

# Proving Lightning Payment Network UC-secure

Aggelos Kiayias<sup>1,2</sup> and Orfeas Stefanos Thyfronitis Litos<sup>1</sup>

<sup>1</sup> University of Edinburgh

<sup>2</sup> IOHK

akiayias@inf.ed.ac.uk, o.thyfronitis@ed.ac.uk

**Abstract.** Blockchains have enabled the decentralized exchange of digital currencies with minimal trust assumptions [1,2]. Nevertheless, the massive replication of data amongst participating nodes has a detrimental effect on scalability [3]. One approach to addressing the issue is layer-2 payment networks, which enable users to perform the vast majority of their transactions without adding them to the blockchain [4,5,6,7]. The most widely adopted layer-2 solution, which is deployed on top of Bitcoin, is the Lightning Payment Network (LPN) [4]. In this work we prove that LPN is secure in the Universal Composability framework [8]. We formally describe the LPN protocol<sup>3</sup>, abstract its intended purpose in a functionality and prove that the specified protocol UC-realizes it.

## 1 State of a channel

Consider a channel between *Alice* and *Bob*. Each party holds some data locally that are enough to ensure ownership of some funds in the channel. This is the data *Alice* holds:

- keys:
  - local funding secret key ( $s_{Alice,F}$ )
  - remote funding public key ( $p_{Bob,F}$ )
  - local payment ( $sb_{Alice,pay}$ ), htlc ( $sb_{Alice,htlc}$ ), delayed payment ( $sb_{Alice,dpay}$ ), revocation ( $sb_{Alice,rev}$ ) basepoint secrets
  - remote payment ( $pb_{Bob,pay}$ ), htlc ( $pb_{Bob,htlc}$ ), delayed payment ( $pb_{Bob,dpay}$ ), revocation ( $pb_{Bob,rev}$ ) basepoints
  - seed (for deriving local per commitment keypairs ( $s_{Alice,com,n}, p_{Alice,com,n}$ ))
  - one remote per commitment secret ( $\forall i \in [1, \dots, n], s_{Bob,com,i}$ ) for each COMMITMENTSIGNED received
  - the current and the next remote per commitment points ( $p_{Bob,com,n}$  and  $p_{Bob,com,n+1}$ )
- *Alice*’s coins (integer)
- *Bob*’s coins (integer)

---

<sup>3</sup> <https://github.com/lightningnetwork/lightning-rfc/>

- every HTLC that is included in the latest irrevocably committed (local or remote) commitment:
  - direction ( $Alice \rightarrow Bob$  or  $Bob \rightarrow Alice$ )
  - hash
  - preimage (or  $\perp$  if still unresolved)
  - coins (integer)
  - Is it included in latest  $\text{localCom}_n$ ? (boolean)
  - HTLC number
- signatures:
  - $\forall i \in [1, \dots, n]$ , signature of  $\text{localCom}_i$  generated with  $s_{Bob,F}$
  - for every HTLC included in  $\text{localCom}_i$ , if HTLC is outgoing, a signature for HTLC-timeout, else a signature for HTLC-success with  $s_{Bob,htlc,i}$

Every other piece of data used in the protocol can be derived by the above.

Representation of a channel's state (from the point of view of *Alice*):

- *Alice*'s coins  $c_{Alice}$
- *Bob*'s coins  $c_{Bob}$
- list of (coins, state  $\in \{\text{proposed}, \text{committed}\}$ ) preimage, whether we have a signature), HTLCs
  - negative coins are outgoing, positive are incoming
  - HTLCs can either be simply proposed (not in an irrevocably committed remote transaction) or committed (the opposite). After the preimage is supplied (no matter the direction), the HTLC is considered settled and is discarded.

I.e.  $\text{State}_{Alice, pchid} = (c_{Alice}, c_{Bob}, ((c_1, \text{state}_1), \dots, (c_k, \text{state}_k)))$

E.g.  $\text{State}_{Alice, pchid} = (4, 5, ((0.1, \text{proposed}), (-0.2, \text{signed})))$

We do not include in the state elements whose contents are irrelevant (e.g. sigs, keys, hashes).

## 2 UC conventions

- send (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign reply to  $\Sigma$  ...
  - =
  - {
  - send (READ) to  $\mathcal{G}_{\text{Ledger}}$

- upon receiving delayed output  $\Sigma$  ...
  - }
- every output that is returned by  $\mathcal{F}_{\text{PayNet}}$  or a player to  $\mathcal{E}$  is in fact a delayed output: It is handed over to  $\mathcal{A}$ , who in turn decides when to give it to  $\mathcal{E}$ .

### 3 Differences from LND

- They use an ad-hoc construction for generating progressive secrets from seed and index, we use a PRF.
- To generate several public keys from one piece of info, they use the basepoint and the per commitment point and take advantage of EC homomorphic properties. We use an Identity Based Signature scheme.
- They also provide a way to cooperatively close a channel. **we should do this as well**
- In LND there are more messages that cover errors in transmission etc. There are also rules that govern message retransmission upon connection failure.
- We don't use the concept of “dust transactions/outputs”.
- In our case, the **delay** of a player is set once, at her registration. In contrast to LN, it can't be changed later.

### 4 Transaction Structure

A well-formed transaction contains:

- A list of inputs
- A list of outputs
- An arbitrary payload (optional)

Each input must be connected to a single valid, previously unconnected (unspent) output in the state.

We assume a one-way, collision-free hash function  $\mathcal{H}$  that creates the id of each transaction.

A well-formed output contains:

- A value in coins
- A list of spending methods. An input that spends this output must specify exactly one of the available spending methods.

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 made with the private keys that correspond to the public keys of one of the conjunctions. If empty, no signatures are needed.
- Absolute locktime in block height, transaction height or time. The output can be spent by an input to a transaction that is added to the state after the specified block height, transaction height or time.
- Relative locktime in block height, transaction height or time. The output can be spent by an input that is added to the state after the current output has been part of the state for the specified number of blocks, transactions or time.
- Hashlock value. The output can be spent by an input that contains a preimage that hashes to the hashlock value. If empty, the input does not need to specify a preimage.

If both the absolute and the relative locktime are empty, output can be spent immediately after being added to the state.

A well-formed input contains:

- A reference to the output and the spending method it spends
- A set of signatures that correspond to one of the conjunctions of public keys in the referred spending method (if needed)
- A preimage that hashes to the hashlock value of the referred spending method (if needed)

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.

## 5 Lightning Protocol

**Protocol  $\Pi_{LN}$**  (self is *Alice* always) - support

```

1: Initialisation:
2:   channels, pendingOpen, pendingPay, pendingClose  $\leftarrow \emptyset$ 
3:   newChannels, closedChannels, updatesToReport  $\leftarrow \emptyset$ 
4:   unclaimedOfferedHTLCs, unclaimedReceivedHTLCs, pendingGetPaid  $\leftarrow \emptyset$ 

5: Upon receiving (REGISTER, delay, relayDelay) from  $\mathcal{E}$ :
6:   delay  $\leftarrow$  delay // Must check chain at least once every delay blocks
7:   relayDelay  $\leftarrow$  relayDelay
8:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign largest block number to lastPoll
   maybe remove lastPoll from real world?
9:    $(pk_{\text{Alice}}, sk_{\text{Alice}}) \leftarrow \text{KeyGen}()$ 
10:  send (REGISTER, Alice, delay, relayDelay,  $pk_{\text{Alice}}$ ) to  $\mathcal{E}$ 

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

15: Upon receiving any message ( $M$ ) except for (REGISTER):
16:  if if haven't received (REGISTER) from  $\mathcal{E}$  then
17:    send (INVALID,  $M$ ) to  $\mathcal{E}$  and ignore message
18:  end if

19: function GetKeys
20:   $(p_F, s_F) \leftarrow \text{KeyGen}()$  // For  $F$  output
21:   $(p_{\text{pay}}, s_{\text{pay}}) \leftarrow \text{MKeyGen}()$  // For com output to remote
22:   $(p_{\text{dpay}}, s_{\text{dpay}}) \leftarrow \text{MKeyGen}()$  // For com output to self
23:   $(p_{\text{htlc}}, s_{\text{htlc}}) \leftarrow \text{MKeyGen}()$  // For htlc output to self
24:  seed  $\xleftarrow{\$} U(k)$  // For per com point
25:   $(p_{\text{rev}}, s_{\text{rev}}) \leftarrow \text{MKeyGen}()$  // For revocation in com
26:  return  $((p_F, s_F), (p_{\text{pay}}, s_{\text{pay}}), (p_{\text{dpay}}, s_{\text{dpay}}),$ 
27:     $(p_{\text{htlc}}, s_{\text{htlc}}), \text{seed}, (p_{\text{rev}}, s_{\text{rev}}))$ 
28: end function

```

Fig. 1.

**Protocol  $\Pi_{LN}$  - 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:    $((ph_F, sh_F), (phb_{pay}, shb_{pay}), (phb_{dpay}, shb_{dpay}), (phb_{htlc}, shb_{htlc}), \mathbf{seed}, (phb_{rev}, shb_{rev})) \leftarrow \mathbf{GetKeys}()$
- 4:    $\text{prand}_1 \leftarrow \mathbf{PRF}(\mathbf{seed}, 1)$
- 5:    $(sh_{com,1}, ph_{com,1}) \leftarrow \mathbf{KeyShareGen}(1^k; \text{prand}_1)$
- 6:   associate keys with *tid*
- 7:   add (*Alice*, *Bob*, *x*, *tid*,  $(ph_F, sh_F), (phb_{pay}, shb_{pay}), (phb_{dpay}, shb_{dpay}), (phb_{htlc}, shb_{htlc}), (phb_{com,1}, shb_{com,1}), (phb_{rev}, shb_{rev}), tid$ ) to **pendingOpen**
- 8:   send (OPENCHANNEL, *x*, **delay** + *k* + (2 + *r*) **windowSize**,  $ph_F, phb_{pay}, phb_{dpay}, phb_{htlc}, ph_{com,1}, phb_{rev}, tid$ ) to *Bob*

**Fig. 2.**

**Protocol  $\Pi_{LN}$  - OPENCHANNEL from *Bob***

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

**Fig. 3.**

**Protocol  $\Pi_{LN}$  - ACCEPTCHANNEL**

- 1: Upon receiving (ACCEPTCHANNEL, **remoteDelay**,  $pt_F$ ,  $ptb_{pay}$ ,  $ptb_{dpay}$ ,  $ptb_{htlc}$ ,  $pt_{com,1}$ ,  $ptb_{rev}$ ,  $tid$ ) 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  $\mathcal{G}_{Ledger}$  and assign reply to  $\Sigma_{Alice}$
- 5:   assign to **prevout** a transaction output found in  $\Sigma_{Alice}$  that is currently exclusively spendable by *Alice* and has value  $y \geq x$
- 6:    $F \leftarrow \text{TX}$  {input spends **prevout** with a **signature**(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:    $pt_{rev,1} \leftarrow \text{CombinePubKey}(ptb_{rev}, ph_{com,1})$
- 10:    $ph_{dpay,1} \leftarrow \text{PubKeyGen}(phb_{dpay}, ph_{com,1})$
- 11:    $ph_{pay,1} \leftarrow \text{PubKeyGen}(phb_{pay}, ph_{com,1})$
- 12:   **remoteCom**  $\leftarrow$  **remoteCom**<sub>1</sub>  $\leftarrow$  TX {input: output 1 of  $F$ , outputs:  $(x, ph_{pay,1}), (0, ph_{rev,1} \vee (pt_{dpay,1}, \text{delay} + k + (2 + r) \text{windowSize relative}))$ }
- 13:   **localCom**  $\leftarrow$  TX {input: output 1 of  $F$ , outputs:  $(x, pt_{rev,1} \vee (ph_{dpay,1}, \text{remoteDelay relative})), (0, pt_{pay,1})$ }
- 14:   add **remoteCom** and **localCom** to channel entry in **pendingOpen**
- 15:   sig  $\leftarrow$  **signature**(**remoteCom**<sub>1</sub>,  $sh_F$ )
- 16:   **lastRemoteSigned**  $\leftarrow$  0
- 17:   send (FUNDINGCREATED,  $tid$ ,  $pchid$ , sig) to *Bob*

Fig. 4.

