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

Amira Bouguera, Robert Kiel, Dr. Sebastian Bürgel, Qianchen Yu

November 2021, v0.1

1 Introduction

Internet privacy is a subset of data privacy and a fundamental human right as defined by the United Nations [UN, 2018]. In order to retain and exercise this right to privacy, each internet user must have full control of the transmission, storage, use, and disclosure of their personally identifiable information. However, the infrastructure and economics of our increasingly technologically driven world put great pressure on privacy, due to the various advantages (economic and otherwise) which knowledge of users' private information provides.

To combat this trend, multiple solutions are being developed to restore internet users' control of their private information. These solutions generally centre on providing truly anonymous communication, such that neither the content of the communication nor information about the communicators can be discovered by anyone, including the communicating parties.

HOPR is a decentralized incentivized mixnet [Chaum, 1981] that employs privacy-by-design protocols. HOPR aims to protect users' transport-level metadata privacy, giving them the freedom to use online services safely and privately. HOPR leverages existing mechanisms such as the Sphinx packet format [Danezis and Goldberg, 2009] and packet mixing to achieve its privacy goals, but adds an innovative incentive framework to promote network growth and reliability. HOPR uses the Ethereum blockchain [Wood, 2021] to facilitate this incentive framework, specifically to perform probabilistic payments via payment channels.

2 Path Selection

HOPR currently utilizes free-route sender-selected paths which are known to provide better privacy than other topologies, as well as better packet delivery

in the event of partial network failure [Dingledine et al., 2005]. HOPR makes it easy to customize various different routing strategies. In this section, we present the free-routing strategy that HOPR nodes are currently following. Another strategy is later introduced in the cover traffic section.

2.1 Objectives and Approach

A packet in the HOPR network is sent via multiple mix node hops before it is delivered to its final recipient. A sender has to consider three partially antagonistic points when selecting a path

1. selected nodes should be effective at mixing packets
2. path selection must not be deterministic
3. offline nodes should be avoided

As anyone can run a node in the HOPR network, it is also important to have neutral Schelling points for effective mix nodes. The default path selection mechanism uses the number of tokens that a node has staked in a particular payment channel as a signal for how much traffic the node is willing to relay in a fashion that aligns its incentives with the rest of the network. Thus, stake is used as a Schelling point for which nodes will mix packets more effectively than others with lower traffic.

To meet HOPR's stated **??**, it is crucial for nodes in the network to keep the chosen path secret. While the individual nodes along a path are selected, based on their stake, a node's individual path selection must remain hard to predict for third parties in order to provide sender-receipient unlinkability. HOPR achieves that by randomizing channel stake by multiplying its stake with a random number.

In order to keep sender-receiver packet loss low, the sender only chooses nodes that it considers to be online. The path selection mechanism uses a heartbeat mechanism for this.

2.2 Randomization Stake as Edge Weights

The path selection algorithm outlined below uses edge weights that determine likelihood of randomly choosing a specific edge. The edge weights are initialized with the payment channel balance and the payment channel topology. The more tokens a node stakes to an outgoing payment channel, the more likely the edge is selected on a path, hence the more likely the connected nodes are chosen as relayers by other nodes.

Since this initialization step based on channel stake is entirely deterministic and potential adversaries could thus deduce the path of an individual packet from public information, each node score is randomized by multiplying it with a random number, $0 \leq r_i \leq 1$. Over long time scales, the expectation value of the weight is still proportional to the channels's score and thus provides a predictable Schelling point.

$$weight(n_i) := balance(n_i) * r_i$$

Note that the random numbers are assigned once at the beginning of the process and reassigned upon the selection of a subsequent packet.

2.3 Heartbeat

HOPR nodes use a heartbeat mechanism to estimate the availability of nodes in the network. Nodes keep scores of each other that measure their health. They utilize a exponential backoff to efficiently measure health scores.

2.3.1 Health Score

Each node keeps a score of all other nodes and increases that score if it has been observed online and decreases if it could not be reached. Each node that has an funded outgoing payment channel is initially listed with $healthScore = 0.2$. Responding to a *ping* with a *pong* packet or sending its own *ping* increases a node's score by 0.1. Not responding to a *ping* reduces the node's score by 0.1. The minimal node score is 0 and the maximal score is 1. Nodes with a $healthScore \geq healthThreshold$ considered online and utilized in paths, nodes with lower scores are omitted. The default $healthThreshold$ is 0.5.

2.3.2 Exponential Backoff

Since the network status can change abruptly, e.g., due to electricity outages or unstable network links, availability needs to be measured frequently on an ongoing basis. On the other hand, it does not make sense to constantly probe nodes that are known to be offline. To provide a dynamic trade-off for both cases, HOPR utilizes a heartbeat with exponential backoff, the time until the next *ping* is sent to a node increases with the number of failed ping attempts n_{fail} since the last sucessful attempt or the network start. A successful response to *ping* resets the backoff timer.

$$t_{bo} = t_{base}{}^{f_{bo}{}^{n_{fail}}}$$

where t_{bo} is the backoff time, $t_{base} = 2s$ is the initial backoff time and $f_{bo} = 1.5$.

The maximal backoff time of 512 seconds corresponds to $n_{fail} = 5$.

2.4 Path selection

The HOPR network uses source-selected routing. This means a node must sample the entire path the mixnet packet will take before sending it to the first relayer.

To achieve the aforementioned privacy guarantees, paths must include at least one intermediate node. However, using the Sphinx packet format (see Section 3) increases both the size of the header and the computation needed to generate it, which makes it possible for adversaries to distinguish packets on longer paths. Therefore, all nodes in the HOPR network are strongly encouraged to use a $targetLength$ of three. Packets with longer paths are dropped by relayers. It is possible but discouraged for nodes to use paths of one or two hops.

Nodes can only be chosen once per path. For example, when choosing the third node in the path $A \rightarrow B$, if A is found to be the only node with an open channel to B, the search will fail and a new path will be generated.

The algorithm terminates once a path with $targetLength$ is found. To prevent the algorithm from visiting too many nodes, the number of iterations is bound by $maxIterations$ and the longest known path is returned if no path of length $targetLength$ was found.

The default path selection algorithm is a best-first search of edges which got initialized with randomized payment channels stakes as outlined above. It omits paths that would contain loops and nodes with $healthScore < healthThreshold$

and paths that would be shorter than *targetLength*.

Algorithm 1: Path selection

Input: nodes V, edges E

$queue \leftarrow$ **new** $\mathsf{PriorityQueue}_{\mathsf{pathWeight}}()$
$queue.addAll(\{(x, y) \in E \mid x = self)$
$closed \leftarrow \emptyset$
$iterations \leftarrow 0$

while $size(queue) > 0$ and $iterations < maxIterations$ **do**
  $path \leftarrow queue.peek()$
  if $length(path) = targetLength$ **then**
    return $path$
  end

  $current \leftarrow last(path)$
  $open \leftarrow \emptyset$

  foreach $next \in \{y \in V \mid (x, y) \in E \ \wedge \ x = current\}$ **do**
    if $healthScore(next) \geq healthThreshold$ **and** $y \notin$ $closed$ **and** $y \notin path$ **then**
      $open.push(y)$
    end
  end

  if $size(open) = 0$ **then**
    $queue.pop()$
    $closed.add(current)$
  else
    foreach $node \in open$ **do**
      $queue.push(\ (...path, node)\ )$
    end
  end

  $iterations \leftarrow iterations + 1$
end
return $queue.peek()$

3 Sphinx Packet Format

HOPR uses the Sphinx packet format [Danezis and Goldberg, 2009] to encapsulate and route data packets in order to ensure sender and recipient unlinkability. The Sphinx packet format determines how mixnet packets are created and trans-

formed before relaying them to the next downstream node in a way that does not leak path information to relayers or other parties. Each Sphinx packet consists of two parts, a header and an onion-encrypted payload:

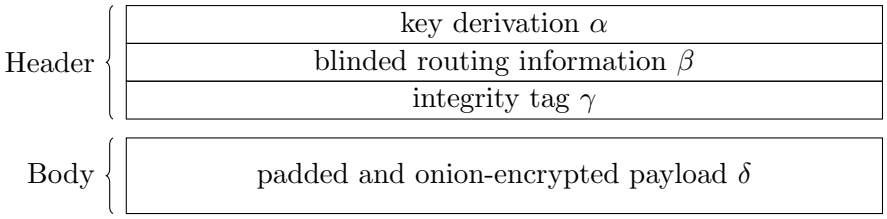

Figure 1: Schematic overview of a SPHINX packet

3.1 Construction

The following explains how a Sphinx packet is created and how it is transformed at each hop before arriving at its final destination. At first, the sender chooses a path (see Section 2), and derives shared keys with each node on the path. The shared keys serve as a master secret to derive subkeys. These subkeys are used to blind the routing information in such a way that nodes can solely determine the next downstream node, and also to create an authentication tag to check the integrity of the header. In addition, the sender applies one layer of encryption for each node along the chosen path to the payload. This accomplished, the sender finally sends the packet to the first hop.

