UC security of Lightning Payment Network

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

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

Abstract. 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[3], 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 ($sb_{Alice,\mathrm{pay}}$), htlc ($sb_{Alice,\mathrm{htlc}}$), delayed payment ($sb_{Alice,\mathrm{dpay}}$), revocation ($sb_{Alice,\mathrm{rev}}$) basepoint secrets
  - remote payment ($pb_{Bob,\mathrm{pay}}$), htlc ($pb_{Bob,\mathrm{htlc}}$), delayed payment ($pb_{Bob,\mathrm{dpay}}$), revocation ($pb_{Bob,\mathrm{rev}}$) basepoints
  - seed (for deriving local per commitment keypairs ($s_{Alice,\mathrm{com},n}, p_{Alice,\mathrm{com},n}$))
  - one remote per commitment secret ($\forall i \in [1, \dots, n], s_{Bob,\mathrm{com},i}$) for each COMMITMENTSIGNED received
  - the current and the next remote per commitment points ($p_{Bob,\mathrm{com},n}$ and $p_{Bob,\mathrm{com},n+1}$)
- *Alice*'s coins (integer)
- *Bob*'s coins (integer)

[3] 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 $\texttt{localCom}_n$? (boolean)
  - HTLC number
- signatures:
  - $\forall i \in [1, \dots, n]$, signature of $\texttt{localCom}_i$ generated with $s_{Bob,F}$
  - for every HTLC included in $\texttt{localCom}_i$, if HTLC is outgoing, a signature for `HTLC-timeout`, else a signature for `HTLC-success` with $s_{Bob,\mathrm{htlc},i}$

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 Σ ...
  =
  {
  send (READ) to $\mathcal{G}_{\text{Ledger}}$

upon receiving delayed output Σ ...
}

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

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

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

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

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

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

Fig. 1.

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

1: Upon receiving (openChannel, *Alice*, *Bob*, x, *tid*) from $\mathcal{E}$:
2: ensure *tid* hasn't been used for opening another channel before
3: $\left(\left(ph_F, sh_F\right), \left(phb_{\mathrm{pay}}, shb_{\mathrm{pay}}\right), \left(phb_{\mathrm{dpay}}, shb_{\mathrm{dpay}}\right), \left(phb_{\mathrm{htlc}}, shb_{\mathrm{htlc}}\right), \mathtt{seed}, \left(phb_{\mathrm{rev}}, shb_{\mathrm{rev}}\right)\right) \leftarrow \mathtt{GetKeys}\left(\right)$
4: $\mathrm{prand}_1 \leftarrow \mathtt{PRF}\left(\mathtt{seed}, 1\right)$
5: $\left(sh_{\mathrm{com},1}, ph_{\mathrm{com},1}\right) \leftarrow \mathrm{KeyShareGen}\left(1^k; \mathrm{prand}_1\right)$
6: associate keys with *tid*
7: add $\left(Alice, Bob, x, \mathtt{tid}, \left(ph_F, sh_F\right), \left(phb_{\mathrm{pay}}, shb_{\mathrm{pay}}\right), \left(phb_{\mathrm{dpay}}, shb_{\mathrm{dpay}}\right) \left(phb_{\mathrm{htlc}}, shb_{\mathrm{htlc}}\right), \left(phb_{\mathrm{com},1}, shb_{\mathrm{com},1}\right), \left(phb_{\mathrm{rev}}, shb_{\mathrm{rev}}\right), tid\right)$ to `pendingOpen`
8: send (openChannel, $x, \mathtt{delay} + k + (2 + r)\, \mathtt{windowSize}, ph_F, phb_{\mathrm{pay}}, phb_{\mathrm{dpay}}, phb_{\mathrm{htlc}}, ph_{\mathrm{com},1}, phb_{\mathrm{rev}}$, *tid*) to *Bob*

Fig. 2.

Protocol Π_{LN} - openChannel from *Bob*

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

Fig. 3.

Protocol Π_{LN} - $\textsc{acceptChannel}$

Upon receiving ($\textsc{acceptChannel}$, $\texttt{remoteDelay}, pt_F, ptb_{\text{pay}}, ptb_{\text{dpay}}, ptb_{\text{htlc}}, pt_{\text{com},1}, ptb_{\text{rev}}$, tid) from Bob:

- ensure there is a temporary ID tid with Bob in $\texttt{pendingOpen}$ on which $\textsc{acceptChannel}$ hasn't been received
- associate received keys with tid
- send ($\textsc{read}$) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
- assign to $\texttt{prevout}$ a transaction output found in Σ_{Alice} that is currently exclusively spendable by $Alice$ and has value $y \geq x$
- $F \leftarrow \text{TX}$ {input spends $\texttt{prevout}$ with a $\texttt{signature}(\text{TX}, sk_{Alice})$, output 0 pays $y - x$ to pk_{Alice}, output 1 pays x to $tid.ph_F \wedge pt_F$}
- $pchid \leftarrow \mathcal{H}(F)$
- add $pchid$ to $\texttt{pendingOpen}$ entry with id tid
- $pt_{\text{rev},1} \leftarrow \texttt{CombinePubKey}(ptb_{\text{rev}}, ph_{\text{com},1})$
- $ph_{\text{dpay},1} \leftarrow \texttt{PubKeyGen}(phb_{\text{dpay}}, ph_{\text{com},1})$
- $ph_{\text{pay},1} \leftarrow \texttt{PubKeyGen}(phb_{\text{pay}}, ph_{\text{com},1})$
- $\texttt{remoteCom} \leftarrow \texttt{remoteCom}_1 \leftarrow \text{TX}$ {input: output 1 of F, outputs: $(x, ph_{\text{pay},1}), (0, ph_{\text{rev},1} \vee (pt_{\text{dpay},1}, \texttt{delay} + k + (2 + r)\,\texttt{windowSize}$ relative$))$}
- $\texttt{localCom} \leftarrow \text{TX}$ {input: output 1 of F, outputs: $(x, pt_{\text{rev},1} \vee (ph_{\text{dpay},1}, \texttt{remoteDelay}$ relative$)), (0, pt_{\text{pay},1})$}
- add $\texttt{remoteCom}$ and $\texttt{localCom}$ to channel entry in $\texttt{pendingOpen}$
- $\text{sig} \leftarrow \texttt{signature}(\texttt{remoteCom}_1, sh_F)$
- $\texttt{lastRemoteSigned} \leftarrow 0$
- send ($\textsc{fundingCreated}$, tid, $pchid$, sig) to Bob

Fig. 4.

Protocol Π_{LN} - FUNDINGCREATED

Upon receiving (FUNDINGCREATED, *tid*, *pchid*, $\texttt{BobSig}_1$) from *Bob*:

ensure there is a temporary ID *tid* with *Bob* in $\texttt{pendingOpen}$ on which we have sent up to ACCEPTCHANNEL

$ph_{\text{rev},1} \leftarrow \texttt{CombinePubKey}\left(phb_{\text{rev}} pt_{\text{com},1}\right)$

$pt_{\text{dpay},1} \leftarrow \texttt{PubKeyGen}\left(ptb_{\text{dpay}}, pt_{\text{com},1}\right)$

$pt_{\text{pay},1} \leftarrow \texttt{PubKeyGen}\left(ptb_{\text{pay}}, pt_{\text{com},1}\right)$

$\texttt{localCom} \leftarrow \texttt{localCom}_1 \leftarrow \text{TX}$ {input: output 1 of F, outputs: $\left(x, pt_{\text{pay},1}\right), \left(0, pt_{\text{rev},1} \vee \left(ph_{\text{dpay},1}, \texttt{remoteDelay} \text{ relative}\right)\right)$}

ensure $\texttt{verify}\left(\texttt{localCom}_1, \texttt{BobSig}_1, pt_F\right) = \texttt{True}$

$\texttt{remoteCom} \leftarrow \texttt{remoteCom}_1 \leftarrow \text{TX}$ {input: output 1 of F, outputs: $\left(x, ph_{\text{rev},1} \vee \left(pt_{\text{dpay},1}, \texttt{delay} + k + (2+r)\,\texttt{windowSize} \text{ relative}\right)\right), \left(0, ph_{\text{pay},1}\right)$}

add $\texttt{BobSig}_1, \texttt{remoteCom}_1$ and $\texttt{localCom}_1$ to channel entry in $\texttt{pendingOpen}$

$\text{sig} \leftarrow \texttt{signature}\left(\text{remoteCom}_1, sh_F\right)$

mark channel as “broadcast, no FUNDINGLOCKED”

$\texttt{lastRemoteSigned}, \texttt{lastLocalSigned} \leftarrow 0$

send (FUNDINGSIGNED, *pchid*, sig) to *Bob*

Fig. 5.

Protocol Π_{LN} - FUNDINGSIGNED

Upon receiving (FUNDINGSIGNED, *pchid*, $\texttt{BobSig}_1$) from *Bob*:

ensure there is a channel ID *pchid* with *Bob* in $\texttt{pendingOpen}$ on which we have sent up to FUNDINGCREATED

ensure $\texttt{verify}\left(\texttt{localCom}, \texttt{BobSig}_1, pb_F\right) = \texttt{True}$

$\texttt{localCom}_1 \leftarrow \texttt{localCom}$

$\texttt{lastLocalSigned} \leftarrow 0$

add $\texttt{BobSig}_1$ to channel entry in $\texttt{pendingOpen}$

$\text{sig} \leftarrow \texttt{signature}\left(F, sk_{Alice}\right)$

mark *pchid* in $\texttt{pendingOpen}$ as “broadcast, no FUNDINGLOCKED”

send (SUBMIT, (sig, F)) to $\mathcal{G}_{\text{Ledger}}$

Fig. 6.

Protocol Π_{LN} - CHECKFORNEW

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

Fig. 7.

Protocol Π_{LN} - FUNDINGLOCKED

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

Fig. 8.