**Protocol  $\Pi_{LN}$  - FUNDINGCREATED**

- 1: Upon receiving (FUNDINGCREATED,  $tid$ ,  $pchid$ ,  $BobSig_1$ ) from *Bob*:
- 2:   ensure there is a temporary ID  $tid$  with *Bob* in **pendingOpen** on which we have sent up to **ACCEPTCHANNEL**
- 3:    $ph_{rev,1} \leftarrow \text{CombinePubKey}(ph_{rev}pt_{com,1})$
- 4:    $pt_{dpay,1} \leftarrow \text{PubKeyGen}(pt_{dpay}, pt_{com,1})$
- 5:    $pt_{pay,1} \leftarrow \text{PubKeyGen}(pt_{pay}, pt_{com,1})$
- 6:    $localCom \leftarrow localCom_1 \leftarrow \text{TX}$  {input: output 1 of  $F$ , outputs:  $(x, pt_{pay,1}), (0, pt_{rev,1} \vee (ph_{dpay,1}, \text{remoteDelay relative}))$ }
- 7:   ensure  $\text{verify}(localCom_1, BobSig_1, pt_F) = \text{True}$
- 8:    $remoteCom \leftarrow remoteCom_1 \leftarrow \text{TX}$  {input: output 1 of  $F$ , outputs:  $(x, ph_{rev,1} \vee (pt_{dpay,1}, \text{delay} + k + (2 + r) \text{windowSize relative})), (0, ph_{pay,1})$ }
- 9:   add  $BobSig_1, remoteCom_1$  and  $localCom_1$  to channel entry in **pendingOpen**
- 10:    $sig \leftarrow \text{signature}(remoteCom_1, sh_F)$
- 11:   mark channel as “broadcast, no FUNDINGLOCKED”
- 12:    $lastRemoteSigned, lastLocalSigned \leftarrow 0$
- 13:   send (FUNDINGSIGNED,  $pchid$ ,  $sig$ ) to *Bob*

**Fig. 5.**

**Protocol  $\Pi_{LN}$  - FUNDINGSIGNED**

- 1: Upon receiving (FUNDINGSIGNED,  $pchid$ ,  $BobSig_1$ ) from *Bob*:
- 2:   ensure there is a channel ID  $pchid$  with *Bob* in **pendingOpen** on which we have sent up to **FUNDINGCREATED**
- 3:   ensure  $\text{verify}(localCom, BobSig_1, pb_F) = \text{True}$
- 4:    $localCom_1 \leftarrow localCom$
- 5:    $lastLocalSigned \leftarrow 0$
- 6:   add  $BobSig_1$  to channel entry in **pendingOpen**
- 7:    $sig \leftarrow \text{signature}(F, sk_{Alice})$
- 8:   mark  $pchid$  in **pendingOpen** as “broadcast, no FUNDINGLOCKED”
- 9:   send (SUBMIT, (sig,  $F$ )) to  $\mathcal{G}_{Ledger}$

**Fig. 6.**



**Protocol  $\Pi_{LN}$  - CHECKFORNEW**

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

**Fig. 7.**

**Protocol  $\Pi_{LN}$  - FUNDINGLOCKED**

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

**Fig. 8.**

**Protocol  $\Pi_{LN}$  - poll**

```

1: Upon receiving (POLL) from  $\mathcal{E}$ :
2:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign reply to  $\Sigma_{\text{Alice}}$ 
3:   assign largest block number in  $\Sigma_{\text{Alice}}$  to lastPoll
4:   toSubmit  $\leftarrow \emptyset$ 
5:   for all  $\tau \in \text{unclaimedOfferedHTLCs}$  do
6:     if input of  $\tau$  has been spent then // by remote HTLC-success
7:       remove  $\tau$  from unclaimedOfferedHTLCs
8:       if we are intermediary then
9:         retrieve preimage  $R$ ,  $pchid'$  of previous channel on the path of
the HTLC, and  $\text{HTLCNo}'$  of the corresponding HTLC' in  $pchid'$ 
10:        add  $(\text{HTLCNo}', R)$  to  $\text{pendingFulfills}_{pchid'}$ 
11:      end if
12:    else if input of  $\tau$  has not been spent and timelock is over then
13:      remove  $\tau$  from unclaimedOfferedHTLCs
14:      add  $\tau$  to toSubmit
15:    end if
16:  end for
17:  run loop of Fig. 10
18:  for all honestly closed  $\text{remoteCom}_n$  that were processed above, with
channel id  $pchid$  do
19:    for all received HTLC outputs  $i$  of  $\text{remoteCom}_n$  do
20:      if there is an entry in  $\text{pendingFulfills}_{pchid}$  with the same  $\text{HTLCNo}$ 
and  $R$  then
21:         $\text{TX} \leftarrow \{\text{input: } i \text{ HTLC output of } \text{remoteCom}_n \text{ with } (ph_{\text{htlc}, n}, R)$ 
as method, output:  $pk_{\text{Alice}}\}$ 
22:         $\text{sig} \leftarrow \text{signature}(\text{TX}, sh_{\text{htlc}, n})$ 
23:        add  $(\text{sig}, \text{TX})$  to toSubmit
24:        remove entry from  $\text{pendingFulfills}_{pchid}$ 
25:      end if
26:    end for
27:  end for
28:  send (SUBMIT, toSubmit) to  $\mathcal{G}_{\text{Ledger}}$ 

29: Upon receiving (GETNEWS) from Alice:
30:   clear newChannels, closedChannels, updatesToReport and send them to
Alice with message name NEWS

```

**Fig. 9.**

Loop over closed channels for poll

```

1: for all  $\text{remoteCom}_n \in \Sigma_{\text{Alice}}$  that spend  $F$  of a  $\text{channel} \in \text{channels}$  do
2:   if we do not have  $sh_{\text{rev},n}$  then // Honest closure
3:     for all unspent offered HTLC outputs  $i$  of  $\text{remoteCom}_n$  do
4:        $\text{TX} \leftarrow \{\text{input: } i \text{ HTLC output of } \text{remoteCom}_n \text{ with } ph_{\text{htlc},n} \text{ as}$ 
        method, output:  $pk_{\text{Alice}}\}$ 
5:        $\text{sig} \leftarrow \text{signature}(\text{TX}, sh_{\text{htlc},n})$ 
6:       if timelock has not expired then
7:         add (sig, TX) to  $\text{unclaimedOfferedHTLCs}$ 
8:       else if timelock has expired then
9:         add (sig, TX) to  $\text{toSubmit}$ 
10:      end if
11:    end for
12:    for all spent offered HTLC output  $i$  of  $\text{remoteCom}_n$  do
13:      if we are intermediary then
14:        retrieve preimage  $R$ ,  $pchid'$  of previous channel on the path of
        the HTLC, and  $\text{HTLCNo}'$  of the corresponding HTLC' in  $pchid'$ 
15:        add  $(\text{HTLCNo}', R)$  to  $\text{pendingFulfills}_{pchid'}$ 
16:      end if
17:    end for
18:  else // malicious closure
19:     $\text{rev} \leftarrow \text{TX} \{\text{inputs: all } \text{remoteCom}_n \text{ outputs, choosing } ph_{\text{rev},n} \text{ method,}$ 
      output:  $pk_{\text{Alice}}\}$ 
20:     $\text{sig} \leftarrow \text{signature}(\text{rev}, sh_{\text{rev},n})$ 
21:    add (sig, rev) to  $\text{toSubmit}$ 
22:  end if
23:  move  $\text{channel}$  from  $\text{channels}$  to  $\text{closedChannels}$ 
24: end for

```

Fig. 10.

**Protocol  $\Pi_{LN}$  - invoice**

- 1: Upon receiving  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}})$  from  $\mathcal{E}$ :
- 2: ensure that  $\overrightarrow{\text{path}}$  consists of syntactically valid  $(\text{pchid}, \text{CltvExpiryDelta})$  pair // Payment completes only if  
 $\forall i \in \overrightarrow{\text{path}}, \text{CltvExpiryDelta}_i \geq 3k + \text{RelayDelay}_i$
- 3: ensure that the first  $\text{pchid} \in \overrightarrow{\text{path}}$  corresponds to an open channel  $\in \text{channels}$  in which we own at least  $x$  in the irrevocably committed state.
- 4: choose unique payment ID  $\text{payid}$  // unique for *Alice* and *Bob*
- 5: add  $(\text{Bob}, x, \overrightarrow{\text{path}}, \text{payid}, \text{"waiting for invoice"})$  to **pendingPay**
- 6: send  $(\text{SENDINVOICE}, \text{payid})$  to *Bob*
  
- 7: Upon receiving  $(\text{SENDINVOICE}, \text{payid})$  from *Bob*:
- 8: ensure there is no  $(\text{Bob}, \text{payid})$  entry in **pendingGetPaid**
- 9: choose random, unique preimage  $R$
- 10: add  $(\text{Bob}, R, \text{payid})$  to **pendingGetPaid**
- 11: send  $(\text{INVOICE}, \mathcal{H}(R), \text{relayDelay} + 3k + 2(2 + r)\text{windowSize} - 1, \text{payid})$  to *Bob*
  
- 12: Upon receiving  $(\text{INVOICE}, h, \text{payid})$  from *Bob*:
- 13: ensure there is a  $(\text{Bob}, x, \overrightarrow{\text{path}}, \text{payid}, \text{"waiting for invoice"})$  entry in **pendingPay**
- 14: ensure  $h$  is valid (in the range of  $\mathcal{H}$ )
- 15: remove entry from **pendingPay**
- 16: send  $(\text{READ})$  to  $\mathcal{G}_{\text{Ledger}}$  and assign largest block number to  $t$
- 17:  $l \leftarrow |\overrightarrow{\text{path}}|$
- 18:  $m \leftarrow$  the concatenation of  $l(x, \text{OutgoingCltvExpiry})$  pairs, where  
 $\text{OutgoingCltvExpiry}_l \leftarrow t, \forall i \in \{1, \dots, l-1\}, \text{OutgoingCltvExpiry}_{l-i} \leftarrow \text{OutgoingCltvExpiry}_{l-i+1} + \text{CltvExpiryDelta}_{l-i+1}$
- 19:  $(\mu_0, \delta_0) \leftarrow \text{SphinxCreate}(m, \text{public keys of } \overrightarrow{\text{path}} \text{ parties})$
- 20: let **remoteCom<sub>n</sub>** the latest signed remote commitment tx
- 21:  $\text{CltvExpiry} \leftarrow \text{OutgoingCltvExpiry}_1 + \text{relayDelay} + 2k + (2 + r)\text{windowSize} - 1$
- 22: reduce simple payment output in **remoteCom** by  $x$
- 23: add an additional  $(x, \text{ph}_{\text{htlc}, n+1} \vee (\text{ph}_{\text{htlc}, n+1} \wedge \text{pt}_{\text{htlc}, n+1}, \text{ on preimage of } h) \vee \text{ph}_{\text{htlc}, n+1}, \text{CltvExpiry absolute})$  output (all with  $n+1$  keys) to **remoteCom**, marked with **HTLCNo**
- 24: reduce delayed payment output in **localCom** by  $x$
- 25: add an additional  $(x, \text{pt}_{\text{rev}, n+1} \vee (\text{pt}_{\text{htlc}, n+1}, \text{ on preimage of } h) \vee (\text{ph}_{\text{htlc}, n+1} \wedge \text{pt}_{\text{htlc}, n+1}, \text{CltvExpiry absolute}))$  output (all with  $n+1$  keys) to **localCom**, marked with **HTLCNo**
- 26: increment **HTLCNo<sub>pchid</sub>** by one and associate  $x, h, \text{pchid}$  with it
- 27: mark **HTLCNo** as "sender"
- 28: send  $(\text{UPDATEADDHTLC}, \text{first } \text{pchid} \text{ of } \overrightarrow{\text{path}}, \text{HTLCNo}_{\text{pchid}}, x, h, \text{CltvExpiry}, (\mu_0, \delta_0))$  to  $\text{pchid}$  channel counterparty

**Fig. 11.**

**Protocol  $\Pi_{LN}$  - UPDATEADDHTLC**

- 1: Upon receiving (UPDATEADDHTLC,  $pchid$ , HTLCNo,  $x$ ,  $h$ , CltvExpiry,  $M$ ) from *Bob*:
- 2:   ensure  $pchid$  corresponds to an open channel in **channels** where *Bob* has at least  $x$
- 3:   ensure HTLCNo = HTLCNo $_{pchid}$  + 1
- 4:   ( $pchid'$ ,  $x'$ , CltvExpiry',  $\delta$ )  $\leftarrow$  SphinxPeel( $sk_{Alice}$ ,  $M$ )
- 5:   **if**  $\delta = \text{receiver}$  **then**
- 6:     ensure  $pchid' = \perp$ ,  $x = x'$ , CltvExpiry  $\geq$   
CltvExpiry' + relayDelay +  $2k + (2 + r) \text{windowSize} - 1$
- 7:     mark HTLCNo as “receiver”
- 8:   **else** // We are an intermediary
- 9:     ensure  
 $x = x'$ , CltvExpiry  $\geq$  CltvExpiry' + relayDelay +  $3k + 2(2 + r) \text{windowSize} - 1$
- 10:    ensure  $pchid'$  corresponds to an open channel in **channels** where we have at least  $x$
- 11:    mark HTLCNo as “intermediary”
- 12:   **end if**
- 13:   increment HTLCNo $_{pchid}$  by one
- 14:   let **remoteCom** $_n$  the latest signed remote commitment tx
- 15:   reduce delayed payment output in **remoteCom** by  $x$
- 16:   add an ( $x$ ,  $ph_{rev,n+1} \vee (ph_{htlc,n+1} \wedge pt_{htlc,n+1}$ , CltvExpiry absolute)  $\vee$   
 $ph_{htlc,n+1}$ , on preimage of  $h$ ) htlc output (all with  $n + 1$  keys) to **remoteCom**,  
marked with HTLCNo
- 17:   reduce simple payment output in **localCom** by  $x$
- 18:   add an ( $x$ ,  $pt_{rev,n+1} \vee pt_{htlc,n+1}$ , CltvExpiry absolute)  $\vee$   
 $((pt_{htlc,n+1} \wedge ph_{htlc,n+1}$ , on preimage of  $h$ )) htlc output (all with  $n + 1$  keys)  
to **remoteCom**, marked with HTLCNo
- 19:   **if**  $\delta = \text{receiver}$  **then**
- 20:     retrieve  $R : \mathcal{H}(R) = h$  from **pendingGetPaid** and clear entry
- 21:     add (HTLCNo,  $R$ ) to **pendingFulfills** $_{pchid}$
- 22:   **else if**  $\delta \neq \text{receiver}$  **then** // Send HTLC to next hop
- 23:     retrieve  $pchid'$  data
- 24:     let **remoteCom**' $_n$  the latest signed remote commitment tx
- 25:     reduce simple payment output in **remoteCom**' by  $x$
- 26:     add an additional ( $x$ ,  $ph_{rev,n+1} \vee (ph_{htlc,n+1} \wedge pt_{htlc,n+1}$ , on preimage  
of  $h$ )  $\vee ph_{htlc,n+1}$  CltvExpiry' absolute) output (all with  $n + 1$  keys) to  
**remoteCom**', marked with HTLCNo'
- 27:     reduce delayed payment output in **localCom**' by  $x$
- 28:     add an additional ( $x$ ,  $pt_{rev,n+1} \vee (pt_{htlc,n+1}$ , on preimage  
of  $h$ )  $\vee (pt_{htlc,n+1} \wedge ph_{htlc,n+1}$  CltvExpiry' absolute)) output (all with  $n + 1$   
keys) to **remoteCom**', marked with HTLCNo'
- 29:     increment HTLCNo' by 1
- 30:      $M' \leftarrow$  SphinxPrepare( $M$ ,  $\delta$ ,  $sk_{Alice}$ )
- 31:     add (HTLCNo',  $x$ ,  $h$ , CltvExpiry',  $M'$ ) to **pendingAdds** $_{pchid'}$
- 32:   **end if**