Once a node receives a mixnet packet, it first derives the key that it has shared with the sender of the packet and checks the integrity of the header. Next, it unblinds the routing information to determine the next downstream node and removes one layer of encryption from the encrypted payload. At this point, the node is able to decide whether it is the final recipient of the message or it is supposed to forward the packet to the next hop.

3.1.1 Key Extraction

Path selection, see section 2, results in a list $n_0, \ldots, n_r$ of mixnodes which are supposed to process and forward the packet before the message reaches its destination *dest*. HOPR nodes use their public keys as addresses, hence the public keys $Y_0, \ldots, Y_r$ are known to the creator of a packet.

Key Exchange Having this knowledge allows the creator of the packet to perform an offline Diffie-Hellman key exchange with each of the mix nodes $n_0, \ldots, n_r$ and the destination *dest*, resulting in shared secrets $s_0, \ldots, s_r$ and s_{dest}.

To provide perfect forward secrecy and thereby minimize the risk of key corruption, the sender first samples a random value $x \in \mathbb{F}$ and derives $\alpha_0 = x \cdot G$. It further derives blinding factor b_0 as $b_0 = \mathsf{KDF}(\alpha_0, x \cdot Y_0)$ where KDF is a key derivation function, see appendix A, and derives the shared secret as $s_0 = x \cdot b_0 \cdot Y_0$. Hence, deriving the shared secret s_0 requires the knowledge of the sender's field element $\alpha_0 = x \cdot G$ as well as the receivers' field element $x \cdot b_0 \cdot Y_0$.

The first relayer n_0 is then able to compute s_0 by first deriving $x \cdot Y_0$ as

$$y_0 \cdot \alpha_0 = y_0 \cdot x \cdot G = x \cdot (y_0 \cdot G) = x \cdot Y_0$$

yielding b_0 and $s_0 = y_0 \cdot b_0 \cdot \alpha_0$ where y_0 refers to the private key of node n_0. The value s_0 then servers as a master secret to perform further key derivations as described in appendix A.

Transformation Once the key extraction is done, the each relayer n_i transforms its value α_i into α_{i+1} by setting $\alpha_{i+1} = b_i \cdot \alpha_i$ such that

$$\alpha_i = x \cdot \left(\prod_{j=0}^{i} b_j\right) \cdot G = \left(\prod_{j=0}^{i} b_j\right) \cdot \alpha_0 = \left(\prod_{j=k+1}^{i} b_j\right) \cdot \alpha_k$$

By adding an additional blinding for every node along the selected path, each incoming α_i and each outgoing α_{i+1} become indistinguishable from random numbers of the same length. Hence, an adversary who observes incoming and outgoing packets cannot use α to track packets.

Since each s_{i+1} depends on b_{i+1} and therefore on all $\alpha_0, \ldots, \alpha_i$, the key extraction can only be done in the foreseen order; first derive s_0, then derive s_1 etc. Hence, an adversary is unable to extract keys in a more favorable order, e.g. because has it managed to compromise *some* but not *all* nodes on the path.

Example The following example shows the key extraction process for a sender A, three mix nodes B, C, D and a receiver Z.

$$\begin{aligned}
(\alpha_B, b_B, s_B) &= (x \cdot b_A \cdot G, \mathsf{KDF}(\alpha_B, y_B \cdot \alpha_B), y_B \cdot b_B \cdot \alpha_B \cdot G \\
(\alpha_C, b_C, s_C) &= (x \cdot b_A\ b_B \cdot G, \mathsf{KDF}(\alpha_C, y_C \cdot \alpha_C), y_C \cdot b_C \cdot \alpha_C \cdot G \\
(\alpha_D, b_D, s_D) &= (x \cdot b_A\ b_B\ b_C \cdot G, \mathsf{KDF}(\alpha_D, y_D \cdot \alpha_D), y_D \cdot b_D \cdot \alpha_D \cdot G \\
(\alpha_Z, b_Z, s_Z) &= (x \cdot b_A\ b_B\ b_C\ b_D \cdot G, \mathsf{KDF}(\alpha_Z, y_Z \cdot \alpha_Z), y_Z \cdot b_Z \cdot \alpha_Z \cdot G
\end{aligned}$$

where $b_A = x \cdot G$ and y_i for $i \in \{B, C, D, Z\}$ refers to the nodes' private key.

Packets consist of (α, β, γ) where β is described in section 3.1.3 and γ in section 3.1.5, leaving both underspecified for the moment. Let further be $packet_i = (\alpha_i, \beta_i, \gamma_i)$, leading to:

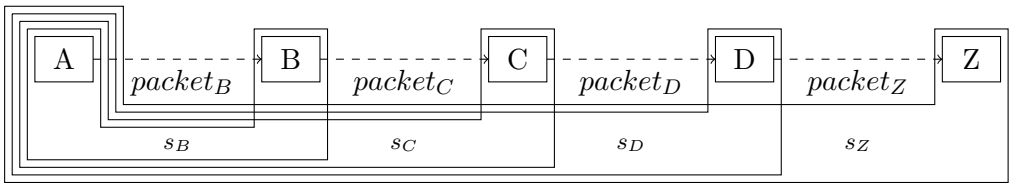

Figure 2: Node A derives shared keys s_B, s_C, s_D, s_Z with node B, C, D, Z using their public keys Y_B, Y_C, Y_D, Y_Z.

3.1.2 Replay protection

The creator of a packet picks a path that the packet is supposed to take. As seen in section 3.1.1, the key derivation is done such that the packet cannot be processed in a order than the one chosen by its creator.

This behavior prevents the adversary from changing the route but also allows no other route. Hence the adversary can be sure that there is no second possible route. Therefore, it can try to replicate the packet and send it multiple to see which connections get used more and thus reveals the route of the packet.

To prevent from this attack scenario, each node n_i computes a fingerprint s_i^{tag} of each processed packet and stores it in order to refuse the processing of already seen packets. The value s_i^{tag} is generated from master secret s_i as seen in section 3.1.1 using the key derivation described in section A.

3.1.3 Routing information

Each node on the path needs to decide whether it is destined to be the receiver of the message and if not, to whom it is supposed to forward the packet. To achieve the privacy properties, each node must only know its direct successor and is allowed to know its predecessor. More precisely, the node must not be able to determine *if* and *where* the next downstream node is going to forward the packet to. Nor shall it know where the packet came from before it was sent to its predecessor.

This is achieved by applying multiple layers of blinding and sequentially removing layer by layer at each hop such that the address of the next hop is only visible for one hop along the path.

To ensure that the header has not been tampered while traveling through and network and beign transformed by the nodes along the path, there is an integrity tag that is sent in addition to the public key of the next hop. The routing information is thus given by (y_i, γ_i) where y_i denotes the public key and γ_i the integrity tag for the next downstream node. See section 3.1.5 for more details on the utilized integrity scheme.

Adresses Nodes in the network are distinguished by the ECDSA public keys, hence the header includes the public key of the next downstream node and in case of the very last node, a distinguished byte sequence END.

ECDSA public keys are given by tuple of two 32-byte field elements (x, y), upon which it is sufficient to solely store the first component x and the sign of y, resulting in a compressed elliptic curve point 0x02<x> for positive y and 0x03<x> otherwise. The sequence END is given as 0x04 such that the last node can ignore all subsequent bytes if END is present.

Blinding The header uses multiple blindings and their aggregations to make certain sections of the header visible to only a single node. Blindings are generated by a pseudorandomness generator (PRG), see appendix C for a detailed description of the utilized PRG.

As a result of the Diffie-Helman key exchange done in section 3.1.1, each node along the path is able to derive a shared secret s_i with the creator of the packet and is therefore able to derive a sub-key s_i^{bl}. Both, creator of the packet and each node n_i along the path use s_i^{bl} as a seed for the PRG, yielding $blinding_i$.

The routing information for each node n_i is blinded by XORing the content with $blinding_i$ as well as the blindings $blinding_0, \ldots, blinding_{i-1}$ of all previous hops. Each node that receives the packet, removes their own blinding from the header and is thus able to extract the routing information destined for them. By removing the blinding from the header, it allows the next downstream node to extract their routing information since it is now only blinded by their blinding.

