

# Payment Channels Overview

Orfeas Stefanos Thyfronitis Litos

University of Edinburgh  
o.thyfronitis@ed.ac.uk

**Abstract.** We provide a payment network functionality and prove that the Lightning Network [1] UC-realizes it.

## 1 State of a channel

Consider a channel between *Alice* and *Bob*. Both parties hold some data locally that correspond to ownership of some funds in the channel. Here we define a concise way of representing this data.

What *Alice* has to hold, specific for this channel:

- keys:
  - local funding secret key
  - remote funding public key
  - local {payment, htlc, delayed\_payment, revocation}\_basepoint\_secret
  - remote {payment, htlc, delayed\_payment, revocation}\_basepoint
  - seed (for local per\_commitment\_secrets)
  - remote per\_commitment\_secret<sub>1,...,m-1</sub>
  - remote per\_commitment\_point<sub>m,m+1</sub>
- *Alice*'s coins
- *Bob*'s coins
- every HTLC that is included in the latest irrevocably committed (local or remote) commitment:
  - direction (*Alice* → *Bob* or *Bob* → *Alice*)
  - hash
  - preimage (or  $\perp$  if still unresolved)
  - coins
  - Is it included in local commitment<sub>*n*</sub>?
  - HTLC number
- signatures:
  - signature of local commitment<sub>*n*</sub> with secret key corresponding to remote funding public key

- for every HTLC included in local commitment<sub>n</sub>, one signature of HTLC-Timeout if outgoing, HTLC-Success if incoming with secret key corresponding to remote htlc\_pubkey<sub>n</sub> (= htlc\_basepoint +  $\mathcal{H}(\text{remote per\_commitment\_point}_n || \text{remote htlc\_basepoint}) \cdot G$ )

The rest of the things 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  $\in \{\text{proposed, committed}\}$ ) 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.
- In LND there are more messages that cover errors in transmission etc. There are also rules that govern message retransmission upon connection failure.

## 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  $\leftarrow \emptyset$ 
4:   unclaimedOfferedHTLCs, unclaimedReceivedHTLCs, pendingGetPaid  $\leftarrow \emptyset$ 

5: Upon receiving (REGISTER, delay, relayDelay) from  $\mathcal{E}$ :
6:   delay  $\leftarrow$  delay
7:   relayDelay  $\leftarrow$  relayDelay
8:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  and assign largest block number to lastPoll
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 change font
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:   $\text{seed} \xleftarrow{\$} U(k)$  // 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:    $((ph_F, sh_F), (ph_{b_{pay}}, sh_{b_{pay}}), (ph_{b_{dpay}}, sh_{b_{dpay}}), (ph_{b_{htlc}}, sh_{b_{htlc}}), \mathbf{seed}, (ph_{b_{rev}}, sh_{b_{rev}})) \leftarrow \mathbf{GetKeys}()$
- 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*,  $(ph_F, sh_F), (ph_{b_{pay}}, sh_{b_{pay}}), (ph_{b_{dpay}}, sh_{b_{dpay}}), (ph_{b_{htlc}}, sh_{b_{htlc}}), (ph_{b_{com,1}}, sh_{b_{com,1}}), (ph_{b_{rev}}, sh_{b_{rev}}), tid$ ) to **pendingOpen**
- 8:   send (OPENCHANNEL, *x*, **delay**, **relayDelay**,  $ph_F, ph_{b_{pay}}, ph_{b_{dpay}}, ph_{b_{htlc}}, ph_{com,1}, ph_{b_{rev}}, tid$ ) to *Bob*

**Fig. 2.**

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

- 1: Upon receiving (OPENCHANNEL, *x*, *BobDelay*,  $pt_F, pt_{b_{pay}}, pt_{b_{dpay}}, pt_{b_{htlc}}, pt_{com,1}, pt_{b_{rev}}, tid$ ) from *Bob*:
- 2:   ensure *tid* has not been used yet with *Bob*
- 3:    $((ph_F, sh_F), (ph_{b_{pay}}, sh_{b_{pay}}), (ph_{b_{dpay}}, sh_{b_{dpay}}), (ph_{b_{htlc}}, sh_{b_{htlc}}), \mathbf{seed}, (ph_{b_{rev}}, sh_{b_{rev}})) \leftarrow \mathbf{GetKeys}()$
- 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**, **relayDelay**,  $ph_F, ph_{b_{pay}}, ph_{b_{dpay}}, ph_{b_{htlc}}, ph_{com,1}, ph_{b_{rev}}, tid$ ) 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  $F$ , output:  $(x, ph_{pay,1})$ }
- 13:   **localCom**  $\leftarrow$  TX {input: output 1 of  $F$ , output:  $(x, pt_{rev,1} \vee (ph_{dpay,1}, \text{remoteDelay} + k + 1 \text{ relative}))$ }
- 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:  $localCom \leftarrow localCom_1 \leftarrow TX \{input: output\ 1\ of\ F, output: (x, pt_{pay,1})\}$
- 7: ensure  $\text{verify}(localCom_1, BobSig_1, pt_F) = \text{True}$
- 8:  $remoteCom \leftarrow remoteCom_1 \leftarrow TX \{input: output\ 1\ of\ F, output: (x, ph_{rev} \vee (pt_{dpay}, delay + k + 1\ \text{relative}))\}$
- 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  $\Pi_{LN}$  - CHECKNEW**

- 1: **explicitly add keys et al to channel**
- 2: Upon receiving (CHECKNEW, *Alice*, *Bob*, *tid*) from  $\mathcal{E}$ : // new message:  
represents lnd polling daemon
- 3: ensure there is a matching **channel** in **pendingOpen** with id *pchid*, with a  
“broadcast” mark, funded with *x* coins
- 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:  $\text{prand}_2 \leftarrow \text{PRF}(\text{seed}, 2)$
- 7:  $(sh_{\text{com},2}, ph_{\text{com},2}) \leftarrow \text{KeyShareGen}(1^k; \text{prand}_2)$
- 8: add TX to **channel** data
- 9: replace “broadcast” mark in **channel** with “in state”
- 10: **if** **channel** is marked as “in state, FUNDINGLOCKED” **then**
- 11:     move channel data from **pendingOpen** to **channels**
- 12:     add receipt of channel to **newChannels**
- 13: **end if**
- 14: send (FUNDINGLOCKED, *pchid*, *ph<sub>com,2</sub>*) to *Bob*

Fig. 7.

