Payment Channels Overview

Orfeas Stefanos Thyfronitis Litos

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

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

1 State of a channel

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

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

- keys:
  - local funding secret key
  - remote funding public key
  - local {payment, htlc, delayed_payment, revocation}_basepoint_secret
  - remote {payment, htlc, delayed_payment, revocation}_basepoint
  - seed (for local per_commitment_secrets)
  - remote per_commitment_secret$_{1,\dots,m-1}$
  - remote per_commitment_point$_{m,m+1}$
- *Alice*'s coins
- *Bob*'s coins
- 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
  - Is it included in local commitment$_n$?
  - HTLC number
- signatures:
  - signature of local commitment$_n$ with secret key corresponding to remote funding public key

- for every HTLC included in local commitment$_n$, one signature of HTLC-Timeout if outgoing, HTLC-Success if incoming with secret key corresponding to remote htlc_pubkey$_n$ (= htlc_basepoint + $\mathcal{H}$(remote per_commitment_point$_n$ ||remote htlc_basepoint) $\cdot G$)

The rest of the things used in the protocol can be derived by the above.

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

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

I.e. $\text{State}_{Alice,pchid} = (c_{Alice}, c_{Bob}, ((c_1, \text{state}_1), \dots, (c_k, \text{state}_k)))$
E.g. $\text{State}_{Alice,pchid} = (4, 5, ((0.1, \text{proposed}), (-0.2, \text{signed})))$
We do not include in the state elements whose contents are irrelevant (e.g. sigs, keys, hashes).

2 UC conventions

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

find out what fu (\tochain) must be

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

Initialisation:
  `channels`, `pendingOpen`, `pendingPay`, `pendingClose` $\leftarrow \emptyset$
  `newChannels`, `closedChannels` $\leftarrow \emptyset$
  `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`
  $(pk_{Alice}, sk_{Alice}) \leftarrow \mathrm{KeyGen}()$
  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 $\mathcal{E}$

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

Fig. 2.

Protocol Π_{LN} - openChannel from *Bob*

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

Fig. 3.

Protocol Π_{LN} - ACCEPTCHANNEL

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

  ensure there is a temporary ID *tid* with *Bob* in `pendingOpen` on which ACCEPTCHANNEL hasn't been received

  associate received keys with *tid*

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

  assign to `prevout` a transaction output found in Σ_{Alice} that is currently exclusively spendable by *Alice* and has value $y \geq x$

  $F \leftarrow$ TX {input spends `prevout` with a `signature`(TX, sk_{Alice}), output 0 pays $y - x$ to pk_{Alice}, output 1 pays x to $tid.ph_F \wedge pt_F$}

  $pchid \leftarrow \mathcal{H}(F)$

  add *pchid* to `pendingOpen` entry with id *tid*

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

  $ph_{\text{dpay},1} \leftarrow \texttt{PubKeyGen}(phb_{\text{dpay}}, ph_{\text{com},1})$

  $ph_{\text{pay},1} \leftarrow \texttt{PubKeyGen}(phb_{\text{pay}}, ph_{\text{com},1})$

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

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

  add `remoteCom` and `localCom` to channel entry in `pendingOpen`

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

  $\texttt{lastRemoteSigned} \leftarrow 0$

  send (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 `pendingOpen` on which we have sent up to ACCEPTCHANNEL

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

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

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

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

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

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

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

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

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 `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 `pendingOpen`

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

mark *pchid* in `pendingOpen` as "broadcast, no FUNDINGLOCKED"

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

Fig. 6.

Protocol Π_{LN} - CHECKNEW

Upon receiving (CHECKNEW, *Alice*, *Bob*, *tid*) from $\mathcal{E}$: // new message: represents lnd polling daemon

  ensure there is a matching `channel` in `pendingOpen` with id *pchid*, with a "broadcast" mark, funded with x coins

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

  ensure $\exists$ unspent TX in Σ_{Alice} with ID *pchid* and a $(x, ph_F \wedge pt_F)$ output

  $\text{prand}_2 \leftarrow \texttt{PRF}\left(\texttt{seed}, 2\right)$

  $(sh_{\text{com},2}, ph_{\text{com},2}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_2\right)$

  add TX to `channel` data

  replace "broadcast" mark in `channel` with "in state"

  if `channel` is marked as "in state, FUNDINGLOCKED" **then**

    move channel data from `pendingOpen` to `channels`

    add receipt of channel to `newChannels`

  end if

  send (FUNDINGLOCKED, *pchid*, $ph_{\text{com},2}$) to *Bob*

Fig. 7.

Protocol Π_{LN} - FUNDINGLOCKED

Upon receiving (FUNDINGLOCKED, *pchid*, $pt_{\text{com},2}$) from *Bob*:

  ensure there is a `channel` with ID *pchid* with *Bob* in `pendingOpen` with a "no FUNDINGLOCKED" mark

  ensure $pk\left(st_{\text{com},n}\right) = pt_{\text{com},n}$

  replace "no FUNDINGLOCKED" mark in `channel` with "FUNDINGLOCKED"

  ensure `channel` has an "in state" mark

  generate 2nd remote delayed payment, htlc, payment keys

  add TX to `channel` data

  move channel data from `pendingOpen` to `channels`

  add receipt of channel to `newChannels`

Fig. 8.