Example The following assumes the sender A is generating routing information for B, C, D, Z and applies their corresponding blinding before sending the header to the first relayer B. The blindings are visualized by different hatchings.

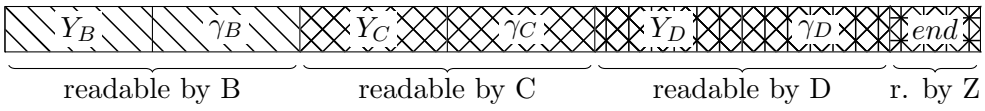

Figure 3: Blinded routing information sent to first relayer B.

3.1.4 Delete and shift

Unblinding the β_i reveals the public key Y_{i+1} of the next downstream node as well as the integrity tag γ_{i+1} that allows the next downstream to verify the integrity of β_{i+1}. The *rest* of β_i is kept hidden to the node as it contains random data, see section shorter paths, *or* blindings from other nodes which is assumed to be indistinguishable by that node.

$$\{Y_{i+1}, \gamma_{i+1}, rest\} = \beta_i \oplus \mathsf{PRG}_{s_i^{bl}}(0, |\beta|)$$

After extracting Y_{i+1} and γ_{i+1}, the node shifts *rest* to the beginning and thereby overwrites its own routing information Y_{i+1} and γ_{i+1}. In addition, it fills the hole in the end by adding its own blinding:

$$\beta_{i+1} = rest \;||\; \mathsf{PRG}_{s_i^{bl}}(|\beta|, |\beta| + |Y| + |\gamma|)$$

Note that the blindings are created using different boundaries. The first blinding is created from 0 to length of β and the second one is created as if public key and integrity were appended in the end of β, hence boundaries are set from $|\beta|$ to $|\beta| + |y| + |\gamma|$.

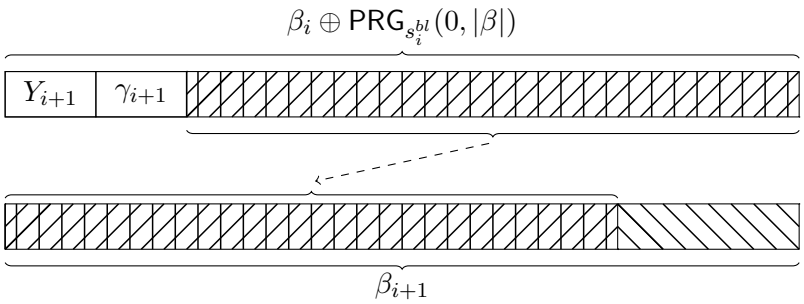

Figure 4: Shifting in the header

3.1.5 Integrity check

Upon reception of a packet, each node first derives the shared key s_i with the creator of the packet. Before it unblinds the routing information, it checks their integrity and *should* drop the packet if the header has been tampered.

Integrity tag As by section 3.1.1, each node along the path shares a secret s_i with the creator of the packet. To create an integrity tag γ_i, the node first derives a sub-key s_i^{int} as described in appendix A, yielding:

$$\gamma_i = HMAC_{s_i^{int}}(\beta_i)$$

where $HMAC$ is instantiated with the hash function BLAKE2s and the output size is set to 32 bytes.

Packet transformation From the predecessor, which can be the creator of the packet *or* a relayer, the node receives γ_i and uses it to verify the integrity of the received header by recomputing the integrity tag γ_i' using the received β_i as well as the derived sub-key s_i^{int} and checking whether $\gamma_i = \gamma_i'$. If this is not the case, the node drops the packet.

The node then applies the transformations from sections 3.1.3 and 3.1.4, and thus receives not only the public key y_{i+1} but also the integrity tag that the next downstream node needs to verify the integrity of the packet header.

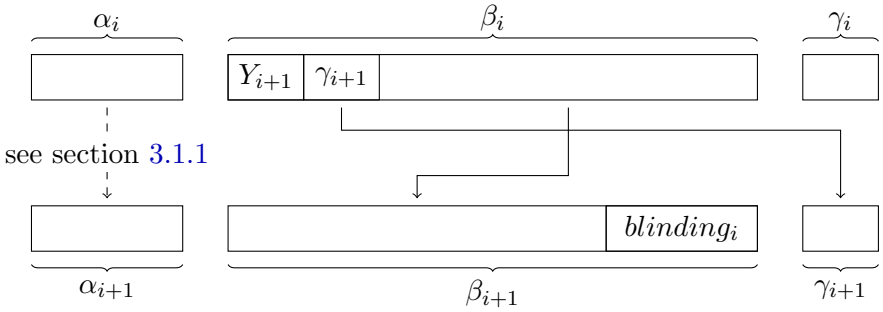

Figure 5: Extraction of γ_{i+1} and creation of the next packet

The received header $(\alpha_i, \beta_i, \gamma_i)$ thereby transforms into $(\alpha_{i+1}, \beta_{i+1}, \gamma_{i+1})$.

Filler In order to compute the integrity tag, the sender needs to know *how* the packet is going to look like when it reaches the node that checks its integrity. Hence, the creator of the packet needs to perform the whole transformations that are done by the relayers along the path *in advance* before sending the packet.

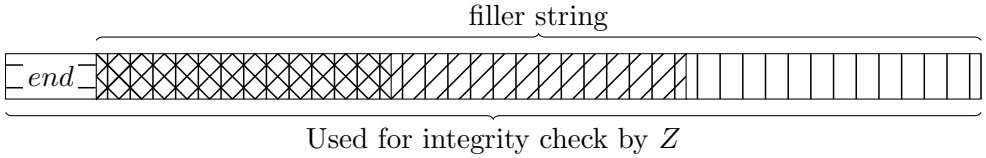

Figure 6: Node A is sending a packet through mix nodes B, C, D to final destination Z such that the readable part for Z includes the END tag and the blindings $blinding_B, blinding_C, blinding_D$ that were appended by the predecessors.

Since each node, shifts the routing information to left, thereby deletes their own routing information from the header and appends their own padding in the end, the routing information of the very last node consists of the END message as desribed section 3.1.3 and the paddings that got appended by the previous nodes. To compute the integrity tag for that node, the creator of the packet needs to compute a filler string ϕ and use it to compute

$$\gamma_i = HMAC_{s_i^{int}}(END \mid\mid \phi_i)$$

3.1.6 Encrypt and decrypt

In contrast to the header, the integrity of the payload is not directly protected by the protocol. To ensure potential manipulations of the message remain visible to the final recipient, the content of the payload is hidden using a pseudorandom permutation scheme (PRP) and its inverse is used to undo the transformation. This comes with the property that if there were any modifications to the payload, such as a bit flip, the probability that the decoded message contains any relevant information is expected to be negligible.

To implement the PRP, HOPR uses the LIONESS [Anderson and Biham, 1996] wide-block cipher scheme, instantiated by using Chacha20 as a stream cipher and BLAKE2s as a hash function as suggested by Katzenpost. See appendix B for a detailed description and the chosen parameters.

As seen in Section 3.1.1, while creating the packet the sender derives a shared key s_i with each node along the chosen path and uses them to create subkeys s_i^{prp} to key the PRP. See Appendix A for more details about the key derivation.

To allow the final recipient to determine whether a message is meaningful content or not, each message is padded by a protocol-specific tag τ and 0s to fit the packet size of 500 bytes, yielding m_{pad}. Decoded payloads that do not include τ are considered invalid and should be dropped.

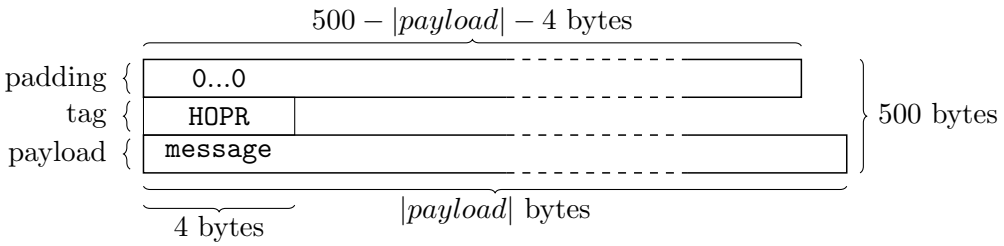

Figure 7: Padded message consisting of 0-padding, protocol tag τ (`0x484f5052`, ASCII-encoded "HOPR"), and payload m.

The sender takes the padded message and encrypts it using PRP.permutate with the derived subkeys in reverse order $(..., s_{i+1}^{prp}, s_i^{prp}, s_{i-1}^{prp}, ...)$

$$\delta_i = \mathsf{PRP.permutate}_{s_i^{prp}}(\delta_{i+1})$$

where $\delta_4 = m_{pad}$ and δ_0 is the ciphertext that is sent to the first relayer when using three intermediate hops.