**Fig. 12.**

**Protocol  $\Pi_{LN}$  - UPDATEFULFILLHTLC**

```

1: Upon receiving (UPDATEFULFILLHTLC,  $pcid$ , 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. 18)
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  $pcid'$  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  $\Sigma_{\text{Alice}}$ 
13:    if latest remoteCom' $_n \in \Sigma_{\text{Alice}}$  then // counterparty has gone on-chain
14:      TX  $\leftarrow$  {input: (remoteCom' HTLC output with number HTLCNo',  $R$ ),
   output:  $pk_{\text{Alice}}$ }
15:      sig  $\leftarrow$  signature(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. 17, l. 15)
19:      send (UPDATEFULFILLHTLC,  $pcid'$ , HTLCNo',  $R$ ) to counterparty
20:    end if
21:  end if

```

**Fig. 13.**

**Protocol  $\Pi_{LN}$  - 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 **remoteCom<sub>n</sub>** in **channel**
- 4:   ensure **remoteCom**  $\neq$  **remoteCom<sub>n</sub>** // there are uncommitted updates
- 5:   ensure **channel** is not marked as “waiting for REVOKEANDACK”
- 6:   **remoteCom<sub>n+1</sub>**  $\leftarrow$  **remoteCom**
- 7:   **ComSig**  $\leftarrow$  **signature**(**remoteCom<sub>n+1</sub>**,  $sh_F$ )
- 8:   **HTLCSigs**  $\leftarrow \emptyset$
- 9:   **for**  $i$  from **lastRemoteSigned** to **HTLCNo** **do**
- 10:     **remoteHTLC<sub>n+1,i</sub>**  $\leftarrow$  TX {input: HTLC output  $i$  of **remoteCom<sub>n+1</sub>**,  
output:  
( $C_{htlc,i}, ph_{rev,n+1} \vee (pt_{dpay,n+1}, delay + k + (2 + r)windowSize \text{ relative})$ )}
- 11:     add **signature**(**remoteHTLC<sub>n+1,i</sub>**,  $sh_{htlc,n+1}$ ) to **HTLCSigs**
- 12:   **end for**
- 13:   add **signature**(**remoteHTLC<sub>n+1,m+1</sub>**,  $sh_{htlc,n+1}$ ) to **HTLCSigs**
- 14:   **lastRemoteSigned**  $\leftarrow$  **HTLCNo**
- 15:   mark **channel** as “waiting for REVOKEANDACK”
- 16:   send (COMMITMENTSIGNED,  $pchid$ , **ComSig**, **HTLCSigs**) to  $pchid$  counterparty

**Fig. 14.**

**Protocol  $\Pi_{LN}$  - COMMITMENTSIGNED**

- 1: Upon receiving (COMMITMENTSIGNED,  $pchid$ ,  $comSig_{n+1}$ ,  $HTLCSigs_{n+1}$ ) from *Bob*:
- 2:   ensure that there is a **channel**  $\in$  **channels** with ID  $pchid$  with *Bob*
- 3:   retrieve latest local commitment tx  $localCom_n$  in **channel**
- 4:   ensure  $localCom \neq localCom_n$  and  $localCom \neq pendingLocalCom$  // there are uncommitted updates
- 5:   **if**  $verify(localCom, comSig_{n+1}, pt_F) = false \vee |HTLCSigs_{n+1}| \neq HTLCNo - lastLocalSigned + 1$  **then**
- 6:     close channel (as in Fig. 18)
- 7:     **return**
- 8:   **end if**
- 9:   **for**  $i$  from  $lastLocalSigned$  to  $HTLCNo$  **do**
- 10:      $localHTLC_{n+1,i} \leftarrow TX \{input: HTLC \text{ output } i \text{ of } localCom, output: (c_{htlc,i}, ph_{rev,n+1} \vee (pt_{dpay,n+1}, remoteDelay \text{ relative}))\}$
- 11:     **if**  $verify(localHTLC_{n+1,i}, HTLCSigs_{n+1,i}, pt_{htlc,n+1}) = false$  **then**
- 12:       close channel (as in Fig. 18)
- 13:       **return**
- 14:     **end if**
- 15:   **end for**
- 16:    $pendingLocalCom \leftarrow localCom$
- 17:   mark  $pendingLocalCom$  as “irrevocably committed”
- 18:    $prand_{n+2} \leftarrow PRF(seed, n + 2)$
- 19:    $(sh_{com,n+2}, ph_{com,n+2}) \leftarrow KeyShareGen(1^k; prand_{n+2})$
- 20:   send (REVOKEANDACK,  $pchid$ ,  $prand_n$ ,  $ph_{com,n+2}$ ) to *Bob*

**Fig. 15.**



**Protocol  $\Pi_{LN}$  - REVOKEANDACK**

- 1: Upon receiving (REVOKEANDACK,  $pchid$ ,  $st_{com,n}$ ,  $pt_{com,n+2}$ ) from *Bob*:
- 2:   ensure there is a **channel**  $\in$  **channels** with *Bob* with ID  $pchid$  marked as “waiting for REVOKEANDACK”
- 3:   **if**  $pk(st_{com,n}) \neq pt_{com,n}$  **then** // wrong  $st_{com,n}$  - closing
- 4:     close channel (as in Fig. 18)
- 5:     **return**
- 6:   **end if**
- 7:   mark  $remoteCom_{n+1}$  as “irrevocably committed”
- 8:    $localCom_{n+1} \leftarrow pendingLocalCom$
- 9:   unmark **channel**
- 10:   add **receipt(channel)** to **updatesToReport**
- 11:    $sh_{rev,n} \leftarrow CombineKey(shb_{rev}, phb_{rev}, st_{com,n}, pt_{com,n})$
- 12:    $ph_{rev,n+2} \leftarrow CombinePubKey(phb_{rev}, pt_{com,n+2})$
- 13:    $pt_{rev,n+2} \leftarrow CombinePubKey(ptb_{rev}, ph_{com,n+2})$
- 14:    $ph_{dpay,n+2} \leftarrow PubKeyGen(phb_{dpay}, ph_{com,n+2})$
- 15:    $pt_{dpay,n+2} \leftarrow PubKeyGen(ptb_{dpay}, pt_{com,n+2})$
- 16:    $ph_{pay,n+2} \leftarrow PubKeyGen(phb_{pay}, ph_{com,n+2})$
- 17:    $pt_{pay,n+2} \leftarrow PubKeyGen(ptb_{pay}, pt_{com,n+2})$
- 18:    $ph_{htlc,n+2} \leftarrow PubKeyGen(phb_{htlc}, ph_{com,n+2})$
- 19:    $pt_{htlc,n+2} \leftarrow PubKeyGen(ptb_{htlc}, pt_{com,n+2})$

**Fig. 16.**

**Protocol  $\Pi_{LN}$  - 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 **pendingFulfills** $_{pchid}$  that is both in an “irrevocably committed” **remoteCom** $_n$  and **localCom** $_n$
- 4:   send (READ) to  $\mathcal{G}_{Ledger}$  and assign reply to  $\Sigma_{Alice}$
- 5:   remove (HTLCNo,  $R$ ) from **pendingFulfills** $_{pchid}$
- 6:   **if** **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: (**remoteCom** $_n$  HTLC output with number HTLCNo,  $R$ ), output:  $pk_{Alice}$ }
- 10:    sig  $\leftarrow$  **signature**(TX,  $sh_{htlc,n}$ )
- 11:    send (SUBMIT, (sig, TX)) to  $\mathcal{G}_{Ledger}$  // shouldn't be already spent by **remote HTLC****Timeout**
- 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 (HTLCNo,  $x, h, CltvExpiry, M$ ) of **pendingAdds** $_{pchid}$  that is both in an “irrevocably committed” **remoteCom** $_n$  and **localCom** $_n$
- 16:   remove chosen entry from **pendingAdds** $_{pchid}$
- 17:   send (UPDATEADDHTLC,  $pchid$ , HTLCNo,  $x, h, CltvExpiry, M$ ) to  $pchid$  counterparty
- 18: Upon receiving (FULFILLONCHAIN) from  $\mathcal{E}$ :
- 19:   send (READ) to  $\mathcal{G}_{Ledger}$  and assign largest block number to  $t$
- 20:   **toSubmit**  $\leftarrow \emptyset$
- 21:   **for all** channels **do**
- 22:     **if** there exists an HTLC in latest **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  $[t, t + k + (2 + r) \text{windowSize}]$  **then**
- 23:       add **localCom** $_n$  of the channel and all corresponding valid HTLC-successes and HTLC-timeouts (for both **localCom** $_n$  and **remoteCom** $_n$ <sup>a</sup>), along with their signatures to **toSubmit**
- 24:     **end if**
- 25:   **end for**
- 26:   send (SUBMIT, **toSubmit**) to  $\mathcal{G}_{Ledger}$

<sup>a</sup> Ensures funds retrieval if counterparty has gone on-chain

**Fig. 17.**

**Protocol  $\Pi_{LN}$  - close**

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

**Fig. 18.**

## 6 Payment Network Functionality

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

Interface: **check**

- from  $\mathcal{E}$ :
  - (REGISTER, delay, relayDelay)
  - (REGISTERED)
  - (OPENCHANNEL, *Alice*, *Bob*, *x*, *tid*)
  - (CHECKFORNEW, *Alice*, *Bob*, *tid*)
  - (PAY, *Bob*, *x*,  $\overrightarrow{\text{path}}$ , **receipt**)
  - (CLOSECHANNEL, **receipt**)
  - (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*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *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*,  $\overrightarrow{\text{path}}$ , **receipt**, *payid*)
  - (CONTINUE)
  - (CLOSECHANNEL, *fchid*, *Alice*)
  - (POLL,  $\Sigma_{\text{Alice}}$ , *Alice*)
  - (PUSHFULFILL, *pchid*, *Alice*)
  - (PUSHADD, *pchid*, *Alice*)
  - (COMMIT, *pchid*, *Alice*)
  - (FULFILLONCHAIN, *t*, *Alice*)

Fig. 19.

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

- 1: Initialisation:
- 2:    **channels**, **pendingPay**, **pendingOpen**, **corrupted**,  $\Sigma \leftarrow \emptyset$
- 3: Upon receiving (REGISTER, delay, relayDelay) from *Alice*:
- 4:    **delay**(*Alice*)  $\leftarrow$  delay // Must check chain at least once every **delay**(*Alice*) blocks
- 5:    **relayDelay**(*Alice*)  $\leftarrow$  relayDelay
- 6:    **updatesToReport**(*Alice*), **newChannels**(*Alice*)  $\leftarrow \emptyset$
- 7:    **polls**(*Alice*)  $\leftarrow \emptyset$
- 8:    **focs**(*Alice*)  $\leftarrow \emptyset$
- 9:    send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice*, store reply to  $\Sigma_{\text{Alice}}$ , add  $\Sigma_{\text{Alice}}$  to  $\Sigma$  and add largest block number to **polls**(*Alice*)
- 10:    **checkClosed**( $\Sigma_{\text{Alice}}$ )
- 11:    send (REGISTER, *Alice*, delay, relayDelay, lastPoll) to  $\mathcal{S}$
- 12: Upon receiving (REGISTERDONE, *Alice*, pubKey) from  $\mathcal{S}$ :
- 13:    **pubKey**(*Alice*)  $\leftarrow$  pubKey
- 14:    send (REGISTER, *Alice*, **delay**(*Alice*), **relayDelay**(*Alice*), pubKey) to *Alice*
- 15: Upon receiving (REGISTERED) from *Alice*:
- 16:    send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and store reply to  $\Sigma_{\text{Alice}}$
- 17:    **checkClosed**( $\Sigma_{\text{Alice}}$ )
- 18:    assign the sum of all output values that are exclusively spendable by *Alice* to **onChainBalance**
- 19:    send (REGISTERED) to *Alice*
- 20: Upon receiving any message except for (REGISTER) from *Alice*:
- 21:    ignore message if *Alice* has not registered
- 22: Upon receiving (CORRUPTED, *Alice*) from  $\mathcal{S}$ :
- 23:    add *Alice* to **corrupted**
- 24:    for the rest of the execution, upon receiving any message for *Alice*, bypass normal execution and simply forward it to  $\mathcal{S}$

**Fig. 20.**

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

- 1: Upon receiving (OPENCHANNEL, *Alice*, *Bob*, *x*, *tid*) from *Alice*:
- 2:   ensure *tid* hasn't been used by *Alice* for opening another channel before
- 3:   choose unique channel ID *fchid*
- 4:   **pendingOpen**(*fchid*)  $\leftarrow$  (*Alice*, *Bob*, *x*, *tid*)
- 5:   send (OPENCHANNEL, *Alice*, *Bob*, *x*, *fchid*, *tid*) to  $\mathcal{S}$
  
- 6: Upon receiving (CHANNELANNOUNCED, *Alice*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *fchid*, *pchid*, *tid*) from  $\mathcal{S}$ :
- 7:   ensure that there is a **pendingOpen**(*fchid*) entry with temporary id *tid*
- 8:   add "*Alice* announced",  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *pchid* to **pendingOpen**(*fchid*)
  
- 9: Upon receiving (CHECKFORNEW, *Alice*, *Bob*, *tid*) from *Alice*:
- 10:   ensure there is a matching **channel** in **pendingOpen**(*fchid*), marked with "*Alice* announced"
- 11:   (*funder*, *fundee*, *x*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ )  $\leftarrow$  **pendingOpen**(*fchid*)
- 12:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and store reply to  $\Sigma_{\text{Alice}}$
- 13:   **checkClosed**( $\Sigma_{\text{Alice}}$ )
- 14:   ensure that there is a TX  $F \in \Sigma_{\text{Alice}}$  with a (*x*, ( $p_{\text{funder},F} \wedge p_{\text{fundee},F}$ )) output
- 15:   mark **channel** with "waiting for FUNDINGLOCKED"
- 16:   send (FUNDINGLOCKED, *Alice*,  $\Sigma_{\text{Alice}}$ , *fchid*) to  $\mathcal{S}$
  