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

- 1: Upon receiving (FUNDINGLOCKED, *pchid*, *pt<sub>com,2</sub>*) from *Bob*:
- 2: ensure there is a **channel** with ID *pchid* with *Bob* in **pendingOpen** with a  
“no FUNDINGLOCKED” mark
- 3: ensure  $pk(st_{\text{com},n}) = pt_{\text{com},n}$
- 4: replace “no FUNDINGLOCKED” mark in **channel** with “FUNDINGLOCKED”
- 5: ensure **channel** has an “in state” mark
- 6: generate 2nd remote delayed payment, htlc, payment keys
- 7: add TX to **channel** data
- 8: move channel data from **pendingOpen** to **channels**
- 9: add receipt of channel to **newChannels**

Fig. 8.

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

```

1: Upon receiving (POLL) from  $\mathcal{E}$ :
2:   send (READ) to  $\mathcal{G}_{Ledger}$  and assign reply to  $\Sigma_{Alice}$ 
3:   assign largest block number in  $\Sigma_{Alice}$  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:       remember preimage - hash combination
9:     else if input of  $\tau$  has not been spent and timelock is over then
10:      remove  $\tau$  from unclaimedOfferedHTLCs
11:      add  $\tau$  to toSubmit
12:     end if
13:   end for
14:   for all  $\text{remoteCom}_n \in \Sigma_{Alice}$  that spend  $F$  of a  $\text{channel} \in \text{channels}$  do
15:     if we do not have  $sh_{rev,n}$  then // Honest closure
16:       for all received HTLC outputs  $i$  of  $\text{remoteCom}_n$  do
17:         if we know the preimage  $R$  then
18:            $\text{TX} \leftarrow \{\text{input: } i \text{ HTLC output of } \text{remoteCom}_n \text{ with}$ 
19:              $(ph_{htlc,n}, R) \text{ as method, output: } pk_{Alice}\}$ 
20:            $\text{sig} \leftarrow \text{signature}(\text{TX}, sh_{htlc,n})$ 
21:           add (sig, TX) to toSubmit
22:         else
23:           add ( $\text{channel}, \text{remoteCom}_n, h, sh_{htlc,n}$ ) to
24:             unclaimedReceivedHTLCs
25:         end if
26:       end for
27:       for all unspent offered HTLC outputs  $i$  of  $\text{remoteCom}_n$  do
28:          $\text{TX} \leftarrow \{\text{input: } i \text{ HTLC output of } \text{remoteCom}_n \text{ with } ph_{htlc,n} \text{ as}$ 
29:           method, output:  $pk_{Alice}\}$ 
30:          $\text{sig} \leftarrow \text{signature}(\text{TX}, sh_{htlc,n})$ 
31:         if timelock has not expired then
32:           add (sig, TX) to unclaimedOfferedHTLCs
33:         else if timelock has expired then
34:           add (sig, TX) to toSubmit
35:         end if
36:       end for
37:     else // malicious closure
38:        $\text{rev} \leftarrow \text{TX}$  {inputs: all  $\text{remoteCom}_n$  outputs, choosing  $ph_{rev,n}$ 
39:         method, output:  $pk_{Alice}\}$ 
40:        $\text{sig} \leftarrow \text{signature}(\text{rev}, sh_{rev,n})$ 
41:       add (sig, rev) to toSubmit
42:     end if
43:   move  $\text{channel}$  from channels to closedChannels
44: end for
45:   send (SUBMIT, toSubmit) to  $\mathcal{G}_{Ledger}$ 

46: Upon receiving (GETNEW) from Alice:
47:   clear newChannels, closedChannels, pendingUpdates and send them to
48:     Alice

```

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

- 1: Upon receiving  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt})$  from  $\mathcal{E}$ :
- 2:   ensure that  $\overrightarrow{\text{path}}$  consists of valid  $pchids$
- 3:   ensure that the first  $pchid \in \overrightarrow{\text{path}}$  has the same  $pchid$  as in **receipt**
- 4:   ensure that **receipt** corresponds to the latest version of an open **channel**  $\in$  **channels** in which we have at least  $x$ .
- 5:   choose unique payment ID  $payid$  // unique for *Alice* and *Bob*
- 6:   add  $(\text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt}, \text{payid}, \text{"waiting for invoice"})$  to **pendingPay**
- 7:   send  $(\text{SENDINVOICE}, \text{payid})$  to *Bob*
  
- 8: Upon receiving  $(\text{SENDINVOICE}, \text{payid})$  from *Bob*:
- 9:   ensure there is no  $(\text{Bob}, \text{payid})$  entry in **pendingGetPaid**
- 10:   choose random, unique preimage  $R$
- 11:   add  $(\text{Bob}, R, \text{payid})$  to **pendingGetPaid**
- 12:   send  $(\text{INVOICE}, \mathcal{H}(R), \text{payid})$  to *Bob*
  