Each node n_i along the chosen path then removes one layer of encryption by setting

$$\delta_{i-1} = \mathsf{PRP.inverse}_{s_i^{prp}}(\delta_i)$$

yielding $\delta_0 = m_{pad}$ in case the node is the final recipient.

3.1.7 Shorter Paths

SPHINX packets define a maximum packet length r[1], meaning r intermediate mix nodes and one destination. They also allow the usage of shorter paths of length v with $0 \leq v < r$. Despite these packets have the same size and are due to their construction indistinguishable from packet with maximum path length, it not advisable to shorter paths since they introduce observable patterns that deviate from normal usage. By observing these patterns, an adversary can be able to reconstruct route of distinguished packets.

Creating a header for shorter paths require less routing information, and thus leaves empty space in the end of β, which is filled as suggested by [Kuhn et al., 2019] with *random* data. This is necessary because the very last node, i.e. the

[1]HOPR nodes use by default a path length of three and refuse the processing of packets with greater path lengths.

one for which the message is destined, is otherwise able to determine the utilized path length.

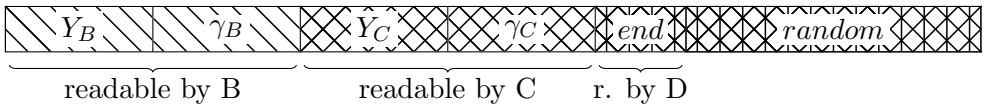

Figure 8: Node A has created routing information for B, C and the final recipient D, filling the empty part of β with random data.

By applying the transformations seen in section 3.1.4 through the mixnet nodes, all blindings got removed, hence the plaintext of β becomes visible and the node is able to decide which parts of β were not used.

Figure 9: Node D receives a packet which was relayed by two intermediate nodes.

4 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. Tickets are redeemable at most once and lose their value when the associated payment channel is closed or when the commitment is reset. A commitment is a secret on-chain value used to verify whether a ticket is a winner or not when an attempt is made to redeem it.

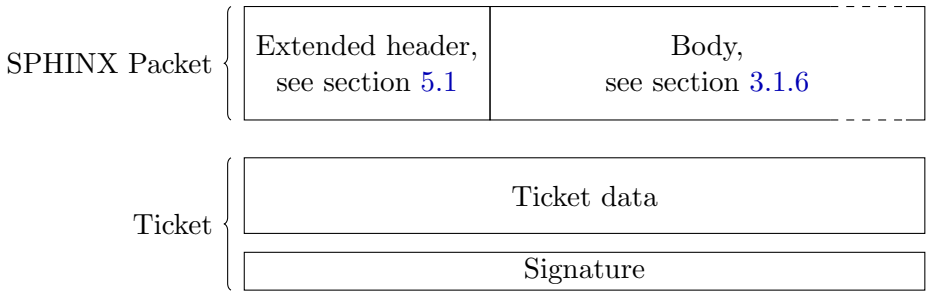

Figure 10: Schematic overview of a mixnet packet that is sent together with a ticket.

4.1 Ticket Issuance

A ticket can be issued once two nodes have established a payment channel with each other. By definition this means at least one of them has staked HOPR tokens. A ticket is issued by a node for the next downstream relay node along the path. In the following, we describe this process between the sender A and the first downstream node B but the same process applies for the ticket that is issued by B for C and following nodes.

The ticket issuer A (who could also be the packet creator) sets a winning probability and relay fee to use and sets the amount to:

$$\sigma = \frac{L \times F}{P_w}$$

where σ is the amount of HOPR tokens set in the ticket, L is the path length, F is the relay fee, and P_w is the ticket's winning probability. These are currently static values which apply network wide.

A issues a ticket for the next downstream node. The challenge is given together with the routing information by the packet. A does not know whether the ticket is a winner or not.

A sets the content of the ticket to:

$$t = (R, \sigma, P_w, \alpha, I, T_c, \zeta),$$

where t has the following components, in addition to those already defined above:

- **Recipient's Ethereum address** R: a unique identifier derived from the ticket recipient's public key.
- **Ticket epoch** α: used as a mechanism to prevent cheating by turning non-winning tickets into winning ones. This is done by increasing the value of α whenever a node resets a commitment, which helps keep track of updates to the on-chain commitments and invalidates tickets from earlier epochs.
- **Ticket index** I: set by the ticket issuer and increases with every issued ticket. The recipient verifies that the index increases with every packet and drops any packets where this is not the case. Redeeming a ticket with index n invalidates all tickets with index $I < n$, hence the relayer has a strong incentive to not accept tickets with an unchanged index.
- **Ticket challenge** T_c: set by the ticket issuer and used to check whether a ticket is redeemable before the packet is been relayed. If it is not redeemable, the packet is dropped.
- **Channel epoch** ζ: used to give each incarnation of the payment channel a new identifier such that tickets issued for previous instances of the channel

become invalid once a channel is reopened (α's count restarts again). This is due to the fact that ζ increments whenever a closed channel is (re)opened.

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

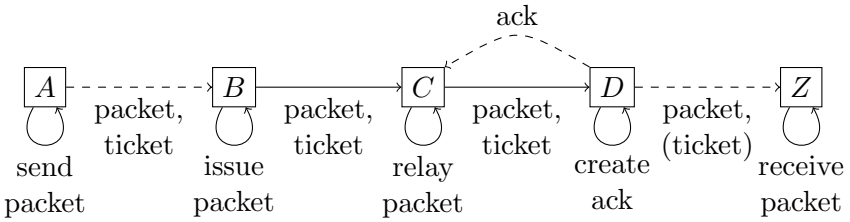

Figure 11: Ticket workflow

4.2 Ticket Validation

Tickets are received together with packets. To this end, 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 as introduced in section key extraction. Ticket validation requires the following steps:

Validate response Once A receives $s_{i+1}^{(1)}$ from B via secret sharing, it can compute

$$r_i = s_i^{(0)} + s_{i+1}^{(1)}$$

where r_i is the response r at iteration i such that it verifies

$$r_i * G = T_{c_i}$$

Validate hint Once the recipient transforms the packet, it can compute s_i. The recipient can now also extract the routing information from the packet. This includes a hint to the value s_{i+1}, given as

$$H_i = s_{i+1}^{(1)} * G,$$

which is stored in the Sphinx packet header.

The unacknowledged ticket is stored in the database under the hint to the promised value to ensure the acknowledgement can be later linked to the unacknowledged ticket.

Together with $s_i^{(0)}$, the node can verify that

$$T_{c_i} = s_i^{(0)} * G + H_i$$

with

$$s_i^{(0)} * G + H_i = s_i^{(0)} * G + s_{i+1}^{(1)} * G = (s_i^{(0)} + s_{i+1}^{(1)}) * G$$

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

4.3 Ticket Redemption

In order to unlock a ticket, the recipient node stores the ticket within its database until it receives an acknowledgement containing s_{i+1} from the next downstream node. HOPR uses acknowledgements to prove the correct transformation of mixnet packets as well as their delivery to the next downstream node.

The challenge can be computed from the acknowledgement as $T_{c_i} = Ack_i * G$. The node checks the following in order to redeem its ticket:

- **Stored ticket** Once the node receives an acknowledgement, it checks whether it is storing an unacknowledged ticket corresponding to the received acknowledgement. If it does not, the node should drop the acknowledgement.

  The node then computes the response to the challenge (also referred to as the *proof of relay secret*) given in the ticket as

  $$r_i = (s_i^{(0)} + s_{i+1}^{(1)}) * G$$

- **Sufficient challenge information** The node checks whether the information gained from the packet transformation is sufficient to fulfil the challenge sent along with the ticket,

  $$r_i * G = T_{c_i}$$

  The node then replies with an acknowledgement which includes a response to the challenge. If this is not the case, the node should drop the packet.

Both the node and the smart contract then perform the following checks:

- **Channel existence** The node checks that the appropriate chan nel **exists** and is **open** or **pending to close**. If these checks do not pass, the node should drop the packet and the smart contract check will revert. To prevent

metadata leakage, the check happens locally rather than on-chain using the blockchain indexer. If the node has no record of the channel or considers the channel to be in a state different from **open** or **pending to close**, the ticket is dropped and receipt of the accompanying packet is rejected.

