent to $\mathcal{G}_{\text{Ledger}}$ from *Alice* in both cases (Fig. 19, line 8 and Fig. 9, line 9).

Additionally, $\mathcal{S}_{\text{LN-Reg}}$ 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. 19 change the state of *Alice* ITI in the same way as lines 2-5 of Fig. 40.

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 4 ensures that the rest of the messages are responded in an indistinguishable way in both worlds. □

Lemma 6. $\text{EXEC}^{\mathcal{F}_{\text{PayNet,Reg}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg}},\mathcal{E}} = \text{EXEC}^{\mathcal{F}_{\text{PayNet,Open}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open}},\mathcal{E}}$

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

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

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

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

Fig. 40.

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

1: For messages register, registerDone, toppedUp, openChannel, channelAnnounced and checkForNew, act like $\mathcal{F}_{\mathrm{PayNet}}$, but skip lines that call `checkClosed`().

2: Upon receiving any other message M from *Alice*:
3: **if** M is a valid $\mathcal{F}_{\mathrm{PayNet}}$ 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 $\mathcal{F}_{\mathrm{PayNet}}$ message from $\mathcal{S}$ **then**
8: send M to *Alice*
9: **end if**

Fig. 41.

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

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

Upon receiving (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
  if both *Alice* and *Bob* are honest **then**
    Simulate the interaction between *Alice* and *Bob* in their respective ITI, as defined in Figures 20-24. 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 $\mathcal{F}_{\text{PayNet},\text{Open}}$.
    After submitting F to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, send (channelAnnounced, $Alice, p_{Alice,F}, p_{Bob,F}$, *fchid*, *pchid*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$.
  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 20, 22, and 24.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 $\mathcal{F}_{\text{PayNet},\text{Open}}$.
  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 21 and 23. 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 $\mathcal{F}_{\text{PayNet},\text{Open}}$.
  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 (fundingLocked, *Alice*, Σ_{Alice}, *fchid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
  execute lines 5-9 of Fig. 25 with *Alice*'s ITI, using Σ_{Alice} from message
  if *Bob* is honest **then**
    expect the delivery of *Alice*'s (fundingLocked) message from $\mathcal{A}$
    send (fundingLocked, *fchid*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$
    upon receiving (fundingLocked, *Bob*, Σ_{Bob}, *fchid*) from $\mathcal{F}_{\text{PayNet},\text{Open}}$:
    simulate Fig. 26 with message from *Alice* in *Bob*'s ITI, using Σ_{Bob} from $\mathcal{F}_{\text{PayNet},\text{Open}}$'s message
  end if

Upon receiving the (fundingLocked) message with the simulated *Alice* ITI:
  simulate Fig. 26 receiving the message with *Alice*'s ITI
  send (channelOpened, *fchid*) to $\mathcal{F}_{\text{PayNet},\text{Open}}$

Fig. 42.

Proof. When $\mathcal{E}$ sends (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) to *Alice*, the interaction of Figures 20-24 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.

We observe that the input validity check executed by $\mathcal{F}_{\text{PayNet,Open}}$ (Fig. 10, line 2) filters only messages that would be ignored by the real protocol as well and would not change its state either (Fig. 20, line 2).

We also observe that, upon receiving the message openChannel or channelAnnounced, $\mathcal{F}_{\text{PayNet,Open}}$ 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. 25 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. 20, line 7 or in Fig. 21, 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. 23, 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. 24, line 8) (which also needs a valid *Alice-Bob* interaction up to that point)

When $\mathcal{E}$ sends (checkForNew, *Alice*, *Bob*, *tid*) to *Alice* in the ideal world, line 10 of Fig. 10 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 8 of Fig. 10 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ such that the entry `pendingOpen`(*fchid*) exists and has temporary ID *tid*, as mandated by line 7 of Fig. 10. Such a message is sent by $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ of Fig. 42 either in lines 5/8, or in lines 4/12. One of the first pair of lines is executed only if $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ 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 10 of Fig. 10 in the ideal and line 2 of Fig. 25 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. 10, lines 13-14) and the real world (Fig. 25, lines 3-4).

