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

Dr. Sebastian Bürgel, Robert Kiel

November 2019, V1

0.1 OnChain Commitment

0.2 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).

0.3 Sphinx Packet Format

A sphinx packet consists of two parts:

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

Notation Let k be a security parameter. An adversary will have to do about 2^k work to break the security of Sphinx with non negligible probability. We suggest using $k = 128$. Let r be the maximum number of nodes that a Sphinx mix message will traverse before being delivered to its destination. G is a prime order cyclic group satisfying the Decisional Diffie-Hellman Assumption. The element g is a generator of G and q is the (prime) order of G, with $q \approx 2^k$. G^* is the set of non-identity elements of G. h_b is a hash function which we model by random oracles such that: $h : G^* \times G^* \to \mathbb{Z}_q^*$ where $\mathbb{Z}_q^*$ is the field of non-identity elements of $\mathbb{Z}_q$ (field of integers). Each node $n \in \mathbb{N}$ has a private key $x_n \in \mathbb{Z}_q^*$ and a public key $y_n = g^{x_n} \in G^*$ where $\mathbb{N} \subset \{0,1\}^k$is a set of mix nodes identifiers.

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 β_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). The unblinding works as follow:

1. B computes the keyed-hash of the encrypted routing information β_0 as $HMAC(s_0, \beta_0)$ and compares with the integrity tag γ_0 attached in the packet header. If the integrity check fails because the header has been tampered with, the packet is dropped. Otherwise, the mix-node proceeds to step 2.

2. B is now ready to decrypt the attached β_0. In order to extract the routing instructions, the mix-node B first appends a zero-byte padding at the end of β_0 and decrypts the padded block of routing information B by XORing it with $(h(s_0))$. Where $\varrho : \{0,1\}^k \rightarrow \{0,1\}^{(2r+3)k}$ is a pseudo-random generator (PRG) and $h_\varrho : G^* \rightarrow \{0,1\}^k$ is a hash function used to key ϱ .

3. (B) parses the routing instructions from (A) in order to obtain the address of the next mix-node (C), as well the new integrity tag γ_1 and β_1, which should be forwarded to the next hop.

4. (B) blinds the key share $a_0 = g^x$ by setting it to $a1 = g^{xb_0}$.

(B) also removes one layer of encryption from the payload. The payload δ_0 in the Sphinx packet is computed using a wide-block cipher to ensure that, if an adversary modifies the payload at any point, the message content is irrecoverably lost.
Once (C) receives the packet, it derives the shared key s_c and removes the blinding, extracts the public key of (D) and deletes the routing information from the packet. Afterwards, it fills the empty space with its own blinding which is different from the one of (B).
Same happens at (D): key derivation, unblinding, deleting, shifting, decryption and blinding.
Last but not least, the packet arrives at (E), the final destination of the packet. Like the other nodes, (E) first derives its shared key s_E and removes the blinding. In contrast to the previous nodes, namely (B), then (C), then (D), it finds a message that symbolizes the end of path and tells (E) that it's the recipient.

Integrity In addition to the routing information that is necessary to determine to which the transformed packet should be sent, each node along the path receives an authentication tag δ_i in the form of a message authentication code (MAC).
By using the derived shared secret, e.g. s_b, each node is able to recompute the authentication tag and check the integrity of the received packet as follows:

$$\delta_0 = HMAC(s_0, \beta_0)$$

The integrity tag encoded in the header allows checks only on whether the packet header was not also modified. The authentication tags are computed

by the sender of the packet and cover also the blindings that nodes add to the packet after transforming it for the next downstream node. This is possible as the sender knows all shared secrets and the blindings are generated pseudo-randomly by using the shared secrets.

0.4 Tickets

Desired properties

- Tickets must be bound to specific payment channels, i.e. they cannot be spent in foreign payment channels
- Tickets must be redeemable at most once
- Ticket must lose its value once the payment channel is closed and must remain invalid if the channel is reopened
- The validity of tickets must be verifiable at ticket reception
- Ticket win must not be determinable at ticket issuance
- Ticket win must not be determinable before the response to the given challenge is known
- Tickets must lose its value once the on-chain commitment is reset
- The winning probability must rely on entropy that is only known by the issuer but not by the recipient and on entropy that is known by the recipient but not by the issuer
- Ticket issuance and ticket redemption must be independent of each other
- Ticket issuance must be non-interactive
- Ticket issuance and ticket redemption must be efficiently computable and the required cryptographic building blocks must be built-in functions of Ethereum.

Nodes that have staked funds within a payment channel can issue tickets that are used for payment to other nodes.