Protocol Π_{LN} - poll

Upon receiving (POLL) from $\mathcal{E}$:
    send (READ) to $\mathcal{G}_{\text{Ledger}}$ and assign reply to Σ_{Alice}
    assign largest block number in Σ_{Alice} to `lastPoll`
    `toSubmit` $\leftarrow \emptyset$
    for all $\tau \in$ `unclaimedOfferedHTLCs` **do**
        if input of τ has been spent **then** // by `remote HTLC-success`
            remove τ from `unclaimedOfferedHTLCs`
            if we are intermediary **then**
                retrieve preimage R, *pchid*′ of previous channel on the path of the HTLC, and `HTLCNo`′ of the corresponding HTLC′ in *pchid*′
                add (`HTLCNo`′, R) to $\texttt{pendingFulfills}_{pchid'}$
            end if
        else if input of τ has not been spent and timelock is over **then**
            remove τ from `unclaimedOfferedHTLCs`
            add τ to `toSubmit`
        end if
    end for
    run loop of Fig. 10
    for all honestly closed $\texttt{remoteCom}_n$ that were processed above, with channel id *pchid* **do**
        for all received HTLC outputs i of $\texttt{remoteCom}_n$ **do**
            if there is an entry in $\texttt{pendingFulfills}_{pchid}$ with the same `HTLCNo` and R **then**
                TX $\leftarrow$ {input: i HTLC output of $\texttt{remoteCom}_n$ with $(ph_{\text{htlc},n}, R)$ as method, output: pk_{Alice}}
                sig $\leftarrow$ `signature` $(TX, sh_{\text{htlc},n})$
                add (sig, TX) to `toSubmit`
                remove entry from $\texttt{pendingFulfills}_{pchid}$
            end if
        end for
    end for
    send (SUBMIT, `toSubmit`) to $\mathcal{G}_{\text{Ledger}}$

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

Fig. 9.

Loop over closed channels for poll

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

Fig. 10.

Protocol Π_{LN} - invoice

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

Upon receiving ($\textsc{sendInvoice}$, $payid$) from Bob:
  ensure there is no (Bob, $payid$) entry in $\texttt{pendingGetPaid}$
  choose random, unique preimage R
  add (Bob, R, $payid$) to $\texttt{pendingGetPaid}$
  send $\left(\textsc{invoice}, \mathcal{H}(R), \texttt{relayDelay} + 3k + 2(2+r)\,\texttt{windowSize} - 1, payid\right)$ to Bob

Upon receiving ($\textsc{invoice}$, h, $payid$) from Bob:
  ensure there is a (Bob, x, $\overrightarrow{\texttt{path}}$, $payid$, "waiting for invoice") entry in $\texttt{pendingPay}$
  ensure h is valid (in the range of $\mathcal{H}$)
  remove entry from $\texttt{pendingPay}$
  send ($\textsc{read}$) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign largest block number to t
  $l \leftarrow |\left(\overrightarrow{\texttt{path}}\right)|$
  $m \leftarrow$ the concatenation of $l\,(x, \texttt{OutgoingCltvExpiry})$ pairs, where $\texttt{OutgoingCltvExpiry}_l \leftarrow t, \forall i \in \{1, \ldots, l-1\}, \texttt{OutgoingCltvExpiry}_{l-i} \leftarrow \texttt{OutgoingCltvExpiry}_{l-i+1} + \texttt{CltvExpiryDelta}_{l-i+1}$
  $(\mu_0, \delta_0) \leftarrow \texttt{SphinxCreate}\left(m, \text{ public keys of } \overrightarrow{\texttt{path}} \text{ parties}\right)$
  let $\texttt{remoteCom}_n$ the latest signed remote commitment tx
  $\texttt{CltvExpiry} \leftarrow \texttt{OutgoingCltvExpiry}_1 + \texttt{relayDelay} + 2k + (2+r)\,\texttt{windowSize} - 1$
  reduce simple payment output in $\texttt{remoteCom}$ by x
  add an additional $(x, ph_{\mathrm{rev},n+1} \vee (ph_{\mathrm{htlc},n+1} \wedge pt_{\mathrm{htlc},n+1}, \text{ on preimage of } h) \vee ph_{\mathrm{htlc},n+1}, \texttt{CltvExpiry}$ absolute) output (all with $n+1$ keys) to $\texttt{remoteCom}$, marked with $\texttt{HTLCNo}$
  reduce delayed payment output in $\texttt{localCom}$ by x
  add an additional $(x, pt_{\mathrm{rev},n+1} \vee (pt_{\mathrm{htlc},n+1}, \text{ on preimage of } h) \vee (ph_{\mathrm{htlc},n+1} \wedge pt_{\mathrm{htlc},n+1}, \texttt{CltvExpiry}$ absolute)) output (all with $n+1$ keys) to $\texttt{localCom}$, marked with $\texttt{HTLCNo}$
  increment $\texttt{HTLCNo}_{pchid}$ by one and associate $x, h, pchid$ with it
  mark $\texttt{HTLCNo}$ as "sender"
  send ($\textsc{updateAddHtlc}$, first $pchid$ of $\overrightarrow{\texttt{path}}, \texttt{HTLCNo}_{pchid}, x, h, \texttt{CltvExpiry}, (\mu_0, \delta_0)$) to $pchid$ channel counterparty

Fig. 11.

Protocol Π_{LN} - UPDATEADDHTLC

Upon receiving (UPDATEADDHTLC, *pchid*, $\texttt{HTLCNo}, x, h, \texttt{CltvExpiry}, M$) from *Bob*:

  ensure *pchid* corresponds to an open `channel` in `channels` where *Bob* has at least x

  ensure $\texttt{HTLCNo} = \texttt{HTLCNo}_{pchid} + 1$

  $(pchid', x', \texttt{CltvExpiry}', \delta) \leftarrow \texttt{SphinxPeel}\left(sk_{Alice}, M\right)$

  if $\delta = \texttt{receiver}$ **then**

    ensure $pchid' = \bot, x = x', \texttt{CltvExpiry} \geq \texttt{CltvExpiry}' + \texttt{relayDelay} + 2k + (2 + r)\,\texttt{windowSize} - 1$

    mark `HTLCNo` as "receiver"

  else // We are an intermediary

    ensure $x = x', \texttt{CltvExpiry} \geq \texttt{CltvExpiry}' + \texttt{relayDelay} + 3k + 2\,(2 + r)\,\texttt{windowSize} - 1$

    ensure $\texttt{pchid}'$ corresponds to an open channel in `channels` where we have at least x

    mark `HTLCNo` as "intermediary"

  end if

  increment $\texttt{HTLCNo}_{pchid}$ by one

  let $\texttt{remoteCom}_n$ the latest signed remote commitment tx

  reduce delayed payment output in `remoteCom` by x

  add an $(x, ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1}, \texttt{CltvExpiry}$ absolute$) \vee ph_{\text{htlc},n+1},$ on preimage of $h)$ htlc output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`

  reduce simple payment output in `localCom` by x

  add an $(x, pt_{\text{rev},n+1} \vee pt_{\text{htlc},n+1}, \texttt{CltvExpiry}$ absolute$) \vee ((pt_{\text{htlc},n+1} \wedge ph_{\text{htlc},n+1},$ on preimage of $h))$ htlc output (all with $n+1$ keys) to `remoteCom`, marked with `HTLCNo`

  if $\delta = \texttt{receiver}$ **then**

    retrieve $R : \mathcal{H}(R) = h$ from `pendingGetPaid` and clear entry

    add $(\texttt{HTLCNo}, R)$ to $\texttt{pendingFulfills}_{pchid}$

  else if $\delta \neq \texttt{receiver}$ **then** // Send HTLC to next hop

    retrieve $pchid'$ data

    let $\texttt{remoteCom}'_n$ the latest signed remote commitment tx

    reduce simple payment output in $\texttt{remoteCom}'$ by x

    add an additional $(x, ph_{\text{rev},n+1} \vee (ph_{\text{htlc},n+1} \wedge pt_{\text{htlc},n+1},$ on preimage of $h) \vee ph_{\text{htlc},n+1}\texttt{CltvExpiry}'$ absolute$)$ output (all with $n+1$ keys) to $\texttt{remoteCom}'$, marked with $\texttt{HTLCNo}'$

    reduce delayed payment output in $\texttt{localCom}'$ by x

    add an additional $(x, pt_{\text{rev},n+1} \vee (pt_{\text{htlc},n+1},$ on preimage of $h) \vee (pt_{\text{htlc},n+1} \wedge ph_{\text{htlc},n+1}\texttt{CltvExpiry}'$ absolute$))$ output (all with $n+1$ keys) to $\texttt{remoteCom}'$, marked with $\texttt{HTLCNo}'$

    increment $\texttt{HTLCNo}'$ by 1

    $M' \leftarrow \texttt{SphinxPrepare}\left(M, \delta, sk_{Alice}\right)$

    add $(\texttt{HTLCNo}', x, h, \texttt{CltvExpiry}', M')$ to $\texttt{pendingAdds}_{pchid'}$

  end if

Fig. 12.

Protocol Π_{LN} - UPDATEFULFILLHTLC

Upon receiving (UPDATEFULFILLHTLC, *pchid*, `HTLCNo`, R) from *Bob*:
    if $\texttt{HTLCNo} > \texttt{lastRemoteSigned} \vee \texttt{HTLCNo} > \texttt{lastLocalSigned} \vee \mathcal{H}(R) \neq h$, where h is the hash in the HTLC with number `HTLCNo` **then**
        close channel (as in Fig. 18)
        return
    end if
    ensure `HTLCNo` is an offered HTLC (`localCom` has h tied to a public key that we own)
    add value of HTLC to delayed payment of `remoteCom`
    remove HTLC output with number `HTLCNo` from `remoteCom`
    add value of HTLC to simple payment of `localCom`
    remove HTLC output with number `HTLCNo` from `localCom`
    if we have a channel *phcid*′ that has a received HTLC with hash h with number `HTLCNo′` **then** // We are intermediary
        send (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign reply to Σ_{Alice}
        if latest $\texttt{remoteCom}'_n \in \Sigma_{Alice}$ **then** // counterparty has gone on-chain
            TX ← {input: (`remoteCom′` HTLC output with number `HTLCNo′`, R), output: pk_{Alice}}
            sig ← $\texttt{signature}(\mathrm{TX}, sh_{\mathrm{htlc},n})$
            send (SUBMIT, (sig, TX)) to $\mathcal{G}_{\mathrm{Ledger}}$ // shouldn't be already spent by remote `HTLCTimeout`
        else // counterparty still off-chain
            // Not having the HTLC irrevocably committed is impossible (Fig. 17, l. 15)
            send (UPDATEFULFILLHTLC, *pchid*′, `HTLCNo′`, R) to counterparty
        end if
    end if

Fig. 13.