- **Commitment value check** Additionally, the node and the smart contract retrieve the next commitment value, $comm_{i-1}$. They then check that this value is not empty and that the commitment is the opening of the next commitment as follows:

$$comm_{i-1} \neq 0 \; and \; comm_i = h(comm_{i-1})$$

- **Commitment verification** The node then verifies that r_i and $comm_{i-1}$ lead to a winning ticket. This is the case if

$$h(t_h, comm_{i-1}, r_i) < P_w$$

where $t_h = h(t)$ is the ticket hash. The values are first ABI encoded, then hashed using keccak256 and last but not least converted to uint256.

If the check is not valid, the node should drop the packet. The final recipient of the packet does not receive a ticket because packet receipt is not incentivized by the HOPR protocol.

- **Issuer identity** The ticket signer must be same as the ticket issuer. Both node and smart contract verify whether the public key associated to the private key used to sign

$$T = (t, Sig_I(t))$$

is the issuer's public key. The packet is dropped if this test fails and ticket redemption reverts.

- **Prior redemption** The ticket must not have been already redeemed and the ticket index must be strictly greater than the current value in the smart contract (replay protection).

$$I_c < I$$

where I_c is the current value in the smart contract and I is the ticket index.

- **Valid ticket amount** The amount of ticket must be greater than 0:

$$\sigma > 0$$

where σ is the ticket amount.

- **Liquidity check** The channel must have enough funds to cover the value transfer for the ticket:

$$C_b > \sigma$$

where C_b is the channel balance.

Finally, the smart contract also performs an epoch check:

- **Epoch check** The ticket epoch, α, and channel epoch, ζ, must be equal to the current values in the smart contract

$$\alpha = \alpha_c \ \ and \ \ \zeta = \zeta_c$$

  where α_c and ζ_c represent the current values in the smart contract.

5 Incentivization Mechanism

The HOPR protocol provides incentives to nodes in the network to achieve correct transformation and delivery of mixnet packets. This is accomplished through a combination of proof of relay, a novel mechanism which is cost effective and privacy preserving, and probabilistic payments, together with an on-chain commitment. The high-level overview of the motivation behind this incentive scheme was covered in the introduction. This section focuses on the technical details used to implement the mechanism.

Construction

- Every packet is sent together with a ticket.
- Each ticket contains a challenge.
- The validity of a ticket can be checked on receipt of the packet, but the on-chain logic enforces a solution to the challenge stated in the ticket.
- The challenge can be solved after the packet has been forwarded.

5.1 Proof of Relay

HOPR incentivizes packet transformation and delivery using a mechanism called **proof of relay**. This mechanism guarantees that a node's relay services are verifiable.

Secret sharing Each node derives two keys, s_i^{own} and $_i^{ack}$, by using the s_i as given by the SPHINX packet (see the key derivation section for more details). s_i^{own} and s_{i+1}^{ack} serve as key shares of a 2-out-of-2 secret sharing between a node n_i and the next downstream node n_{i+1} along the chosen path. Once a node knows *both* key shares, s_i^{own} and s_{i+1}^{ack}, it is able to reconstruct $s_i^{response}$ to redeem the received ticket on-chain.

Whilst s_i^{own} is derivable upon reception of a packet, s_{i+1}^{ack} requires the cooperation of the next downstream node n_{i+1} and is sent as an *acknowledgement* if n_{i+1} has received the transformed packet and considers both the packet and embedded ticket valid.

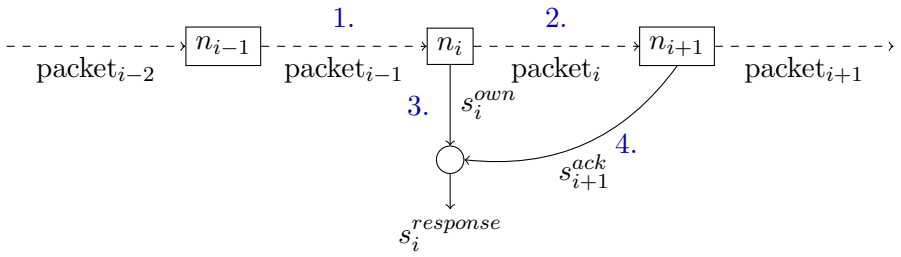

Figure 12: 1. Node n_i receives $packet_{i-1}$, validates it, transforms it and 2. sends it to node n_{i+1}. 3. While processing $packet_i$, node n_i derives s_i^{own} and 1. once node n_{i+1} considers $packet_i$ valid, it *acknowledges* the receipt of $packet_i$ and thereby reveals s_{i+1}^{ack} to node n_i which allows node n_i to reconstruct $s_i^{response}$.

Challenge response Tickets sent next to a mixnet packet include a challenge C_i which is computed as

$$C_i = s_i^{response} \cdot G = (s_i^{own} + s_{i+1}^{ack}) \cdot G$$

where G refers to the base point and $\cdot$ means scalar multiplication on the curve. Hence, in order to solve the challenge, it is necessary to know s_i^{own} as well as s_{i+1}^{ack}.

Challenge and Hint Once a node receives a packet, it is able to derive s_i^{own} but it is unable to decide whether s_{i+i}^{ack} will ever lead to $s_i^{response}$ that solves the challenge. Since the underlying field preserves the distributivity, it holds that

$$C_i = s_i^{response} \cdot G = (s_i^{own} + s_{i+1}^{ack}) \cdot G = s_i^{own} \cdot G + s_{i+1}^{ack} \cdot G$$

Hence by knowing $hint_i = s_{i+1}^{ack} \cdot G$, the node can verify that $s_i^{own} \cdot G + hint_i = C_i$ and thereby check that the creator of the challenge must have known a value $\tilde{s}_{i+1}^{ack}$ that led to $hint_i$. Due to the infeasibility of inverting scalar multiplication on the chosen curve, knowing $hint_i$ does not reveal s_{i+1}^{ack}. Hence, by embedding $hint_i$ into the part of β within the SPHINX packet that is readable by node n_i, the sender makes the validity of the embedded challenge verifiable.

As the next downstream node would not accept a packet without a ticket[2], the node n_i not only needs to transform the packet but must also issue a ticket to

[2]By default, nodes only forward packets that include incentives. Nevertheless, the protocol does not prevent them from processing packets without enforcing an incentive.

the next downstream node n_{i+1}. Therefore, it needs to know which challenge $\tilde{C}_{i+1}$ to put into the ticket issued for node n_{i+1}. As in the previous section, this is done with the help of the creator of the packet, who embeds C_{i+1} into the part of the SPHINX packet that is readable by node n_{i+1}.

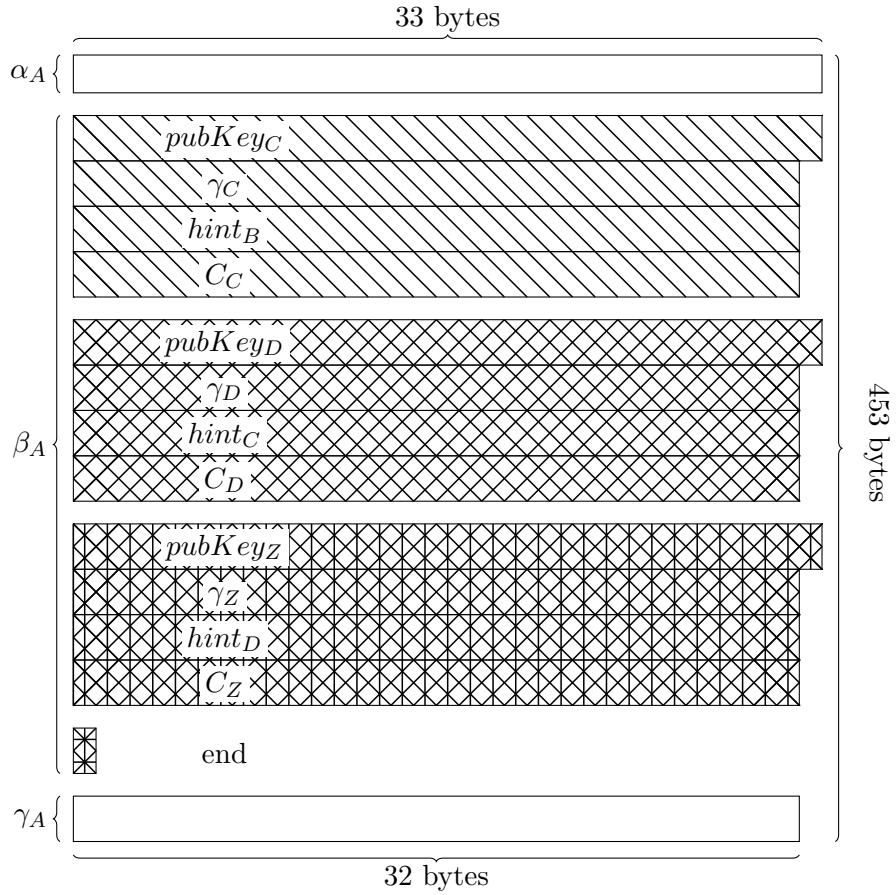

Figure 13: Mixnet packet header with PoR fields that is sent from the sender A to B and supposed to be forwarded through nodes C, D to Z.