0.4.1 Ticket issuance

The issuer selects the recipient of the payment and retrieves the current epoch counter of the recipient from the blockchain. In addition, the issuer determines the channel identifier $channelId$ of the channel between the issuer and the recipient and retrieves the corresponding channel iteration counter from the blockchain. The channel iteration counter is used to give each reincarnation of the payment channel a new identifier such that tickets issued for previous instances of the channel lose their validity once a channel is reopened.
Afterwards the issuer sets:

- Value of the ticket
- Winning probability
- Challenge

If the issuer is also the sender of the packet, it can set the challenge arbitrarily. If the issuer receives the packet from another node and attempts to issue a ticket for the next downstream node, the challenge is given together with the routing information by the packet. The content of a ticket is given as the tuple:$(recipient, challenge, account.counter, amount, winProb, channel.teration, index)$ Let t be the content of a ticket. The issuer then signs the ticket with its private key and sends $ticket := (t, Sig_{Issuer}(t))$ to the recipient.

0.4.2 Ticket validation

Once a node receives a ticket, the ticket gets validated:

- **Economically:**
  - Does the ticket value fit to the currently staked money and the previously redeemed tickets?
  - Is the winning probability set reasonably?
  - Has the ticket already been processed?
- **Cryptographically:**
  - Is the signature valid?
  - Is the given challenge derivable?

Tickets are received together with packets which means that the recipient and the next downstream node share a secret s whose key shares s_a and s_b are derivable by the recipient and the next downstream node. The secret sharing between the issuer is "2-out-of-2" secret sharing, meaning that the reconstruction of the key requires both key shares, s_a and s_b.
Once the recipient transforms the packet, namely applying the elliptic curve math to extract the shared secret with the sender from Diffie-Hellman key exchange, it is able to compute s_a. The recipient is now also able to extract the routing information from the packet. This includes a hint to the value s_b given as $hint := s_b * G$ where G is the base-point of the utilized curve. Together with s_a, the node can verify that

$$ticket.challenge = s_a * G + hint$$

with

$$s_a * G + hint = s_a * G + s_b * G = (s_a + s_b) * G$$

This allows the recipient to verify that the promised value s_b 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. The node then takes the ticket and stores it within the database until it receives an acknowledgement containing sb from the next downstream node. The unacknowledged ticket is stored in the database under the hint to the promised value, namely $hint = s_b * G$, to make sure that the acknowledgement can be afterwards linked to the unacknowledged ticket.

0.4.3 Ticket redemption

Once a node receives an acknowledgement from another party, 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 can store the value but keeping it will most likely not bring any value.
Acknowledgements are sent on every packet, even if the ticket issuer is the sender of the packet and therefore does not need any acknowledgement to unlock a ticket. The node then computes the response to the challenge given in the ticket. Therefore, it fetches its part of the PoR secret sharing as computed from the corresponding relayed packet from the database.
Let sa be the value from the database and s_b be the value extracted from the acknowledgement. The response is then computed as $response := (s_a + s_b) * G$. 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 as it does not give any additional value.
If none of the aforementioned checks have failed, the node submits the ticket to the on-chain logic by calling $redeemTicket()$. The on-chain logic repeats the previously done checks and additionally checks whether the:

- signature of the ticket issuer is valid
- ticket has been already spent (replay protection)
- Channel exists
- Channel balance towards the ticket recipient is sufficient to cover the costs for the ticket
- Ticket index is strictly greater than the current value in the smart contract (reorder protection)

The replay protection is done by storing a fingerprint of the ticket in the storage of the on-chain logic. The fingerprint is given as $fingerprint := hash(ticketData)$. If $fingerprint$ exists, the on-chain logic must reject the ticket because the ticket could be a double-spend attempt.

The reorder protection is necessary to enforce a total order in which ticket can be redeemed. This means that not every ticket must be redeemed but if a ticket is redeemed, then it invalidates all tickets between the index of the ticket and the index of the previously redeemed ticket. If the on-chain were not doing the check, nodes are able to check their opening value open against multiple tickets which increases the winning probability and allows nodes to increase their pay-out. If none of the previous checks have failed, the on-chain logic performs the following state changes:

- Update channel balance
- Set $channel.counter[issuer]$ to $ticket.index$
- Write ticket fingerprint in storage
- Update on-chain commitment

0.4.4 Ticket issuance vs Ticket redemption

The values that are used to sign the ticket and the ones that are needed to redeem the ticket are slightly different, more precisely: *channel.iteration* and *account.counter* are already stored within the on-chain logic, hence they don't need to be part of the redeem transaction. Analogously *response* and *challenge*. The signature is computed over *challenge* but the redeemer provides response from which the smart contract computes challenge.
This means that $ticketHash$ is computed from given values by the redeem transaction, the storage of the on-chain logic as well as computed values. The validity of the signature is then checked against $ticketHash$, hence *channel.iteration*, *account.counter* as well as *challenge* are checked implicitly. This does not reduce the security of the signature verification since the probability that an adversary is able to find values for *channel.iteration* or *account.counter* or *response* that result in a valid signature can be assumed to be negligible.