- 17: Upon receiving (FUNDINGLOCKED, *fchid*) from  $\mathcal{S}$ :
- 18:   ensure a **channel** is in **pendingOpen**(*fchid*), marked with "waiting for FUNDINGLOCKED" and replace mark with "waiting for CHANNELOPENED"
- 19:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Bob* and store reply to  $\Sigma_{\text{Bob}}$
- 20:   **checkClosed**( $\Sigma_{\text{Bob}}$ )
- 21:   ensure that there is a TX  $F \in \Sigma_{\text{Bob}}$  with a (*x*, ( $p_{\text{funder},F} \wedge p_{\text{fundee},F}$ )) output
- 22:   add **receipt**(**channel**) to **newChannels**(*Bob*)
- 23:   send (FUNDINGLOCKED, *Bob*,  $\Sigma_{\text{Bob}}$ , *fchid*) to  $\mathcal{S}$
  
- 24: Upon receiving (CHANNELOPENED, *fchid*) from  $\mathcal{S}$ :
- 25:   ensure a **channel** is in **pendingOpen**(*fchid*), marked with "waiting for CHANNELOPENED" and remove mark
- 26:   offChainBalance(*funder*)  $\leftarrow$  offChainBalance(*funder*) + *x* **[Orfeas: remove on/offChainBalance?]**
- 27:   onChainBalance(*funder*)  $\leftarrow$  offChainBalance(*funder*) - *x*
- 28:   **channel**  $\leftarrow$  (*funder*, *fundee*, *x*, 0, 0, *fchid*, *pchid*)
- 29:   add **channel** to **channels**
- 30:   add **receipt**(**channel**) to **newChannels**(*Alice*)
- 31:   clear **pendingOpen**(*fchid*) entry

**Fig. 21.**

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

- 1: Upon receiving  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}})$  from *Alice*:
- 2:     choose unique payment ID *payid*
- 3:     add  $(\text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{payid})$  to **pendingPay**
- 4:     send  $(\text{PAY}, \text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{payid}, \text{STATE}, \Sigma)$  to  $\mathcal{S}$
  
- 5: Upon receiving  $(\text{UPDATE}, \text{receipt}, \text{Alice})$  from  $\mathcal{S}$ :
- 6:     add **receipt** to **updatesToReport**(*Alice*) // trust  $\mathcal{S}$  here, check on  
      RESOLVEPAYS
- 7:     send (CONTINUE) to  $\mathcal{S}$

**Fig. 22.**

**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 Charlie keys  $\in$  charged do
3:     for all (Dave, payid)  $\in$  charged(Charlie) do
4:       retrieve (Alice, Bob, x,  $\overrightarrow{\text{path}}$ ) with ID payid and remove it from
   pendingPay
5:       if Dave =  $\perp$  then
6:         return
7:       end if
8:       calculate IncomingCltvExpiry, OutgoingCltvExpiry of Dave (as in
   Fig. 11, l. 18)
9:       if Dave  $\neq$  Alice  $\wedge$  Dave  $\notin$  corrupted  $\wedge$  (( $\Sigma_{\text{Dave}}$  contains in block  $h_{\text{tx}}$ 
   an old remoteComm that does not contain the HTLC and a tx that spends the
   delayed output of remoteComm  $\wedge$  polls(Dave) contains an element in
   [ $h_{\text{tx}} + k, h_{\text{tx}} + k + \text{delay}(\text{Dave}) - 1$ ])  $\vee$  ( $\Sigma_{\text{Dave}}$  does not contain such a
   remoteComm  $\wedge$  polls(Dave) contains two elements in [ $\text{OutgoingCltvExpiry} +$ 
    $k + (2 + r)\text{windowSize} + 1, \text{IncomingCltvExpiry} - k - (2 + r)\text{windowSize}$ ]
   that have a difference of at least  $k + (2 + r)\text{windowSize} \wedge$  focs(Dave)
   contains  $\text{IncomingCltvExpiry} - k - (2 + r)\text{windowSize} \wedge$  the element in
   polls(Dave) was added before the element in focs(Dave))) then
10:        halt
11:      end if
12:      for all channels  $\in \overrightarrow{\text{path}}$  starting from the one where Dave pays do
13:        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
14:        if payer has x or more in channel then
15:          update channel to the next version and transfer x from
   payer to payee
16:        else
17:          revert all updates done in this loop
18:        end if
19:      end for
20:      if Dave  $\notin$  corrupted then
21:         $\text{offChainBalance}(\text{Dave}) \leftarrow \text{offChainBalance}(\text{Dave}) - x$ 
22:      end if
23:       $\text{offChainBalance}(\text{Bob}) \leftarrow \text{offChainBalance}(\text{Bob}) + x$ 
24:    end for
25:  end for

```

**Fig. 23.**



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

```

1: Upon receiving (CLOSECHANNEL, receipt) from Alice
2:   ensure that there is a channel  $\in$  channels : receipt(channel) = receipt
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, Alice) to  $\mathcal{S}$ 