- 13: Upon receiving  $(\text{INVOICE}, h, \text{payid})$  from *Bob*:
- 14:   ensure there is a  $(\text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt}, \text{payid}, \text{"waiting for invoice"})$  entry in **pendingPay**
- 15:   ensure  $h$  is valid (in the range of  $\mathcal{H}$ )
- 16:   remove entry from **pendingPay**
- 17:   send  $(\text{READ})$  to  $\mathcal{G}_{\text{Ledger}}$  and assign largest block number to  $t$
- 18:    $m \leftarrow$  the concatenation of **length path**  $(x, \text{remoteDelay}_i)$  pairs, where the last **remoteDelay** is  $t + 2k + 1 + \text{BobDelay}$  and every previous **remoteDelay** is incremented by  $3k + \text{RHSDelay}$  *Alice doesn't know these Bob, RHS delays*
- 19:    $(\mu_0, \delta_0) \leftarrow \text{SphinxCreate}(m, \text{public keys of } \overrightarrow{\text{path}} \text{ parties})$
- 20:   let **remoteCom** $_n$  the latest signed remote commitment tx
- 21:   reduce simple payment output in **remoteCom** by  $x$
- 22:   add an additional  $(x, ph_{\text{rev}, n+1} \vee (ph_{\text{htlc}, n+1} \wedge pt_{\text{htlc}, n+1}, \text{ on preimage of } h) \vee ph_{\text{htlc}, n+1}, \text{largest remoteRelayDelay absolute})$  output (all with  $n + 1$  keys) to **remoteCom**, marked with **HTLCNo**
- 23:   increment **HTLCNo** $_{pchid}$  by one and associate  $x, h, pchid$  with it
- 24:   mark **HTLCNo** as "sender"
- 25:   send  $(\text{UPDATEADHTLC}, \text{first } pchid \text{ of } \overrightarrow{\text{path}}, \text{HTLCNo}_{pchid}, x, h, \text{largest remoteDelay}, (\mu_0, \delta_0))$  to  $pchid$  channel counterparty

**Fig. 10.**

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

```

1: Upon receiving (UPDATEADDHTLC,  $pchid$ , HTLCNo,  $x$ ,  $h$ , remoteDelay,  $M$ ) 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:   ( $pchid'$ ,  $x'$ , remoteDelay',  $\delta$ )  $\leftarrow$  SphinxPeel( $sk_{Alice}$ ,  $M$ )
5:   if  $\delta = \text{receiver}$  then
6:     ensure
7:      $pchid' = \perp$ ,  $x = x'$ , remoteDelay = remoteDelay' =  $2k + 1 + \text{delay}$ 
8:     increment HTLCNo $_{pchid}$  by one
9:     mark HTLCNo as "receiver"
10:  else // We are an intermediary
11:    ensure  $x = x'$ , remoteDelay = remoteDelay' +  $3k + \text{delay}$ 
12:    ensure  $pchid'$  corresponds to an open channel in channels where we
    have at least  $x$ 
13:    increment HTLCNo $_{pchid}$  by one
14:    mark HTLCNo as "intermediary"
15:  end if
16:  let remoteCom $_n$  the latest signed remote commitment tx
17:  reduce delayed payment output in remoteCom by  $x$ 
18:  add an
    ( $x$ ,  $ph_{\text{rev}, n+1} \vee (ph_{\text{htlc}, n+1} \wedge pt_{\text{htlc}, n+1}$ , remoteRelayDelay absolute)  $\vee$ 
     $ph_{\text{htlc}, n+1}$ , 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
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 ( $x$ ,  $ph_{\text{rev}, n+1} \vee (ph_{\text{htlc}, n+1} \wedge pt_{\text{htlc}, n+1}$ , on preimage
    of  $h$ )  $\vee ph_{\text{htlc}, n+1}$  remoteRelayDelay' absolute) output (all with  $n + 1$  keys) to
    remoteCom, marked with HTLCNo
27:    increment HTLCNo by 1
28:     $M' \leftarrow$  SphinxPrepare( $M$ ,  $\delta$ ,  $sk_{Alice}$ )
29:    send (UPDATEADDHTLC,  $pchid'$ , HTLCNo,  $x$ ,  $h$ , remoteDelay',  $M$ ) to
     $pchid'$  counterparty
30:  end if

```

**Fig. 11.**

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

```

1: Upon receiving (UPDATEFULFILLHTLC,  $pchid$ , HTLC_no,  $R$ ) from Bob:
2:   ensure HTLC_no  $\leq$  lastRemoteSigned, HTLC_no  $\leq$  lastLocalSigned
3:   ensure HTLC_no is an offered HTLC (localCom has  $h$  tied to a public key
   that we own)
4:   ensure  $\mathcal{H}(R) = h$ , where  $h$  is the hash in the HTLC with number
   HTLC_no
5:   add value of HTLC to delayed payment of remoteCom
6:   remove HTLC output with number HTLC_no from remoteCom
7:   if we have a channel  $pchid'$  that has a received HTLC with hash  $h$  with
   number HTLCNo' then // We are intermediary
8:     if HTLCNo'  $\leq$  lastRemoteSigned' then // HTLC committed
9:       send (READ) to  $\mathcal{G}_{Ledger}$  and assign reply to  $\Sigma_{Alice}$ 
10:      if latest remoteCom'_n  $\in \Sigma_{Alice}$  then // counterparty has gone
   on-chain
11:        TX  $\leftarrow$  {input: remoteCom' HTLC output with number HTLCNo',
   output:  $pk_{Alice}$ }
12:        sig  $\leftarrow$  signature(TX,  $sh_{htlc,n}$ )
13:        send (SUBMIT, (sig,  $R$ , TX)) to  $\mathcal{G}_{Ledger}$  // shouldn't be already
   spent by remote HTLCTimeout
14:      else // counterparty still off-chain
15:        send (UPDATEFULFILLHTLC,  $pchid'$ , HTLCNo,  $R$ ) to counterparty
16:      end if
17:    else // we haven't received REVOKEANDACK
18:      add (HTLCNo',  $R$ ) to pendingFulfills_ $pchid'$ 
19:    end if
20:  end if

```

**Fig. 12.**

**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: ( $c_{htlc,i}, ph_{rev,n+1} \vee (pt_{dpay,n+1}, delay + k + 1 \text{ relative})$ )}
- 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. 13.**