Protocol Π_{LN} - COMMIT

Upon receiving (COMMIT, *pchid*) from $\mathcal{E}$:
    ensure that there is a `channel` $\in$ `channels` with ID *pchid*
    retrieve latest remote commitment tx $\texttt{remoteCom}_n$ in `channel`
    ensure $\texttt{remoteCom} \neq \texttt{remoteCom}_n$ // there are uncommitted updates
    ensure `channel` is not marked as "waiting for REVOKEANDACK"
    $\texttt{remoteCom}_{n+1} \leftarrow \texttt{remoteCom}$
    $\text{ComSig} \leftarrow \texttt{signature}\left(\texttt{remoteCom}_{n+1}, sh_F\right)$
    $\text{HTLCSigs} \leftarrow \emptyset$
    for i from `lastRemoteSigned` to `HTLCNo` **do**
        $\texttt{remoteHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of $\texttt{remoteCom}_{n+1}$, output: $(c_{\text{htlc,i}}, ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}, \texttt{delay} + k + (2+r)\,\texttt{windowSize}\ \text{relative}))$}
        add $\texttt{signature}\left(\texttt{remoteHTLC}_{n+1,i}, sh_{\text{htlc},n+1}\right)$ to HTLCSigs
    end for
    add $\texttt{signature}\left(\texttt{remoteHTLC}_{n+1,m+1}, sh_{\text{htlc},n+1}\right)$ to HTLCSigs
    $\texttt{lastRemoteSigned} \leftarrow \texttt{HTLCNo}$
    mark `channel` as "waiting for REVOKEANDACK"
    send (COMMITMENTSIGNED, *pchid*, ComSig, HTLCSigs) to *pchid* counterparty

Fig. 14.

Protocol Π_{LN} - COMMITMENTSIGNED

Upon receiving (COMMITMENTSIGNED, *pchid*, $\texttt{comSig}_{n+1}$, $\texttt{HTLCSigs}_{n+1}$) from *Bob*:
  ensure that there is a `channel` $\in$ `channels` with ID *pchid* with *Bob*
  retrieve latest local commitment tx $\texttt{localCom}_n$ in `channel`
  ensure $\texttt{localCom} \neq \texttt{localCom}_n$ and $\texttt{localCom} \neq \texttt{pendingLocalCom}$ // there are uncommitted updates
  if $\texttt{verify}\left(\texttt{localCom}, \texttt{comSig}_{n+1}, pt_F\right) = \texttt{false} \lor |\texttt{HTLCSigs}_{n+1}| \neq \texttt{HTLCNo} - \texttt{lastLocalSigned} + 1$ **then**
    close channel (as in Fig. 18)
    return
  end if
  for i from `lastLocalSigned` to `HTLCNo` **do**
    $\texttt{localHTLC}_{n+1,i} \leftarrow \mathrm{TX}$ {input: HTLC output i of `localCom`, output: $(c_{\mathrm{htlc,i}}, ph_{\mathrm{rev},n+1} \lor (pt_{\mathrm{dpay},n+1}, \texttt{remoteDelay}$ relative$))$}
    if $\texttt{verify}\left(\texttt{localHTLC}_{n+1,i}, \texttt{HTLCSigs}_{n+1,i}, pt_{\mathrm{htlc},n+1}\right) = \texttt{false}$ **then**
      close channel (as in Fig. 18)
      return
    end if
  end for
  $\texttt{pendingLocalCom} \leftarrow \texttt{localCom}$
  mark `pendingLocalCom` as "irrevocably committed"
  $\mathrm{prand}_{n+2} \leftarrow \texttt{PRF}\left(\texttt{seed}, n+2\right)$
  $(sh_{\mathrm{com},n+2}, ph_{\mathrm{com},n+2}) \leftarrow \mathrm{KeyShareGen}\left(1^k; \mathrm{prand}_{n+2}\right)$
  send (REVOKEANDACK, *pchid*, $\mathrm{prand}_n, ph_{\mathrm{com},n+2}$) to *Bob*

Fig. 15.

Protocol Π_{LN} - REVOKEANDACK

Upon receiving (REVOKEANDACK, *pchid*, $st_{\mathrm{com},n}, pt_{\mathrm{com},n+2}$) from *Bob*:

  ensure there is a `channel` $\in$ `channels` with *Bob* with ID *pchid* marked as "waiting for REVOKEANDACK"

  if $pk\left(st_{\mathrm{com},n}\right) \neq pt_{\mathrm{com},n}$ **then** // wrong $st_{\mathrm{com},n}$ - closing

    close channel (as in Fig. 18)

    return

  end if

  mark `remoteCom`$_{n+1}$ as "irrevocably committed"

  `localCom`$_{n+1}$ $\leftarrow$ `pendingLocalCom`

  unmark `channel`

  add `receipt`(`channel`) to `updatesToReport`

  $sh_{\mathrm{rev},n} \leftarrow$ `CombineKey` $\left(shb_{\mathrm{rev}}, phb_{\mathrm{rev}}, st_{\mathrm{com}n}, pt_{\mathrm{com},n}\right)$

  $ph_{\mathrm{rev},n+2} \leftarrow$ `CombinePubKey` $\left(phb_{\mathrm{rev}}, pt_{\mathrm{com},n+2}\right)$

  $pt_{\mathrm{rev},n+2} \leftarrow$ `CombinePubKey` $\left(ptb_{\mathrm{rev}}, ph_{\mathrm{com},n+2}\right)$

  $ph_{\mathrm{dpay},n+2} \leftarrow$ `PubKeyGen` $\left(phb_{\mathrm{dpay}}, ph_{\mathrm{com},n+2}\right)$

  $pt_{\mathrm{dpay},n+2} \leftarrow$ `PubKeyGen` $\left(ptb_{\mathrm{dpay}}, pt_{\mathrm{com},n+2}\right)$

  $ph_{\mathrm{pay},n+2} \leftarrow$ `PubKeyGen` $\left(phb_{\mathrm{pay}}, ph_{\mathrm{com},n+2}\right)$

  $pt_{\mathrm{pay},n+2} \leftarrow$ `PubKeyGen` $\left(ptb_{\mathrm{pay}}, pt_{\mathrm{com},n+2}\right)$

  $ph_{\mathrm{htlc},n+2} \leftarrow$ `PubKeyGen` $\left(phb_{\mathrm{htlc}}, ph_{\mathrm{com},n+2}\right)$

  $pt_{\mathrm{htlc},n+2} \leftarrow$ `PubKeyGen` $\left(ptb_{\mathrm{htlc}}, pt_{\mathrm{com},n+2}\right)$

Fig. 16.

Protocol Π_{LN} - PUSH

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

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

Upon receiving (FULFILLONCHAIN) from $\mathcal{E}$:
  send (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ and assign largest block number to t
  `toSubmit` $\leftarrow \emptyset$
  for all channels **do**
    if there exists an HTLC in latest $\texttt{localCom}_n$ for which we have sent both UPDATEFULFILLHTLC and COMMITMENTSIGNED to a transaction without that HTLC to counterparty, but have not received the corresponding REVOKEANDACK AND the HTLC expires within $[t, t + k + (2 + r)\,\texttt{windowSize}]$ **then**
      add $\texttt{localCom}_n$ of the channel and all corresponding valid `HTLC-success`es and `HTLC-timeout`s (for both $\texttt{localCom}_n$ and $\texttt{remoteCom}_n$[a]), along with their signatures to `toSubmit`
    end if
  end for
  send (SUBMIT, `toSubmit`) to $\mathcal{G}_{\mathrm{Ledger}}$

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

Fig. 17.

Protocol Π_{LN} - close

Upon receiving ($\textsc{closeChannel}$, $\texttt{receipt}$) from $\mathcal{E}$:

    ensure $\texttt{receipt}$ corresponds to an open $\texttt{channel} \in \texttt{channels}$

    assign latest $\texttt{channel}$ sequence number to n

    $\mathrm{HTLCs} \leftarrow \emptyset$

    for every HTLC output $\in \texttt{localCom}_n$ with number i **do**

        $\mathrm{sig} \leftarrow \texttt{signature}\left(\texttt{localHTLC}_{n,i}, sh_{\mathrm{htlc},n}\right)$

        add $\left(\mathrm{sig}, \texttt{HTLCSigs}_{n,i}, \texttt{localHTLC}_{n,i}\right)$ to HTLCs

    end for

    $\mathrm{sig} \leftarrow \texttt{signature}\left(\texttt{localCom}_n, sh_F\right)$

    add $\texttt{receipt}(\texttt{channel})$ to $\texttt{closedChannels}$

    remove $\texttt{channel}$ from $\texttt{channels}$

    send ($\textsc{submit}$, $\left(\mathrm{sig}, \texttt{remoteSig}_n, \texttt{localCom}_n\right)$, HTLCs) to $\mathcal{G}_{\mathrm{Ledger}}$

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, $\overrightarrow{\texttt{path}}$, `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*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*)
  - (UPDATE, `receipt`, *Alice*)
  - (RESOLVEPAYS, *payid*, `charged`)
- to $\mathcal{S}$:
  - (REGISTER, *Alice*, delay, relayDelay, lastPoll)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*)
  - (CHANNELOPENED, *Alice*, *fchid*)
  - (PAY, *Alice*, *Bob*, x, $\overrightarrow{\texttt{path}}$, `receipt`, *payid*)
  - (CONTINUE)
  - (CLOSECHANNEL, *fchid*, *Alice*)
  - (POLL, Σ_{Alice}, *Alice*)
  - (PUSHFULFILL, *pchid*, *Alice*)
  - (PUSHADD, *pchid*, *Alice*)
  - (COMMIT, *pchid*, *Alice*)
  - (FULFILLONCHAIN, t, *Alice*)

Fig. 19.