Protocol Π_{LN} - poll

```
Upon receiving (POLL) from E:
    send (READ) to G_Ledger and assign reply to Σ_Alice
    assign largest block number in Σ_Alice to lastPoll
    toSubmit ← ∅
    for all τ ∈ unclaimedOfferedHTLCs do
        if input of τ has been spent then // by remote HTLC-success
            remove τ from unclaimedOfferedHTLCs
            remember preimage - hash combination
        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
    for all remoteCom_n ∈ Σ_Alice that spend F of a channel ∈ channels do
        if we do not have sh_rev,n then // Honest closure
            for all unspent offered HTLC outputs i of remoteCom_n do
                TX ← {input: i HTLC output of remoteCom_n with ph_htlc,n as method, output: pk_Alice}
                sig ← signature (TX, sh_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 ← TX {inputs: all remoteCom_n outputs, choosing ph_rev,n method, output: pk_Alice}
            sig ← signature (rev, sh_rev,n)
            add (sig, rev) to toSubmit
        end if
        move channel from channels to closedChannels
    end for
    for all honestly closed remoteCom_n that were processed above, with channel id pchid do
        for all received HTLC outputs i of remoteCom_n do
            if there is an entry in pendingFulfills_pchid with the same HTLCNo and R then
                TX ← {input: i HTLC output of remoteCom_n with (ph_htlc,n, R) as method, output: pk_Alice}
                sig ← signature (TX, sh_htlc,n)
                add (sig, TX) to toSubmit
                remove entry from pendingFulfills_pchid
            end if
        end for
    end for
    send (SUBMIT, toSubmit) to G_Ledger

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

Protocol $\mathit{\Pi}_{\mathrm{LN}}$ - invoice

Upon receiving (PAY, Bob, x, $\overrightarrow{\texttt{path}}$) from $\mathcal{E}$:

  ensure that $\overrightarrow{\texttt{path}}$ consists of syntactically valid ($pchid$, `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 `channel` $\in$ `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 `pendingPay`

  send (SENDINVOICE, $payid$) to Bob

Upon receiving (SENDINVOICE, $payid$) from Bob:

  ensure there is no (Bob, $payid$) entry in `pendingGetPaid`

  choose random, unique preimage R

  add (Bob, R, $payid$) to `pendingGetPaid`

  send $(\text{INVOICE}, \mathcal{H}(R), \texttt{relayDelay} + 3k + 2fu - 1, payid)$ to Bob

Upon receiving (INVOICE, h, $payid$) from Bob:

  ensure there is a (Bob, x, $\overrightarrow{\texttt{path}}$, $payid$, "waiting for invoice") entry in `pendingPay`

  ensure h is valid (in the range of $\mathcal{H}$)

  remove entry from `pendingPay`

  send (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

  $\texttt{CltvExpiry} \leftarrow \texttt{OutgoingCltvExpiry}_1 + \texttt{relayDelay} + 2k + fu - 1$

  reduce simple payment output in `remoteCom` by x

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

  reduce delayed payment output in `localCom` by x

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

  increment $\texttt{HTLCNo}_{pchid}$ by one and associate $x, h, pchid$ with it

  mark `HTLCNo` as "sender"

  send (UPDATEADDHTLC, first $pchid$ of $\overrightarrow{\texttt{path}}, \texttt{HTLCNo}_{pchid}, x, h, \texttt{CltvExpiry}, (\mu_0, \delta_0)$) to $pchid$ channel counterparty

Fig. 10.

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

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

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

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

    mark `HTLCNo` as "receiver"

  else // We are an intermediary

    ensure $x = x', \texttt{CltvExpiry} \geq \texttt{CltvExpiry}' + \texttt{relayDelay} + 3k + 2fu - 1$

    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

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

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

  end if

Fig. 11.

Protocol Π_{LN} - updateFulfillHtlc

Upon receiving (updateFulfillHtlc, *pchid*, `HTLCNo`, R) from *Bob*:
if `HTLCNo` $>$ `lastRemoteSigned` $\vee$ `HTLCNo` $>$ `lastLocalSigned` $\vee \mathcal{H}(R) \neq h$, where h is the hash in the HTLC with number `HTLCNo` **then**
close channel (as in Fig. 17)
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 `remoteCom`$'_n \in \Sigma_{Alice}$ **then** // counterparty has gone on-chain
TX $\leftarrow$ {input: (`remoteCom`$'$ HTLC output with number `HTLCNo`$'$, R), output: pk_{Alice}}
sig $\leftarrow$ `signature` (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. 16, l. 15)
send (updateFulfillHtlc, *pchid*$'$, `HTLCNo`$'$, R) to counterparty
end if
end if

Fig. 12.

Protocol Π_{LN} - COMMIT

Upon receiving (COMMIT, *pchid*) from $\mathcal{E}$:

ensure that there is a $\texttt{channel} \in \texttt{channels}$ with ID *pchid*

retrieve latest remote commitment tx $\texttt{remoteCom}_n$ in $\texttt{channel}$

ensure $\texttt{remoteCom} \neq \texttt{remoteCom}_n$ // there are uncommitted updates

ensure $\texttt{channel}$ is not marked as "waiting for REVOKEANDACK"

$\texttt{remoteCom}_{n+1} \leftarrow \texttt{remoteCom}$

$\mathrm{ComSig} \leftarrow \texttt{signature}\left(\texttt{remoteCom}_{n+1}, sh_F\right)$

$\mathrm{HTLCSigs} \leftarrow \emptyset$

for i from $\texttt{lastRemoteSigned}$ to $\texttt{HTLCNo}$ **do**

$\texttt{remoteHTLC}_{n+1,i} \leftarrow$ TX {input: HTLC output i of $\texttt{remoteCom}_{n+1}$, output: $(c_{\mathrm{htlc,i}}, ph_{\mathrm{rev},n+1} \vee (pt_{\mathrm{dpay},n+1}, \texttt{delay} + k + fu$ relative$))$}

add $\texttt{signature}\left(\texttt{remoteHTLC}_{n+1,i}, sh_{\mathrm{htlc},n+1}\right)$ to HTLCSigs

end for

add $\texttt{signature}\left(\texttt{remoteHTLC}_{n+1,m+1}, sh_{\mathrm{htlc},n+1}\right)$ to HTLCSigs

$\texttt{lastRemoteSigned} \leftarrow \texttt{HTLCNo}$

mark $\texttt{channel}$ as "waiting for REVOKEANDACK"

send (COMMITMENTSIGNED, *pchid*, ComSig, HTLCSigs) to *pchid* counterparty

Fig. 13.

Protocol Π_{LN} - commitmentSigned

Upon receiving (commitmentSigned, *pchid*, $\texttt{comSig}_{n+1}$, $\texttt{HTLCSigs}_{n+1}$) from *Bob*:

  ensure that there is a $\texttt{channel} \in \texttt{channels}$ with ID *pchid* with *Bob*

  retrieve latest local commitment tx $\texttt{localCom}_n$ in $\texttt{channel}$

  ensure $\texttt{localCom} \neq \texttt{localCom}_n$ and $\texttt{localCom} \neq \texttt{pendingLocalCom}$ // there are uncommitted updates

  if $\texttt{verify}\left(\texttt{localCom}, \texttt{comSig}_{n+1}, pt_F\right) = \texttt{false} \vee |\texttt{HTLCSigs}_{n+1}| \neq \texttt{HTLCNo} - \texttt{lastLocalSigned} + 1$ **then**

    close channel (as in Fig. 17)

    return

  end if