7: function checkClosed( $\Sigma_{\text{Alice}}$ ) // Called after every (READ), ensures requested
   closes eventually happen
8:   for all entries (fchid, receipt, h)  $\in$  pendingClose(Alice) do
9:     if there is a closing transaction in  $\Sigma_{\text{Alice}}$  for open channel with ID fchid
       with a balance that corresponds to receipt then
10:      let x, y the balances of Alice and the channel counterparty Bob
       respectively
11:      offChainBalance(Alice)  $\leftarrow$  offChainBalance(Alice) + x
12:      onChainBalance(Alice)  $\leftarrow$  onChainBalance(Alice) - x
13:      offChainBalance(Bob)  $\leftarrow$  offChainBalance(Bob) + y
14:      onChainBalance(Bob)  $\leftarrow$  onChainBalance(Bob) - y
15:      remove channel from channels
16:      remove entry from pendingClose(Alice)
17:      if there is an (fchid,  $\_$ ,  $\_$ ) entry in pendingClose(Bob) then
18:        remove it from pendingClose(Bob)
19:      end if
20:    else if there is a closing transaction in block of height h in  $\Sigma_{\text{Alice}}$  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
21:      if polls(Alice) contains an entry in  $[h + k, h + k + \text{delay}(\text{Alice}) - 1]$ 
       then
22:        halt
23:      end if
24:    else if there is no such closing transaction  $\wedge h = \perp$  then
25:      assign largest block number of  $\Sigma_{\text{Alice}}$  to h of entry
26:    else if there is no such closing transaction  $\wedge h \neq \perp \wedge$  (largest block
       number of  $\Sigma_{\text{Alice}} \geq h + (2 + r) \text{windowSize}$ ) then
27:      halt
28:    end if
29:  end for
30: end function

```

**Fig. 24.**

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

- 1: Upon receiving (POLL) from *Alice*:
- 2:     send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and store reply to  $\Sigma_{\text{Alice}}$
- 3:     add largest block number in  $\Sigma_{\text{Alice}}$  to  $\text{polls}(\text{Alice})$
- 4:     **checkClosed**( $\Sigma_{\text{Alice}}$ )
- 5:     **for all**  $\text{channels} \in \Sigma_{\text{Alice}}$  that contain *alice* and are maliciously closed by a remote commitment  $\text{tx}$  (one with an older channel version than the irrevocably committed one) in block with height  $h_{\text{tx}}$  **do**
- 6:         **if** the delayed output (of the counterparty) has been spent AND  $\text{polls}(\text{Alice})$  has an element in  $[h_{\text{tx}} + k, h_{\text{tx}} + k + \text{delay}(\text{Alice}) - 1]$  **then**
- 7:             halt // *Alice* wasn't negligent but couldn't punish - bad event
- 8:         **end if**
- 9:     **end for**
- 10:     send (POLL,  $\Sigma_{\text{Alice}}$ , *Alice*) to  $\mathcal{S}$

**Fig. 25.**

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

- 1: Upon receiving (PUSHFULFILL,  $pchid$ ) from *Alice*:
- 2:     send (PUSHFULFILL,  $pchid$ , *Alice*, STATE,  $\Sigma$ ) to  $\mathcal{S}$
  
- 3: Upon receiving (PUSHADD,  $pchid$ ) from *Alice*:
- 4:     send (PUSHADD,  $pchid$ , *Alice*, STATE,  $\Sigma$ ) to  $\mathcal{S}$
  
- 5: Upon receiving (COMMIT,  $pchid$ ) from *Alice*:
- 6:     send (COMMIT,  $pchid$ , *Alice*, STATE,  $\Sigma$ ) to  $\mathcal{S}$
  
- 7: Upon receiving (FULFILLONCHAIN) from *Alice*:
- 8:     send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice*, store reply to  $\Sigma_{\text{Alice}}$  and assign largest block number to  $t$
- 9:     **checkClosed**( $\Sigma_{\text{Alice}}$ )
- 10:    add  $t$  to **focs**(*Alice*)
- 11:    send (FULFILLONCHAIN,  $t$ , *Alice*) to  $\mathcal{S}$
  
- 12: Upon receiving (CLOSEDCHANNEL, **channel**, *Alice*) from  $\mathcal{S}$ :
- 13:    add **channel** to **closedChannels**(*Alice*) // trust  $\mathcal{S}$
- 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

**Fig. 26.**

## 7 Security Proof

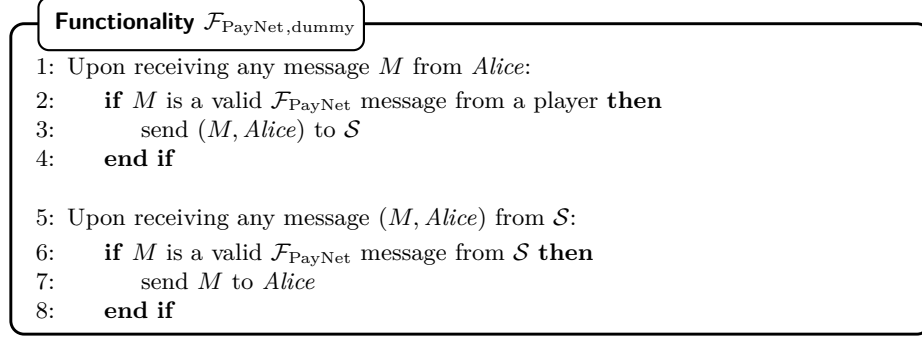


Fig. 27.

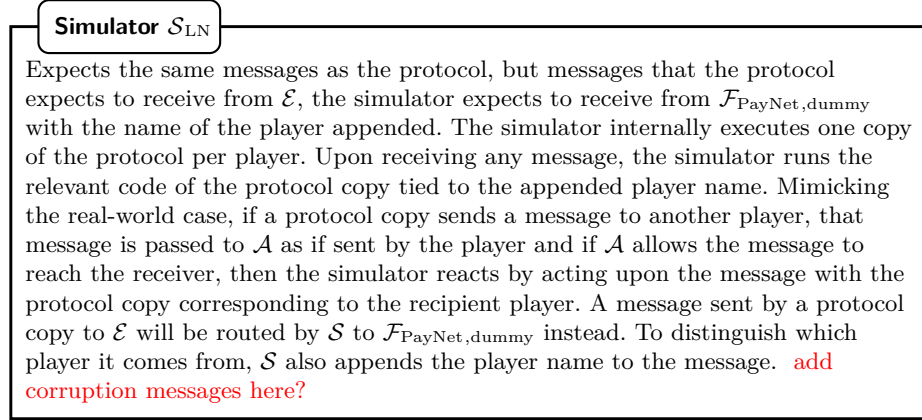
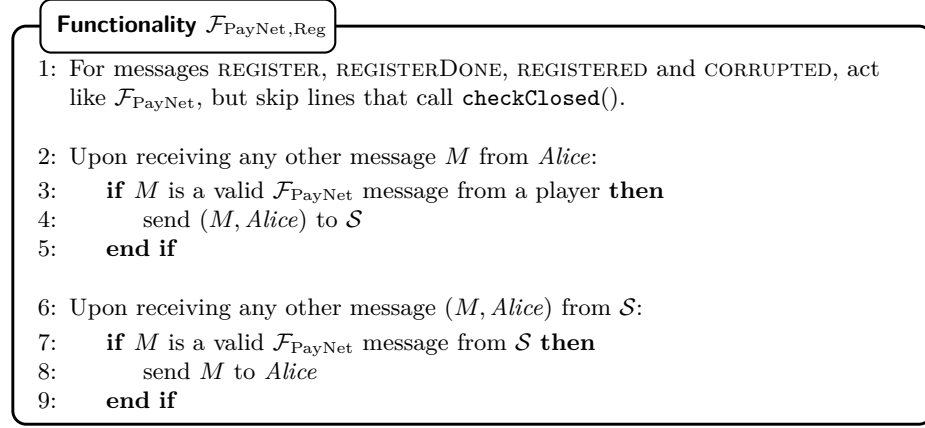


Fig. 28.

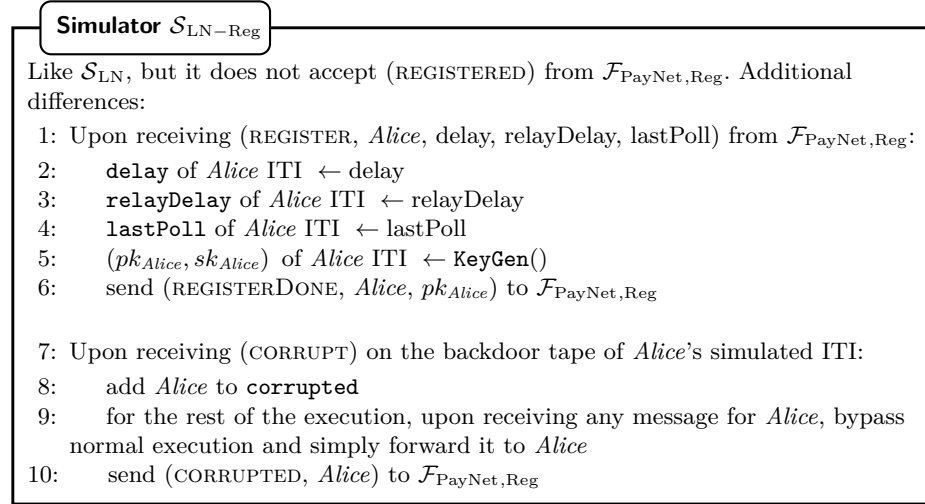
**Lemma 1.**  $\text{EXEC}_{\mathcal{H}_{\text{LN}}, \mathcal{A}_d, \mathcal{E}}^{\mathcal{G}_{\text{Ledger}}} = \text{EXEC}_{\mathcal{S}_{\text{LN}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet}, \text{dummy}}, \mathcal{G}_{\text{Ledger}}}$

*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  $\mathcal{F}_{\text{PayNet}, \text{dummy}}$ . 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.  $\square$



**Fig. 29.**



**Fig. 30.**

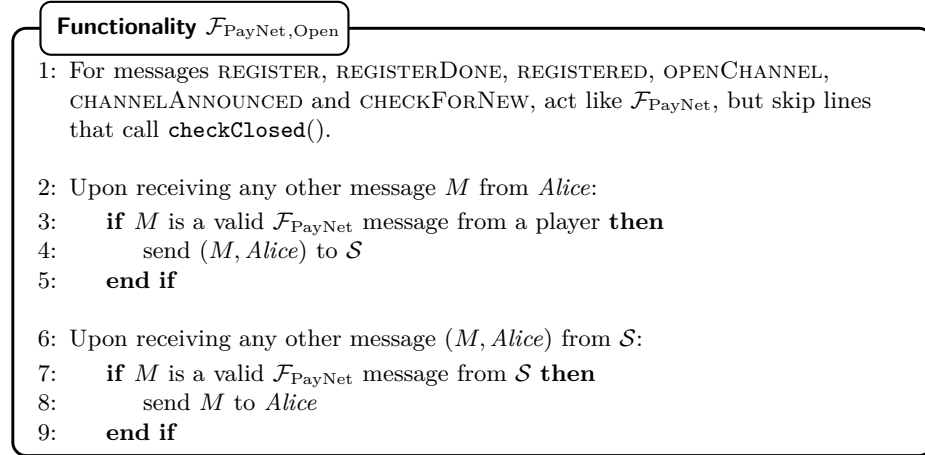
**Lemma 2.**  $\text{EXEC}_{\mathcal{S}_{\text{LN}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet,dummy}}, \mathcal{G}_{\text{Ledger}}} = \text{EXEC}_{\mathcal{S}_{\text{LN-Reg}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet,Reg}}, \mathcal{G}_{\text{Ledger}}}$

*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. 1, line 9) and in the ideal world (c.f. Fig. 30, line 5).

Furthermore, one (READ) is sent to  $\mathcal{G}_{\text{Ledger}}$  from *Alice* in both cases (Fig. 1, line 8 and Fig. 20, 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. 1 change the state of *Alice* ITI in the same way as lines 2-5 of Fig. 30.

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 1 ensures that the rest of the messages are responded in an indistinguishable way in both worlds.  $\square$



**Fig. 31.**

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

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

- 1: Upon receiving (OPENCHANNEL, *Alice*, *Bob*, *x*, *fchid*, *tid*) from  $\mathcal{F}_{\text{PayNet,Open}}$ :
- 2:   **if** both *Alice* and *Bob* are honest **then**
- 3:       Simulate the interaction between *Alice* and *Bob* in their respective ITI, as defined in Figures 2-6. All messages should be handed to and received from  $\mathcal{A}$ , as in the real world execution.
- 4:       After sending (FUNDINGSIGNED) as *Bob* to *Alice*, send (CHANNELANNOUNCED, *Bob*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *fchid*, *pchid*, *tid*) to  $\mathcal{F}_{\text{PayNet,Open}}$ .
- 5:       After submitting *F* to  $\mathcal{G}_{\text{Ledger}}$  as *Alice*, send (CHANNELANNOUNCED, *Alice*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *fchid*, *pchid*) to  $\mathcal{F}_{\text{PayNet,Open}}$ .
- 6:       **else if** *Alice* is honest, *Bob* is corrupted **then**
- 7:       Simulate *Alice*'s part of the interaction between *Alice* and *Bob* in *Alice*'s ITI, as defined in Figures 2, 4, and 6. All messages should be handed to and received from  $\mathcal{A}$ , as in the real world execution.
- 8:       After submitting *F* to  $\mathcal{G}_{\text{Ledger}}$  as *Alice*, send (CHANNELANNOUNCED, *Alice*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *fchid*, *pchid*) to  $\mathcal{F}_{\text{PayNet,Open}}$ .
- 9:       **else if** *Alice* is corrupted, *Bob* is honest **then**
- 10:       send (OPENCHANNEL, *Alice*, *Bob*, *x*, *fchid*, *tid*) to simulated (corrupted) *Alice*
- 11:       Simulate *Bob*'s part of the interaction between *Alice* and *Bob* in *Bob*'s ITI, as defined in Figures 3 and 5. All messages should be handed to and received from  $\mathcal{A}$ , as in the real world execution.
- 12:       After sending (FUNDINGSIGNED) as *Bob* to *Alice*, send (CHANNELANNOUNCED, *Bob*,  $p_{\text{Alice},F}$ ,  $p_{\text{Bob},F}$ , *fchid*, *pchid*) to  $\mathcal{F}_{\text{PayNet,Open}}$ .
- 13:       **else if** both *Alice* and *Bob* are corrupted **then**
- 14:       forward message to  $\mathcal{A}$  //  $\mathcal{A}$  may open the channel or not
- 15:       **end if**
- 16: Upon receiving (FUNDINGLOCKED, *Alice*,  $\Sigma_{\text{Alice}}$ , *fchid*) from  $\mathcal{F}_{\text{PayNet,Open}}$ :
- 17:   execute lines 5-9 of Fig. 7 with *Alice*'s ITI, using  $\Sigma_{\text{Alice}}$  from message
- 18:   **if** *Bob* is honest **then**
- 19:       expect the delivery of *Alice*'s (FUNDINGLOCKED) message from  $\mathcal{A}$
- 20:       send (FUNDINGLOCKED, *fchid*) to  $\mathcal{F}_{\text{PayNet,Open}}$
- 21:       upon receiving (FUNDINGLOCKED, *Bob*,  $\Sigma_{\text{Bob}}$ , *fchid*) from  $\mathcal{F}_{\text{PayNet,Open}}$ :
- 22:       simulate Fig. 8 with message from *Alice* in *Bob*'s ITI, using  $\Sigma_{\text{Bob}}$  from  $\mathcal{F}_{\text{PayNet,Open}}$ 's message
- 23:       **end if**
- 24: Upon receiving the (FUNDINGLOCKED) message with the simulated *Alice* ITI:
- 25:   simulate Fig. 8 receiving the message with *Alice*'s ITI
- 26:   send (CHANNELOPENED, *fchid*) to  $\mathcal{F}_{\text{PayNet,Open}}$

**Fig. 32.**

**Lemma 3.**  $\text{EXEC}_{\mathcal{S}_{\text{LN-Reg}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet,Reg}}, \mathcal{G}_{\text{Ledger}}} = \text{EXEC}_{\mathcal{S}_{\text{LN-Reg-Open}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet,Open}}, \mathcal{G}_{\text{Ledger}}}$

*Proof.* When  $\mathcal{E}$  sends (OPENCHANNEL, *Alice*, *Bob*, *x*, *fchid*, *tid*) to *Alice*, the interaction of Figures 2-6 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  $\mathcal{S}_{\text{LN-Reg-Open}}$ , so their state will be identical to that of the parties in the real execution. Furthermore, since  $\mathcal{S}_{\text{LN-Reg-Open}}$  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. 21, line 2) filters only messages that would be ignored by the real protocol as well and would not change its state either (Fig. 2, line 2).

We also observe that, upon receiving OPENCHANNEL or CHANNELANNOUNCED,  $\mathcal{F}_{\text{PayNet,Open}}$  does not send any messages to parties other than  $\mathcal{S}_{\text{LN-Reg-Open}}$ , 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. 7 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. 2, line 7 or in Fig. 3, 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. 5, line 11) (which needs a valid *Alice-Bob* interaction up to that point **more in-depth?**), and in the second case the “broadcast” mark can be added only before *Alice* sends (SUBMIT, (sig, *F*)) to  $\mathcal{G}_{\text{Ledger}}$  (Fig. 6, line 8) (which also needs a valid *Alice-Bob* interaction up to that point **more in-depth?**)

When  $\mathcal{E}$  sends (CHECKFORNEW, *Alice*, *Bob*, *tid*) to *Alice* in the ideal world, line 10 of Fig. 21 will allow execution to continue if there exists an entry with temporary id *tid* and member *Alice* marked as “*Alice* announced” in **pendingOpen**(*fchid*) for some *fchid*. This can only happen if line 8 of Fig. 21 is executed, where **pendingOpen**(*fchid*) contains *tid* as temporary id. This line in turn can only be executed if  $\mathcal{F}_{\text{PayNet,Open}}$  received (CHANNELANNOUNCED, *Alice*, *pAlice,F*, *pBob,F*, *fchid*, *pchid*, *tid*) from  $\mathcal{S}_{\text{LN-Reg-Open}}$  such that **pendingOpen**(*fchid*) exists and has temporary id *tid*, as mandated by line 7 of Fig. 21. Such a message is sent by  $\mathcal{S}_{\text{LN-Reg-Open}}$  of Fig. 32 either in lines 5/8, or in lines 4/12. One of the first pair of lines is executed only if  $\mathcal{S}_{\text{LN-Reg-Open}}$  receives (OPENCHANNEL,



$Alice, Bob, x, fchid, tid$ ) from  $\mathcal{F}_{\text{PayNet,Open}}$  and the simulated  $\mathcal{A}$  allows a valid *Alice-Bob* interaction up to the point where *Alice* sends (SUBMIT) to  $\mathcal{G}_{\text{Ledger}}$ , whereas one of the second pair of lines is executed only if  $\mathcal{S}_{\text{LN-Reg-Open}}$  receives (OPENCHANNEL, *Bob, Alice, x, fchid, tid*) from  $\mathcal{F}_{\text{PayNet,Open}}$  and the simulated  $\mathcal{A}$  allows a valid *Alice-Bob* interaction up to the point where *Alice* sends (FUNDINGSIGNED) to *Bob*.

The last two points lead us to deduce that line 10 of Fig. 21 in the ideal and line 2 of Fig. 7 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. 21, lines ??-14) and the real world (Fig. 7, lines 3-4).

Moving on, in the real world lines 5-9 of Fig. 7 are executed by *Alice* and, given that  $\mathcal{A}$  allows it, the code of Fig. 8 is executed by *Bob*. Likewise, in the ideal world, the functionality executes lines ??-16 and as a result it (always) sends (CHANNELOPENED, *Alice, fchid*) to  $\mathcal{S}_{\text{LN-Reg-Open}}$ . In turn  $\mathcal{S}_{\text{LN-Reg-Open}}$  simulates lines 5-9 of Fig. 7 with *Alice*'s ITI and, if  $\mathcal{A}$  allows it,  $\mathcal{S}_{\text{LN-Reg-Open}}$  simulates the code of Fig. 8 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.  $\square$

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

- 1: For messages REGISTER, REGISTERDONE, REGISTERED, 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. 33.**

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

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

- 1: Upon receiving  $(\text{FULFILLONCHAIN}, t, \text{Alice})$  from  $\mathcal{F}_{\text{PayNet, Pay}}$ :
- 2:     execute lines 20-26 of Fig. 17 as *Alice*, using  $t$  from message
- 3: Upon receiving  $(\text{PAY}, \text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt}, \text{payid})$  from  $\mathcal{F}_{\text{PayNet, Pay}}$ :
- 4:     add  $(\overrightarrow{\text{path}}, \text{payid})$  to **payids**
- 5:     strip  $\text{payid}$ , simulate receiving the message with *Alice* ITI and further execute the parts of  $\Pi_{\text{LN}}$  that correspond to honest parties (Fig. 11- Fig. 13)
- 6:     **if** any “ensure” in  $\Pi_{\text{LN}}$  fails until *Bob* processes **UPDATEADDHTLC** **then** // payment failed
- 7:         add  $(\perp, \text{payid})$  to **charged**(*Alice*)
- 8:         remove  $(\overrightarrow{\text{path}}, \text{payid})$  from **payids**
- 9:     **end if**
- 10: Upon receiving  $(\text{POLL}, \Sigma_{\text{Alice}}, \text{Alice})$  from  $\mathcal{F}_{\text{PayNet, Pay}}$ :
- 11:     simulate Fig. 9, lines 3-28 receiving (POLL), using  $\Sigma_{\text{Alice}}$  from the message, with *Alice*’s ITI

**Fig. 34.**

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

- 1: Upon receiving (PUSHFULFILL,  $pchid$ ,  $Alice$ ) from  $\mathcal{F}_{\text{PayNet, Pay}}$ :
- 2:     simulate Fig. 17, 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. 17, 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. 14 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. 16 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. 16, line 10)
- 9:     upon receiving (CONTINUE) from  $\mathcal{F}_{\text{PayNet, Pay}}$ , carry on with the simulation
- 10:    **end if**

**Fig. 35.**

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