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

- 1: Upon receiving (COMMITMENTSIGNED,  $pchid$ , **comSig<sub>n+1</sub>**, **HTLCSigs<sub>n+1</sub>**) from *Bob*:
- 2:   ensure that there is a **channel**  $\in$  **channels** with ID  $pchid$  with *Bob*
- 3:   retrieve latest local commitment tx **localCom<sub>n</sub>** in **channel**
- 4:   ensure **localCom**  $\neq$  **localCom<sub>n</sub>** and **localCom**  $\neq$  **pendingLocalCom** // there are uncommitted updates
- 5:   ensure **verify**(**localCom**, **comSig<sub>n+1</sub>**,  $pt_F$ ) = **true**
- 6:   **for**  $i$  from **lastLocalSigned** to **HTLCNo** **do**
- 7:     **localHTLC<sub>n+1,i</sub>**  $\leftarrow$  TX {input: HTLC output  $i$  of **localCom**, output: ( $c_{htlc,i}, ph_{rev,n+1} \vee (pt_{dpay,n+1}, remoteDelay + k + 1 \text{ relative})$ )}
- 8:     ensure **verify**(**localHTLC<sub>n+1,i</sub>**, **HTLCSigs<sub>n+1,i</sub>**,  $pt_{htlc,n+1}$ ) = **true**
- 9:   **end for**
- 10:   **pendingLocalCom**  $\leftarrow$  **localCom**
- 11:   mark **pendingLocalCom** as “irrevocably committed”
- 12:    $prand_{n+2} \leftarrow \text{PRF}(\text{seed}, n + 2)$
- 13:   ( $sh_{com,n+2}, ph_{com,n+2}$ )  $\leftarrow$  **KeyShareGen**( $1^k$ ;  $prand_{n+2}$ )
- 14:   send (REVOKEANDACK,  $pchid$ ,  $prand_n$ ,  $ph_{com,n+2}$ ) to *Bob*

**Fig. 14.**

**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:   ensure  $pk(st_{com,n}) = pt_{com,n}$
- 4:   mark **remoteCom** $_{n+1}$  as “irrevocably committed”
- 5:   **localCom** $_{n+1} \leftarrow$  **pendingLocalCom**
- 6:   unmark **channel**
- 7:    $sh_{rev,n} \leftarrow \text{CombineKey}(shb_{rev}, phb_{rev}, st_{com,n}, pt_{com,n})$
- 8:    $ph_{rev,n+2} \leftarrow \text{CombinePubKey}(phb_{rev}, pt_{com,n+2})$
- 9:    $pt_{rev,n+2} \leftarrow \text{CombinePubKey}(ptb_{rev}, ph_{com,n+2})$
- 10:    $ph_{dpay,n+2} \leftarrow \text{PubKeyGen}(phb_{dpay}, ph_{com,n+2})$
- 11:    $pt_{dpay,n+2} \leftarrow \text{PubKeyGen}(ptb_{dpay}, pt_{com,n+2})$
- 12:    $ph_{pay,n+2} \leftarrow \text{PubKeyGen}(phb_{pay}, ph_{com,n+2})$
- 13:    $pt_{pay,n+2} \leftarrow \text{PubKeyGen}(ptb_{pay}, pt_{com,n+2})$
- 14:    $ph_{htlc,n+2} \leftarrow \text{PubKeyGen}(phb_{htlc}, ph_{com,n+2})$
- 15:    $pt_{htlc,n+2} \leftarrow \text{PubKeyGen}(ptb_{htlc}, pt_{com,n+2})$

**Fig. 15.**

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

- 1: Upon receiving (PUSH,  $pchid$ ) from  $\mathcal{E}$ :
- 2:   ensure that there is a **channel**  $\in$  **channels** with ID  $pchid$
- 3:   choose a member (HTLC\_no,  $R$ ) of **pendingFulfills** $_{pchid}$  that is both in an “irrevocably committed” **remoteCom** $_n$  and **localCom** $_n$
- 4:   remove (HTLC\_no,  $R$ ) from **pendingFulfills** $_{pchid}$
- 5:   send (UPDATEFULFILLHTLC,  $pchid$ , HTLC\_no,  $R$ ) to  $pchid$  counterparty

**Fig. 16.**

**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** $_{n,i}$ ,  $sh_{\text{htlc},n}$ )
- 7:     add (**sig**, **HTLCSigs** $_{n,i}$ , **localHTLC** $_{n,i}$ ) to HTLCs
- 8:   **end for**
- 9:   **sig**  $\leftarrow$  **signature**(**localCom** $_n$ ,  $sh_F$ )
- 10:   remove **channel** from **channels**
- 11:   send (SUBMIT, (**sig**, **remoteSig** $_n$ , **localCom** $_n$ ), HTLCs) to  $\mathcal{G}_{\text{Ledger}}$

**Fig. 17.**





## 6 Payment Network Functionality

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

*Parameters:*

- one-way, collision-free hash function  $\mathcal{H}$  (for generating transaction IDs)

*Interface (messages from  $\mathcal{E}$ ):* **check**

- (REGISTER)
- (SETDELAY, delay)
- (OPENCHANNEL, self, peer, selfCoins)
- (CLOSECHANNEL, receipt)
- (PAY, peer, coins, path, receipt)

*Initialisation:*

- 1: Initialisation:
- 2:    **channels**, **pendingPay**, **pendingOpen**, **corrupted**  $\leftarrow \emptyset$
- 3: Upon receiving (REGISTER, delay, relayDelay) from *Alice*:
- 4:    **delay**(*Alice*)  $\leftarrow$  delay
- 5:    **relayDelay**(*Alice*)  $\leftarrow$  relayDelay
- 6:    **pendingUpdates**(*Alice*), **newChannels**(*Alice*)  $\leftarrow \emptyset$
- 7:    **negligent**(*Alice*), **relayNegligent**(*Alice*)  $\leftarrow \emptyset$
- 8:    send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and assign largest block number to **lastPoll**(*Alice*)
- 9:    send (REGISTER, *Alice*, delay, relayDelay, lastPoll) to  $\mathcal{S}$
- 10: Upon receiving (REGISTERDONE, *Alice*, pubKey) from  $\mathcal{S}$ :
- 11:    **pubKey**(*Alice*)  $\leftarrow$  pubKey
- 12:    send (REGISTER, *Alice*, **delay**, **relayDelay**, pubKey) to *Alice*
- 13: Upon receiving (REGISTERED) from *Alice*:
- 14:    send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and assign reply to  $\Sigma_{\text{Alice}}$
- 15:    assign the sum of all output values that are exclusively spendable by *Alice* to **onChainBalance**
- 16:    send (REGISTERED) to *Alice*
- 17: Upon receiving any message except for (REGISTER) from *Alice*:
- 18:    ignore message if *Alice* has not registered
- 19: Upon receiving (CORRUPTED, *Alice*) from  $\mathcal{S}$ :
- 20:    add *Alice* to **corrupted**
- 21: At the end of each activation: **[Orfeas: can this part completely go?]**
- 22:    verify onChainbalance() for all parties is consistent with ledger (if not roll back the state and ignore command of activation).