  for i from $\texttt{lastLocalSigned}$ to $\texttt{HTLCNo}$ **do**

    $\texttt{localHTLC}_{n+1,i} \leftarrow \text{TX}$ {input: HTLC output i of $\texttt{localCom}$, output: $(c_{\text{htlc,i}}, ph_{\text{rev},n+1} \vee (pt_{\text{dpay},n+1}, \texttt{remoteDelay}$ relative$))$}

    if $\texttt{verify}\left(\texttt{localHTLC}_{n+1,i}, \texttt{HTLCSigs}_{n+1,i}, pt_{\text{htlc},n+1}\right) = \texttt{false}$ **then**

      close channel (as in Fig. 17)

      return

    end if

  end for

  $\texttt{pendingLocalCom} \leftarrow \texttt{localCom}$

  mark $\texttt{pendingLocalCom}$ as "irrevocably committed"

  $\text{prand}_{n+2} \leftarrow \texttt{PRF}\left(\texttt{seed}, n+2\right)$

  $(sh_{\text{com},n+2}, ph_{\text{com},n+2}) \leftarrow \text{KeyShareGen}\left(1^k; \text{prand}_{n+2}\right)$

  send (revokeAndAck, *pchid*, $\text{prand}_n, ph_{\text{com},n+2}$) to *Bob*

Fig. 14.

Protocol Π_{LN} - revokeAndAck

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

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

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

    close channel (as in Fig. 17)

    return

  end if

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

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

  unmark `channel`

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

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

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

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

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

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

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

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

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

Fig. 15.

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 ($\texttt{HTLCNo}, x, h, \texttt{CltvExpiry}, M$) of $\texttt{pendingAdds}_{pchid}$ that is both in an "irrevocably committed" $\texttt{remoteCom}_n$ and $\texttt{localCom}_n$

  remove chosen entry from $\texttt{pendingAdds}_{pchid}$

  send (UPDATEADDHTLC, *pchid*, `HTLCNo`, $x, h, \texttt{CltvExpiry}, M$) 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 $[t, 2?k + fu - 1 + t]$ **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. 16.

Protocol Π_{LN} - close

remove receipt?

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

- ensure `receipt` corresponds to an open `channel` $\in$ `channels`
- assign latest `channel` sequence number to n
- HTLCs $\leftarrow \emptyset$
- **for** every HTLC output $\in$ $\texttt{localCom}_n$ with number i **do**
    - 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**
- sig $\leftarrow$ $\texttt{signature}\left(\texttt{localCom}_n, sh_F\right)$
- remove `channel` from `channels`
- send (submit, $\left(\mathrm{sig}, \texttt{remoteSig}_n, \texttt{localCom}_n\right)$, HTLCs) to $\mathcal{G}_{\mathrm{Ledger}}$

Fig. 17.

6 Payment Network Functionality

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

Parameters:

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

Interface: check

– from $\mathcal{E}$:
  - (REGISTER, delay, relayDelay)
  - (REGISTERED)
  - (OPENCHANNEL, *Alice*, *Bob*, x, *tid*)
  - (CHECKNEW, *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, pendingUpdates)
– from $\mathcal{S}$:
  - (REGISTERDONE, *Alice*, pubKey)
  - (CORRUPTED, *Alice*)
  - (CHANNELANNOUNCED, *Alice*, $p_{Alice,F}, p_{Bob,F}$, *fchid*, *pchid*, *tid*)
  - (RESOLVEPAYS, *payid*, `charged`)
  - (CHANNELCLOSED, *fchid*)
  - (CHANNELSCLOSED, `details`, *Alice*, *reportid*)
– 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*)
  - (CLOSECHANNEL, *fchid*, *Alice*)
  - (GETCLOSEDFUNDS, `toReport`, Σ_{Alice}, *Alice*, *reportid*)
  - (PUSHFULFILL, *pchid*, *Alice*)
  - (PUSHADD, *pchid*, *Alice*)
  - (COMMIT, *pchid*, *Alice*)
  - (FULFILLONCHAIN, t, *Alice*)

Fig. 18.

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

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

Upon receiving $(\textsc{register}, \text{delay}, \text{relayDelay})$ from *Alice*:
  $\texttt{delay}\,(Alice) \leftarrow \text{delay}$ // Must check chain at least once every $\texttt{delay}(Alice)$ blocks
  $\texttt{relayDelay}\,(Alice) \leftarrow \text{relayDelay}$
  $\texttt{pendingUpdates}\,(Alice)\,, \texttt{newChannels}\,(Alice) \leftarrow \emptyset$
  $\texttt{polls}\,(Alice) \leftarrow \emptyset$
  $\texttt{focs}\,(Alice) \leftarrow \emptyset$
  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 $\texttt{polls}(Alice)$
  send $(\textsc{register}, Alice, \text{delay}, \text{relayDelay}, \text{lastPoll})$ to $\mathcal{S}$

Upon receiving $(\textsc{registerDone}, Alice, \text{pubKey})$ from $\mathcal{S}$:
  $\texttt{pubKey}\,(Alice) \leftarrow \text{pubKey}$
  send $(\textsc{register}, Alice, \texttt{delay}(Alice), \texttt{relayDelay}(Alice), \text{pubKey})$ to *Alice*

Upon receiving $(\textsc{registered})$ from *Alice*:
  send $(\textsc{read})$ to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
  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 $\texttt{corrupted}$

Fig. 19.

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 (checkNew, *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}
    ensure that there is a TX $F \in \Sigma_{Alice}$ with a $(x, (p_{\text{funder},F} \wedge p_{\text{fundee},F}))$ output such that $\mathcal{H}(F) = pchid$
    mark `pendingOpen`(*fchid*) with "*Alice* checked"
    if `pendingOpen`(*fchid*) is not marked with "noted" **then**
        mark `pendingOpen`(*fchid*) with "noted"
        if funder $=$ *Alice* **then**
            $\text{offChainBalance}(Alice) \leftarrow \text{offChainBalance}(Alice) + x$ **[Orfeas: remove on/offChainBalance?]**
            $\text{onChainBalance}(Alice) \leftarrow \text{offChainBalance}(Alice) - x$
            `channel` $\leftarrow (Alice, Bob, x, 0, 0, fchid, pchid)$
        else// *Bob* is the funder
            $\text{offChainBalance}(Bob) \leftarrow \text{offChainBalance}(Bob) + x$ **[Orfeas: remove on/offChainBalance?]**
            $\text{onChainBalance}(Bob) \leftarrow \text{offChainBalance}(Bob) - x$
            `channel` $\leftarrow (Bob, Alice, x, 0, 0, fchid, pchid)$
        end if
        add `channel` to `channels`
    end if
    add `receipt`(`channel`) to `newChannels`(*Alice*)
    if `pendingOpen`(*fchid*) is marked with "*Alice* checked" and "*Bob* checked" **then**
        clear `pendingOpen`(*fchid*) entry
    end if
    send (channelOpened, *Alice*, *fchid*) to $\mathcal{S}$

Fig. 20.

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

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

2: choose unique payment ID $payid$