```

1: Upon receiving any message with a concatenated (STATE,  $\Sigma$ ) part from
    $\mathcal{F}_{\text{PayNet, Pay}}$ : // 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
       $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$  then
4:     for all  $\Sigma_{\text{Alice}} \in \Sigma$  do
5:       for all  $(\overrightarrow{\text{path}}, \text{payid}) \in \text{payids} : \text{Alice} \in \overrightarrow{\text{path}}$  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  $\Sigma_{\text{Alice}}$ )), AND the next honest player
           Bob down the line successfully timed out the HTLC on-chain (i.e.
           HTLC-timeout is in  $\Sigma_{\text{Bob}}$ ) 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  $(\overrightarrow{\text{path}}, \text{payid})$  from payids
9:           else if  $\Sigma_{\text{Alice}}$  contains an old remoteComm of the channel before
           Alice (closer to payer) on the  $\overrightarrow{\text{path}}$  that does not contain the relevant HTLC
           and a tx that spends the delayed output of remoteComm  $\vee$  ( $\Sigma_{\text{Alice}}$  contains the
           most recent remoteComn or localComn of the channel before Alice and the
           HTLC-success of the relevant HTLC  $\vee$  Alice's latest irrevocably committed
           remoteComn for the channel before Alice does not contain the HTLC)  $\wedge \Sigma_{\text{Alice}}$ 
           contains the most recent remoteComl or localComl 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  $(\overrightarrow{\text{path}}, \text{payid})$  from payids
12:            else if Alice is the payer in  $\overrightarrow{\text{path}}$  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  $\Sigma_{\text{Alice}}$ ) then // honest payment
           completed
13:              add (Alice, payid) to charged(Alice)
14:              remove  $(\overrightarrow{\text{path}}, \text{payid})$  from payids
15:            end if
16:          end for
17:        end for
18:      end if
19:      clear charged and send (RESOLVEPAYS, charged) to  $\mathcal{F}_{\text{PayNet, Pay}}$ 

```

**Fig. 36.**

**Lemma 4.**  $\text{EXEC}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet, Open}}, \mathcal{G}_{\text{Ledger}}} \stackrel{c}{\approx} \text{EXEC}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet, Pay}}, \mathcal{G}_{\text{Ledger}}}$

*Proof.* When  $\mathcal{E}$  sends  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt})$  to *Alice* in the ideal world,  $\mathcal{S}_{\text{LN-Reg-Open}}$  is always notified (Fig. 22, line 4) and simulates the relevant execution of the real world (Fig. 34, 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  $(\text{PUSHADD}, pchid)$ ,  $(\text{PUSHFULFILL}, pchid)$ ,  $(\text{COMMIT}, pchid)$  to *Alice* in the ideal world, it is forwarded to  $\mathcal{S}_{\text{LN-Reg-Open}}$  (Fig. 26, 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. 35, lines 2, 4, 6). Neither  $\mathcal{F}_{\text{PayNet,Pay}}$  nor  $\mathcal{S}_{\text{LN-Reg-Open}}$  alter their state as a result of these messages, apart from the state of *Alice*'s simulated ITI and the state of other simulated ITIs that receive and handle messages that were sent as a result of *Alice*'s ITI simulation. The states of these ITIs are modified in the exact same way as they would in the real world. We deduce that these three messages do not introduce any opportunity for  $\mathcal{E}$  to distinguish the real and the ideal world.

When  $\mathcal{E}$  sends  $(\text{FULFILLONCHAIN})$  to *Alice* in the real world, lines 18-26 of Fig. 17 are executed by *Alice*. In the ideal world on the other hand,  $\mathcal{F}_{\text{PayNet,Pay}}$  sends  $(\text{READ})$  to  $\mathcal{G}_{\text{Ledger}}$  (Fig. 26, line 9) as *Alice* and subsequently lets  $\mathcal{S}_{\text{LN-Reg-Open}}$  simulate *Alice*'s ITI receiving  $(\text{FULFILLONCHAIN})$  (Fig. 34, lines 1-2). Observe that during this simulation a second  $(\text{READ})$  message to  $\mathcal{G}_{\text{Ledger}}$  (that would not match any message in the real world) is avoided because  $\mathcal{S}_{\text{LN-Reg-Open}}$  skips line 19 of Fig. 17, using as  $t$  the one received from  $\mathcal{F}_{\text{PayNet,Pay}}$  in the message  $(\text{FULFILLONCHAIN}, t, \text{Alice})$ . Since  $\mathcal{F}_{\text{PayNet,Pay}}$  sends  $(\text{READ})$  to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and given that after  $\mathcal{G}_{\text{Ledger}}$  replies, control is given directly to  $\mathcal{S}_{\text{LN-Reg-Open}}$ , 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 distinguishability.

When  $\mathcal{E}$  sends  $(\text{POLL})$  to *Alice*, the first action is sending  $(\text{READ})$  as *Alice* to  $\mathcal{G}_{\text{Ledger}}$  both in the ideal (Fig. 25, line 4) and the real (Fig. 9, line 2) worlds. Subsequently, in the real world lines 3-28 of Fig. 9 are executed by *Alice*, whereas in the ideal world, given that the check of line 6 does not lead to a bad event (and thus given that the functionality does not halt in line 7), a  $(\text{POLL})$  message is sent to  $\mathcal{S}_{\text{LN-Reg-Open}}$ . We will prove later that  $\mathcal{F}_{\text{PayNet,Pay}}$  halts here only with negligible probability.

Upon receiving (POLL),  $\mathcal{S}_{\text{LN-Reg-Open}}$  simulates receiving (POLL) with *Alice*'s ITI (Fig. 34, line 11), but does not READ from  $\mathcal{G}_{\text{Ledger}}$  and uses instead the  $\Sigma_{\text{Alice}}$  provided along with the message. A reasoning identical to that found in the previous paragraph shows that this  $\Sigma_{\text{Alice}}$  is exactly the same as that which *Alice*'s ITI would obtain had it executed line 2 of Fig. 9 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. 9.

Let  $E$  the “bad” event in which  $\mathcal{F}_{\text{PayNet,Pay}}$  executes line 7 of Fig. 25 and halts. We will now prove that, during  $\text{EXEC}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet,Pay}}, \mathcal{G}_{\text{Ledger}}}$ , it is  $\Pr[E] \leq \text{negl.}$  The condition of Fig. 25, line 6 is triggered if the delayed output (that of the malicious party) of  $\text{tx}_1$  has been spent by the transaction  $\text{tx}_2$  in  $\Sigma_{\text{Alice}}$  (event  $E_1$ ) and  $\text{polls}(\text{Alice})$  contains an element in  $[h_1 + k, h_1 + k + \text{delay}(\text{Alice}) - 1]$ , where  $h_1$  is the block height where  $\text{tx}_1$  is (event  $E_2$ ). Observe that  $E = E_1 \wedge E_2$ . We note that the elements in  $\text{polls}(\text{Alice})$  correspond to the block heights of  $\Sigma_{\text{Alice}}$  at the moments when *Alice* POLLS (Fig. 25, line 3). Consider the following two events:  $E_{1,1} : \text{tx}_2$  spends the delayed output with a signature valid by the delayed payment public key after the locktime expires.  $E_{1,2} : \text{tx}_2$  spends the delayed output with a signature valid by the revocation public key  $p_{\text{Alice,rev}}$ . 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, since only *Alice* knows the revocation secret key  $s_{\text{Alice,rev}}$ , forgery of the signature required in  $E_{1,2}$  can happen only with negligible probability, i.e.  $\Pr[E_{1,2}] \leq \text{negl.}$  We now concentrate on the event  $E_{1,1}$ . Due to the fact that  $\text{tx}_2$  spends an output locked with a relative timelock of length  $\text{delay}(\text{Alice}) + k + (2 + r)\text{windowSize}$ , the commitment transaction  $\text{tx}_1$  can reside in a block of maximum height  $h_1 \leq h_2 - \text{delay}(\text{Alice}) - k - (2 + r)\text{windowSize}$ , where  $h_2$  is the block height where  $\text{tx}_2$  is. If *Alice* POLLS on a moment when  $|\Sigma_{\text{Alice}}| \geq h_1 + k$ ,  $\Sigma_{\text{Alice}}$  necessarily contains  $\text{tx}_1$ . Furthermore, if *Alice* POLLS on a moment when  $|\Sigma_{\text{Alice}}| \leq h_1 + k + \text{delay}(\text{Alice}) - 1 \leq h_2 - (2 + r)\text{windowSize} - 1$ , she sees  $\text{tx}_1$  and directly submits the punishment transaction  $\text{tx}_3$  (which she has, given that a maliciously closed channel is defined as one where the non-closing party has the punishment transaction) (Fig. 10, lines 19-21). Given that  $\text{tx}_3$  is broadcast when  $|\Sigma_{\text{Alice}}| \leq h_2 - (2 + r)\text{windowSize} - 1$ , it is guaranteed to be on-chain in a block  $h_3 \leq h_2 - 1$  (according to Proposition 1). Since  $\text{tx}_3$  spends the same funds as  $\text{tx}_2$ , the two cannot be part of the chain simultaneously. Since  $E_{1,1} \Rightarrow \Sigma_{\text{Alice}}$  contains  $\text{tx}_2$  and  $E_2 \Rightarrow \Sigma_{\text{Alice}}$  contains  $\text{tx}_3$ ,  $E_{1,1}$  and  $E_2$  are mutually exclusive. There-

fore  $\Pr[E] = \Pr[(E_{1,1} \vee E_{1,2}) \wedge E_2] = \Pr[(E_{1,1} \wedge E_2) \vee (E_{1,2} \wedge E_2)] \leq \Pr[E_{1,1} \wedge E_2] + \Pr[E_{1,2} \wedge E_2] = \Pr[E_{1,2} \wedge E_2] \leq \Pr[E_{1,2}] \leq \text{negl.}$  We conclude that POLL introduces only a negligible opportunity for distinguishability.

We now treat the effects of the (STATE,  $\Sigma$ ) message that  $\mathcal{F}_{\text{PayNet,Pay}}$  sends to  $\mathcal{S}_{\text{LN-Reg-Open}}$  as a concatenation 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. 36, 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. 11, line 6, when the first message is  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}})$  and *Bob* is corrupted), or if the handling of the message results in sending (SUBMIT) to  $\mathcal{G}_{\text{Ledger}}$  (e.g. in Fig. 17, line 11 when the first message is (PUSHFULFILL, *pchid*) and counterparty has gone on-chain). In that case, the (STATE) message is simply ignored (Fig. 36, line 3) and does not influence execution in any way.

In the case when (STATE,  $\Sigma$ ) is handled,  $\mathcal{S}_{\text{LN-Reg-Open}}$  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. 36, 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. 23, where, if it does not halt (line 10), it updates the state of each affected channel (line 15) and does not send any message, thus control returns to  $\mathcal{E}$ . Therefore we have to prove that  $\mathcal{F}_{\text{PayNet,Pay}}$  halts with at most negligible probability in order to conclude that the handling of a (STATE) message does not introduce opportunity for distinguishability.

$\mathcal{F}_{\text{PayNet,Pay}}$  halts if the player *Dave* charged for a payment is an honest intermediary of that payment, (has POLLED in time to catch a malicious closure (event *A*) but a malicious closure succeeded (event *B*)) or (no malicious closure succeeded ( $\neg B$ ) and *Dave* has POLLED in time to learn the preimage of the HTLC early enough (event *C*) and has attempted to fulfill on chain at the right moment (event *D*)) (Fig. 23, line 9) – i.e. halts in the event  $(A \wedge B) \vee (\neg B \wedge C \wedge D)$ .  $\mathcal{S}_{\text{LN-Reg-Open}}$  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. 36, line 9), – i.e. *Dave* is charged in the event  $B \vee (F \wedge G)$ .

We will now show that  $\Pr[A \wedge B] \leq \text{negl} \wedge \Pr[(C \wedge D) \wedge (F \wedge G)] \leq \text{negl}$ , from which we can deduce that  $\Pr[(A \wedge B) \vee ((C \wedge D) \wedge (F \wedge G))] \leq \text{negl}$  and thus  $\Pr[((A \wedge B) \vee (\neg B \wedge C \wedge D)) \wedge (B \vee (F \wedge G))] \leq \text{negl}$ . 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  $\Pr[A \wedge B] \leq \text{negl}$ .

The only way for event  $C$  to be true is if  $\mathcal{E}$  sends (POLL) to *Dave* during the prescribed time period (Fig. 25, line 3) – note that the addition to `polls(Dave)` during registration (Fig. 20, 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. 11, line 18), registration happens necessarily before handling (INVOICE) (Fig. 20, line 21) and the element added to `polls(Dave)` at registration is the chain height at that time (Fig. 20, line 9). When *Dave* receives (POLL),  $\mathcal{F}_{\text{PayNet, Pay}}$  sends (GETCLOSEDFUNDS) to  $\mathcal{S}_{\text{LN-Reg-Open}}$  (Fig. 25, line 10) with overwhelming probability (since, as we saw earlier,  $\mathcal{F}_{\text{PayNet, Pay}}$  halts in line 7 only with negligible probability).

Event  $G$  happens only when the previous counterparty successfully appends HTLC-timeout to  $\Sigma_{\text{Dave}}$ , 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 happen only with negligible probability. Thus, given that  $F$  happens until a moment when  $|\Sigma_{\text{Dave}}| \leq \text{IncomingCltvExpiry} - k - (2 + r)\text{windowSize}$ , *Dave* has the time to successfully fulfill the HTLC with overwhelming probability. Given  $C$ , *Dave* has POLLED at two moments  $h_1, h_2 \in [\text{OutgoingCltvExpiry} + k + (2 + r)\text{windowSize} + 1, \text{IncomingCltvExpiry} - k - (2 + r)\text{windowSize}]$ , such that  $h_2 \geq h_1 + k + (2 + r)\text{windowSize}$ . If  $\Sigma_{\text{Dave}}$  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. 17, 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. 17, 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. 10, line 9), which either ends up in  $\Sigma_{Dave}$  by block height  $h_1 + (2 + r) \text{windowSize}$  (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 all but negligible cases *Dave* is not charged for the payment, i.e.  $\Pr[(C \wedge D) \wedge (F \wedge G)] \leq \text{negl.}$   $\square$

#### Simulator $\mathcal{S}$

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

- 1: Upon receiving (CLOSECHANNEL, **receipt**, *Alice*) from  $\mathcal{F}_{\text{PayNet}}$ :
- 2:     simulate Fig. 18 receiving (CLOSECHANNEL, **receipt**) with *Alice*'s ITI
- 3: every time **closedChannels** of *Alice* is updated with **channel** (Fig. 18, line 10 and Fig. 10, line 23), send (CLOSECHANNEL, **channel**, *Alice*) to  $\mathcal{F}_{\text{PayNet}}$  and expect (CONTINUE) from  $\mathcal{F}_{\text{PayNet}}$  to resume simulation

Fig. 37.

**Lemma 5.**  $\text{EXEC}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet}}, \text{Pay}, \mathcal{G}_{\text{Ledger}}} \stackrel{c}{\approx} \text{EXEC}_{\mathcal{S}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet}}, \mathcal{G}_{\text{Ledger}}}$

*Proof.* When  $\mathcal{E}$  sends (CLOSECHANNEL, **receipt**) to *Alice*, in the ideal world, if it is not the first closing message to *Alice* the message is ignored (Fig. 24, line 5). Similarly in the real world, if there has been another such message, *Alice* ignores it (Fig. 18, lines 10 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. 24, 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. 37). 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. 18 always passes and the rest of Fig. 18 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. 18 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. 26, lines 15-16). In the real world, *Alice* sends (NEWS, **newChannels**, **closedChannels**, **updatesToReport**) to  $\mathcal{E}$  and empties these fields as well (Fig. 9, 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. 21, line 30). This happens when the simulated *Alice* ITI handles a FUNDINGLOCKED message from *Bob* (Fig. 32, line 26). In the real world *Alice* would have modified her **newChannels** while handling *Bob*'s FUNDINGLOCKED (Fig. 8, 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. 21, 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. 32, 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. 26, line 13), which in turn happens exactly when the simulated *Alice* ITI adds the **channel** to her **closedChannels** (Fig. 37, 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. 16. In the ideal world on the other hand, it is updated in line 6 of Fig. 22, which is triggered only by an (UPDATE) message by  $\mathcal{S}$ . This message is sent only when line 10 of Fig. 16 is simulated by  $\mathcal{S}$  (Fig. 35, 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. 16, line 2 and Fig. 15, lines 5 and 15), 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. 26, 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. 35, 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. 24, lines 7-30) 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 happens only with negligible probability.

$\mathcal{F}_{\text{PayNet}}$  halts in line 22 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 21) would cause  $\mathcal{F}_{\text{PayNet}}$  to alert  $\mathcal{S}$  (Fig. 25, line 10), who in turn would submit the punishment transaction in time to prevent the counterparty from spending the delayed payment (Fig. 10, 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 themselves. This however can only happen with negligible probability, since the counterparty would need to forge a signature valid by the public revocation key.

$\mathcal{F}_{\text{PayNet}}$  halts in line 27 of Fig. 24 in case  $\mathcal{E}$  has asked for the channel to close, but too much time has passed since. This event cannot happen, because  $\mathcal{F}_{\text{PayNet}}$  forwards a `CLOSECHANNEL` message to  $\mathcal{S}$  (Fig. 24, line 6) for every element that it adds to `pendingPay` (Fig. 24, line 4) and this causes  $\mathcal{S}$  to submit the closing transaction to  $\mathcal{G}_{\text{Ledger}}$  (Fig. 18, line 11). This transaction is necessarily valid, because there is no other transaction that spends the funding transaction of the channel, according to the first check of line 26 of Fig. 24.  $\mathcal{F}_{\text{PayNet}}$  halts in this case only if it is sure that the chain has grown by  $(2 + r) \cdot \text{windowSize}$  blocks, and thus if the closing transaction had been submitted when `CLOSECHANNEL` was received, it should have been necessarily included (Proposition 1). Therefore  $\mathcal{F}_{\text{PayNet}}$  cannot halt in line 27 of Fig. 24. We deduce that the execution of `checkClosed` by  $\mathcal{F}_{\text{PayNet}}$  increases the probability of distinguishability only by a negligible amount.  $\square$

### Theorem 1 (Lightning Payment Network Security).

$$\text{EXEC}_{\mathcal{H}_{\text{LN}}, \mathcal{A}_{\text{d}}, \mathcal{E}}^{\mathcal{G}_{\text{Ledger}}} \stackrel{c}{\approx} \text{EXEC}_{\mathcal{S}, \mathcal{E}}^{\mathcal{F}_{\text{PayNet}}, \mathcal{G}_{\text{Ledger}}}$$

*Proof.* The theorem is a direct result of Lemmas 1-5.  $\square$

## 8 Combined Sign primitive

### 8.1 Algorithms

- $(mpk, msk) \leftarrow \text{MASTERKEYGEN}(1^k)$
- $(pk, sk) \leftarrow \text{KEYSHAREGEN}(1^k)$
- $(cpk_l, csk_l) \leftarrow \text{COMBINEKEY}(msk, mpk, sk, pk)$
- $cpk_l \leftarrow \text{COMBINEPUBKEY}(mpk, pk)$
- $\sigma \leftarrow \text{SIGN}(csk, m)$
- $\{0, 1\} \leftarrow \text{VERIFY}(cpk, m, \sigma)$

### 8.2 Correctness

- $\forall k \in \mathbb{N}$ ,  
 $\Pr[(mpk, msk) \leftarrow \text{MASTERKEYGEN}(1^k),$   
 $(pk, sk) \leftarrow \text{KEYSHAREGEN}(1^k),$   
 $(cpk_1, csk_1) \leftarrow \text{COMBINEKEY}(msk, mpk, sk, pk),$   
 $cpk_2 \leftarrow \text{COMBINEPUBKEY}(mpk, pk),$   
 $cpk_1 = cpk_2] = 1$
- $\forall k \in \mathbb{N}, m \in \mathcal{M}$ ,  
 $\Pr[(mpk, msk) \leftarrow \text{MASTERKEYGEN}(1^k),$   
 $(pk, sk) \leftarrow \text{KEYSHAREGEN}(1^k),$   
 $(cpk, csk) \leftarrow \text{COMBINEKEY}(mpk, msk, pk, sk),$   
 $\text{VERIFY}(cpk, m, \text{SIGN}(csk, m)) = 1] = 1$

### 8.3 Security

#### Game share-EUF<sup>A</sup>(1<sup>k</sup>)

