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

Amira Bouguera, Robert Kiel, Dr. Sebastian Bürgel

May 2021, V2

Abstract

1 Introduction

Internet privacy, also commonly referred to as online privacy, is a subset of data privacy and a fundamental human right. We consider privacy to revolve around control, use and disclosure of one's personally identifiable information. However, our increasingly technologically-driven world puts great pressure on privacy. This is why several actors in the space are continuously developing different solutions to achieve anonymous communication and thus leveraging internet privacy.

1.1 State of the art

1.1.1 Privacy protocols

VPN or Virtual Private Network is a privacy technology that allows creating a secure connection on the internet by building an encrypted tunnel between the client and server. VPN also provides a new IP address (the one of the proxy) to help bypass censorship and geolocalization blocks. Due to the centralized trust model VPNs has, they suffer from inherent weaknesses like being a single point of failure. They are also vulnerable to powerful network eavedropers who can track the routed network traffic.
Those privacy concerns have motivated the creation of a new model of VPNs, dVPNs (Decentralized Virtual Private Networks) with no central authority and backed by blockchain technology. Orchid, Sentinel and Mysterium are the most known projects in the blockchain ecosystem providing dVPN solutions. These projects however lack privacy, performance and reliability guarantees due to the fact that regular internet users can be both bandwidth providers and normal users. One of the main reasons behind using decentralized VPN instead of a centralized one is to implement a no traffic logging policy as a way to increase

privacy, this however introduces the risk of using the service illegally (terrorism, drug smugling, child pornography...etc) which holds service providers reliable to government authorities without them having any legal protection as large VPN providers would have. There have also been some incidents reported, where unaware dVPN users have been (ab)used as exit nodes through which DDoS attacks were performed. Similarly to VPN, users have no guarantee whether a dVPN might inspect, log, and share any of their traffic. A promising project called VPN^0 started by the Brave team which provides better privacy guarantees by leveraging zero knowledge proofs to hide traffic content to relay nodes with traffic accounting and traffic blaming capabilities as a way to combat the weaknesses of other dVPN solutions. This project is still research oriented though.

Another existing solution for network privacy is Tor, an open-source software for enabling anonymous communication. Tor is based on onion routing which encapsulates messages in layers of encryption and transmits them through a series of network nodes called onion routers. Tor however is susceptible to end-to-end correlation attacks conducted by an adversary who can eavesdrop the communication channels.These attacks reveal a wide range of information like the identity of the communicating peers. Another project based on onion routing is I2P peer-to-peer network. I2P has different design choices from those of Tor:

- Packet switched instead of circuit switched: Tor allocates connection to long lived circuits, this allocation does not change until either the connection or circuit closes. On the other hand, routers in I2P maintain multiple tunnels per destination which increases significantly the scalability and resilience against failures since packets are used in parallel.
- Unidirectional instead of bidirectional tunnels: which makes deanonymization harder since tunnel participants see half as much data in unidirectional tunnels and need two sets of peers to be profiled.
- Peer profiles instead of directory authorities: I2P's network information is stored in a DHT (information in the DHT is inherently untrusted) while Tor's relay network is managed by a set of nine Directory Authorities.

I2P are vulnerable to eclipse attacks since no I2P router has a full view of the global network (similar to other peer-to-peer networks) and they also protect against only local adversaries (like Tor) and thus vulnerable to timing, intersection and traffic analysis attacks. I2P have also showed to be vulnerable to sybil and predecessor attacks inspite of the different contermeasures implemented to defeat them.