By chaining this principle, nodes are forced to *always* issue a ticket to the next downstream node because they are unable to claim their own incentive without the help of the next downstream node.

Since the very last node, namely the final recipient, of the packet, does not need to forward the packet to anyone else, it has no direct incentive to acknowledge tickets, hence there are no direct consequences for not acknowledging packet. In its current version, the protocol does not prevent this kind of behaviour, but research is being conducted into the necessity and feasibility of solving this issue via a reputation system.

5.2 On-chain Commitment

HOPR uses a commitment scheme to deposit values on-chain and reveal them once a node redeems an incentive for relaying packets. This comes with the benefit that the redeeming party must disclose a secret that is unknown to the issuer of the incentive until it is claimed on-chain. The **opening** Open and the **response** r to the proof-of-relay challenge (called *nextCommitment* and *proofOfRelaySecret* in the smart contract) are then used to prove to the smart contract that the node has a legitimate claim to the funds. HOPR uses a computationally hiding and binding commitment scheme:

Commitment phase Once a node is the destination of a HOPR unidirectional channel, it generates a master key $comm_0$ randomly and uses it to create an iterated commitment $comm_i$ such that for every $i \in \mathbb{N}_0$ and $i > 0$ it holds that

$$\mathsf{Open}(comm_i, comm_{i-1}) = \top,$$

which means opening $comm_i$ with $comm_{i-1}$ holds true.

The iterated commitment is computed as

$$comm_n = h^n(comm_0),$$

where h is a pre-image resistant hash function (we use the keccak256 hash function, which is also used in Ethereum). The master key should be pseudorandom, such that all intermediate commitments $comm_i$ for $i \in \mathbb{N}_0$ and $0 < i \leq n$ are indistinguishable for the ticket issuer from random numbers of the same length. This is necessary to ensure that the ticket issuer is unable to determine whether a ticket is a winner or not when issuing the ticket. This makes it infeasible for the ticket issuer to tweak the challenge to such that it cannot be a winner.

When dispatching a transaction that opens the payment channel, the commitment $comm_n$ is stored in the channel structure in the smart contract and the smart contract will force the ticket recipient to reveal $comm_{n-1}$ when redeeming a ticket issued in this channel. The number of iterations n can be chosen as a constant and should reflect the number of tickets a node intends to redeem within a channel.

Opening phase In order to redeem a ticket, a node must reveal the opening to the current commitment $comm_i$ that is stored in the smart contract for the channel. Since the opening $comm_{i-1}$ allows the ticket issuer to determine whether a ticket is going to be a winner, the ticket recipient should keep $comm_{i-1}$ until it is used to redeem a ticket. Tickets lead to a win if:

$$\mathsf{h}(t_h, r_i, comm_{i-1}) < P_w,$$

where P_w is the ticket's winning probability and

$$t_h = \mathsf{h}(t) \; and \; \mathsf{Open}(comm_i, comm_{i-1}) = \top.$$

Since $comm_0$ is known to the ticket recipient, the ticket recipient can compute the opening as $comm_{n-1} = \mathsf{h}^{n-1}(comm_0)$. On redeeming a ticket, the smart contract verifies that

$$\mathsf{Open}(comm_i, comm_{i-1}) = \top$$

and sets $channel.comm[redeemer] \leftarrow comm_{i-1}$. Thus the next time the node redeems a ticket, it must reveal $comm_{i-2}$. In addition, each node is granted the right to reset the commitment to a new value. This is particularly necessary once a node reveals $comm_0$ and therefore is with high probability unable to compute a value r such that

$$\mathsf{Open}(comm_0, r) \neq \bot,$$

where $\bot$ represents the truth value "false". Since this mechanism can be abused by the ticket recipient to tweak the entropy used to determine whether a ticket is a winner or not, the smart contract keeps track of resets of the on-chain commitment and sets

$$channel[redeemer].ticketEpoch \leftarrow channel[redeemer].ticketEpoch + 1,$$

thereby invalidating all previously unredeemed tickets.

5.3 Probabilistic Payments

In traditional payment channels, two parties A and B lock some funds within a smart contract, make transactions off-chain and only commit the aggregation on-chain. Thus, a payment channel is bidirectional, which means both A and B can send and receive transactions within the same payment channel. The HOPR protocol uses unidirectional payment channels to implement bidirectional payment channel behaviour, where one payment channel is created from A to B and another from B to A. The payment channel creator is the sole owner of funds in the payment channel and the only one able to create **tickets**, encapsulated funds which are described in detail in Section 4. A payment channel created from party A to party B is different from a payment channel created from party B to party A.

$$A \rightarrow B \neq B \rightarrow A$$

This separation reflects the directional nature of packets flowing through the network. It also brings the advantage that each payment channel's logic is easier to verify.

Acknowledgements are messages which allow every node to acknowledge the processing of a packet to the previous node. This acknowledgement (ACK) contains the cryptographic material needed to unlock the possible payout for the previous node. Note that an acknowledgement is always sent to the previous node, and using acknowledgments with vanilla payment channels results in accumulated incentives, where the latest acknowledgement contains all previous incentives plus the incentive for the most recent interaction, as explained below:

$$value(ACK_n) = \sum_{i=1}^{n} fee_{packet_i}, \tag{1}$$

where n is the total number of mixnet packets transformed.

An issue arises when B receives ACK_n for $packet_n$ before sending $packet_{n-1}$. At this point B would have no incentive to process $packet_{n-1}$ rather than $packet_n$. To avoid such false incentives, the HOPR protocol utilizes probabilistic payments. A *ticket* can be either a win or a loss, determined based on some winning probability lower than 1. This means nodes are incentivized to continue relaying packets, as they do not know which tickets will result in a payout. From a node's perspective, each ticket has the same value until it is claimed; therefore, the HOPR protocol encourages nodes to claim tickets independently from each other.

$$value(ACK_i) = value(ACK_j) \quad for \quad i, j \in \{1, n\} \tag{2}$$

If we assume constant costs, there is no added value in pretending packet loss or intentionally changing the order in which packets are processed. On the contrary, a node would reduce its potential payouts if it were to forward packets slowly or not at all.

5.4 Payment Channel Management

For node A to transfer packets to node B, it must first open a payment channel. There are four distinct payment channel states, represented in the following scheme:

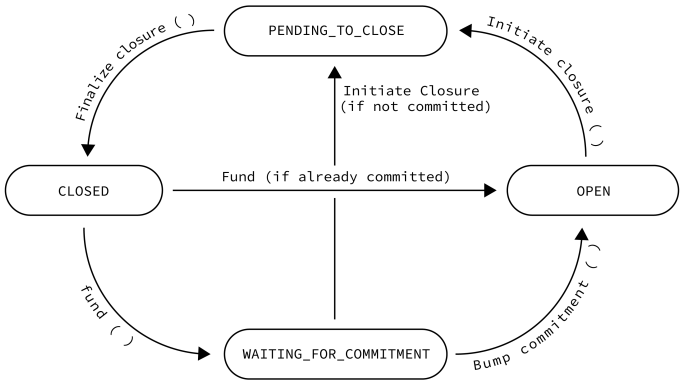

Figure 14: Payment channel states

Initially, each payment channel is *Closed*.

Opening a channel Node A can open a channel by transferring funds to the payment channels contract *HoprChannels* and including the following *userdata*:

$$[A : address, B : address, \lambda : uint8, \mu : uint8], \mu = 0,$$

where λ is the amount to be staked by A. This call will trigger an on-chain event *ChannelFunded* and open a unidirectional payment channel from A to B. The payment channel will start in state *Waiting for commitment*. The destination address of the payment channel must now set an on-chain commitment in order for the payment channel between both parties to become *Open*. This is done by B calling the *bumpChannel()* function to make a new set of commitments towards this payment channel. This call will trigger an on-chain event *ChannelOpened* and bumps the ticket epoch to ensure tickets with the previous epochs are invalidated. Every time the channel changes its state, an on-chain event *ChannelUpdated* is emitted.

Redeeming tickets As long as the channel remains open, nodes can claim their incentives for forwarding packets via tickets. Tickets are redeemed by dispatching a *redeemTicket()* call to an *Open* payment channel.