Fig. 18.

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

- 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 (CHECKNEW, *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 assign reply to  $\Sigma_{\text{Alice}}$
- 13:   ensure that there is a TX  $F \in \Sigma_{\text{Alice}}$  with a  $(x, (p_{\text{funder},F} \wedge p_{\text{fundee},F}))$  output such that  $\mathcal{H}(F) = \textit{pchid}$
- 14:   mark **pendingOpen**(*fchid*) with "*Alice* checked"
- 15:   **if** **pendingOpen**(*fchid*) is not marked with "noted" **then**
- 16:     mark **pendingOpen**(*fchid*) with "noted"
- 17:     **if** *funder* = *Alice* **then**
- 18:       offChainBalance(*Alice*)  $\leftarrow$  offChainBalance(*Alice*) + *x*   **[Orfeas: remove on/offChainBalance?]**
- 19:       onChainBalance(*Alice*)  $\leftarrow$  offChainBalance(*Alice*) - *x*
- 20:       **channel**  $\leftarrow$  (*Alice*, *Bob*, *x*, 0, 0, *fchid*, *pchid*)
- 21:     **else** // *Bob* is the funder
- 22:       offChainBalance(*Bob*)  $\leftarrow$  offChainBalance(*Bob*) + *x*   **[Orfeas: remove on/offChainBalance?]**
- 23:       onChainBalance(*Bob*)  $\leftarrow$  offChainBalance(*Bob*) - *x*
- 24:       **channel**  $\leftarrow$  (*Bob*, *Alice*, *x*, 0, 0, *fchid*, *pchid*)
- 25:     **end if**
- 26:     add **channel** to **channels**
- 27:   **end if**
- 28:   add receipt(**channel**) to newChannels(*Alice*)
- 29:   **if** **pendingOpen**(*fchid*) is marked with "*Alice* checked" and "*Bob* checked" **then**
- 30:     clear **pendingOpen**(*fchid*) entry
- 31:   **end if**
- 32:   send (CHANNELOPENED, *Alice*, *fchid*) to  $\mathcal{S}$

**Fig. 19.**

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

- 1: Upon receiving  $(\text{PAY}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt})$  from *Alice*:
- 2:   ensure that  $(\text{Alice}, c) \in \text{receipt}$  and  $c \geq x$
- 3:   ensure that there is a  $\text{channel} \in \text{channels} : \text{receipt}(\text{channel}) = \text{receipt}$
- 4:   ensure that  $\overrightarrow{\text{path}}$  consists of  $\text{channels} \in \text{channels}$
- 5:   ensure that each consecutive pair of channels in  $\overrightarrow{\text{path}}$  has a common member
- 6:   choose unique payment ID  $\text{payid}$
- 7:   add  $(\text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{payid})$  to  $\text{pendingPay}$
- 8:   send  $(\text{PAY}, \text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt}, \text{payid})$  to  $\mathcal{S}$
  
- 9: Upon receiving  $(\text{RESOLVEPAY}, \text{payid}, \text{Charlie})$  from  $\mathcal{S}$ :
- 10:   retrieve  $(\text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}})$  with ID  $\text{payid}$  and remove it from  $\text{pendingPay}$
- 11:   **if**  $(\text{Charlie} \neq \text{Alice} \text{ and } \text{Charlie} \notin \text{corrupted})$  or  $\text{Charlie} \notin \overrightarrow{\text{path}}$  **then**
- 12:     halt
- 13:   **end if**
- 14:   **for all**  $\text{channels} \in \overrightarrow{\text{path}}$  starting from the one where *Charlie* pays **do**
- 15:     in the first iteration, *Charlie* is **payer**. In subsequent iterations, **payer** is the unique player that has received but has not given. The other **channel** party is **payee**
- 16:     **if** **payer** has  $x$  or more in **channel** **then**
- 17:       update **channel** to the next version and transfer  $x$  from **payer** to **payee**
- 18:       add  $\text{receipt}(\text{channel})$  to both parties'  $\text{pendingUpdates}$
- 19:     **else**
- 20:       revert all updates and remove them from  $\text{pendingUpdates}$
- 21:     **end if**
- 22:   **end for**
- 23:    $\text{offChainBalance}(\text{Charlie}) \leftarrow \text{offChainBalance}(\text{Charlie}) - x$
- 24:    $\text{offChainBalance}(\text{Bob}) \leftarrow \text{offChainBalance}(\text{Bob}) + x$

Fig. 20.

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

- 1: Upon receiving (CLOSECHANNEL, **receipt**) from *Alice* [Aggelos: (or  $\mathcal{S}$ ) ?? ]  
[Orfeas: @Aggelos: why  $\mathcal{S}$ ?]
- 2:   ensure that there is a **channel**  $\in$  **channels** : **receipt**(**channel**) = **receipt**
- 3:   retrieve *fchid* from **channel**
- 4:   **pendingClose**(*fchid*)  $\leftarrow$  *Alice*
- 5:   send (CLOSECHANNEL, *fchid*, *Alice*) to  $\mathcal{S}$
  
- 6: Upon receiving (CHANNELCLOSED, *fchid*) from  $\mathcal{S}$ :
- 7:   *Alice*  $\leftarrow$  **pendingClose**(*fchid*)
- 8:   retrieve *Charlie*, *Bob*, *x*, *y*, *pchid* from **channel** with ID *fchid*
- 9:   ensure that *Charlie* = *Alice*
- 10:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as *Alice* and assign reply to  $\Sigma_{\text{Alice}}$
- 11:   ensure that transaction with ID *pchid* is in  $\Sigma_{\text{Alice}}$ , is spent, *x* of its coins  
are spendable or will be spendable exclusively by *Alice* and *y* of its coins are  
spendable exclusively by *Bob*
- 12:   **pendingClose**(*fchid*)  $\leftarrow \perp$
- 13:   add receipt of **channel** to **closedChannels**(*Bob*)
- 14:   remove **channel** from **channels**
- 15:   **onChainBalance**(*Alice*)  $\leftarrow$  **onChainBalance**(*Alice*) + *x*
- 16:   **onChainBalance**(*Bob*)  $\leftarrow$  **onChainBalance**(*Bob*) + *y*
- 17:   **offChainBalance**(*Alice*)  $\leftarrow$  **offChainBalance**(*Alice*) - *x*
- 18:   **offChainBalance**(*Bob*)  $\leftarrow$  **offChainBalance**(*Bob*) - *y*
- 19:   send (CHANNELCLOSED, receipt from **channel**) to *Alice*