Moving on, in the real world lines 5-9 of Fig. 25 are executed by *Alice* and, given that $\mathcal{A}$ allows it, the code of Fig. 26 is executed by *Bob*. Likewise, in the ideal world, the functionality executes lines 15-16 of Fig. 25 and as a result it (always) sends (FUNDINGLOCKED, *Alice*, Σ_{Alice}, *fchid*) to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$. In turn $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ simulates lines 5-9 of Fig. 25 with *Alice*'s ITI and, if $\mathcal{A}$ allows it, $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ simulates the code of Fig. 26 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 $\mathcal{F}_{\text{PayNet,Pay}}$

1: For messages REGISTER, REGISTERDONE, TOPPEDUP, OPENCHANNEL, CHANNELANNOUNCED, CHECKFORNEW, POLL, PAY, PUSHADD, PUSHFULFILL, FULFILLONCHAIN and COMMIT, 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. 43.

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

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

1: Upon receiving (FULFILLONCHAIN, t, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
2: execute lines 20-26 of Fig. 35 as *Alice*, using t from message

3: Upon receiving $\left(\text{PAY}, \textit{Alice}, \textit{Bob}, x, \overrightarrow{\texttt{path}}, \texttt{receipt}, \textit{payid}\right)$ from $\mathcal{F}_{\text{PayNet,Pay}}$:
4: add $(\overrightarrow{\texttt{path}}, \textit{payid})$ to `payids`
5: strip *payid*, simulate receiving the message with *Alice* ITI and further execute the parts of Π_{LN} that correspond to honest parties (Fig. 29- Fig. 31)
6: **if** any "ensure" in Π_{LN} fails until *Bob* processes UPDATEADDHTLC **then** // payment failed
7: add $(\perp, \textit{payid})$ to `charged`(*Alice*)
8: remove $(\overrightarrow{\texttt{path}}, \textit{payid})$ from `payids`
9: **end if**

10: Upon receiving (POLL, $\Sigma_{\textit{Alice}}$, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
11: simulate Fig. 27, lines 3-28 receiving (POLL), using $\Sigma_{\textit{Alice}}$ from the message, with *Alice*'s ITI

Fig. 44.

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. 35, lines 1-12 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: Upon receiving (PUSHADD, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
4: simulate Fig. 35, lines 13-17 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

5: Upon receiving (COMMIT, *pchid*, *Alice*) from $\mathcal{F}_{\text{PayNet,Pay}}$:
6: simulate Fig. 32 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
7: **if** during the simulation above, line 10 of Fig. 34 is simulated in *Alice*'s ITI **then**
8: send (UPDATE, `receipt`, *Alice*) to $\mathcal{F}_{\text{PayNet,Pay}}$, where `receipt` is the receipt just added to the simulated `updatesToReport` (Fig. 34, line 10)
9: upon receiving (CONTINUE) from $\mathcal{F}_{\text{PayNet,Pay}}$, carry on with the simulation
10: **end if**

Fig. 45.

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

1: Upon receiving any message with a concatenated (STATE, Σ) part from $\mathcal{F}_{\text{PayNet},\text{Pay}}$: // PAY, PUSHFULFILL, PUSHADD, COMMIT
2: handle first part of the message normally
3: **if** at the end of the simulation above, control is still held by $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}}$ **then**
4: **for all** $\Sigma_{Alice} \in \Sigma$ **do**
5: **for all** $(\overrightarrow{\texttt{path}}, payid) \in \texttt{payids} : Alice \in \overrightarrow{\texttt{path}}$ **do**
6: **if** *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}) **then** // no or bad communication with *Bob*'s previous player
7: add to `charged`(*Alice*) a tuple (`corrupted`, *payid*) where `corrupted` is set to one of the corrupted parties between *Alice* and *Bob*
8: remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`
9: **else if** 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 and a tx that spends the delayed output of $\texttt{remoteCom}_m \vee$ ((Σ_{Alice} contains the most recent $\texttt{remoteCom}_n$ or $\texttt{localCom}_n$ of the channel before *Alice* and the `HTLC-success` 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-timeout` or an `HTLC-success` that pays the counterparty) for HTLC of the channel after *Alice*) **then** // *Alice* did not fulfill in time
10: add (*Alice*, *payid*) to `charged`(*Alice*)
11: remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`
12: **else if** *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}) **then** // honest payment completed
13: add (*Alice*, *payid*) to `charged`(*Alice*)
14: remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`
15: **end if**
16: **end for**
17: **end for**
18: **end if**
19: clear `charged` and send (RESOLVEPAYS, `charged`) to $\mathcal{F}_{\text{PayNet},\text{Pay}}$

Fig. 46.

Lemma 7.