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

4: send $\left(\textsc{pay}, Alice, Bob, x, \overrightarrow{\texttt{path}}, payid, \textsc{state}, \Sigma\right)$ to $\mathcal{S}$

Fig. 21.

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

Upon receiving any message with a concatenated (resolvePays, $\texttt{charged}$) from $\mathcal{S}$: // from resolvePay, channelsClosed, from pushFulfill, from pushAdd, from commit

  for all *Charlie* keys $\in \texttt{charged}$ **do**

    for all $(Dave, payid) \in \texttt{charged}\,(Charlie)$ **do**

      retrieve $\left(Alice, Bob, x, \overrightarrow{\texttt{path}}\right)$ with ID *payid* and remove it from $\texttt{pendingPay}$

      calculate $\texttt{IncomingCltvExpiry}$, $\texttt{OutgoingCltvExpiry}$ of *Dave* (as in Fig. 10, l. 18)

      if $Dave = \perp$ **then**

        return

      else if $Dave \neq Alice \vee Dave \notin \texttt{corrupted} \vee (\texttt{polls}(Dave)$ contains an element in $[\texttt{OutgoingCltvExpiry} + 2k + fu - 1, \texttt{IncomingCltvExpiry} - 2?k - fu - 1] \wedge$ $\texttt{focs}(Dave)$ contains $\texttt{OutgoingCltvExpiry} - 2k - fu - 1)$ **then**

        halt

      end if

      for all $\texttt{channels} \in \overrightarrow{\texttt{path}}$ starting from the one where *Dave* pays **do**

        in the first iteration, $\texttt{payer}$ is *Dave*. In subsequent iterations, $\texttt{payer}$ is the unique player that has received but has not given. The other $\texttt{channel}$ party is $\texttt{payee}$

        if $\texttt{payer}$ has x or more in $\texttt{channel}$ **then**

          update $\texttt{channel}$ to the next version and transfer x from $\texttt{payer}$ to $\texttt{payee}$

          add $\texttt{receipt}(\texttt{channel})$ to both parties' $\texttt{pendingUpdates}$

        else

          revert all updates and remove them from $\texttt{pendingUpdates}$

        end if

      end for

      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. 22.

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`
    `pendingClose` (*fchid*) $\leftarrow$ *Alice*
    send (closeChannel, *fchid*, *Alice*) to $\mathcal{S}$

Upon receiving (channelClosed, *fchid*) from $\mathcal{S}$:

    Alice $\leftarrow$ `pendingClose` (*fchid*)
    retrieve *Charlie*, Bob, x, y, *pchid* from `channel` with ID *fchid*
    ensure that *Charlie* = *Alice*
    send (read) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and store reply to Σ_{Alice}
    ensure that transaction with ID *pchid* is in Σ_{Alice}, is spent, x of its coins are spendable or will be spendable exclusively by *Alice* and y of its coins are spendable exclusively by *Bob*
    `pendingClose` (*fchid*) $\leftarrow \perp$
    add receipt of `channel` to `closedChannels`(*Bob*)
    remove `channel` from `channels`
    `onChainBalance` (*Alice*) $\leftarrow$ `onChainBalance` (*Alice*) $+ x$
    `onChainBalance` (*Bob*) $\leftarrow$ `onChainBalance` (*Bob*) $+ y$
    `offChainBalance` (*Alice*) $\leftarrow$ `offChainBalance` (*Alice*) $- x$
    `offChainBalance` (*Bob*) $\leftarrow$ `offChainBalance` (*Bob*) $- y$
    send (channelClosed, receipt from `channel`) to *Alice*

Fig. 23.

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*)
  generate unique *reportid*
  `toReport` (*reportid*) $\leftarrow \emptyset$
  scan Σ_{Alice} for honestly closed channels that contain *Alice* and exist in `channels` (txs that spend funding txs that have the same channel version as stored), remove them from `channels` and add them to `toReport`(*reportid*) (marked as "honest")
  scan Σ_{Alice} for malicioulsy closed channels that contain *Alice* and exist in `channels` (txs that spend funding txs that have an older channel version than stored)
  for all maliciously closed `channel`s by a remote committment `tx` 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
    add `channel` to `toReport`(*reportid*) (marked as "malicious")
  end for
  send (GETCLOSEDFUNDS, `toReport`(*reportid*), Σ_{Alice}, *Alice*, *reportid*, STATE, Σ) to $\mathcal{S}$

Fig. 24.

Functionality $\mathcal{F}_{\mathrm{PayNet}}$- verify closed channels

// Expected after resolutions are visible on-chain

Upon receiving a message that contains (CHANNELSCLOSED, `details`, *Alice*, *reportid*) from $\mathcal{S}$:

  if `toReport`(*reportid*) not in storage **then**

    ignore message

  end if

  the next time (READ) is sent to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice* and just after a reply Σ_{Alice} is received, do the following:

    for all `channel` $\in$ `details` **do**

      ensure `channel` $\in$ `toReport` (*reportid*)

      if `channel` is marked as "malicious" **then**

        ensure that transactions that spend the funding tx of `channel` and pay *Alice* the entire `channel` value exist in Σ_{Alice}, otherwise halt

      else // `channel` is marked as "honest"

        ensure that transactions that spend the funding tx of `channel` and pay *Alice* her part in the latest state of `channel` exist in Σ_{Alice} AND she received funds from all "received HTLCs" for which she knew the preimage when she received the associated POLL AND received funds from all "offered HTLCs" that had timed out when she received the associated POLL, otherwise halt

      end if

      add the receipt of `channel` to `closedChannels`(*Alice*)

      remove `channel` from `channels`

    end for

    remove `toReport`(*reportid*) from storage

  handle the first part of the received message

Fig. 25.

Functionality $\mathcal{F}_{\text{PayNet}}$- 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 $\mathcal{G}_{\text{Ledger}}$ as *Alice*, store reply to Σ_{Alice} and assign largest block number to t
    add t to `focs`(*Alice*)
    send (fulfillOnChain, t, *Alice*) to $\mathcal{S}$

Upon receiving (getNews) from *Alice*:
    clear `newChannels`(*Alice*), `closedChannels`(*Alice*), `pendingUpdates`(*Alice*) and send them to *Alice* with header news

Fig. 26.

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. 27.

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

Expects the same messages as the protocol, but messages that the protocol expects to receive from $\mathcal{E}$, the simulator expects to receive from $\mathcal{F}_{\mathrm{PayNet,dummy}}$ 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 $\mathcal{F}_{\mathrm{PayNet,dummy}}$ instead. To distinguish which player it comes from, $\mathcal{S}$ also appends the player name to the message. add corruption messages here?

Fig. 28.

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