If B tries to redeem a ticket from the channel $A \rightarrow B$ (spending channel), but there is an open channel $B \rightarrow A$ (earning channel), B's rewards will be transferred to $B \rightarrow A$ (earning channel). Otherwise, rewards will be sent directly to B.

Closing a channel Nodes can close a payment channel in order to access their previously staked funds. Only the payment channel creator can initi-

ate the process by calling *initiateChannelClosure()*. This changes the state to *Pendingtoclose* and triggers a grace period during which the destination node can redeem any unredeemed tickets. Nodes should actively monitor blockchain events to be aware of this payment channel state change.

Once the grace period has elapsed, the payment channel creator can call *finalizeChannelClosure()* which changes the payment channel to *Closed*. When a payment channel is closed, the remaining funds are automatically transferred to the payment channel creator. The channel epoch increments, meaning any unredeemed tickets can no longer be redeemed.

6 Cover Traffic

Cover traffic (CT), also known as chaff, is randomly generated packets injected into a mixnet to increase its anonymity set. Since cover traffic is transported and mixed using the same procedures as 'real' packets, cover traffic is indistinguishable from real traffic. Deploying cover traffic makes it more difficult for an attacker to conduct passive attacks against the network. This is a direct result of there being more traffic: passive attacks such as traffic analysis attacks become more expensive the more traffic there is to analyse.

Cover traffic also increases the anonymity level of the mix network by improving sender-recipient unlinkability. This is particularly important in the early stages of HOPR's lifecycle, since it can be reasonably expected that traffic through the network will initially be low. Without cover traffic, it would be easier for a global passive adversary to link the sender and recipient of a particular packet.

A drawback of cover traffic is that it increases bandwidth overhead, which ultimately reduces the capacity of the network. This trade-off between capacity, latency, and anonymity is a result of the "anonymity trilemma" [Das et al., 2018]. Since HOPR is a privacy network, it is reasonable to prioritise anonymity here. However, it is important to mitigate the deleterious effects of cover traffic as far as possible, particularly avoiding wasted transmissions. This is achieved through judicious choice of which nodes to open cover traffic channels to and select paths to, based on node importance, and closing covert traffic channels to nodes which are found to be offline.

6.1 Cover traffic nodes

Cover traffic nodes (CT nodes) are HOPR nodes with a cover traffic service enabled, initially run by the HOPR Association (or third parties sponsored by the HOPR Association) for the purpose of generating cover traffic. In the future, anyone who fulfils the requirements for running a CT node in the HOPR network

will be able to start one.

6.2 Opening cover traffic channels

CT nodes must open and fund payment channels before they can send cover traffic packets via that counterparty. The number of channels a CT node can open is predefined and cannot be exceeded. Whenever payment channels are closed, the CT node replaces them by opening new channels, provided it still has funds. Channel opening is randomly weighted according to the importance of a node. A node's importance, $\Omega(N)$, is defined as follows:

$$\Omega(N) = st(N) * sum(w(C), \forall\ outgoingChannels(N))$$

where

$$st(N) = uT(N) + tB(N)$$

and

$$w(C) = \sqrt{(B(C)/st(Cs) * st(Cd))}$$

where N is the node, w is the channel's weight, st is the number of HOPR tokens staked by the node, and C is a payment channel between two nodes in the HOPR network. For this channel, Cs is the node which opened the payment channel, while Cd is channel counterparty. $uT(N)$ is the balance of unreleased tokens for a node N. B is the channel balance and tB is the sum of the outgoing balances of each channel from node N.

In accordance with this definition, a node's importance score increases with the channel balances of channels that node shares with other nodes with high total stake. This means that the odds of channel being opened to a particular node is not simply proportional to that node's stake. Cover traffic packets are allocated in proportion to a node's importance rather than channel stake as a way to mitigate against selfish node operators seeking to maximize their own profits.

The way importance and weight are calculated prevents nodes from opening a few minimally funded channels to nodes with large stake and incentivizes nodes to stake in as many channels as possible, including re-allocating their stake to their downstream nodes to receive cover traffic there. This discourages centralization and prevents "thin traffic", which limits the size of the anonymity set and increases the risk of traceable packets, even with high per-hop latency.

Since distributing cover traffic comes at a cost to network bandwidth, it is important to try and minimise wasted transmission. Therefore, CT nodes distribute cover traffic based on node importance rather than node uptime. This is because individual nodes are liable to suddenly lose connectivity. Of course, this distribution method will still result in sending cover traffic to offline nodes. If a

mix node is found to be offline, the channel to that node is closed and new one to a different node is opened. The offline node will have to wait until the CT node is ready to open a new channel to stand a chance of being reselected.

6.3 Path selection and payout

After opening channels, the CT node sends cover traffic packets at regular intervals. The path selection algorithm used for cover traffic is the same one used to select nodes for real traffic (see Section 2 for more details) where nodes are selected at random but weighted by importance, starting at the CT node.

The first relay node is chosen with a probability proportional to the amount funded by the CT node, so every node has an equal chance of getting selected. For subsequent hops, nodes are selected with a probability weighted proportional to their importance. We use a weighted priority queue of potential paths to choose the next node. If the queue is empty then it fails.

As with real traffic, rewards are distributed to all selected nodes in a path as tickets which can be redeemed for HOPR tokens with a certain probability (see Section 4). Thus there is a direct correlation between a node's importance and the cover traffic rewards it receives, ensuring that rewards are distributed fairly. These cover traffic rewards also serve as an incentive for node operators to open meaningful payment channels instead of, e.g., Sibyls or nodes with insignificant stake who would thus not be able to relay a lot of traffic.

6.4 Closing cover traffic channels

To minimize wasted cover traffic, CT nodes will close any channel deemed likely to result in failed distribution and open new channel in its place. A cover traffic channel is closed and a new one opened if any of the following factors falls below its threshold:

- **Channel balance** The channel balance must remain higher than the minimum stake value, defined as:

  $$B(C) < L * \sigma$$

  where $B(C)$ is the channel balance, L is the path length, and σ is the ticket payout amount, which is currently hardcoded as 0.1 HOPR per ticket.

- **Connection quality** Connection quality, q, is defined as the fraction of 'pings' the node has responded to within a 5 second cut-off. q must not drop below the defined upper bound.

- **Number of failed packets** The number of failed packets must stay above the failed packet threshold.

If a node is detected as having fallen below any of these thresholds, the CT node calls *initiateChannelClosure()* and emits two events: *ChannelUpdate* and *ChannelClosureInitiated*.

7 Conclusion

7.1 Privacy, Reliability, and Scalability

This paper explained how the HOPR protocol, particularly its proof-of-relay mechanism, provides a first solution to building a scalable and robust incentivized network which provides full anonymity with minimal trade-offs in latency and bandwidth. Although HOPR is indebted to previous developments in the space, particularly the Sphinx packet format and the Ethereum blockchain, we believe its unique combination of probabilistic payments and verifiable but unlinkable cryptographic tickets is the first viable approach to creating a global privacy network which does not rely on any centralized entity to provide data transmission or incentive management.

The HOPR protocol is still under active development, with much research and implementation work still to be completed, but a working proof of concept is already live and available to use.

7.2 Future work

7.2.1 Path position leak

In HOPR, payments are performed at each hop along a packet's route. These incentives weaken the unlinkability guarantees inherited from the Sphinx packet format [Danezis and Goldberg, 2009] as they reveal the identity of the packet sender, who must transfer these incentives using their signature. To solve this problem, HOPR forwards incentives along with the packet.

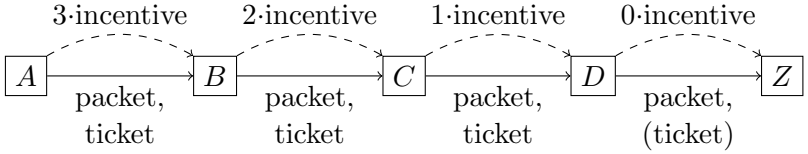

Figure 15: Incentive workflow

However, this leaks the relayer's position within the selected path, since the value of the ticket is set according to the current relay fee and the number of intermediate hops, more precisely

$$amount := \frac{(hops - 1) * relayFee}{winProb}$$

This leak is considered low severity, but further research will be conducted on the subject.

7.2.2 Reputation (aggregated trust matrix)

In HOPR, we assume the majority of nodes are honest and act properly. Nevertheless, there might be nodes who actively try to attack the network by:

- Dropping packets or acknowledgements
- Sending false packets, tickets, or acknowledgements

Since nodes need to monitor the network to select paths, they need to filter nodes that behave inappropriately. To help nodes achieve this, HOPR is considering implementing a transitive reputation system which assigns a score to each node that acts as a relayer. A node's behaviour will affect its reputation, which will in turn affect its probability of being chosen as a relayer.