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

Initialisation:

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

Upon receiving ($\textsc{register}$, delay, relayDelay) from *Alice*:

    $\texttt{delay}\,(Alice) \leftarrow$ delay // Must check chain at least once every $\texttt{delay}(Alice)$ blocks

    $\texttt{relayDelay}\,(Alice) \leftarrow$ relayDelay

    $\texttt{updatesToReport}\,(Alice)\,, \texttt{newChannels}\,(Alice) \leftarrow \emptyset$

    $\texttt{polls}\,(Alice) \leftarrow \emptyset$

    $\texttt{focs}\,(Alice) \leftarrow \emptyset$

    send ($\textsc{read}$) to $\mathcal{G}_{\text{Ledger}}$ as *Alice*, store reply to Σ_{Alice}, add Σ_{Alice} to Σ and add largest block number to $\texttt{polls}(Alice)$

    $\texttt{checkClosed}(\Sigma_{Alice})$

    send ($\textsc{register}$, *Alice*, delay, relayDelay, lastPoll) to $\mathcal{S}$

Upon receiving ($\textsc{registerDone}$, *Alice*, pubKey) from $\mathcal{S}$:

    $\texttt{pubKey}\,(Alice) \leftarrow$ pubKey

    send ($\textsc{register}$, *Alice*, $\texttt{delay}(Alice)$, $\texttt{relayDelay}(Alice)$, pubKey) to *Alice*

Upon receiving ($\textsc{registered}$) from *Alice*:

    send ($\textsc{read}$) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}

    $\texttt{checkClosed}(\Sigma_{Alice})$

    assign the sum of all output values that are exclusively spendable by *Alice* to $\texttt{onChainBalance}$

    send ($\textsc{registered}$) to *Alice*

Upon receiving any message except for ($\textsc{register}$) from *Alice*:

    ignore message if *Alice* has not registered

Upon receiving ($\textsc{corrupted}$, *Alice*) from $\mathcal{S}$:

    add *Alice* to $\texttt{corrupted}$

    for the rest of the execution, upon receiving any message for *Alice*, bypass normal execution and simply forward it to $\mathcal{S}$

Fig. 20.

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

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

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

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

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

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

Fig. 21.

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

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

2: choose unique payment ID *payid*

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

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

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

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

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

Fig. 22.

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

Upon receiving (resolvePays, $\texttt{charged}$) from $\mathcal{S}$: // after first sending pay, pushFulfill, pushAdd, commit
    for all *Alice* keys $\in \texttt{charged}$ **do**
        for all $(Dave, payid) \in \texttt{charged}(Alice)$ **do**
            retrieve $\left(Alice, Bob, x, \overrightarrow{\texttt{path}}\right)$ with ID *payid* and remove it from $\texttt{pendingPay}$
            if $Dave = \perp$ **then**
                continue with next loop iteration
            end if
            calculate $\texttt{IncomingCltvExpiry}$, $\texttt{OutgoingCltvExpiry}$ of *Dave* (as in Fig. 11, l. 18)
            if $Dave \neq Alice \land Dave \notin \texttt{corrupted} \land$
$((\Sigma_{Dave}$ contains in block $h_{\texttt{tx}}$ an old $\texttt{remoteCom}_m$ that does not contain the HTLC and a tx that spends the delayed output of $\texttt{remoteCom}_m \land$
$\texttt{polls}(Dave)$ contains an element in $[h_{\texttt{tx}} + k, h_{\texttt{tx}} + k + \texttt{delay}(Dave) - 1]) \lor$
$(\Sigma_{Dave}$ does not contain an old $\texttt{remoteCom}_m \land \texttt{polls}(Dave)$ contains two elements in $[\texttt{OutgoingCltvExpiry} + k + (2 + r)\,\texttt{windowSize} + 1, \texttt{IncomingCltvExpiry} - k - (2 + r)\,\texttt{windowSize}]$ that have a difference of at least $k + (2 + r)\,\texttt{windowSize} \land$
$\texttt{focs}(Dave)$ contains $\texttt{IncomingCltvExpiry} - k - (2 + r)\,\texttt{windowSize} \land$
the element in $\texttt{polls}(Dave)$ was added before the element in $\texttt{focs}(Dave)))$ **then**
                halt
            end if
            run code of Fig. 24
            if $Dave \notin \texttt{corrupted}$ **then**
                $\texttt{offChainBalance}(Dave) \leftarrow \texttt{offChainBalance}(Dave) - x$
            end if
            $\texttt{offChainBalance}(Bob) \leftarrow \texttt{offChainBalance}(Bob) + x$
        end for
    end for

Fig. 23.

Loop over payment hops for update and check

for all open `channel`s $\in \overrightarrow{\texttt{path}}$ that are not in any `closedChannels`, starting from the one where *Dave* pays **do**
    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`
    if `payer` has x or more in `channel` **then**
        update `channel` to the next version and transfer x from `payer` to `payee`
    else
        revert all updates done in this loop
    end if
end for
for all updated `channel`s in the previous loop **do**
    ensure that a corresponding element has been added to the `updatesToReport` of each honest counterparty, otherwise halt
end for

Fig. 24.

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

Upon receiving (closeChannel, `receipt`) from *Alice*
    ensure that there is a `channel` $\in$ `channels` : `receipt` (`channel`) = `receipt`
    retrieve *fchid* from `channel`
    add (*fchid*, `receipt`(`channel`), $\perp$) to `pendingClose`(*Alice*)
    do not serve any other (pay or closeChannel) message from *Alice* for this channel
    send (closeChannel, `receipt`, *Alice*) to $\mathcal{S}$

function `checkClosed`(Σ_{Alice}) // Called after every (read), ensures requested closes eventually happen
    for all entries ($fchid$, `receipt`, h) $\in$ `pendingClose`(*Alice*) $\cup$ `closedChannels`(*Alice*) **do**
        if there is a closing transaction in Σ_{Alice} for open `channel` with ID *fchid* with a balance that corresponds to `receipt` **then**
            let x, y the balances of *Alice* and the `channel` counterparty *Bob* respectively
            offChainBalance (*Alice*) $\leftarrow$ offChainBalance (*Alice*) $+ x$
            onfChainBalance (*Alice*) $\leftarrow$ onChainBalance (*Alice*) $- x$
            offChainBalance (*Bob*) $\leftarrow$ offChainBalance (*Bob*) $+ y$
            onChainBalance (*Bob*) $\leftarrow$ onChainBalance (*Bob*) $- y$
            remove `channel` from `channels`
            remove entry from `pendingClose`(*Alice*)
            if there is an (*fchid*, _, _) entry in `pendingClose`(*Bob*) **then**
                remove it from `pendingClose`(*Bob*)
            end if
        else if there is a closing transaction in block of height h in Σ_{Alice} for open `channel` with ID *fchid* with a balance that does not correspond to the `receipt` and the delayed output has been spent by the counterparty **then**
            if `polls`(*Alice*) contains an entry in $[h + k, h + k + \texttt{delay}(Alice) - 1]$ **then**
                halt
            end if
        else if there is no such closing transaction $\wedge\ h = \perp$ **then**
            assign largest block number of Σ_{Alice} to h of entry
        else if there is no such closing transaction $\wedge\ h \neq \perp \wedge$ (largest block number of Σ_{Alice}) $\geq h + (2 + r)\,\texttt{windowSize}$ **then**
            halt
        end if
    end for
end function

Fig. 25.

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

Upon receiving (POLL) from *Alice*:

 send (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}

 add largest block number in Σ_{Alice} to `polls`(*Alice*)

 `checkClosed`(Σ_{Alice})

 for all `channel`$s \in \Sigma_{Alice}$ that contain *Alice* and are maliciously closed by a remote commitment `tx` (one with an older channel version than the irrevocably committed one) in block with height $h_{\texttt{tx}}$ **do**

  if the delayed output (of the counterparty) has been spent AND `polls`(*Alice*) has an element in $[h_{\texttt{tx}} + k, h_{\texttt{tx}} + k + \texttt{delay}(Alice) - 1]$ **then**

   halt // *Alice* wasn't negligent but couldn't punish - bad event

  end if

 end for

 send (POLL, Σ_{Alice}, *Alice*) to $\mathcal{S}$

Fig. 26.

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

Upon receiving (pushFulfill, *pchid*) from *Alice*:
    send (pushFulfill, *pchid*, *Alice*, state, Σ) to $\mathcal{S}$

Upon receiving (pushAdd, *pchid*) from *Alice*:
    send (pushAdd, *pchid*, *Alice*, state, Σ) to $\mathcal{S}$

Upon receiving (commit, *pchid*) from *Alice*:
    send (commit, *pchid*, *Alice*, state, Σ) to $\mathcal{S}$

Upon receiving (fulfillOnChain) from *Alice*:
    send (read) to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice*, store reply to Σ_{Alice} and assign largest block number to t
    `checkClosed`(Σ_{Alice})
    add t to `focs`(*Alice*)
    send (fulfillOnChain, t, *Alice*) to $\mathcal{S}$

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

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

Fig. 27.

7 Security Proof

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

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

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

Fig. 28.

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

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

Fig. 29.

Lemma 1. $\textsc{Exec}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}} = \textsc{Exec}^{\mathcal{F}_{\text{PayNet,dummy}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN}},\mathcal{E}}$

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

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

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

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

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

Fig. 30.