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 $\mathcal{S}_{\mathrm{LN}}$ through $\mathcal{F}_{\mathrm{PayNet,dummy}}$. 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 and REGISTERED, act like $\mathcal{F}_{\mathrm{PayNet}}$.

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. 29.

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:

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

Fig. 30.

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. 30, line 5).

Furthermore, one (READ) is sent to $\mathcal{G}_{\mathrm{Ledger}}$ from *Alice* in both cases (Fig. 1, line 8 and Fig. 19, line 9).

Additionally, $\mathcal{S}_{\mathrm{LN-Reg}}$ ensures that the state of *Alice* ITI is exactly the same as what would have been in the case of $\mathcal{S}_{\mathrm{LN}}$, as lines 6-9 of Fig. 1 change the state of *Alice* ITI in the same way as lines 2-5 of Fig. 30.

Lastly, the fact that the state of the *Alice* ITIs are changed in the same way in both worlds, along with the same argument as in the proof of Lemma 1 ensures that the rest of the messages are responded in an indistinguishable way in both worlds. $\square$

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

1: For messages REGISTER, REGISTERDONE, REGISTERED, OPENCHANNEL, CHANNELANNOUNCED and CHECKNEW, act like $\mathcal{F}_{\text{PayNet}}$.

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

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

Fig. 31.

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

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

Upon receiving (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 (FUNDINGSIGNED) as *Bob* to *Alice*, send (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 (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 (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 (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 (FUNDINGSIGNED) as *Bob* to *Alice*, send (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 (CHANNELOPENED, *Alice*, *fchid*) from $\mathcal{F}_{\mathrm{PayNet,Open}}$:
  execute lines 5-13 of Fig. 7 with *Alice*'s ITI
  if *Bob* is honest **then**
    expect the delivery of *Alice*'s (FUNDINGLOCKED) message from $\mathcal{A}$
    simulate Fig. 8 with received message in *Bob*'s ITI
  end if

maybe remove?

Upon receiving (CHECKNEW, *Alice*, *Bob*, *tid*) from $\mathcal{F}_{\mathrm{PayNet,Open}}$: // *Alice* should be corrupted
  send (CHECKNEW, *Alice*, *Bob*, *tid*) as $\mathcal{E}$ to $\mathcal{A}$
  if *Bob* is honest **then**
    expect a (FUNDINGLOCKED) message from $\mathcal{A}$
    simulate Fig. 8 with received message in *Bob*'s ITI
  end if

Fig. 32.

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

Proof. When $\mathcal{E}$ sends (OPENCHANNEL, *Alice*, *Bob*, x, *fchid*, *tid*) to *Alice*, the interaction of Figures 2-6 will be executed in both the real and the ideal world. In more detail, in the ideal world the execution of the honest parties will be simulated by the respective ITIs run by $\mathcal{S}_{\text{LN-Reg-Open}}$, so their state will be identical to that of the parties in the real execution. Furthermore, since $\mathcal{S}_{\text{LN-Reg-Open}}$ executes faithfully the protocol code, it generates the same messages as would be generated by the parties themselves in the real-world setting.

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

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

When $\mathcal{E}$ sends (CHECKNEW, *Alice*, *Bob*, *tid*) to *Alice* in the real world, line 2 of Fig. 7 will allow execution to continue if there exists an entry with temporary id *tid* in `pendingOpen` marked as "broadcast". Such an entry can be added either in Fig. 2, line 7 or in Fig. 3, line 6. The former event can happen only in case *Alice* received a valid OPENCHANNEL message from *Bob* with temporary id *tid*, which in turn can be triggered only by a valid OPENCHANNEL message with the same temporary id from $\mathcal{E}$ to *Bob*, whereas the latter only in case *Alice* received a valid OPENCHANNEL message from $\mathcal{E}$ with the same temporary id. Furthermore, in the first case the "broadcast" mark can be added only before *Alice* sends (FUNDINGSIGNED, *pchid*, sig) to *Bob* (Fig. 5, line 11) (which needs a valid *Alice*-*Bob* interaction up to that point more in-depth?), and in the second case the "broadcast" mark can be added only before *Alice* sends (SUBMIT, (sig, F)) to $\mathcal{G}_{\text{Ledger}}$ (Fig. 6, line 8) (which also needs a valid *Alice*-*Bob* interaction up to that point more in-depth?)

When $\mathcal{E}$ sends (CHECKNEW, *Alice*, *Bob*, *tid*) to *Alice* in the ideal world, line 10 of Fig. 20 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. 20 is executed, where `pendingOpen`(*fchid*) contains *tid* as temporary id. This line in turn can only be executed if $\mathcal{F}_{\text{PayNet,Open}}$ received (CHANNELANNOUNCED, *Alice*, $p_{Alice,F}$, $p_{Bob,F}$, *fchid*, *pchid*, *tid*) from $\mathcal{S}_{\text{LN-Reg-Open}}$ such that `pendingOpen`(*fchid*) exists and has temporary id *tid*, as mandated by line 7 of Fig. 20. Such a message is sent by $\mathcal{S}_{\text{LN-Reg-Open}}$ of Fig. 32 either in lines 5/8, or in lines 4/12. One of the first pair of lines is executed only if $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ receives (openChannel, *Alice*, *Bob*, x, *fchid*, *tid*) from $\mathcal{F}_{\mathrm{PayNet,Open}}$ and the simulated $\mathcal{A}$ allows a valid *Alice*-*Bob* interaction up to the point where *Alice* sends (submit) to $\mathcal{G}_{\mathrm{Ledger}}$, whereas one of the second pair of lines is executed only if $\mathcal{S}_{\mathrm{LN-Reg-Open}}$ receives (openChannel, *Bob*, *Alice*, x, *fchid*, *tid*) from $\mathcal{F}_{\mathrm{PayNet,Open}}$ 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. 20 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 $\mathcal{G}_{\mathrm{Ledger}}$ and performs identical checks in both the ideal (Fig. 20, lines 12-13) and the real world (Fig. 7, lines 3-4).

Moving on, in the real world lines 5-13 of Fig. 7 are executed by *Alice* and, given that $\mathcal{A}$ allows it, the code of Fig. 8 is executed by *Bob*. Likewise, in the ideal world, the functionality executes lines 14-32 and as a result it (always) sends (channelOpened, *Alice*, *fchid*) to $\mathcal{S}$. In turn $\mathcal{S}$ simulates lines 5-13 of Fig. 7 with *Alice*'s ITI and, if $\mathcal{A}$ allows it, $\mathcal{S}$ simulates the code of Fig. 8 with *Bob*'s ITI. Once more we conclude that both worlds appear to behave identically to both $\mathcal{E}$ and $\mathcal{A}$ under the same inputs from them. □

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