**Fig. 21.**

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

```

1: Upon receiving (POLL) from Alice:
2:   toReport (Alice)  $\leftarrow \emptyset$ 
3:   send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as Alice and assign reply to  $\Sigma_{\text{Alice}}$ 
4:   assign largest block number in  $\Sigma_{\text{Alice}}$  to  $t$ 
5:   if  $\text{lastPoll}(\text{Alice}) + \text{delay}(\text{Alice}) < t$  then
6:     add  $[\text{lastpoll}(\text{Alice}), t - \text{delay}(\text{Alice}) - 1]$  to negligent(Alice)
7:   end if
8:   if  $\text{lastPoll}(\text{Alice}) + \text{relayDelay}(\text{Alice}) < t$  then
9:     add  $[\text{lastpoll}(\text{Alice}), t - \text{relayDelay}(\text{Alice}) - 1]$  to
    relayNegligent(Alice)
10:  end if
11:   $\text{lastPoll}(\text{Alice}) \leftarrow t$ 
12:  scan  $\Sigma_{\text{Alice}}$  for honestly closed channels that contain Alice and exist in
    channels (txs that spend funding txs that have the same channel version as
    stored), remove them from channels and add them to toReport(Alice)
    (marked as “honest”)
13:  scan  $\Sigma_{\text{Alice}}$  for maliciously closed channels that contain Alice and exist in
    channels (txs that spend funding txs that have an older channel version than
    stored)
14:  for all maliciously closed channels of which the spending txs can still be
    spent by Alice do // If Alice is negligent, she may be unable to punish
15:    if Alice cannot spend those spending txs and Alice has not been
    negligent in the last interval then
16:      halt
17:    end if
18:    add channel to toReport(Alice) (marked as “malicious”)
19:  end for
20:  send (GETCLOSEDFUNDS, toReport(Alice), Alice) to  $\mathcal{S}$ 

21: Upon receiving (CHANNELSCLOSED, details, Alice) from  $\mathcal{S}$ :
22:  send (READ) to  $\mathcal{G}_{\text{Ledger}}$  as Alice and assign reply to  $\Sigma_{\text{Alice}}$ 
23:  for all channel  $\in$  details do
24:    ensure channel  $\in$  toReport (Alice)
25:    if channel is marked as “malicious” then
26:      ensure that transactions that spend the funding tx of channel and
      pay Alice the entire channel value exist in  $\Sigma_{\text{Alice}}$ 
27:    else // channel is marked as “honest”
28:      ensure that transactions that spend the funding tx of channel and
      pay Alice the her part in channel exist in  $\Sigma_{\text{Alice}}$ 
29:      ensure that Alice has not suffered losses for multi-hop payments
      where she was not the payer
30:    end if
31:    add the receipt of channel to closedChannels(Alice)
32:    remove channel from channels and toReport(Alice)
33:  end for

```

**Fig. 22.**

**Functionality  $\mathcal{F}_{\text{PayNet}}^{\text{miscellaneous}}$**

- 1: Upon receiving (PUSH,  $pchid$ ) from *Alice*:
- 2:     send (PUSH,  $pchid$ , *Alice*) to  $\mathcal{S}$
- 3: Upon receiving (COMMIT,  $pchid$ ) from *Alice*:
- 4:     send (COMMIT,  $pchid$ , *Alice*) to  $\mathcal{S}$
- 5: Upon receiving (GETNEW) from *Alice*:
- 6:     clear **newChannels**(*Alice*), **closedChannels**(*Alice*), **pendingUpdates**(*Alice*)  
      and send them to *Alice*

**Fig. 23.**

## 7 Security Proof

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

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

**Fig. 24.**

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

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

**Fig. 25.**

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

*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  $\mathcal{F}_{\text{PayNet}, \text{dummy}}$ . Thus the two outputs are indistinguishable.  $\square$

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

- 1: For messages REGISTER, REGISTERDONE and REGISTERED, act like  $\mathcal{F}_{\text{PayNet}}$ .
- 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  $(M, \text{Alice})$  to  $\mathcal{S}$
- 5:   **end if**
- 6: Upon receiving any other message  $(M, \text{Alice})$  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. 26.**



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

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

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

**Fig. 27.**

**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,  $pk_{\text{Alice}}$ ) where  $pk_{\text{Alice}}$  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. 27, line 5).

Furthermore, one (READ) is sent to  $\mathcal{G}_{\text{Ledger}}$  from *Alice* in both cases (Fig. 1, line 8 and Fig. 18, line 8).

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. 27.

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.  $\square$

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

- 1: For messages REGISTER, REGISTERDONE, REGISTERED, OPENCHANNEL, CHANNELANNOUNCED and CHECKNEW, act like  $\mathcal{F}_{\text{PayNet}}$ .
- 2: Upon receiving any other message  $M$  from *Alice*:
- 3:   **if**  $M$  is a valid  $\mathcal{F}_{\text{PayNet}}$  message from a player **then**
- 4:     send  $(M, \textit{Alice})$  to  $\mathcal{S}$
- 5:   **end if**
- 6: Upon receiving any other message  $(M, \textit{Alice})$  from  $\mathcal{S}$ :
- 7:   **if**  $M$  is a valid  $\mathcal{F}_{\text{PayNet}}$  message from  $\mathcal{S}$  **then**
- 8:     send  $M$  to *Alice*
- 9:   **end if**