Transitive trust evaluation

Reputation within a network can be defined as "a peer's belief in another peer's capabilities, honesty, and reliability based on other peers recommendations" [Wang and Vassileva, 2003]. Trust is represented by a triplet (trust, distrust, uncertainty) where:

- Trust: $td^t(d, e, x, k) = \frac{n}{m}$ where m is the number of all experiences and n are the positive ones
- Distrust: $tdd^t(d, e, x, k) = \frac{l}{m}$ where l stands for the number of the trustor's negative experience.
- Uncertainty = 1 - trust - distrust.

Commitment derivation

In the future $comm_0$ will be derived from the private key of the node as

$$comm_0 = \mathsf{h}(privKey, chainId, contractAddr, channelId, channelEpoch)$$

And there will be further research to determine whether such a design leaks information about the private key or not.

References

[Anderson and Biham, 1996] Anderson, R. J. and Biham, E. (1996). Two practical and provably secure block ciphers: BEARS and LION. In Gollmann, D., editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 113–120. Springer. https://doi.org/10.1007/3-540-60865-6_48.

[Chaum, 1981] Chaum, D. (1981). Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88. http://doi.acm.org/10.1145/358549.358563.

[Danezis and Goldberg, 2009] Danezis, G. and Goldberg, I. (2009). Sphinx: A compact and provably secure mix format. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 269–282. IEEE Computer Society. https://doi.org/10.1109/SP.2009.15.

[Das et al., 2018] Das, D., Meiser, S., Mohammadi, E., and Kate, A. (2018). Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 108–126. IEEE Computer Society. https://doi.org/10.1109/SP.2018.00011.

[Dingledine et al., 2005] Dingledine, R., Shmatikov, V., and Syverson, P. (2005). Synchronous batching: From cascades to free routes. In Martin, D. and Serjantov, A., editors, *Privacy Enhancing Technologies*, pages 186–206, Berlin, Heidelberg. Springer Berlin Heidelberg. http://web.mit.edu/arma/Public/sync-batching.pdf.

[Kuhn et al., 2019] Kuhn, C., Beck, M., and Strufe, T. (2019). Breaking and (partially) fixing provably secure onion routing. *CoRR*, abs/1910.13772.

[UN, 2018] UN (2018). The right to privacy in the digital age : report of the united nations high commissioner for human rights. page 17 p. http://digitallibrary.un.org/record/1640588.

[Wang and Vassileva, 2003] Wang, Y. and Vassileva, J. (2003). Trust and reputation model in peer-to-peer networks. *Peer-to-Peer Computing*. https://ieeexplore.ieee.org/document/913204.

[Wood, 2021] Wood, G. (2021). Ethereum: A secure decentralised generalised transaction ledger (istanbul version). https://ethereum.github.io/yellowpaper/paper.pdf.

A Key Derivation

During a protocol execution, a node derives a master secret s_i from the SPHINX header that is then used to derive multiple sub-secrets for multiple purposes by using the BLAKE2s hash function within HKDF. HKDF is given by two algorithms: extract and expand, where extract is used to extract the entropy from a given secret s, such as an elliptic-curve point, and produces the intermediate keying material (IKM). The IKM then serves as a master secret to feed expand in order to derive pseudorandom subkeys in the desired length.

A.1 Extraction

As a result of the packet creation and its transformation, the sender is able to derive a shared secret s_i given as a compressed elliptic-curve point (33 bytes) with each of the nodes along the path.

$$s_i^{master} \longleftarrow \mathsf{extract}(s_i, 33, pubKey)$$

By adding their own public key $pubKey$ as a salt, each node derives a unique s_i^{master} for each s_i.

A.2 Expansion

Each subkey s_i^{sub} is used for one purpose, such as keying the *pseudorandomness generator* (PRG).

$$s_i^{sub} \longleftarrow \mathsf{expand}(s_i^{master}, length(purpose), hashKey(purpose))$$

Usage	Purpose	Length	Hash Key (UTF-8)
SPHINX packet	PRG	32	`HASH_KEY_PRG`
	PRP	128 + 64	`HASH_KEY_PRP`
	Packet tag	32	`HASH_KEY_PACKET_TAG`
Proof of Relay	acknowledgement	32*	`HASH_KEY_ACK_KEY`
	ownKey	32*	`HASH_KEY_OWN_KEY`

The values marked with a * are treated as field elements, hence there exists a non-zero probability that expand produces a value outside of the field. In this specific case, the utilized hash key is repeatedly padded by "_" until expand returns a field element.

B LIONESS wide-block cipher scheme

HOPR uses the LIONESS [Anderson and Biham, 1996] wide-block cipher scheme with ChaCha20 as stream cipher and BLAKE2s as hash function, resulting in a key length of 128 bytes and 64 bytes for the initialisation vector.

The key k is split into four chunks of 32 bytes $k = k_1 \mid\mid k_2 \mid\mid k_3 \mid\mid k_4$, where k_1, k_3 are used as keys for the stream cipher and k_2, k_4 are used to key the hash function. The same applies to the initialisation vector iv which is split into four chunks of 16 bytes $iv = iv_1 \mid\mid iv_2 \mid\mid iv_3 \mid\mid iv_4$ with iv_1, iv_3 as an initialisation vector for the stream cipher and iv_2, iv_4 as an initialisation vector for the hash function.

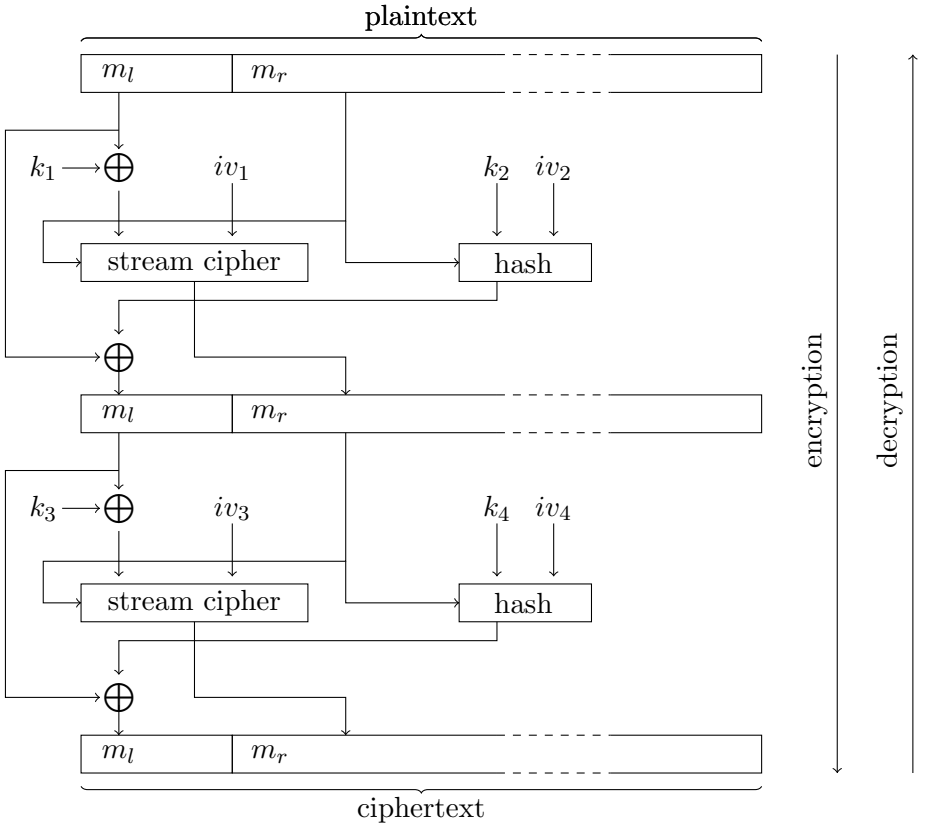

Figure 16: Schematic overview of encryption and decryption of the LIONESS [Anderson and Biham, 1996] scheme

LIONESS is an unbalanced Feistel scheme, hence decryption is achieved by applying the operations used for encryption in the inverse order. In contrast to other Feistel schemes, the plaintext m is not split equally and the size of the parts depend on the output length of used hash function.

As HOPR payloads have a size of 500 bytes, messages are split as $m = m_l \mid\mid m_r$ where $|m_l| = 32$ bytes and $|m_r| = 468$ bytes.

C Pseudo-random generator - PRG

tbd