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

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

Upon receiving (REGISTER, *Alice*, delay, relayDelay, lastPoll) from $\mathcal{F}_{\mathrm{PayNet,Reg}}$:
  `delay` of *Alice* ITI $\leftarrow$ delay
  `relayDelay` of *Alice* ITI $\leftarrow$ relayDelay
  `lastPoll` of *Alice* ITI $\leftarrow$ lastPoll
  (pk_{Alice}, sk_{Alice}) of *Alice* ITI $\leftarrow$ `KeyGen`()
  send (REGISTERDONE, *Alice*, pk_{Alice}) to $\mathcal{F}_{\mathrm{PayNet,Reg}}$

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

Fig. 31.

Lemma 2. $\textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,dummy}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN}},\mathcal{E}} = \textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,Reg}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN-Reg}},\mathcal{E}}$

Proof. When $\mathcal{E}$ sends (REGISTER, delay, relayDelay) to *Alice*, it receives as a response (REGISTER, *Alice*, delay, relayDelay, pk_{Alice}) where pk_{Alice} is a public key generated by `KeyGen`() both in the real (c.f. Fig. 1, line 9) and in the ideal world (c.f. Fig. 31, 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, $\mathcal{S}_{\text{LN}-\text{Reg}}$ ensures that the state of *Alice* ITI is exactly the same as what would have been in the case of $\mathcal{S}_{\text{LN}}$, as lines 6-9 of Fig. 1 change the state of *Alice* ITI in the same way as lines 2-5 of Fig. 31.

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. □

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

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

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

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

Fig. 32.

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

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

Upon receiving ($\textsc{openChannel}$, *Alice*, *Bob*, x, *fchid*, *tid*) from $\mathcal{F}_{\mathrm{PayNet,Open}}$:
  if both *Alice* and *Bob* are honest **then**
    Simulate the interaction between *Alice* and *Bob* in their respective ITI, as defined in Figures 2-6. All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After sending ($\textsc{fundingSigned}$) as *Bob* to *Alice*, send ($\textsc{channelAnnounced}$, $Bob, p_{Alice,F}, p_{Bob,F}, fchid, pchid, tid$) to $\mathcal{F}_{\mathrm{PayNet,Open}}$.
    After submitting F to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice*, send ($\textsc{channelAnnounced}$, $Alice, p_{Alice,F}, p_{Bob,F}, fchid, pchid$) to $\mathcal{F}_{\mathrm{PayNet,Open}}$.
  else if *Alice* is honest, *Bob* is corrupted **then**
    Simulate *Alice*'s part of the interaction between *Alice* and *Bob* in *Alice*'s ITI, as defined in Figures 2, 4, and 6.All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After submitting F to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice*, send ($\textsc{channelAnnounced}$, $Alice, p_{Alice,F}, p_{Bob,F}, fchid, pchid$) to $\mathcal{F}_{\mathrm{PayNet,Open}}$.
  else if *Alice* is corrupted, *Bob* is honest **then**
    send ($\textsc{openChannel}$, *Alice*, *Bob*, x, *fchid*, *tid*) to simulated (corrupted) *Alice*
    Simulate *Bob*'s part of the interaction between *Alice* and *Bob* in *Bob*'s ITI, as defined in Figures 3 and 5. All messages should be handed to and received from $\mathcal{A}$, as in the real world execution.
    After sending ($\textsc{fundingSigned}$) as *Bob* to *Alice*, send ($\textsc{channelAnnounced}$, $Bob, p_{Alice,F}, p_{Bob,F}, fchid, pchid$) to $\mathcal{F}_{\mathrm{PayNet,Open}}$.
  else if both *Alice* and *Bob* are corrupted **then**
    forward message to $\mathcal{A}$ // $\mathcal{A}$ may open the channel or not
  end if

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

Upon receiving the ($\textsc{fundingLocked}$) message with the simulated *Alice* ITI:
  simulate Fig. 8 receiving the message with *Alice*'s ITI
  send ($\textsc{channelOpened}$, *fchid*) to $\mathcal{F}_{\mathrm{PayNet,Open}}$

Fig. 33.

Lemma 3. $\textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,Reg}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN-Reg}},\mathcal{E}} = \textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,Open}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN-Reg-Open}},\mathcal{E}}$

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 $\mathcal{S}_{\mathrm{LN-Reg-Open}}$, so their state will be identical to that of the parties in the real execution. Furthermore, since $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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 $\mathcal{F}_{\mathrm{PayNet,Open}}$ (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, $\mathcal{F}_{\mathrm{PayNet,Open}}$ does not send any messages to parties other than $\mathcal{S}_{\mathrm{LN-Reg-Open}}$, 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 $\mathcal{G}_{\mathrm{Ledger}}$ (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 $\mathcal{F}_{\mathrm{PayNet,Open}}$ received (channelAnnounced, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*) from $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ such that `pendingOpen`(*fchid*) exists and has temporary id *tid*, as mandated by line 7 of Fig. 21. Such a message is sent by $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ of Fig. 33 either in lines 5/8, or in lines 4/12. One of the first pair of lines is executed only if $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ receives (openChannel, *Bob*, *Alice*, x, *fchid*, *tid*) from $\mathcal{F}_{\text{PayNet,Open}}$ and the simulated $\mathcal{A}$ allows a valid *Alice*-*Bob* interaction up to the point where *Alice* sends (fundingSigned) to *Bob*.

The last two points lead us to deduce that line 10 of Fig. 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 $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$. In turn $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ simulates lines 5-9 of Fig. 7 with *Alice*'s ITI and, if $\mathcal{A}$ allows it, $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ 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. □

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

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()`.

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

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

Fig. 34.

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

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

Upon receiving ($\textsc{fulfillOnChain}, t, \mathit{Alice}$) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    execute lines 20-26 of Fig. 17 as *Alice*, using t from message

Upon receiving $\left(\textsc{pay}, \mathit{Alice}, \mathit{Bob}, x, \overrightarrow{\texttt{path}}, \texttt{receipt}, \mathit{payid}\right)$ from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    add ($\overrightarrow{\texttt{path}}$, *payid*) to **`payids`**
    strip *payid*, simulate receiving the message with *Alice* ITI and further execute the parts of Π_{LN} that correspond to honest parties (Fig. 11- Fig. 13)
    if any "ensure" in Π_{LN} fails until *Bob* processes $\textsc{updateAddHtlc}$ **then** // payment failed
        add ($\perp$, *payid*) to **`charged`**(*Alice*)
        remove ($\overrightarrow{\texttt{path}}$, *payid*) from **`payids`**
    end if

Upon receiving ($\textsc{poll}$, Σ_{Alice}, *Alice*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    simulate Fig. 9, lines 3-28 receiving ($\textsc{poll}$), using Σ_{Alice} from the message, with *Alice*'s ITI

Fig. 35.

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

```
Upon receiving (pushFulfill, pchid, Alice) from F_{PayNet,Pay}:
    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 A the messages for corrupted players

Upon receiving (pushAdd, pchid, Alice) from F_{PayNet,Pay}:
    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 A the messages for corrupted players

Upon receiving (commit, pchid, Alice) from F_{PayNet,Pay}:
    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 A the messages for corrupted players
    if during the simulation above, line 10 of Fig. 16 is simulated in Alice's ITI
    then
        send (update, receipt, Alice) to F_{PayNet,Pay}, where receipt is the
        receipt just added to the simulated updatesToReport (Fig. 16, line 10)
        upon receiving (continue) from F_{PayNet,Pay}, carry on with the
        simulation
    end if
```

Fig. 36.

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

Upon receiving any message with a concatenated (STATE, Σ) part from $\mathcal{F}_{\mathrm{PayNet,Pay}}$: // PAY, PUSHFULFILL, PUSHADD, COMMIT

  handle first part of the message normally

  if at the end of the simulation above, control is still held by $\mathcal{S}_{\mathrm{LN-Reg-Open-Pay}}$ **then**

    for all $\Sigma_{Alice} \in \Sigma$ **do**

      for all $(\overrightarrow{\texttt{path}}, payid) \in \texttt{payids} : Alice \in \overrightarrow{\texttt{path}}$ **do**

        if *Alice* sent UPDATEFULFILLHTLC to a corrupted player and either (got the fulfillment of the HTLC irrevocably committed OR fulfilled the HTLC on-chain (i.e. `HTLC-success` is in Σ_{Alice})), AND the next honest player *Bob* down the line successfully timed out the HTLC on-chain (i.e. `HTLC-timeout` is in Σ_{Bob}) **then** // no or bad communication with *Bob*'s previous player

          add to `charged`(*Alice*) a tuple (`corrupted`, *payid*) where `corrupted` is set to one of the corrupted parties between *Alice* and *Bob*

          remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`

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

          add (*Alice*, *payid*) to `charged`(*Alice*)

          remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`

        else if *Alice* is the payer in $\overrightarrow{\texttt{path}}$ AND ((she has received UPDATEFULFILLHTLC AND has subsequently sent COMMIT and REVOKEANDACK) OR player after *Alice* has irrevocably fulfilled the HTLC on-chain (i.e. his `HTLC-success` is in Σ_{Alice}) **then** // honest payment completed

          add (*Alice*, *payid*) to `charged`(*Alice*)

          remove $(\overrightarrow{\texttt{path}}, payid)$ from `payids`

        end if

      end for

    end for

  end if

  clear `charged` and send (RESOLVEPAYS, `charged`) to $\mathcal{F}_{\mathrm{PayNet,Pay}}$

Fig. 37.