```
For messages register, registerDone, registered, openChannel,
  channelAnnounced, checkNew, poll, pay, pushAdd, pushFulfill,
  fulfillOnChain and commit, act like F_PayNet.

Upon receiving any other message M from Alice:
    if M is a valid F_PayNet message from a player then
        send (M, Alice) to S
    end if

Upon receiving any other message (M, Alice) from S:
    if M is a valid F_PayNet message from S then
        send M to Alice
    end if
```

Fig. 33.

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. 16 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 $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    strip *payid*, simulate receiving the message with *Alice* ITI and further execute the parts of Π_{LN} that correspond to honest parties (Fig. 10- Fig. 12)
    if any "ensure" in Π_{LN} fails until receiver processes $\textsc{updateAddHtlc}$ **then** // payment failed
        add $(\bot, \mathit{payid})$ to `charged`(*Alice*)
    else
        add ($\overrightarrow{\texttt{path}}$, *payid*) to `payids`
    end if

Upon receiving ($\textsc{getClosedFunds}$, `toReport`, Σ_{Alice}, *Alice*, *reportid*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    simulate Fig. 9, lines 3-47 on input ($\textsc{poll}$), using Σ_{Alice} from the message, with *Alice*'s ITI
    add all channels in `toSubmit` to `details`(*Alice*)
    The first time we receive a message from $\mathcal{F}_{\mathrm{PayNet,Pay}}$ that contains Σ_{Alice} that is at least k blocks longer than the current one, concatenate ($\textsc{channelsClosed}$, `details`(*Alice*), `reportid`) to the first subsequent message to $\mathcal{F}_{\mathrm{PayNet,Pay}}$ and clear `details`(*Alice*)
    send ($\textsc{submit}$, `toSubmit`) to $\mathcal{G}_{\mathrm{Ledger}}$ as *Alice*

Upon receiving ($\textsc{pushFulfill}$, *pchid*, *Alice*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    simulate Fig. 16, lines 1-12 on input ($\textsc{pushFulfill}$, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players

Upon receiving ($\textsc{pushAdd}$, *pchid*, *Alice*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    simulate Fig. 16, lines 13-17 on input ($\textsc{pushAdd}$, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players

Upon receiving ($\textsc{commit}$, *pchid*, *Alice*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$:
    simulate Fig. 13 on input ($\textsc{commit}$, *pchid*) with *Alice*'s ITI and handle subsequent messages by simulating respective ITIs of honest players or sending to $\mathcal{A}$ the messages for corrupted players

Fig. 34.

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, GETCLOSEDFUNDS, PUSHFULFILL, PUSHADD, COMMIT

  handle first part of the message normally

  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. permanently added `HTLC-success` to $\mathcal{G}_{\mathrm{Ledger}}$)), AND the next honest player *Bob* down the line successfully timed out the HTLC on-chain (i.e. permanently added the relevant `HTLC-Timeout` to $\mathcal{G}_{\mathrm{Ledger}}$) (due to no or bad communication from the previous player) **then**

        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 (player before *Alice* (closer to payer) has put in block h_{com} of Σ_{Alice} an older `commitment` tx (that doesn't contain the HTLC) AND $\mathcal{S}$ hasn't received (GETCLOSEDFUNDS, `report`, *Alice*, *reportid*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$ with the offending tx in `report` during $[h_{\mathrm{com}} + k, h_{\mathrm{com}} + k + \texttt{AliceDelay} - 1]$) OR (player after *Alice* (closer to receiver) has irrevocably fulfilled the HTLC on-chain with the `HTLC-success` tx in block h_{fulfill} of Σ_{Alice} AND $\mathcal{S}$ hasn't received (FULFILLONCHAIN, cltvExpiry $- 2k - fu - 1$, *Alice*) from $\mathcal{F}_{\mathrm{PayNet,Pay}}$, where cltvExpiry is the incoming CLTV expiry of *Alice* in $\overrightarrow{\texttt{path}}$) **then**

        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 with the `HTLC-success` tx) **then** // honest payment completed

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

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

      end if

    end for

  end for

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

Fig. 35.

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. When $\mathcal{E}$ sends $\left(\textsc{pay}, Bob, x, \overrightarrow{\texttt{path}}, \texttt{receipt}\right)$ to *Alice* in the ideal world, $\mathcal{S}$ is always notified (Fig. 21, line 4) and simulates the relevant execution of the real world (Fig. 34, line 4). 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 $\mathcal{S}$ (Fig. 26, lines 2, 4, 6 respectively), who in turn simulates *Alice*'s real-world execution with her simulated ITI and the handling of any subsequent messages sent by *Alice*'s ITI (Fig. 34, lines 16, 18, 20). Neither $\mathcal{F}_{\text{PayNet,Pay}}$ nor $\mathcal{S}$ 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. 16 are executed by *Alice*. In the ideal world on the other hand, $\mathcal{F}_{\text{PayNet,Pay}}$ sends (READ) to $\mathcal{G}_{\text{Ledger}}$ (Fig. 26, line 8) as *Alice* and subsequently lets $\mathcal{S}$ simulate *Alice*'s ITI receiving (FULFILLONCHAIN) (Fig. 34, lines 1-2). Observe that during this simulation a second message to $\mathcal{G}_{\text{Ledger}}$ (that would not match any message in the real world) is avoided because $\mathcal{S}$ skips line 19 of Fig. 16, using as t the one received from $\mathcal{F}_{\text{PayNet,Pay}}$ in the message (FULFILLONCHAIN, t, *Alice*). Since $\mathcal{F}_{\text{PayNet,Pay}}$ sends (READ) to $\mathcal{G}_{\text{Ledger}}$ as *Alice* and given that after $\mathcal{G}_{\text{Ledger}}$ replies, control is given directly to $\mathcal{S}$, 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.

continue When $\mathcal{E}$ sends (POLL) to *Alice* □

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$^{\mathcal{A}}\left(1^k\right)$

```
(aux, mpk, n) ← A (INIT)
for i ← 1 to n do
    (pk_i, sk_i) ← KeyShareGen (1^k)
end for
(cpk*, pk*, m*, σ*) ← A (KEYS, aux, pk_1, ..., pk_n)
if pk* ∈ {pk_1, ..., pk_n} ∧ cpk* = CombinePubKey (mpk, pk*) ∧
Verify (cpk*, m*, σ*) = 1 then
    return 1
else
    return 0
end if
```

Fig. 36.

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

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

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

$(mpk, msk) \leftarrow \text{MasterKeyGen}\left(1^k\right)$
$i \leftarrow 0$
$(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}\left(\text{INIT}, mpk\right)$
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
  $(cpk_i, csk_i) \leftarrow \text{CombineKey}\left(mpk, msk, pk_i, sk_i\right)$
  $\sigma_i \leftarrow \text{Sign}\left(csk_i, m_i\right)$
  $(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{A}\left(\text{SIGNATURE}, \mathtt{aux}_{i-1}, \sigma_i\right)$
end while
parse response as $(cpk^*, pk^*, m^*, \sigma^*)$
if $m^* \notin \{m_1, \ldots, m_i\} \wedge cpk^* = \text{CombinePubKey}\left(mpk, pk^*\right) \wedge$ $\text{Verify}\left(cpk^*, m^*, \sigma^*\right) = 1$ **then**
  return 1
else
  return 0
end if

Fig. 37.

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

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

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

8.4 Construction

output standard signing keypairs to avoid duplication?

Parameters: $\mathcal{H}, G$

function `MasterKeyGen`(1^k, rand)
  Return (rand, $G \cdot$ rand)
end function

function `KeyShareGen`(1^k, rand)
  Return (rand, $G \cdot$ rand)
end function

function `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 `CombinePubKey`(mpk, pk)

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

end function

function $\texttt{Sign}(csk, m)$

like standard sign

end function

function $\texttt{Verify}(cpk, m, \sigma)$

like standard verify

end function

just to remember

$sh_{\mathrm{rev},n} \leftarrow shb_{\mathrm{rev}} \cdot \mathcal{H}(phb_{\mathrm{rev}}||pt_{\mathrm{com},n}) + st_{\mathrm{com},n} \cdot \mathcal{H}(pt_{\mathrm{com},n}||phb_{\mathrm{rev}})$

$pt_{\mathrm{rev},n+2} \leftarrow ptb_{\mathrm{rev}} \cdot \mathcal{H}(ptb_{\mathrm{rev}}||ph_{\mathrm{com},n+2}) + ph_{\mathrm{com},n+2} \cdot \mathcal{H}(ph_{\mathrm{com},n+2}||ptb_{\mathrm{rev}})$

$ph_{\mathrm{rev},n+2} \leftarrow phb_{\mathrm{rev}} \cdot \mathcal{H}(phb_{\mathrm{rev}}||pt_{\mathrm{com},n+2}) + pt_{\mathrm{com},n+2} \cdot \mathcal{H}(pt_{\mathrm{com},n+2}||phb_{\mathrm{rev}})$

Lemma 5. *The construction above is* $\mathsf{share\text{-}EUF}$*-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

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

We construct a PPT distinguisher $\mathcal{A}$ (Fig. 38) such that

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

that breaks the assumption, thus proving Lemma 5.

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 $\mathcal{B}$
$(\mathsf{aux}, mpk, n) \leftarrow \mathcal{A}(\textsc{init})$
for $i \leftarrow 1$ to n **do**
    $(pk_i, sk_i) \leftarrow \textsc{KeyShareGen}(1^k)$
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 $\mathcal{B}$
$(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\textsc{keys}, \mathsf{aux}, pk_1, \ldots, pk_n)$
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. 38.

Let Y be the range of the random oracle. The modified random oracle used in Fig. 38 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. 38, 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

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

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. 38, 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

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

□

Lemma 6. *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. 39) 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 6.

Algorithm $\mathcal{A}(vk)$

$j \xleftarrow{\$} U\left[1, T(\mathcal{B}) + T(\mathcal{A})\right]$ // $T(M)$ is the maximum running time of M
    Random Oracle: for every first-seen query q from $\mathcal{B}$ set $\mathcal{H}(q)$ to a random value
    return $\mathcal{H}(q)$ to $\mathcal{B}$
$(mpk, msk) \leftarrow \textsc{MasterKeyGen}\left(1^k\right)$
    Random Oracle: Let q be the jth first-seen query from $\mathcal{B}$ or $\mathcal{A}$:
    if $q = (mpk \,\|\, x)$ **then**
        if $\mathcal{H}(x \,\|\, mpk)$ unset **then**
            set $\mathcal{H}(x \,\|\, mpk)$ to a random value
        end if
        set $\mathcal{H}(mpk \,\|\, x)$ to $(vk - x \cdot \mathcal{H}(x \,\|\, mpk)) \cdot mpk^{-1}$
    else if $q = (x \,\|\, mpk)$ **then**
        if $\mathcal{H}(mpk \,\|\, x)$ unset **then**
            set $\mathcal{H}(mpk \,\|\, x)$ to a random value
        end if
        set $\mathcal{H}(x \,\|\, mpk)$ to $(vk - mpk \cdot \mathcal{H}(mpk \,\|\, x)) \cdot x^{-1}$
    else
        set $\mathcal{H}(q)$ to a random value
    end if
    return $\mathcal{H}(q)$ to $\mathcal{B}$ or $\mathcal{A}$
$i \leftarrow 0$
$(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{B}(\textsc{init}, mpk)$
while response can be parsed as (pk, sk, m) **do**
    $i \leftarrow i + 1$
    store pk, sk, m as pk_i, sk_i, m_i
    $(cpk_i, csk_i) \leftarrow \textsc{CombineKey}(mpk, msk, pk_i, sk_i)$
    $\sigma_i \leftarrow \textsc{Sign}(csk_i, m_i)$
    $(\mathtt{aux}_i, \text{response}) \leftarrow \mathcal{B}(\textsc{signature}, \mathtt{aux}_{i-1}, \sigma_i)$
end while
parse response as $(cpk^*, pk^*, m^*, \sigma^*)$
$(cpk^*, pk^*, m^*, \sigma^*) \leftarrow \mathcal{B}(\textsc{keys}, \mathtt{aux}, pk_1, \ldots, pk_n)$
if $vk = cpk^* \wedge \mathcal{B}$ wins the master-EUF-CMA game **then** // $\mathcal{A}$ won the EUF-CMA game
    return (m^*, σ^*)
else
    return $\textsc{fail}$
end if

Fig. 39.

The modified random oracle used in Fig. 39 is indistinguishable from the standard random oracle for the same reasons as in the proof of Lemma 5.

Let E denote the event in which $\textsc{CombinePubKey}$ 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. 39, line 31) and thus

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

We can reason like in the proof of Lemma 5 to deduce that

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

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

Once more we can reason in the same fashion as in the proof of Lemma 5 to deduce that

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

□

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

Proof. The construction is combine-EUF-secure as a direct consequence of Lemma 5, Lemma 6 and the definition of combine-EUF-security. □

9 Notes on Lightning Specification

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

10 The Ledger Functionality and its Properties

We next provide the complete description of the ledger functionality that is based on the UC formalisation of [2,3].

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 $\mathcal{G}_{\mathrm{RO}}$, the clock $\mathcal{G}_{\mathrm{CLOCK}}$ and network $\mathcal{F}_{\mathrm{N\text{-}MC}}$. If an honest party maintains registration with all the resources then after `Delay` clock ticks it necessarily becomes synchronised.

The progress of the `state` variable is guaranteed via the ExtendPolicy function that is executed when honest parties submit inputs to the functionality. While we do not specify ExtendPolicy in our paper (we refer to the citations above for the full specification) it is sufficient to note that ExtendPolicy guarantees the following properties:

1. in a period of time equal to $\mathtt{maxTime}_{\mathtt{window}}$, at least `windowSize` blocks are added to `state`.
2. each window of `windowSize` blocks has at most $\mathtt{advBlcks}_{\mathtt{window}}$adversarial blocks included in it.
3. any transaction submitted by an honest party earlier than $\frac{\mathtt{Delay}}{2}$ ticks before the time that the block that is `windowSize` positions before the head of the `state` must be included in an honest block.

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

Proposition 1. *Consider any synchronised honest party that wishes to place a transaction* $\mathtt{tx}$ *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* $t \geq ???$*,* $\mathtt{tx}$ *is guaranteed to be included in the correct block range as long as the party submits* $\mathtt{tx}$ *to the ledger functionality within* **[Orfeas: rounds or blocks?]** $[????, ????]$.

Proof. TBD ∎

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

General: The functionality is parameterized by four algorithms, $\mathsf{Validate}$, $\mathsf{ExtendPolicy}$, $\mathsf{Blockify}$, and $\mathsf{predict\text{-}time}$, along with three parameters: $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$, and $\mathcal{S}_{\text{initStake}} := \{(U_1, s_1), \ldots, (U_n, s_n)\}$. The functionality manages variables $\mathtt{state}$ (the immutable state of the ledger), $\mathtt{NxtBC}$ (a list of transaction identifiers to be added to the ledger), $\mathtt{buffer}$ (the set of pending transactions), τ_L (the rules under which the state is extended), and $\boldsymbol{\tau}_{\mathtt{state}}$ (the time sequence where all immutable blocks where added). The variables are initialized as follows: $\mathtt{state} := \boldsymbol{\tau}_{\mathtt{state}} := \mathtt{NxtBC} := \varepsilon$, $\mathtt{buffer} := \emptyset$, $\tau_L = 0$. For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer $\mathtt{pt}_i$ (initially set to 1) and a current-state view $\mathtt{state}_p := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\boldsymbol{\mathcal{I}}_H^T$ (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 $\tau' < \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.

   (b) For any synchronized party $U_p \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if U_p is not registered to the clock, then consider it desynchronized, i.e., set $\mathcal{P}_{DS} \cup \{U_p\}$.

2. If I was received from an honest party $U_p \in \mathcal{P}$:
   (a) Set $\boldsymbol{\mathcal{I}}_H^T := \boldsymbol{\mathcal{I}}_H^T || (I, U_p, \tau_L)$;

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

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

(d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ 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 $(\texttt{state}, \texttt{buffer}, \mathcal{I}_H^T)$ to $\mathcal{A}$.
   - *Maintaining the ledger state:*
     If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\mathcal{I}_H^T$ as above) $\mathsf{predict\text{-}time}(\mathcal{I}_H^T) = \widehat{\tau} > \tau_L$ 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 $I = (\text{DESYNC-STATE}, (U_{i_1}, \texttt{state}'_{i_1}), \ldots, (U_{i_\ell}, \texttt{state}'_{i_\ell}))$, with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\texttt{state}_{i_j} := \texttt{state}'_{i_j}$ 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. Poon J., Dryja T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments
2. 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)
3. 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)