```

1: (aux, mpk, n) ← A(INIT)
2: for i ← 1 to n do
3:   (pki, ski) ← KEYSHAREGEN(1k)
4: end for
5: (cpk*, pk*, m*, σ*) ← A(KEYS, aux, pk1, ..., pkn)
6: if pk* ∈ {pk1, ..., pkn} ∧ cpk* = COMBINEPUBKEY(mpk, pk*) ∧
   VERIFY(cpk*, m*, σ*) = 1 then
7:   return 1
8: else
9:   return 0
10: end if

```

Fig. 38.

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

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

**Game master-EUF-CMA<sup>A</sup>(1<sup>k</sup>)**

```

1: (mpk, msk) ← MASTERKEYGEN(1k)
2: i ← 0
3: (auxi, response) ← A(INIT, mpk)
4: while response can be parsed as (pk, sk, m) do
5:   i ← i + 1
6:   store pk, sk, m as pki, ski, mi
7:   (cpki, cski) ← COMBINEKEY(mpk, msk, pki, ski)
8:   σi ← SIGN(cski, mi)
9:   (auxi, response) ← A(SIGNATURE, auxi-1, σi)
10: end while
11: parse response as (cpk*, pk*, m*, σ*)
12: if m* ∉ {m1, ..., mi} ∧ cpk* = COMBINEPUBKEY(mpk, pk*) ∧
    VERIFY(cpk*, m*, σ*) = 1 then
13:   return 1
14: else
15:   return 0
16: end if

```

**Fig. 39.**

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

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

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

## 8.4 Construction

output standard signing keypairs to avoid duplication?

Parameters:  $\mathcal{H}, G$

**function** MasterKeyGen( $1^k$ , rand)

Return (rand,  $G \cdot \text{rand}$ )

**end function**

**function** KeyShareGen( $1^k$ , rand)

Return (rand,  $G \cdot \text{rand}$ )

```

end function
function CombineKey( $msk, mpk, sk, pk$ )
    return  $msk \cdot \mathcal{H}(mpk \parallel pk) + sk \cdot \mathcal{H}(pk \parallel mpk)$ 
end function
function CombinePubKey( $mpk, pk$ )
    return  $mpk \cdot \mathcal{H}(mpk \parallel pk) + pk \cdot \mathcal{H}(pk \parallel mpk)$ 
end function
function Sign( $csk, m$ )
    like standard sign
end function
function Verify( $cpk, m, \sigma$ )
    like standard verify
end function
just to remember
 $sh_{\text{rev},n} \leftarrow shb_{\text{rev}} \cdot \mathcal{H}(phb_{\text{rev}} \parallel pt_{\text{com},n}) + st_{\text{com},n} \cdot \mathcal{H}(pt_{\text{com},n} \parallel phb_{\text{rev}})$ 
 $pt_{\text{rev},n+2} \leftarrow ptb_{\text{rev}} \cdot \mathcal{H}(ptb_{\text{rev}} \parallel ph_{\text{com},n+2}) + ph_{\text{com},n+2} \cdot \mathcal{H}(ph_{\text{com},n+2} \parallel ptb_{\text{rev}})$ 
 $ph_{\text{rev},n+2} \leftarrow phb_{\text{rev}} \cdot \mathcal{H}(phb_{\text{rev}} \parallel pt_{\text{com},n+2}) + pt_{\text{com},n+2} \cdot \mathcal{H}(pt_{\text{com},n+2} \parallel phb_{\text{rev}})$ 

```

**Lemma 6.** *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

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

We construct a PPT distinguisher  $\mathcal{A}$  (Fig. 40) such that

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

that breaks the assumption, thus proving Lemma 6.

**Algorithm  $\mathcal{A}(vk)$**

```

1:  $j \xleftarrow{\$} U[1, T(\mathcal{B})]$  //  $T(M)$  is the maximum running time of  $M$ 
2: Random Oracle: for every first-seen query  $q$  from  $\mathcal{B}$  set  $\mathcal{H}(q)$  to a random value
3:   return  $\mathcal{H}(q)$  to  $\mathcal{B}$ 
4:  $(\text{aux}, \text{mpk}, n) \leftarrow \mathcal{A}(\text{INIT})$ 
5: for  $i \leftarrow 1$  to  $n$  do
6:    $(pk_i, sk_i) \leftarrow \text{KEYSHAREGEN}(1^k)$ 
7: end for
8: Random Oracle: Let  $q$  be the  $j$ th first-seen query from  $\mathcal{B}$ :
9:   if  $q = (\text{mpk} \parallel x)$  then
10:    if  $\mathcal{H}(x \parallel \text{mpk})$  unset then
11:      set  $\mathcal{H}(x \parallel \text{mpk})$  to a random value
12:    end if
13:    set  $\mathcal{H}(\text{mpk} \parallel x)$  to  $(vk - x \cdot \mathcal{H}(x \parallel \text{mpk})) \cdot \text{mpk}^{-1}$ 
14:   else if  $q = (x \parallel \text{mpk})$  then
15:     if  $\mathcal{H}(\text{mpk} \parallel x)$  unset then
16:       set  $\mathcal{H}(\text{mpk} \parallel x)$  to a random value
17:     end if
18:     set  $\mathcal{H}(x \parallel \text{mpk})$  to  $(vk - \text{mpk} \cdot \mathcal{H}(\text{mpk} \parallel x)) \cdot x^{-1}$ 
19:   else
20:     set  $\mathcal{H}(q)$  to a random value
21:   end if
22:   return  $\mathcal{H}(q)$  to  $\mathcal{B}$ 
23:  $(\text{cpk}^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\text{KEYS}, \text{aux}, pk_1, \dots, pk_n)$ 
24: if  $vk = \text{cpk}^* \wedge \mathcal{B}$  wins the share-EUF game then //  $\mathcal{A}$  won the EUF-CMA game
25:   return  $(m^*, \sigma^*)$ 
26: else
27:   return FAIL
28: end if

```

**Fig. 40.**

Let  $Y$  be the range of the random oracle. The modified random oracle used in Fig. 40 is indistinguishable from the standard random oracle by PPT algorithms since the statistical distance of the standard random oracle from the modified one is at most  $\frac{1}{2|Y|} < \text{negl}(k)$  as they differ in at most one element.

Let  $E$  denote the event in which  $\mathcal{B}$  does not invoke COMBINEPUBKEY to produce  $\text{cpk}^*$ . In that case the values  $\mathcal{H}(pk^* \parallel \text{mpk})$  and  $\mathcal{H}(\text{mpk} \parallel pk^*)$

are decided after  $\mathcal{B}$  terminates (Fig. 40, line 24) and thus

$$\begin{aligned} \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*) | E] &= \frac{1}{|Y|} < \text{negl}(k) \Rightarrow \\ \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*) \wedge E] &< \text{negl}(k) \quad . \end{aligned} \quad (1)$$

It is

$$\begin{aligned} (\mathcal{B} \text{ wins}) &\rightarrow (cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*)) \Rightarrow \\ \Pr[\mathcal{B} \text{ wins}] &\leq \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*)] \Rightarrow \\ \Pr[\mathcal{B} \text{ wins} \wedge E] &\leq \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*) \wedge E] \stackrel{(1)}{\Rightarrow} \\ &\Pr[\mathcal{B} \text{ wins} \wedge E] < \text{negl}(k) \quad . \end{aligned}$$

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

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

We now focus at the event  $\neg E$ . Let  $F$  the event in which the call of  $\mathcal{B}$  to  $\text{COMBINEPUBKEY}$  to produce  $cpk^*$  results in the  $j$ th invocation of the Random Oracle. Since  $j$  is chosen uniformly at random,  $\Pr[F | \neg E] = \frac{1}{T(\mathcal{B})}$ . 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

$$\begin{aligned} cpk^* &= \text{COMBINEPUBKEY}(mpk, pk^*) = \\ &mpk \cdot \mathcal{H}(mpk \parallel pk^*) + pk^* \cdot \mathcal{H}(pk^* \parallel mpk) \end{aligned}$$

Since  $F$  holds, the  $j$ th invocation of the Random Oracle queried either  $\mathcal{H}(mpk \parallel pk^*)$  or  $\mathcal{H}(pk^* \parallel mpk)$ . In either case (Fig. 40, lines 9-18), it is  $cpk^* = vk$ . This means that  $\text{VERIFY}(vk, m^*, \sigma^*) = 1$ . We conclude that in the event  $(F \wedge \mathcal{B} \text{ wins} \wedge \neg E)$ ,  $\mathcal{A}$  wins the EUF-CMA game. A final observation is that the probability that the events  $(\mathcal{B} \text{ wins} \wedge \neg E)$  and  $F$  are almost independent, thus

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

□



**Lemma 7.** *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

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

We construct a PPT distinguisher  $\mathcal{A}$  (Fig. 41) such that

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

that breaks the assumption, thus proving Lemma 7.

**Algorithm  $\mathcal{A}(vk)$**