Lemma 4. $\textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,Open}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN-Reg-Open}},\mathcal{E}} \overset{c}{\approx} \textsc{Exec}^{\mathcal{F}_{\mathrm{PayNet,Pay}},\mathcal{G}_{\mathrm{Ledger}}}_{\mathcal{S}_{\mathrm{LN-Reg-Open-Pay}},\mathcal{E}}$

Proof. Before focusing on individual messages sent by $\mathcal{E}$, we will first prove that three particular forgery events happen with negligible probability. Let *Alice* be an honest player. Let P be the event in which at some point during the execution a transaction that has the following two characteristics appears in Σ_{Alice}: (a) it spends a funding transaction of a channel that contains *Alice* (and thus has a $p_{Alice,F}$ public key), or it spends a simple output, delayed output or htlc output tied with a public key that was created by *Alice* ($p_{Alice,\mathrm{pay},n}, p_{Alice,\mathrm{dpay},n}, p_{Alice,\mathrm{htlc},n}$ respectively) and (b) it was never signed by *Alice*. Suppose that $\Pr[P] = a >$ negl. In that case, there is an algorithm (Fig. 43) that breaks the Identity Based Signature Scheme with non-negligible probability, thus breaking the security assumption of the IBS scheme used. Therefore we deduce that $\Pr[P] \leq$ negl.

Let Q be the event in which at some point during the execution a transaction that has the following characteristic appears in Σ_{Alice}: it spends the revocation output of a local (for *Alice*) commitment transaction for a channel that contains *Alice* and *Bob* (and thus has a $p_{Bob,\mathrm{rev},n}$ key). Observe that, since *Alice* is honest and according to both the real and the ideal execution, if *Alice* submits her local commitment transaction $\texttt{localCom}_n$ to the ledger, under no circumstances does she subsequently go on to send $s_{Alice,\mathrm{com},n}$ to any party. (This secret information could be used by *Bob* to efficiently compute $s_{Bob,\mathrm{rev},n}$ with $\textsc{CombineKey}(sb_{Bob,\mathrm{rev}}, pb_{Bob,\mathrm{rev}}, s_{Alice,\mathrm{com},n}, p_{Alice,\mathrm{com},n})$.) If $\Pr[Q] = a >$ negl, there is an algorithm (Fig. 44) that wins the share-EUF game of the Combined Signature Scheme with non-negligible probability, thus breaking the security assumption of the combined signature used. Therefore we deduce that $\Pr[Q] \leq$ negl.

Lastly, let R be the event in which at some point during the execution a transaction that has the following two characteristics appears in Σ_{Alice}: (a) it spends the revocation output of a remote (for *Alice*) commitment transaction for a channel that contains *Alice* (and thus has a $p_{Alice,\mathrm{rev},n}$ key) and (b) it was never signed by *Alice*. Observe that, since *Alice* is honest, she has never sent $s_{Alice,\mathrm{rev},n}$ to any party. Suppose that $\Pr[R] = a >$ negl. In that case, there is an algorithm (Fig. 45) that wins the master-EUF-CMA game of the Combined Signature Scheme with non-negligible probability, thus breaking the security assumption of the combined signature used. Therefore we deduce that $\Pr[R] \leq$ negl. The rest of the proof operates in the world where $\neg P \wedge \neg Q \wedge \neg R$ holds.

We can now move on to treating individual messages sent by $\mathcal{E}$ during the execution. When $\mathcal{E}$ sends $\left(\textsc{pay}, Bob, x, \overrightarrow{\texttt{path}}, payid\right)$ to *Alice* in the ideal world, $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ is always notified (Fig. 22, line 4) and simulates the relevant execution of the real world (Fig. 35, line 5). No messages to $\mathcal{G}_{\mathrm{Ledger}}$ or $\mathcal{E}$ that differ from the real world are generated in the process. At the end of this simulation, no further messages are sent (and the control returns to $\mathcal{E}$). Therefore, when $\mathcal{E}$ sends PAY, no opportunity for distinguishability arises.

When $\mathcal{E}$ sends any message of (PUSHADD, *pchid*), (PUSHFULFILL, *pchid*), (COMMIT, *pchid*) to *Alice* in the ideal world, it is forwarded to $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ (Fig. 27, 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. 36, lines 2, 4, 6). Neither $\mathcal{F}_{\mathrm{PayNet,Pay}}$ nor $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ alter their state as a result of these messages, apart from the state of *Alice*'s simulated ITI and the state of other simulated ITIs that receive and handle messages that were sent as a result of *Alice*'s ITI simulation. The states of these ITIs are modified in the exact same way as they would in the real world. We deduce that these three messages do not introduce any opportunity for $\mathcal{E}$ to distinguish the real and the ideal world.

When $\mathcal{E}$ sends (FULFILLONCHAIN) to *Alice* in the real world, lines 18-26 of Fig. 17 are executed by *Alice*. In the ideal world on the other hand, $\mathcal{F}_{\mathrm{PayNet,Pay}}$ sends (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ (Fig. 27, line 9) as *Alice* and subsequently lets $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ simulate *Alice*'s ITI receiving (FULFILLONCHAIN) (Fig. 35, lines 1-2). Observe that during this simulation a second (READ) message to $\mathcal{G}_{\mathrm{Ledger}}$ (that would not match any message in the real world) is avoided because $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ skips line 19 of Fig. 17, using as t the one received from $\mathcal{F}_{\mathrm{PayNet,Pay}}$ in the message (FULFILLONCHAIN, t, *Alice*). Since $\mathcal{F}_{\mathrm{PayNet,Pay}}$ sends (READ) to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice* and given that after $\mathcal{G}_{\mathrm{Ledger}}$ replies, control is given directly to $\mathcal{S}_{\mathrm{LN-Reg-Open}}$, 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 distinghuishability.

When $\mathcal{E}$ sends (POLL) to *Alice*, the first action is sending (READ) as *Alice* to $\mathcal{G}_{\mathrm{Ledger}}$ both in the ideal (Fig. 26, 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 (POLL) message is sent to $\mathcal{S}_{\mathrm{LN-Reg-Open}}$. We will prove later that $\mathcal{F}_{\mathrm{PayNet,Pay}}$ does not halt here. Upon receiving (POLL), $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ simulates receiving (POLL) with *Alice*'s ITI

(Fig. 35, line 11), but does not READ from $\mathcal{G}_{\text{Ledger}}$ and uses instead the Σ_{Alice} provided along with the message. A reasoning identical to that found in the previous paragraph shows that this Σ_{Alice} is exactly the same as that which *Alice*'s ITI would obtain had it executed line 2 of Fig. 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. 26 and halts. We will now prove that, during $\textsc{Exec}^{\mathcal{F}_{\text{PayNet,Pay}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S}_{\text{LN-Reg-Open-Pay}},\mathcal{E}}$, it is $\Pr[E] = 0$. The condition of Fig. 26, line 6 is triggered if the delayed output (that of the malicious party) of $\texttt{tx}_1$ has been spent by the transaction $\texttt{tx}_2$ in Σ_{Alice} (event E_1) and $\texttt{polls}(Alice)$ contains an element in $[h_1 + k, h_1 + k + \texttt{delay}\,(Alice) - 1]$, where h_1 is the block height where $\texttt{tx}_1$ is (event E_2). Observe that $E = E_1 \wedge E_2$. We note that the elements in $\texttt{polls}(Alice)$ correspond to the block heights of Σ_{Alice} at the moments when *Alice* POLLs (Fig. 26, line 3). Consider the following two events: $E_{1,1}$: $\texttt{tx}_2$ spends the delayed output with a signature valid by the delayed payment public key after the locktime expires. $E_{1,2}$: $\texttt{tx}_2$ spends the delayed output with a signature valid by the revocation public key $p_{Alice,\text{rev}}$. Note that $E_1 = E_{1,1} \vee E_{1,2}$ and $E_{1,1}, E_{1,2}$ are mutually exclusive (since the same output cannot be spent twice). Observe that $E_{1,2} \subset R$, thus $\Pr[E_{1,2}|\neg R] = 0$. We now concetrate on the event $E_{1,1}$. Due to the fact that $\texttt{tx}_2$ spends an output locked with a relative timelock of length $\texttt{delay}\,(Alice) + k + (2 + r)\,\texttt{windowSize}$, the commitment transaction $\texttt{tx}_1$ can reside in a block of maximum height $h_1 \leq h_2 - \texttt{delay}\,(Alice) - k - (2 + r)\,\texttt{windowSize}$, where h_2 is the block height where $\texttt{tx}_2$ is. If *Alice* POLLs on a moment when $|\Sigma_{Alice}| \geq h_1 + k, \Sigma_{Alice}$ necessarily contains $\texttt{tx}_1$. Furthermore, if *Alice* POLLs on a moment when $|\Sigma_{Alice}| \leq h_1 + k + \texttt{delay}\,(Alice) - 1 \leq h_2 - (2 + r)\,\texttt{windowSize} - 1$, she sees $\texttt{tx}_1$ and directly submits the punishment transaction $\texttt{tx}_3$ (which she has, given that a maliciously closed channel is defined as one where the non-closing party has the punishment transaction) (Fig. 10, lines 19-21). Given that $\texttt{tx}_3$ is broadcast when $|\Sigma_{Alice}| \leq h_2 - (2 + r)\,\texttt{windowSize} - 1$, it is guaranteed to be on-chain in a block $h_3 \leq h_2 - 1$ (according to Proposition 1). Since $\texttt{tx}_3$ spends the same funds as $\texttt{tx}_2$, the two cannot be part of the chain simultaneously. Since $E_{1,1} \Rightarrow \Sigma_{Alice}$ contains $\texttt{tx}_2$ and $E_2 \Rightarrow \Sigma_{Alice}$ contains $\texttt{tx}_3$, $E_{1,1}$ and E_2 are mutually exclusive. Therefore $\Pr[E] = \Pr[(E_{1,1} \vee E_{1,2}) \wedge E_2] = \Pr[(E_{1,1} \wedge E_2) \vee (E_{1,2} \wedge E_2)] \leq \Pr[E_{1,1} \wedge E_2] + \Pr[E_{1,2} \wedge E_2] = \Pr[E_{1,2} \wedge E_2] \leq \Pr[E_{1,2}] = 0$. We conclude that, given $\neg P \wedge \neg Q \wedge \neg R$ POLL introduces no opportunity for distinghuishability.