**Fig. 28.**

**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 (CHANNELOPENED, *Alice*, *fchid*) from  $\mathcal{F}_{\text{PayNet,Open}}$ :
- 17:     execute lines 6-14 of Fig. 7 with *Alice*'s ITI
- 18:     **if** *Bob* is honest **then**
- 19:         expect the delivery of *Alice*'s (FUNDINGLOCKED) message from  $\mathcal{A}$
- 20:         simulate Fig. 8 with received message in *Bob*'s ITI
- 21:     **end if**
- 22: Upon receiving (CHECKNEW, *Alice*, *Bob*, *tid*) from  $\mathcal{F}_{\text{PayNet,Open}}$ : // *Alice* should be corrupted
- 23:     send (CHECKNEW, *Alice*, *Bob*, *tid*) as  $\mathcal{E}$  to  $\mathcal{A}$
- 24:     **if** *Bob* is honest **then**
- 25:         expect a (FUNDINGLOCKED) message from  $\mathcal{A}$
- 26:         simulate Fig. 8 with received message in *Bob*'s ITI
- 27:     **end if**

**Fig. 29.**

**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  $(\text{OPENCHANNEL}, \text{Alice}, \text{Bob}, x, fchid, tid)$  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. 19, 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  $\text{OPENCHANNEL}$  or  $\text{CHANNELANNOUNCED}$ ,  $\mathcal{F}_{\text{PayNet,Open}}$  does not send any messages to parties other than  $\mathcal{S}_{\text{LN-Reg-Open}}$ , so we don't have to simulate those.

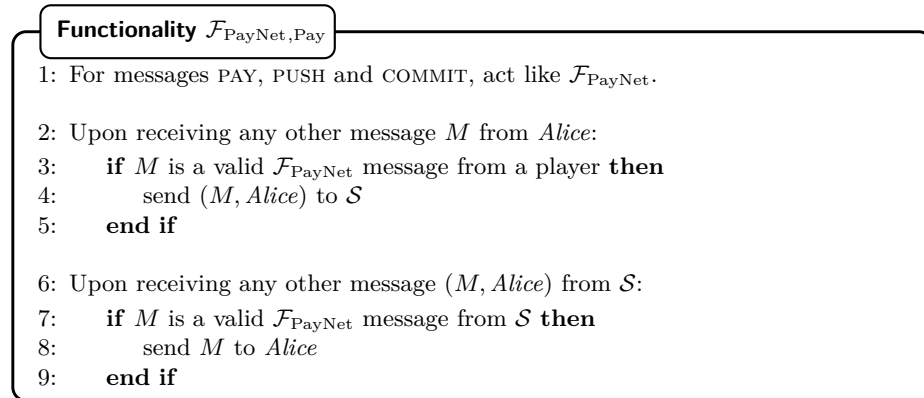
When  $\mathcal{E}$  sends  $(\text{CHECKNEW}, \text{Alice}, \text{Bob}, tid)$  to *Alice* in the real world, line 3 of Fig. 7 will allow execution to continue if there exists an entry with temporary id *tid* in `pendingOpen` marked as “broadcast”. Such an entry can be added either in Fig. 2, line 7 or in Fig. 3, line 6. The former event can happen only in case *Alice* received a valid  $\text{OPENCHANNEL}$  message from *Bob* with temporary id *tid*, which in turn can be triggered only by a valid  $\text{OPENCHANNEL}$  message with the same temporary id from  $\mathcal{E}$  to *Bob*, whereas the latter only in case *Alice* received a valid  $\text{OPENCHANNEL}$  message from  $\mathcal{E}$  with the same temporary id. Furthermore, in the first case the “broadcast” mark can be added only before *Alice* sends  $(\text{FUNDINGSIGNED}, pchid, \text{sig})$  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  $(\text{SUBMIT}, (\text{sig}, F))$  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  $(\text{CHECKNEW}, \text{Alice}, \text{Bob}, tid)$  to *Alice* in the ideal world, line 10 of Fig. 19 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. 19 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  $(\text{CHANNELANNOUNCED}, \text{Alice}, p_{\text{Alice}, F}, p_{\text{Bob}, F}, fchid, pchid, tid)$  from  $\mathcal{S}_{\text{LN-Reg-Open}}$  such that `pendingOpen(fchid)` exists and has temporary id *tid*, as mandated by line 7 of Fig. 19. Such a message is sent by  $\mathcal{S}_{\text{LN-Reg-Open}}$  of Fig. 29 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  $(\text{OPENCHANNEL}, \text{Alice}, \text{Bob}, x, \text{fchid}, \text{tid})$  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  $(\text{OPENCHANNEL}, \text{Bob}, \text{Alice}, x, \text{fchid}, \text{tid})$  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. 19 in the ideal and line 3 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. 19, lines 12-13) and the real world (Fig. 7, lines 4-5).

Moving on, in the real world lines 6-14 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 14-32 and as a result it (always) sends (CHANNELOPENED, *Alice*, *fchid*) to  $\mathcal{S}$ . In turn  $\mathcal{S}$  simulates lines 6-14 of Fig. 7 with *Alice*'s ITI and, if  $\mathcal{A}$  allows it,  $\mathcal{S}$  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.  $\square$



**Fig. 30.**

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

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

- 1: Upon receiving  $(\text{PAY}, \text{Alice}, \text{Bob}, x, \overrightarrow{\text{path}}, \text{receipt}, \text{payid})$  from  $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$ :
- 2: strip  $\text{payid}$ , simulate receiving the message with *Alice* ITI and further execute the parts of the real protocol that correspond to honest parties (Fig. 10- Fig. 12) [Orfeas: submit? 13]
- 3: figure out who paid and name them *Charlie* **continue**
- 4: send  $(\text{RESOLVEPAY}, \text{payid}, \text{Charlie})$  to  $\mathcal{S}_{\text{LN-Reg-Open-Pay}}$