```

1:  $j \xleftarrow{\$} U[1, T(\mathcal{B}) + T(\mathcal{A})]$  //  $T(M)$  is the maximum running time of  $M$ 
2: Random Oracle: for every first-seen query  $q$  from  $\mathcal{B}$  set  $\mathcal{H}(q)$  to a random value
3:   return  $\mathcal{H}(q)$  to  $\mathcal{B}$ 
4:  $(mpk, msk) \leftarrow \text{MASTERKEYGEN}(1^k)$ 
5: Random Oracle: Let  $q$  be the  $j$ th first-seen query from  $\mathcal{B}$  or  $\mathcal{A}$ :
6:   if  $q = (mpk \parallel x)$  then
7:     if  $\mathcal{H}(x \parallel mpk)$  unset then
8:       set  $\mathcal{H}(x \parallel mpk)$  to a random value
9:     end if
10:    set  $\mathcal{H}(mpk \parallel x)$  to  $(vk - x \cdot \mathcal{H}(x \parallel mpk)) \cdot mpk^{-1}$ 
11:   else if  $q = (x \parallel mpk)$  then
12:     if  $\mathcal{H}(mpk \parallel x)$  unset then
13:       set  $\mathcal{H}(mpk \parallel x)$  to a random value
14:     end if
15:     set  $\mathcal{H}(x \parallel mpk)$  to  $(vk - mpk \cdot \mathcal{H}(mpk \parallel x)) \cdot x^{-1}$ 
16:   else
17:     set  $\mathcal{H}(q)$  to a random value
18:   end if
19:   return  $\mathcal{H}(q)$  to  $\mathcal{B}$  or  $\mathcal{A}$ 
20:  $i \leftarrow 0$ 
21:  $(\text{aux}_i, \text{response}) \leftarrow \mathcal{B}(\text{INIT}, mpk)$ 
22: while response can be parsed as  $(pk, sk, m)$  do
23:    $i \leftarrow i + 1$ 
24:   store  $pk, sk, m$  as  $pk_i, sk_i, m_i$ 
25:    $(cpk_i, csk_i) \leftarrow \text{COMBINEKEY}(mpk, msk, pk_i, sk_i)$ 
26:    $\sigma_i \leftarrow \text{SIGN}(csk_i, m_i)$ 
27:    $(\text{aux}_i, \text{response}) \leftarrow \mathcal{B}(\text{SIGNATURE}, \text{aux}_{i-1}, \sigma_i)$ 
28: end while
29: parse response as  $(cpk^*, pk^*, m^*, \sigma^*)$ 
30:  $(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\text{KEYS}, \text{aux}, pk_1, \dots, pk_n)$ 
31: if  $vk = cpk^* \wedge \mathcal{B}$  wins the master-EUF-CMA game then //  $\mathcal{A}$  won the EUF-CMA game
32:   return  $(m^*, \sigma^*)$ 
33: else
34:   return FAIL
35: end if

```

**Fig. 41.**

The modified random oracle used in Fig. 41 is indistinguishable from the standard random oracle for the same reasons as in the proof of Lemma 6.

Let  $E$  denote the event in which `COMBINEPUBKEY` is not invoked to produce  $cpk^*$ . In that case the values  $\mathcal{H}(pk^* \parallel mpk)$  and  $\mathcal{H}(mpk \parallel pk^*)$  are decided after  $\mathcal{B}$  terminates (Fig. 41, line 31) and thus

$$\begin{aligned} \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*) \mid E] &< \text{negl}(k) \Rightarrow \\ \Pr[cpk^* = \text{COMBINEPUBKEY}(mpk, pk^*) \wedge E] &< \text{negl}(k) . \end{aligned} \quad (3)$$

We can reason like in the proof of Lemma 6 to deduce that

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

We now focus at the event  $\neg E$ . Let  $F$  the event in which the call of `COMBINEPUBKEY` that produces  $cpk^*$  results in the  $j$ th invocation of the Random Oracle. Since  $j$  is chosen uniformly at random,  $\Pr[F \mid \neg E] = \frac{1}{T(\mathcal{B}) + T(\mathcal{A})}$ . Observe that  $\Pr[F \mid E] = 0 \Rightarrow \Pr[F] = \Pr[F \mid \neg E] = \frac{1}{T(\mathcal{B}) + T(\mathcal{A})}$ .

Once more we can reason in the same fashion as in the proof of Lemma 6 to deduce that

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

□

**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 direct consequence of Lemma 6, Lemma 7 and the definition of **combine-EUF**-security. □

## 9 Notes on Lightning Specification

- The relevant part of the specification can be found at <https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md>.

## 10 The Ledger Functionality and its Properties

We next provide the complete description of the ledger functionality that is based on the UC formalisation of [9,10].

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  $\mathbf{pt}_i$  for party  $U_i$  below). The functionality has a parameter **windowSize** such that no finalised prefix of any player will be shorter than  $|\mathbf{state}| - \mathbf{windowSize}$ . 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  $\mathbf{maxTime}_{\text{window}}$ , a number of blocks at least **windowSize** are added to **state**.
2. in a period of time equal to  $\mathbf{minTime}_{\text{window}}$ , no more blocks may be added to **state** if **windowSize** blocks have been already added.
3. each window of **windowSize** blocks has at most  $\mathbf{advBlcks}_{\text{window}}$  adversarial blocks included in it.
4. any transaction that (i) is submitted by an honest party earlier than  $\frac{\mathbf{Delay}}{2}$  rounds before the time that the block that is **windowSize** positions before the head of the **state** was included, and (ii) is valid with respect to an honest block that extends **state**, then it must be included in such block.

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

**Proposition 1.** *Consider any synchronised honest party that wishes to place a transaction **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  $t \geq 1 + (2+r)\mathbf{windowSize}$ , where  $r = \lceil (\mathbf{maxTime}_{\text{window}} + \frac{\mathbf{Delay}}{2}) / \mathbf{minTime}_{\text{window}} \rceil$ , **tx** is guaranteed to be included in the intended block range as long as the*

party submits  $\mathbf{tx}$  to the ledger functionality by the time the block indexed by  $h + t - 1 - (2 + r)\mathbf{windowSize}$  is added to **state** in its view.

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

Suppose  $U$  transmits the transaction  $\mathbf{tx}$  to the ledger at time  $\tau_{h+x}^U$ . Observe that as long as  $\tau_{h+x} + \mathbf{maxTime}_{\mathbf{window}}$  is  $\mathbf{Delay}/2$  before the time that block with index  $h + t - 1 - 2\mathbf{windowSize}$  enters **state**, then  $\mathbf{tx}$  is guaranteed to enter the **state** in a block with index up to  $h + t - 1$  since  $\mathbf{advBlcks}_{\mathbf{window}} < \mathbf{windowSize}$ . It follows we need  $\tau_{h+x} + \mathbf{maxTime}_{\mathbf{window}} < \tau_{h+t-1-2\mathbf{windowSize}} - \frac{\mathbf{Delay}}{2}$ . Let  $r = \lceil (\mathbf{maxTime}_{\mathbf{window}} + \frac{\mathbf{Delay}}{2}) / \mathbf{minTime}_{\mathbf{window}} \rceil$ . Recall that in a period of  $\mathbf{minTime}_{\mathbf{window}}$  rounds at most  $\mathbf{windowSize}$  blocks enter **state**. As a result  $r\mathbf{windowSize}$  blocks require at least  $r\mathbf{minTime}_{\mathbf{window}} \geq \mathbf{maxTime}_{\mathbf{window}} + \frac{\mathbf{Delay}}{2}$  rounds. It follows that if  $t \geq 1 + (2 + r)\mathbf{windowSize}$  and  $x = t - 1 - (2 + r)\mathbf{windowSize}$  the inequality follows.  $\square$

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

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

**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 (CLOCK-READ,  $\text{sid}_C$ ) to  $\mathcal{G}_{\text{CLOCK}}$  and upon receiving response (CLOCK-READ,  $\text{sid}_C, \tau$ ) set  $\tau_L := \tau$  and do the following if  $\tau > 0$  (otherwise, ignore input):

1. Updating synchronized/desynchronized party set:
  - (a) Let  $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$  denote the set of desynchronized honest parties that have been registered (continuously) to the ledger, the clock, and the GRO since time  $\tau' < \tau_L - \text{Delay}$ . Set  $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$ .
  - (b) For any synchronized party  $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$ , if  $U_p$  is not registered to the clock, then consider it desynchronized, i.e., set  $\mathcal{P}_{DS} \cup \{U_p\}$ .
2. If  $I$  was received from an honest party  $U_p \in \mathcal{P}$ :
  - (a) Set  $\mathcal{I}_H^T := \mathcal{I}_H^T \parallel (I, U_p, \tau_L)$ ;
  - (b) Compute  $\mathbf{N} = (N_1, \dots, N_\ell) := \text{ExtendPolicy}(\mathcal{I}_H^T, \text{state}, \text{NxtBC}, \text{buffer}, \tau_{\text{state}})$  and if  $\mathbf{N} \neq \varepsilon$  set  $\text{state} := \text{state} \parallel \text{Blockify}(N_1) \parallel \dots \parallel \text{Blockify}(N_\ell)$  and  $\tau_{\text{state}} := \tau_{\text{state}} \parallel \tau_L^\ell$ , where  $\tau_L^\ell = \tau_L \parallel \dots \parallel \tau_L$ .
  - (c) For each  $\text{BTX} \in \text{buffer}$ : if  $\text{Validate}(\text{BTX}, \text{state}, \text{buffer}) = 0$  then delete  $\text{BTX}$  from  $\text{buffer}$ . Also, reset  $\text{NxtBC} := \varepsilon$ .
  - (d) If there exists  $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$  such that  $|\text{state}| - \text{pt}_j > \text{windowSize}$  or  $\text{pt}_j < |\text{state}_j|$ , then set  $\text{pt}_k := |\text{state}|$  for all  $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$ .
3. If the calling party  $U_p$  is *stalled* or *time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input  $I$  and its sender's ID,  $\mathcal{G}_{\text{LEDGER}}$  executes the corresponding code from the following list:
  - *Submitting a transaction:*  
 If  $I = (\text{SUBMIT}, \text{sid}, \text{tx})$  and is received from a party  $U_p \in \mathcal{P}$  or from  $\mathcal{A}$  (on behalf of a corrupted party  $U_p$ ) do the following
    - (a) Choose a unique transaction ID  $\text{txid}$  and set  $\text{BTX} := (\text{tx}, \text{txid}, \tau_L, U_p)$
    - (b) If  $\text{Validate}(\text{BTX}, \text{state}, \text{buffer}) = 1$ , then  $\text{buffer} := \text{buffer} \cup \{\text{BTX}\}$ .
    - (c) Send (SUBMIT, BTX) to  $\mathcal{A}$ .
  - *Reading the state:*  
 If  $I = (\text{READ}, \text{sid})$  is received from a party  $U_p \in \mathcal{P}$  then set  $\text{state}_p := \text{state}_{|\min\{\text{pt}_p, |\text{state}|\}}|$  and return (READ,  $\text{sid}, \text{state}_p$ ) to the requester. If the requester is  $\mathcal{A}$  then send (state, buffer,  $\mathcal{I}_H^T$ ) 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  $\mathcal{I}_H^T$  as above)  $\text{predict-time}(\mathcal{I}_H^T) = \hat{\tau} > \tau_L$  then send  $(\text{CLOCK-UPDATE}, \text{sid}_C)$  to  $\mathcal{G}_{\text{CLOCK}}$ . Else send  $I$  to  $\mathcal{A}$ .

- *The adversary proposing the next block:*

If  $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \dots, \text{txid}_\ell))$  is sent from the adversary, update **NxtBC** as follows:

- Set  $\text{listOfTxid} \leftarrow \epsilon$
- For  $i = 1, \dots, \ell$  do: if there exists  $\text{BTX} := (x, \text{txid}, \text{minerID}, \tau_L, U_j) \in \text{buffer}$  with  $\text{ID txid} = \text{txid}_i$  then set  $\text{listOfTxid} := \text{listOfTxid} \parallel \text{txid}_i$ .
- Finally, set  $\text{NxtBC} := \text{NxtBC} \parallel (\text{hFlag}, \text{listOfTxid})$  and output  $(\text{NEXT-BLOCK}, \text{ok})$  to  $\mathcal{A}$ .

- *The adversary setting state-slackness:*

If  $I = (\text{SET-SLACK}, (U_{i_1}, \hat{\text{pt}}_{i_1}), \dots, (U_{i_\ell}, \hat{\text{pt}}_{i_\ell}))$ , with  $\{U_{i_1}, \dots, U_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$  is received from the adversary  $\mathcal{A}$  do the following:

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

- *The adversary setting the state for desynchronized parties:*

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

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

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  $\text{sid}$  the clock maintains a variable  $\tau_{\text{sid}}$ . 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  $\text{sid}$  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 (CLOCK-READ, 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  $U_p = (\cdot, \text{sid}) \in \mathcal{P}$ .

## References

1. Nakamoto S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
2. 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)
3. 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)
4. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments
5. Decker C., Wattenhofer R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In Symposium on Self-Stabilizing Systems: pp. 3–18: Springer (2015)
6. Lind J., Eyal I., Pietzuch P. R., Sirer E. G.: Teechan: Payment Channels Using Trusted Execution Environments. CoRR: vol. abs/1612.07766: URL <http://arxiv.org/abs/1612.07766> (2016)
7. Dziembowski S., Ekey 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)
8. 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)
9. 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)
10. Badertscher C., Gazi 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)