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

Proof. Before focusing on individual messages sent by $\mathcal{E}$, 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 m is the maximum number of channels that a player can open 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 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 $\text{COMBINEKEY}(pb_{Bob,\text{rev}}, sb_{Bob,\text{rev}}, p_{Alice,\text{com},n}, s_{Alice,\text{com},n})$.) Suppose that p is the maximum total number of opens and updates that a player can perform, m is the maximum number of opens a player can perform 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 $\forall \mathcal{E}, \Pr[S] \leq nm \cdot \text{E-master}(k)$.

We can now move on to treating individual messages sent by $\mathcal{E}$ during the execution. When $\mathcal{E}$ sends $\left(\text{PAY}, Bob, x, \overrightarrow{\texttt{path}}, payid\right)$ to *Alice* in the ideal world,

$\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ is always notified (Fig. 11, line 4) and simulates the relevant execution of the real world (Fig. 44, line 5). 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ (Fig. 17, 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. 45, lines 2, 4, 6). Neither $\mathcal{F}_{\text{PayNet},\text{Pay}}$ nor $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ 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 18-26 of Fig. 35 are executed by *Alice*. In the ideal world on the other hand, $\mathcal{F}_{\text{PayNet},\text{Pay}}$ sends (READ) to $\mathcal{G}_{\text{Ledger}}$ (Fig. 17, line 8) as *Alice* and subsequently instructs $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ to simulate the receiving of (FULFILLONCHAIN) with *Alice*'s ITI (Fig. 44, lines 1-2). 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ skips line 19 of Fig. 35, using as t the one received from $\mathcal{F}_{\text{PayNet},\text{Pay}}$ in the message (FULFILLONCHAIN, t, *Alice*). Since $\mathcal{F}_{\text{PayNet},\text{Pay}}$ 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. 16, line 4) and the real (Fig. 27, line 2) worlds. Subsequently, in the real world lines 3-28 of Fig. 27 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. 44, line 11), 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. 27 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 28 of Fig. 27.

The event E in which $\mathcal{F}_{\text{PayNet},\text{Pay}}$ executes line 6 of Fig. 16 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 $\mathcal{F}_{\text{PayNet,Pay}}$ executes line 11 of Fig. 16 and halts. We will now prove that, during $\text{EXEC}^{\mathcal{F}_{\text{PayNet,Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}},\mathcal{E}}$, it is $\Pr[E|\neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$. The condition of Fig. 16, 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. 16, 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. 28, lines 19-21). Given that $\texttt{tx}_3$ is broadcast when $|\Sigma_{Alice}| \leq h_2 - (2 + r)\,\texttt{windowSize}$, it is guaranteed to be on-chain in a block $h_3 \leq h_2$ (according to Proposition 1). Since $\texttt{tx}_3$ spends the same funds as $\texttt{tx}_2$, the two cannot be part of the chain simultaneously. Since $E'_{1,1} \Rightarrow \Sigma_{Alice}$ contains $\texttt{tx}_2$ and $E'_2 \Rightarrow \Sigma_{Alice}$ contains $\texttt{tx}_3$, $E'_{1,1}$ 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 $\mathcal{S}_{\text{LN-Reg-Open}}$, 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. 46, line 2). It may be the case that at the end of the handling of line 2, $\mathcal{S}_{\text{LN-Reg-Open}}$ 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. 29, line 6, when the first message is $\left(\text{PAY}, Bob, x, \overrightarrow{\texttt{path}}\right)$ and *Bob* is corrupted), or if the handling of the message results in sending (SUBMIT) to $\mathcal{G}_{\text{Ledger}}$ (e.g. in Fig. 35, line 11 when the first message is (PUSHFULFILL, *pchid*) and counter-

party has gone on-chain). In that case, the (STATE) message is simply ignored (Fig. 46, 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. 46, lines 4-17). 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. 12 and 13, where, if it does not halt (Fig. 12, lines 12, 15 and 24 and Fig. 13, line 10), it updates the state of each affected channel (Fig. 13, 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 12 of Fig. 12 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 15 of Fig. 12 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 24 of Fig. 12 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) – i.e. halts in the event $(A \wedge B) \vee (\neg B \wedge C \wedge D)$. $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ 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 F) and his previous counterparty timed out the HTLC (event G) (Fig. 46, line 9), – i.e. *Dave* is charged in the event $B \vee (F \wedge G)$.