**Fig. 31.**

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

*Proof.*

□

## 8 Combined Sign primitive

### 8.1 Algorithms

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

### 8.2 Correctness

- $\forall k \in \mathcal{N},$   
 $\Pr[(\text{mpk}, \text{msk}) \leftarrow \text{MASTERKEYGEN}(1^k),$   
 $(\text{pk}, \text{sk}) \leftarrow \text{KEYSHAREGEN}(1^k),$   
 $(\text{cpk}_1, \text{csk}_1) \leftarrow \text{COMBINEKEY}(\text{msk}, \text{mpk}, \text{sk}, \text{pk}),$   
 $\text{cpk}_2 \leftarrow \text{COMBINEPUBKEY}(\text{mpk}, \text{pk}),$   
 $\text{cpk}_1 = \text{cpk}_2] = 1$
- $\forall k \in \mathcal{N}, m \in \mathcal{M},$   
 $\Pr[(\text{mpk}, \text{msk}) \leftarrow \text{MASTERKEYGEN}(1^k),$   
 $(\text{pk}, \text{sk}) \leftarrow \text{KEYSHAREGEN}(1^k),$   
 $(\text{cpk}, \text{csk}) \leftarrow \text{COMBINEKEY}(\text{mpk}, \text{msk}, \text{pk}, \text{sk}),$   
 $\text{VERIFY}(\text{cpk}, m, \text{SIGN}(\text{csk}, m)) = 1] = 1$

### 8.3 Security

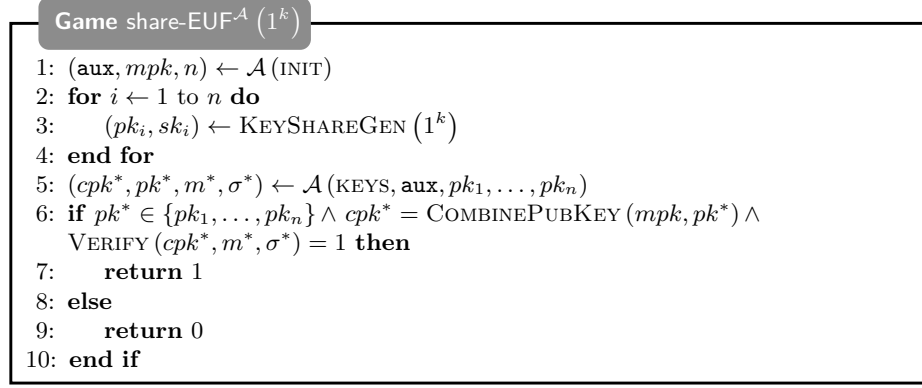


Fig. 32.

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

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

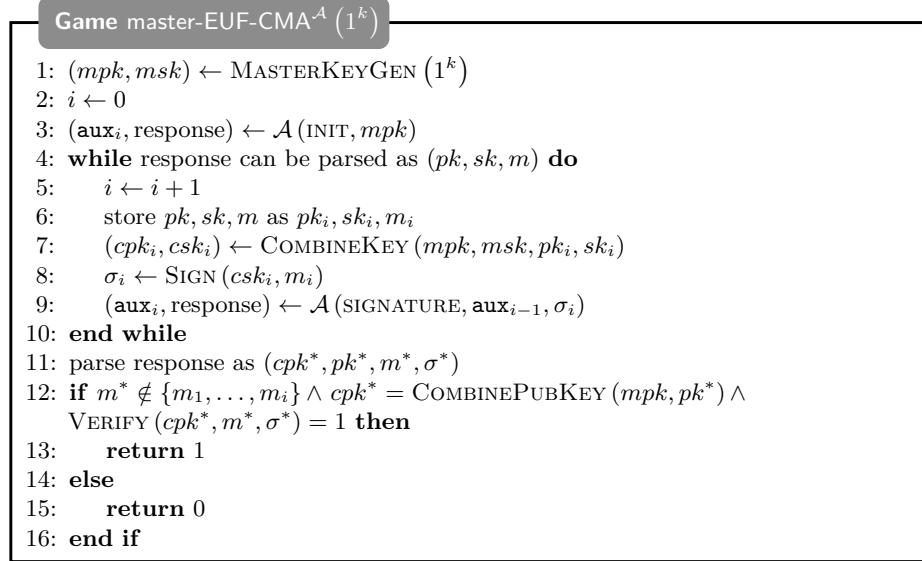


Fig. 33.

**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( $1^k, \text{rand}$ )

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

**end function**

**function** KEYSHAREGEN( $1^k, \text{rand}$ )

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

**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 5.** 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. 34) such that

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

that breaks the assumption, thus proving Lemma 5.

**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:  $(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\text{KEYS}, \text{aux}, pk_1, \dots, pk_n)$ 
24: if  $vk = cpk^* \wedge \mathcal{B}$  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. 34.**

Let  $Y$  be the range of the random oracle. The modified random oracle used in Fig. 34 is indistinguishable from the standard random oracle by PPT algorithms since the statistical distance of the standard random oracle from the modified one is at most  $\frac{1}{2|Y|} < \text{negl}(k)$  as they differ in at most one element.

Let  $E$  denote the event in which  $\mathcal{B}$  does not invoke `COMBINEPUBKEY` to produce  $cpk^*$ . In that case the values  $\mathcal{H}(pk^* \parallel mpk)$  and  $\mathcal{H}(mpk \parallel pk^*)$  are decided after  $\mathcal{B}$  terminates (Fig. 34, line 24) and thus

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

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 `COMBINEPUBKEY` to produce  $cpk^*$  results in the  $j$ th invocation of the Random Oracle. Since  $j$  is chosen uniformly at random,  $\Pr[F \mid \neg E] = \frac{1}{T(\mathcal{B})}$ . Observe that  $\Pr[F \mid E] = 0 \Rightarrow \Pr[F] = \Pr[F \mid \neg E] = \frac{1}{T(\mathcal{B})}$ .

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. 34, 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 6.** *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. 35) 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:  $j \xleftarrow{\$} U[1, T(\mathcal{B}) + T(\mathcal{A})]$  //  $T(M)$  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. 35.**

The modified random oracle used in Fig. 35 is indistinguishable from the standard random oracle for the same reasons as in the proof of Lemma 5.

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. 35, 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 5 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 to 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 5 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 1.** *The construction above is **combine-EUF**-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly **EUF-CMA**-secure.*

*Proof.* The construction is **combine-EUF**-secure as a direct consequence of Lemma 5, Lemma 6 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>.

## References

1. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments