# HOPR - a Decentralized and Metadata-Private Messaging Protocol with Incentives

Dr. Sebastian Bürgel, Robert Kiel

November 2019, V1

## 0.1 On-chain Commitment

### Commitment schemes

A commitment scheme is a protocol between usually two parties $A$ and $B$ and fulfills two properties (hiding and biding).

**Hiding:** The ability to commit a value only known by the sender.
**Binding:** The committed value must be the only one that the sender can compute and that validates during the reveal phase.

## 0.2 Setup phase

Once a node joins the HOPR network, it creates an iterated commitment and stores the opening key in the database. Iterated commitment scheme means openn opens $cm_n$ whereas $open_{n-1}$ opens $cm_{n-1} = open_n$ and so on.
The final commitment is computed as $cm_n := hash_n(r)$ where hash is a preimage-resistant hash function and $r$ is chosen uniformly at random by the node in order to prevent the issuer from knowing whether the ticket will be a win or not.

Furthermore, it will prevent it from tweaking the given challenge such that the ticket cannot be a win. The number of iterations $n$ can be chosen as a constant and should reflect the number of tickets a node intends to redeem.

## 0.3 Opening phase

In order to redeem a ticket, a node has to reveal the opening to the current commitment $cm_n$ that is stored in the smart contract.
The opening shouldn't be revealed otherwise since it discloses whether a ticket is a win or not. The ticket is a win if:

$$hash(ticketHash, response, open) < ticket.winProb$$

where

$$ticketHash := hash(ticketData)$$

The opening is computed as $open_{n-1} = hash_{n-1}(r)$ such that $cm_n = hash(open_{n-1})$.

The on-chain logic verifies whether the latest opening $open_n$ indeed opens the current on-chain commitment $cm_n$. If this is the case, the current on-chain commitment is replaced by the given opening.
The on-chain keeps track of updates to the on-chain commitments to prevent double-spending. So whenever a node resets a commitment, a counter, namely *account.counter* is increased and a call to *updateCommitment* invalidates all previously unredeemed tickets.

## 0.4 Proof Of Relay

HOPR incentivizes packet transformation and delivery using a mechanism called "Proof-Of-Relay". This mechanism makes sure nodes relay services are verifiable.

**Construction**

- Every packet is sent together with a ticket.
- Each ticket contains a challenge.
- The validity of a ticket can only be checked on reception of the packet but the on-chain logic enforces a solution to the challenge stated in the ticket.

Since "Proof-Of-Relay" is used to make the relay services of nodes verifiable, it is the duty of each node to check that given challenges are derivable from the given and the expected information. Packets with inappropriate challenges should be dropped as they might not lead to winning tickets. Therefore, the sender of the packet also provides a hint of the expected value that a node is supposed to get from the next downstream node (as explained in the ticket section).

## 0.5 Sphinx Packet Format

HOPR uses the SPHINX packet format to encapsulate and route data packets in order to provide privacy features such as sender and receiver unlinkability. Sphinx is a cryptographic message format used to relay anonymized messages within a mix network. A sphinx packet consists of two parts:

1. Header:
    - Key derivation
    - Routing information
    - Integrity protection
2. Body:
    - Onion-Encrypted payload

**Key derivation** The sender (A) picks a random $x \in \mathbb{Z}_q^*$ that is used to derive new keys for every packet.
(A) randomly picks a path consisting of intermediate nodes (B), (C),(D) [see section path-finding] and the final destination of the packet (E)
(A) performs an offline Diffie-Hellman key exchange with each of these nodes and derives shared keys with each of these nodes.
(A) computes a sequence of $r$ tuples (in our case r=4)

$$(a_0, s_0, b_0), ....., (a_{r-1}, s_{r-1}, b_{r-1})$$

as follows:

- $a_0 = g^x, s_0 = y_B^x, b_0 = h(a_0, s_0)$
- $a_1 = g^{xb_0}, s_1 = y_C^{xb_0}, b_1 = h(a_1, s_1)$
- $a_2 = g^{xb_0b_1}, s_2 = y_D^{xb_0b_1}, b_2 = h(a_2, s_2)$

Where $y_B, y_C, y_D, y_E$ are the public keys of the nodes $B, C, D$ which we assume are available to $A$ . The $a_i$ are the group elements which, when combined with the nodes' public keys, allows computing a shared key for each via Diffie-Hellman key exchange, and so the first node in the user-chosen route can forward the packet to the next, and only that mix-node can decrypt it. The $s_i$ are the Diffie Hellman shared secrets, and the $b_i$ are the blinding factors.

**Routing information** Each node on the path needs to know the next downstream node. Therefore, the sender $(A)$ generates routing information $\beta_i$ for $(B)$, $(C)$ and (D) as well as message END to tell $(E)$ that it is the final receiver of the message.
As $(A)$ has a shared secret with each of the nodes along the path, it is able to derive blindings for each of them which is symbolised as different hatchings.
Once $(B)$ receives the packet, it derives the shared key $s_0$(for simplicity we call it $s_B$ as it is the shared key with $B$) by computing

$$s_0 = (a_0)^b = (g^x)^b = (g^b)^x = y_B^x$$

and removes its blindings. This allows $(B)$ to unblind the routing info that tells $(B)$ the public key of the next downstream node $(C)$.

$(B)$ also removes one layer of encryption from the payload. Same happens at $(C)$ and $(D)$: key derivation, unblinding, deleting, shifting, decryption and blinding. In addition to all these steps, $(E)$ finds a message that symbolizes the end of the path and tells that it's the recipient.

**Integrity** Each node along the path receives an authentication tag $\gamma_i$ in the form of a message authentication code (MAC) which is encoded in the header and allows to check whether or not the header was modified which guarantees integrity.

## 0.6 Tickets aggregation

The word aggregation means the process of combining things or amounts into a single group. HOPR adds an additional scaling layer that aggregates multiple tickets and redeems them within one transaction.

Since probabilistic payments cannot scale arbitrarily and ethereum gas fees are only increasing it makes sense to aggregate multiple tickets and redeem them within one transaction. The preimage of a winning ticketis stored in the node's database.

The node sends ticket or several winning tickets $ticket_A$ and $ticket_B$ with the corresponding pre-images to the issuer who computes their aggregation $ticket_C$.

The issuer stores on chain $hash(ticket)$ of the redeemed tickets in a space-efficient data storage called Bloom Filters. The issuer computes the modified Bloom Filter by XORing the previous version with the one after inserting the ticket.
Once $ticket_C$ is sent to the smart contract, the intended version of the modified Bloom Filters is recovered by $bloom_{n+1} = bloom_n \vee diff$ and thereby invalidates $ticket_A$, $ticket_B$ as well as $ticket_C$.

## 0.7 Tickets

In the HOPR protocol, nodes that have staked funds within a payment channel can issue tickets that are used for payment to other nodes. Tickets are used for probabilistic payments; every ticket is bound to a specific payment channel and cannot be spent elsewhere. They are redeemable at most once and they lose their value when the channel is closed.

### 0.7.1 Ticket issuance

A ticket can be issued once two nodes have established a payment channel with each other which means that at least one of them has locked HOPR tokens.
The ticket issuer A (A could also be the packet creator) selects the winning probability of the ticket and the relay fee to use and sets amount to:

$$amount := \frac{pathLength * relayFee}{winProb}$$

If the issuer $(A)$ receives the packet from another node $(O)$ (packet creator) and attempts to issue a ticket for the next downstream node, the challenge is given together with the routing information by the packet. We assume $(A)$ and $(O)$ are not the same.

$(A)$ does not know whether the ticket is a win or not.

$(A)$ sets content of a ticket to:

$t = (recipient, challenge, account.counter, amount, winProb, channel.teration, index)$

$(A)$ then signs the ticket with its private key and sends $ticket := (t, Sig_Issuer(t))$ to the recipient together with a mixnet packet.

The data for a ticket is signed by the issuer, and a ticket is the data followed by the signature:

$$ticket := (ticketData, Sig_Issuer(ticketData))$$

where

$$ticketData := (amount, winProb, recipient, index, challenge, iteration, chainId, tag, version)$$

**Challenge**

$(O)$ creates a shared secret $s_i$ with all the relay nodes in the channel (A-B-C-Z) by using an offline version of the Diffie-Hellman key exchange.
The shared secret $s_i$ is used as a seed for a PRG (Pseudo Random Generator) to create secret shares $s_i, s_i'$ for each node along the route. Relayers compute $s_i$ and get $s_i + 1'$ from the next downstream node.
The sender $(A)$ creates $challenge_i := (s_i + s_i + 1') * G$ and a hint for B,C,D,Z (let's suppose these are the relay nodes and $Z$ will be the final destination) a "hint" how the promised value $s'_C, s'_D, s'_Z$ is going to look like. The value "hint" is computed as $hint_i := s_i + 1' * G$

### 0.7.2 Ticket validation

Tickets are received together with packets which means that the recipient and the next downstream node share a secret $s$ whose key shares $s_i$ and $s_i + 1$ are derivable by those nodes.
Once the recipient $(A)$ receives $s_{i+1}^{(1)}$ from (B) by the secret sharing, it can compute $response := s_i^{(0)} + s_{i+1}^{(1)}$ such that it verifies

$$response * G = ticket.challenge$$

Once the recipient transforms the packet, it is able to compute $s_i$. The recipient is now also able to extract the routing information from the packet. This includes a hint to the value $s_{i+1}$ given as $hint_i := s_{i+1} * G$
Together with $s_i$, the node can verify that

$$ticket.challenge = s_i * G + hint_i$$

with

$$s_i * G + hint = si * G + s_{i+1} * G = (s_i + s_{i+1}) * G$$

This allows the recipient to verify that the promised value $s_{i+1}$ indeed leads to a solution of the challenge given in the ticket. If this is not the case, then the node should drop the packet.
Without this check, the sender is able to intentionally create falsy challenges that lead to unredeemable tickets.

### 0.7.3 Ticket redemption

In order to unlock the ticket, the node stores it within its database until it receives an acknowledgement containing $s_{i+1}$ from the next downstream node. The challenge can be computed from acknowledgement as $challenge := ack * G$. Once it receives an acknowledgement, it checks whether it stores an unacknowledged ticket for the received acknowledgement. If this is not the case, the node should drop the acknowledgement.
The node then computes the response to the challenge given in the ticket as

$$response := (s_i + s_{i+1}) * G$$

Additionally, the node retrieves the opening value open to the current on-chain commitment and checks whether response, open leads to a winning ticket. This is the case if

$$hash(ticketHash, response, open) < ticket.winProb$$

where

$$ticketHash := hash(ticketData)$$

If this is not the case, the node should drop the ticket. The final recipient of the packet does not receive a ticket because message reception is not incentivized by the HOPR protocol.
The node checks whether the information gained from the packet transformation is sufficient to fulfill the given challenge sent along with the ticket. It then replies with an acknowledgement that includes a response to the challenge.