We now treat the effects of the (STATE, Σ) message that $\mathcal{F}_{\mathrm{PayNet,Pay}}$ sends to $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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. 37, line 2). It may be the case that at the end of the handling of line 2, $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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 $\left(\text{PAY}, Bob, x, \overrightarrow{\texttt{path}}\right)$ and Bob is corrupted), or if the handling of the message results in sending (SUBMIT) to $\mathcal{G}_{\mathrm{Ledger}}$ (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. 37, line 3) and does not influence execution in any way.

In the case when (STATE, Σ) is handled, $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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 $\mathcal{G}_{\mathrm{Ledger}}$ state (Fig. 37, lines 4-17). The resolution is then sent to $\mathcal{F}_{\mathrm{PayNet,Pay}}$ with the message (RESOLVEPAYS, `charged`). $\mathcal{F}_{\mathrm{PayNet,Pay}}$ handles this message in Fig. 23, where, if it does not halt (line 15), it updates the state of each affected channel (Fig. 24, line 4) and does not send any message, thus control returns to $\mathcal{E}$. Therefore we have to prove that $\mathcal{F}_{\mathrm{PayNet,Pay}}$ halts with at most negligible probability in order to conclude that the handling of a (STATE) message does not introduce opportunity for distinguishability.

$\mathcal{F}_{\mathrm{PayNet,Pay}}$ halts (Fig. 23, line 15) 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 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 14) – i.e. halts in the event $(A \wedge B) \vee (\neg B \wedge C \wedge D)$. $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ 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. 37, line 9), – i.e. *Dave* is charged in the event $B \vee (F \wedge G)$.

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

$(\neg B \wedge C \wedge D)) \wedge (B \vee (F \wedge G))|\neg P \wedge \neg Q \wedge \neg R] = 0$. This last step holds because $(A \wedge B) \vee ((C \wedge D) \wedge (F \wedge G)) = (A \wedge B) \vee (C \wedge D \wedge F \wedge G)$ and $((A \wedge B) \vee (\neg B \wedge C \wedge D)) \wedge (B \vee (F \wedge G)) = (A \wedge B) \vee (\neg B \wedge C \wedge D \wedge F \wedge G)$ and the latter is a subset of the former.

The analysis of the event $A \wedge B$ is identical to the one we did previously for the events E_1, E_2, with A corresponding to E_2 and B to E_1. We thus deduce that $\Pr[A \wedge B|\neg P \wedge \neg Q \wedge \neg R] = 0$.

The only way for event C to be true is if $\mathcal{E}$ sends (POLL) to *Dave* during the prescribed time period (Fig. 26, 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), $\mathcal{F}_{\text{PayNet,Pay}}$ always sends (GETCLOSEDFUNDS) to $\mathcal{S}_{\text{LN}-\text{Reg}-\text{Open}}$ (Fig. 26, line 10) (since, as we saw earlier, $\mathcal{F}_{\text{PayNet,Pay}}$ never halts).

Event G happens only when the previous counterparty successfully appends `HTLC-timeout` to Σ_{Dave}, which is a valid transaction only from the block of height $\texttt{IncomingCltvExpiry} + 1$ and on, or if the previous counterparty learns the preimage of the HTLC and forges a signature valid by *Dave*'s public HTLC key, or if the previous counterparty forges a signature valid by *Dave*'s public revocation key; the two latter scenarios can never happen. Thus, given that F happens until a moment when $|\Sigma_{Dave}| \leq \texttt{IncomingCltvExpiry} - k - (2 + r)\,\texttt{windowSize}$, *Dave* has the time to successfully fulfill the HTLC. Given C, *Dave* has POLLed at two moments $h_1, h_2 \in [\texttt{OutgoingCltvExpiry} + k + (2 + r)\,\texttt{windowSize} + 1, \texttt{IncomingCltvExpiry} - k - (2 + r)\,\texttt{windowSize}]$, such that $h_2 \geq h_1 + k + (2 + r)\,\texttt{windowSize}$. If Σ_{Dave} contains the preimage at moment h_1 or h_2, then *Dave* may try to update the previous channel off-chain if he receives a (PUSHFULFILL) for that channel (Fig. 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 Σ_{Dave} by block height $h_1 +$

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

It remains to be proven that the halt of line 10 in Fig. 24 does not occur with non-negligible probability. Indeed, $\mathcal{S}$ only reports the payment as resolved in RESOLVEPAYS if a party has been irrevocably charged for it (Fig. 37, lines 6, 9, or 12). In all three cases, all channels that follow the `charged` party on the $\overrightarrow{\texttt{path}}$ have either been closed or irrevocably updated to a newer version that includes the new balance. Since $\mathcal{F}_{\text{PayNet}}$ may only halt for a `channel` that has not been declared or confirmed as closed (Fig. 24, lines 1 and 9), all channels that can cause a halt are channels that have the update of this payment irrevocably committed. This only happens when both sides send a REVOKEANDACK that updates the channel from a version that contains the relevant HTLC to a version that doesn't; and when an honest party receives such a REVOKEANDACK message, it logs the update in `updatesToReport` (Fig. 16, line 10) which causes $\mathcal{S}$ to report the update to $\mathcal{F}_{\text{PayNet}}$ (Fig. 36, line 8). We therefore conclude that $\mathcal{F}_{\text{PayNet}}$ never halts on line 10 of Fig. 24. $\square$

Simulator $\mathcal{S}$

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

Upon receiving (CLOSECHANNEL, `receipt`, *Alice*) from $\mathcal{F}_{\text{PayNet}}$:
    simulate Fig. 18 receiving (CLOSECHANNEL, `receipt`) with *Alice*'s ITI

every time `closedChannels` of *Alice* is updated with data from a `channel` (Fig. 18, line 10 and Fig. 10, line 23), send (CLOSEDCHANNEL, `channel`, *Alice*) to $\mathcal{F}_{\text{PayNet}}$ and expect (CONTINUE) from $\mathcal{F}_{\text{PayNet}}$ to resume simulation

Fig. 38.

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

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

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. 25, line 5). Similarly in the real world, if there has been another such message, *Alice* ignores it (Fig.18, lines 11 and 2).

In the case that it is indeed the first closing message, in the ideal world $\mathcal{F}_{\text{PayNet}}$ takes note that this close is pending (Fig. 25, 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. 38). 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. 27, 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. 33, 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. 33, 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. 27, line 13), which in turn happens exactly when the simulated *Alice* ITI adds the `channel` to her `closedChannels` (Fig. 38, 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. 36, 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. 27, 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. 36, 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.25, 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 never happens.

$\mathcal{F}_{\text{PayNet}}$ 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. 26, 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 themself. This however can never happen, since this event would be a subset of either R, if $\texttt{remoteCom}_n$ (i.e. the counterparty closed the channel) is in Σ_{Alice}, or Q, if $\texttt{localCom}_n$ is in Σ_{Alice} (i.e. *Alice* closed the channel).

$\mathcal{F}_{\text{PayNet}}$ halts in line 27 of Fig. 25 in case $\mathcal{E}$ has asked for the channel to close, but too much time has passed since. This event cannot happen, for two reasons. First, regarding elements in `pendingClose`(*Alice*), because $\mathcal{F}_{\text{PayNet}}$ forwards a CLOSECHANNEL message to $\mathcal{S}$ (Fig. 25, line 6) for every element that it adds to `pendingClose` (Fig 25, line 4) and this causes $\mathcal{S}$ to submit the closing transaction to $\mathcal{G}_{\text{Ledger}}$ (Fig. 18, line 12). This transaction is necessarily valid, because there is no other transaction that spends the funding transaction of the channel, according to the first check of line 26 of Fig. 25. $\mathcal{F}_{\text{PayNet}}$ halts in this case only if it is sure that the chain has grown by $(2+r)$ `windowSize` blocks, and thus if the closing transaction had been submitted when closeChannel was received, it should have been necessarily included (Proposition 1). Second, every element added to `closedChannels` (Fig. 18, line 10 and Fig. 10, line 23) corresponds to a submission of a closing transaction for the same channel (Fig. 18, line 12), or to a channel for which the closing transaction is already in the ledger state (Fig. 10, line 1). In both cases, the transaction has been submitted at least $(2+r)$ `windowSize` blocks earlier, thus again by Proposition 1 it is impossible for the transaction not to be in the ledger state. Therefore $\mathcal{F}_{\text{PayNet}}$ cannot halt in line 27 of Fig. 25. We deduce that, given $\neg P \wedge \neg Q \wedge \neg R$, the execution of `checkClosed` by $\mathcal{F}_{\text{PayNet}}$ does not contribute any increase to the probability of distinguishability. □

Theorem 1 (Lightning Payment Network Security).

$$\textsc{Exec}^{\mathcal{G}_{\text{Ledger}}}_{\Pi_{\text{LN}},\mathcal{A}_{\text{d}},\mathcal{E}} \overset{c}{\approx} \textsc{Exec}^{\mathcal{F}_{\text{PayNet}},\mathcal{G}_{\text{Ledger}}}_{\mathcal{S},\mathcal{E}}$$

Proof. The theorem is a direct result of Lemmas 1-5. □

8 Combined Sign primitive

8.1 Algorithms

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

8.2 Correctness

- $\forall k \in \mathbb{N}$,
  $\Pr[(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right),$
  $(pk, sk) \leftarrow \textsc{KeyShareGen}\left(1^k\right),$
  $(cpk_1, csk_1) \leftarrow \textsc{CombineKey}\left(msk, mpk, sk, pk\right),$
  $cpk_2 \leftarrow \textsc{CombinePubKey}\left(mpk, pk\right),$
  $cpk_1 = cpk_2] = 1$

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

8.3 Security

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

$(\mathsf{aux}, mpk, n) \leftarrow \mathcal{A}\left(\textsc{init}\right)$
for $i \leftarrow 1$ to n **do**
    $(pk_i, sk_i) \leftarrow \textsc{KeyShareGen}\left(1^k\right)$
end for
$(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{A}\left(\textsc{keys}, \mathsf{aux}, pk_1, \ldots, pk_n\right)$
if $pk^* \in \{pk_1, \ldots, pk_n\} \wedge cpk^* = \textsc{CombinePubKey}\left(mpk, pk^*\right) \wedge$ $\textsc{Verify}\left(cpk^*, m^*, \sigma^*\right) = 1$ **then**
    return 1
else
    return 0
end if

Fig. 39.