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

The only way for event C to be true is if $\mathcal{E}$ sends (POLL) to *Dave* during the prescribed time period (Fig. 16, line 3) – note that the addition to `polls`(*Dave*) during registration (Fig. 9, line 9) 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. 29, line 19), registration happens necessarily before handling (INVOICE) (Fig. 9, line 21) and the element added to `polls`(*Dave*) at registration is the chain height at that time (Fig. 9, line 9).

When *Dave* receives (POLL), $\mathcal{F}_{\text{PayNet,Pay}}$ always sends (GETCLOSEDFUNDS) to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ (Fig. 16, line 17) (since, as we saw earlier, $\mathcal{F}_{\text{PayNet,Pay}}$ never halts).

Event G 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 F happens until a moment when $|\Sigma_{Dave}| \leq$ `IncomingCltvExpiry` $-(2+r)$ `windowSize`, *Dave* has the time to successfully fulfill the HTLC. Given C, *Dave* has POLLed at two moments $h_1, h_2 \in$ [`OutgoingCltvExpiry`, `IncomingCltvExpiry` - $(2+r)$`windowSize`], such that $h_2 \geq h_1 + (2+r)$ `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. 35, lines 1-11), 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. 35, lines 18-26) 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. 28, 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. $\Pr[(C \wedge D) \wedge (F \wedge G) | \neg P \wedge \neg Q \wedge \neg R \wedge \neg S] = 0$.

It remains to be proven that the halt of line 10 in Fig. 13 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. 46, lines 6, 9, or 12). 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. 13, 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 update in `updatesToReport` (Fig. 34, line 10) which causes $\mathcal{S}$ to report the update to $\mathcal{F}_{\text{PayNet}}$ (Fig. 45, line 8). We therefore conclude that $\mathcal{F}_{\text{PayNet}}$ never halts on line 10 of Fig. 13.

To conclude, given $\neg P \wedge \neg Q \wedge \neg R \wedge \neg S$, it is $\text{EXEC}^{\mathcal{F}_{\text{PayNet,Open}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}},\mathcal{E}} = \text{EXEC}^{\mathcal{F}_{\text{PayNet,Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}-\text{Pay}},\mathcal{E}}$. 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

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

□

Simulator $\mathcal{S}$

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

1: Upon receiving (CLOSECHANNEL, `receipt`, *tid*, *Alice*) from $\mathcal{F}_{\text{PayNet}}$:
2: simulate Fig. 36 receiving (CLOSECHANNEL, `receipt`, *tid*) with *Alice*'s ITI

3: every time `closedChannels` of *Alice* is updated with data from a `channel` (Fig. 36, line 10 and Fig. 28, line 23), send (CLOSEDCHANNEL, `channel`, *Alice*) to $\mathcal{F}_{\text{PayNet}}$ and expect (CONTINUE) from $\mathcal{F}_{\text{PayNet}}$ to resume simulation

Fig. 47.

Lemma 8.

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

Proof. 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 (CLOSECHANNEL, `receipt`, *tid*) to *Alice*, in the ideal world, if it is not the first closing message to *Alice* the message is ignored (Fig. 14, line 5). Similarly in the real world, if there has been another such message, *Alice* ignores it (Fig.36, lines 11 and 2).

In the case that it is indeed the first closing message, in the ideal world $\mathcal{F}_{\text{PayNet}}$ takes note that this close is pending (Fig. 14, lines 3-4) and stops serving more requests for this channel (line 5), before asking $\mathcal{S}$ to carry out channel closing. $\mathcal{S}$ then simulates the response to the original message from $\mathcal{E}$ with *Alice*'s ITI (Fig. 47). 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. 36 always passes and the rest of Fig. 36 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. 36 is executed by the real *Alice* in its entirety. 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*)) to $\mathcal{E}$ and empties these fields (Fig. 17, lines 15-16). In the real world, *Alice* sends (news, `newChannels`, `closedChannels`, `updatesToReport`) to $\mathcal{E}$ and empties these fields as well (Fig. 27, lines 29-30). `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. 10, line 30). This happens when the simulated *Alice* ITI handles a fundingLocked message from *Bob* (Fig. 42, line 26). In the real world *Alice* would have modified her `newChannels` while handling *Bob*'s fundingLocked (Fig. 26, 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. 10, line 22). 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. 42, lines 17-22). 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. 17, line 13), which in turn happens exactly when the simulated *Alice* ITI adds the `channel` to her `closedChannels` (Fig. 47, line 3). Therefore the real and ideal `closedChannels` are always synchronized.