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

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

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

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

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

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

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

Fig. 40.

Definition 3. *A Combined Sign scheme is* master-EUF-CMA-secure *if*

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

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

Definition 4. *A Combined Sign scheme is* master-EUF-CMA-secure *if*

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

Definition 5. *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, $G \cdot$ rand)
end function

function $\texttt{KeyShareGen}(1^k, \text{rand})$
  Return $(\text{rand}, G \cdot \text{rand})$
end function
function $\texttt{CombineKey}(msk, mpk, sk, pk)$
  return $msk \cdot \mathcal{H}\left(mpk \,\|\, pk\right) + sk \cdot \mathcal{H}\left(pk \,\|\, mpk\right)$
end function
function $\texttt{CombinePubKey}(mpk, pk)$
  return $mpk \cdot \mathcal{H}\left(mpk \,\|\, pk\right) + pk \cdot \mathcal{H}\left(pk \,\|\, mpk\right)$
end function
function $\texttt{Sign}(csk, m)$
  like standard sign
end function
function $\texttt{Verify}(cpk, m, \sigma)$
  like standard verify
end function

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

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

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

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

that breaks the assumption, thus proving Lemma 6.

Algorithm $\mathcal{A}(vk)$

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

Fig. 41.

Let Y be the range of the random oracle. The modified random oracle used in Fig. 41 is indistinguishable from the standard random oracle by PPT algorithms since the statistical distance of the standard random oracle from the modified one is at most $\frac{1}{2|Y|} < negl(k)$ as they differ in at most one element.

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

are decided after $\mathcal{B}$ terminates (Fig. 41, line 24) and thus

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

It is

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

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

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

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

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

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

Since F holds, the jth invocation of the Random Oracle queried either $\mathcal{H}\left(mpk \parallel pk^*\right)$ or $\mathcal{H}\left(pk^* \parallel mpk\right)$. In either case (Fig. 41, lines 9-18), it is $cpk^* = vk$. This means that $\textsc{Verify}\left(vk, m^*, \sigma^*\right) = 1$. We conclude that in the event $\left(F \wedge \mathcal{B} \text{ wins} \wedge \neg E\right)$, $\mathcal{A}$ wins the EUF-CMA game. A final observation is that the probability that the events $\left(\mathcal{B} \text{ wins} \wedge \neg E\right)$ and F are almost independent, thus

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

$\square$

Lemma 7. *The construction above is* $\mathsf{master\text{-}EUF\text{-}CMA}$*-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly* $\mathsf{EUF\text{-}CMA}$*-secure and the range of the Random Oracle coincides with that of the underlying signature scheme signing keys.*

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

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

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 42) such that

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

that breaks the assumption, thus proving Lemma 7.

Algorithm $\mathcal{A}(vk)$

```
j <-$ U[1, T(B) + T(A)]   // T(M) is the maximum running time of M
    Random Oracle: for every first-seen query q from B set H(q) to a random value
    return H(q) to B
(mpk, msk) <- MasterKeyGen(1^k)
    Random Oracle: Let q be the jth first-seen query from B or A:
    if q = (mpk || x) then
        if H(x || mpk) unset then
            set H(x || mpk) to a random value
        end if
        set H(mpk || x) to (vk - x · H(x || mpk)) · mpk^{-1}
    else if q = (x || mpk) then
        if H(mpk || x) unset then
            set H(mpk || x) to a random value
        end if
        set H(x || mpk) to (vk - mpk · H(mpk || x)) · x^{-1}
    else
        set H(q) to a random value
    end if
    return H(q) to B or A
i <- 0
(aux_i, response) <- B(INIT, mpk)
while response can be parsed as (pk, sk, m) do
    i <- i + 1
    store pk, sk, m as pk_i, sk_i, m_i
    (cpk_i, csk_i) <- CombineKey(mpk, msk, pk_i, sk_i)
    σ_i <- Sign(csk_i, m_i)
    (aux_i, response) <- B(SIGNATURE, aux_{i-1}, σ_i)
end while
parse response as (cpk*, pk*, m*, σ*)
(cpk*, pk*, m*, σ*) <- B(KEYS, aux, pk_1, ..., pk_n)
if vk = cpk* ∧ B wins the master-EUF-CMA game then   // A won the EUF-CMA game
    return (m*, σ*)
else
    return FAIL
end if
```

Fig. 42.

The modified random oracle used in Fig. 42 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 $\mathcal{H}(pk^* \| mpk)$ and $\mathcal{H}(mpk \| pk^*)$ are decided after $\mathcal{B}$ terminates (Fig. 42, line 31) and thus

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

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

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

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

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

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

□

Theorem 2. *The construction above is combine-EUF-secure in the Random Oracle model under the assumption that the underlying signature scheme is strongly EUF-CMA-secure.*

Proof. The construction is combine-EUF-secure as a direct consequence of Lemma 6, Lemma 7 and the definition of combine-EUF-security. □

9 Forgery algorithms

Algorithm for forging an IBS signature when $\Pr[P] = a >$ negl

Algorithm IBS forgery

1: bla

Fig. 43.

Algorithm for winning the share-EUF game when $\Pr[R] = a >$ negl

Algorithm share-EUF forgery

1: bla

Fig. 44.

Algorithm for winning the master-EUF-CMA game when $\Pr[Q] = a >$ negl

Algorithm master-EUF-CMA forgery

1: bla

Fig. 45.

10 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.

11 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 $\mathtt{pt}_i$ for party U_i below). The functionality has a parameter `windowSize` such that no finalised prefix of any player will be shorter than $|\mathtt{state}| - \mathtt{windowSize}$. 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 $\texttt{state}$ variable is guaranteed via the ExtendPolicy function that is executed when honest parties submit inputs to the functionality. While we do not specify ExtendPolicy in our paper (we refer to the citations above for the full specification) it is sufficient to note that ExtendPolicy guarantees the following properties:

1. in a period of time equal to $\texttt{maxTime}_{\texttt{window}}$, a number of blocks at least $\texttt{windowSize}$ are added to $\texttt{state}$.
2. in a period of time equal to $\texttt{minTime}_{\texttt{window}}$, no more blocks may be added to $\texttt{state}$ if $\texttt{windowSize}$ blocks have been already added.
3. each window of $\texttt{windowSize}$ blocks has at most $\texttt{advBlcks}_{\texttt{window}}$ adversarial blocks included in it.
4. any transaction that (i) is submitted by an honest party earlier than $\frac{\texttt{Delay}}{2}$ rounds before the time that the block that is $\texttt{windowSize}$ positions before the head of the $\texttt{state}$ was included, and (ii) is valid with respect to an honest block that extends $\texttt{state}$, then it must be included in such block.

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

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

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

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

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

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

Party Management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (as discussed below). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a (currently unregistered) honest party is registered at the ledger, *if it is registered with the clock and the global RO already,* then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$.

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

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

1. Updating synchronized/desynchronized party set:
   (a) Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) to the ledger, the clock, and the GRO since time $\tau' < \tau_L - \texttt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
   (b) For any synchronized party $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if U_p is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{U_p\}$.
2. If I was received from an honest party $U_p \in \mathcal{P}$:
   (a) Set $\boldsymbol{\mathcal{I}}_H^T := \boldsymbol{\mathcal{I}}_H^T || (I, U_p, \tau_L)$;

(b) Compute
$\boldsymbol{N} = (\boldsymbol{N}_1, \ldots, \boldsymbol{N}_\ell) := \mathsf{ExtendPolicy}(\boldsymbol{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \boldsymbol{\tau}_{\texttt{state}})$ and if $\boldsymbol{N} \neq \varepsilon$ set $\texttt{state} := \texttt{state}||\mathsf{Blockify}(\boldsymbol{N}_1)||\ldots||\mathsf{Blockify}(\boldsymbol{N}_\ell)$ and $\boldsymbol{\tau}_{\texttt{state}} := \boldsymbol{\tau}_{\texttt{state}}||\tau_L^\ell$, where $\tau_L^\ell = \tau_L||\ldots,||\tau_L$.

(c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then delete $\texttt{BTX}$ from $\texttt{buffer}$. Also, reset $\texttt{NxtBC} := \varepsilon$.

(d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k := |\texttt{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

3. If the calling party U_p is *stalled or time-unaware* (according to the defined party classification), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:

- *Submitting a transaction:*
If $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ and is received from a party $U_p \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party U_p) do the following
  (a) Choose a unique transaction ID txid and set $\texttt{BTX} := (\texttt{tx}, \text{txid}, \tau_L, U_p)$
  (b) If $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 1$, then $\texttt{buffer} := \texttt{buffer} \cup \{\texttt{BTX}\}$.
  (c) Send $(\text{SUBMIT}, \texttt{BTX})$ to $\mathcal{A}$.

- *Reading the state:*
If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\texttt{state}_p := \texttt{state}|_{\min\{\texttt{pt}_p, |\texttt{state}|\}}$ and return $(\text{READ}, \text{sid}, \texttt{state}_p)$ to the requester. If the requester is $\mathcal{A}$ then send $(\texttt{state}, \texttt{buffer}, \boldsymbol{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

- *Maintaining the ledger state:*
If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\boldsymbol{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\boldsymbol{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

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

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

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

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

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

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

Synchronization:

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

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

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., Eckey L., Faust S., Malinowski D.: PERUN: Virtual Payment Channels over Cryptographic Currencies. IACR Cryptology ePrint Archive: vol. 2017, p. 635: URL http://eprint.iacr.org/2017/635 (2017)
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., Gaži P., Kiayias A., Russell A., Zikas V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security: pp. 913–930: ACM (2018)