Regarding `updatesToReport`, in the real world it is populated exclusively in line 10 of Fig. 34. In the ideal world on the other hand, it is updated in line 6 of Fig. 11, which is triggered only by an (update) message by $\mathcal{S}$. This message is sent only when line 10 of Fig. 34 is simulated by $\mathcal{S}$ (Fig. 45, line 8). 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. 34, line 2 and Fig. 33, lines 5 and 17), 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. 17, 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. 45, line 6). We conclude that the `updatesToReport` sent to $\mathcal{E}$ in either the real or the ideal world are always identical.

Lastly, in the ideal world, whenever (READ) is sent to $\mathcal{G}_{\text{Ledger}}$ and a reply is received, the function `checkClosed` (Fig.15) 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 15 of Fig. 15 in case a channel is closed without using a commitment transaction. Similarly to event E in the proof of Lemma 7, 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 18 of Fig. 15 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 17) would cause $\mathcal{F}_{\text{PayNet}}$ to alert $\mathcal{S}$ (Fig. 16, line 17), who in turn would submit the punishment transaction in time to prevent the counterparty from spending the delayed payment (Fig. 28, lines 19-21). 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 25 of Fig. 15 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 (CLOSECHANNEL) message to $\mathcal{S}$ (Fig. 14, line 6) for every element that it adds to `pendingClose` (Fig 14, line 4) and this causes $\mathcal{S}$ to submit the closing transaction to $\mathcal{G}_{\text{Ledger}}$ (Fig. 36, line 12). 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 24 of Fig. 15. $\mathcal{F}_{\text{PayNet}}$ 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 (CLOSECHANNEL) was received, it should have been necessarily included (Proposition 1). Second, every element added to `closedChannels` (Fig. 36, line 10 and Fig. 28, line 23) corresponds to a submission of a closing transaction for the same channel (Fig. 36, line 12), or to a channel for which the closing transaction is already in the ledger state (Fig. 28, 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 25 of Fig. 15. 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 30 of Fig. 15 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 an (UPDATE) message (Fig. 45, line 8) 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. 12, lines 9, 31 and 32). Secondly, upon closure of each channel, $\mathcal{F}_{\text{PayNet}}$ would have halted if the closing balance were not the expected one (Fig. 15, line 17), 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. 10, lines 26 and 27 and Fig. 15, lines 6 or 8 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

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

□

Proof of Theorem 1. The theorem is a direct result of Lemmas 4-8. □

J.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.

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

□

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

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. 19, 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. 48. wins EUF-CMA game

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

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

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

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

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. 49. wins IBS-EUF-CMA game

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

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

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

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. 19, 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. 50. wins share-EUF game

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 $\Pr[R] \in [c - \text{E-prf}(k), c + \text{E-prf}(k)]$.

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

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

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

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

□

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. 19, 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 18 of Fig. 28 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. 51. wins master-EUF-CMA game

Proposition 6. $\forall \mathcal{E}, \Pr[S] \leq nm \cdot \text{E-master}(k)$

Proof. Let $\Pr[S] = d$ hold for the unmodified execution. When it is simulating $\text{EXEC}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}}$, $\mathcal{A}_{\text{share}}$ 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 KeyShareGen() 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

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

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

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

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

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

□

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. Spilman J.: Anti dos for tx replacement. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html (2013)
14. 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)
15. Dziembowski S., Eckey L., Faust S., Malinowski D.: PERUN: Virtual Payment Channels over Cryptographic Currencies. IACR Cryptology ePrint Archive: vol. 2017, p. 635: URL http://eprint.iacr.org/2017/635 (2017)
16. 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)
17. 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)
18. 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)
19. Green M., Miers I.: Bolt: Anonymous Payment Channels for Decentralized Currencies. In Thuraisingham et al. [29]: pp. 473–489: doi:10.1145/3133956.3134093: URL https://doi.org/10.1145/3133956.3134093 (2017)

20. 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/ndss-2017-programme/tumblebit-untrusted-bitcoin-compatible-anonymous-payment-hub/ (2017)
21. Khalil R., Gervais A.: Revive: Rebalancing Off-Blockchain Payment Networks. In Thuraisingham et al. [29]: pp. 439–453: doi:10.1145/3133956.3134033: URL https://doi.org/10.1145/3133956.3134033 (2017)
22. 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)
23. 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)
24. 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)
25. 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)
26. 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)
27. 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)
28. Katz J., Lindell Y.: Introduction to Modern Cryptography, Second Edition. CRC Press: ISBN 9781466570269 (2014)
29. 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)