Mixnets are overlay networks of mix nodes that route messages anonymously similarly to Tor. First mixnet paper in 1981 by David Chaum used a cascade topology where each node receives a batch of encrypted messages, decrypts, randomly permutes packets, and transfers them in parallel. Cascade topology

makes it easy to prove the anonymity properties of a given mixnet design for a particular mix, however, it does not scale well with respect to increasing mixnet traffic and is also susceptible to traffic and active attacks. Since then, research has evolved to provide solutions with low latency while still providing high anonymity by using a method called cover traffic. Cover Traffic is designed to hide communication messages among random noise. An external adversary able to observe the message flow should not be able to discriminate communication messages from random noise messages which increases privacy. What differentiates mixnets from Tor is that mixnets are designed to provide metadata protection from global network adversaries by using cover traffic. Because mixnets add extra latency to network traffic, they are better-suited to applications that are not as sensitive to increased latency, such as messaging or email applications while applications like real-time video streaming are better suited for Tor.
One of the well known projects is Loopix. Loopix leverages cover traffic to resist traffic analysis while still achieving low- to mid-latency. To this end Loopix employs a mixing strategy that we call a Poisson Mix that is based on the independent delaying of messages, which makes the timings of packets unlinkable.

The goal of each one of these projects is to acheive low latency, low bandwidth overhead and strong anonymity or as we call it the anonymity trilemma. We present in the following a comparison table (from the Loopix paper) between different anonymous communication systems.

	Low Latency	Low Communication Overhead	Scalable Deployment	Asynchronous Messaging†	Active Attack Resistant	Offline Storage*	Resistance to GPA
Loopix	✓	✓	✓	✓	✓	✓	✓
Dissent [43]	×	×	×	×	✓	×	✓
Vuvuzela [42]	×	×	✓	×	✓	×	✓
Stadium [41]	×	✓	✓	×	✓	×	✓
Riposte [10]	×	×	✓	×	✓	×	✓
Atom [29]	×	✓	✓	×	✓	×	✓
Riffle [30]	✓	✓	×	×	✓	×	✓
AnonPoP [23]	×	✓	✓	×	×	✓	✓
Tor [19]	✓	✓	✓	✓	×	×	×

Figure 1: Comparison between anonymous communication systems

All the previous mentioned projects except VPN and dVPNs lack the economic incentive which could result in scaling issues and poor performance. Tor and I2P for example rely on donations and government funding only which covers the cost of running a node. This has discouraged volunteers to join the network and the number of routers in both networks hasn't increased much for the last few years. Mixnets are also based on a group of volunteer agents who lack incentives to participate. Some solutions have proposed adding digital coins to messages, such that each volunteer can extract only the digital coin designated as a payment for them. However, malicious volunteers can sabotage the system

by extracting and using their coins without performing their task which consists of forwarding anonymized messages since there is no verification whether the message arrives to its final destination or not. Bandwidth providers in dVPNs share their resources and are granted tokens accordingly as way of payment for their services. This is done using the blockchain technology. A good example of such technologies is Mysterium: an open source dVPN completely built upon a P2P architecture. Mysterium runs a smart contract on top of Ethereum to make sure that the VPN service is paid adequately.

1.1.2 Scalability Layer 2 protocols

Blockchain technology (mostly public blockchains like Bitcoin and Ethereum) suffers from a major scalability issue which is due to the fact that every node in the network needs to process every transaction, validate it and stores a copy of the entire state. The number of transactions Ethereum can process for example cannot exceed that of a single node which is currently 15 transactions per second.

There have been multiple solutions proposed to treat the scalability issue such as sharding and off-chain computation. Both of these solutions intend to create a second layer of computation in order to reduce the load on the blockchain mainnet.
Off chain solutions like Plasma, Truebit and state channels process transactions outside the Blockchain while still guaranteeing a sufficient level of security and finality. State channels are better known as "payment channels". In models like the "Lightning Network", a payment channel is opened between two parties by committing a funding transaction, followed by making any number of transactions that update the channel's funds without broadcasting those to the blockchain, then closing the channel by broadcasting the final version of the settlement transaction.

1.2 The HOPR vision

HOPR is a decentralized incentivized mixnet that leverages privacy by design protocols. HOPR aims to protect people's metadata privacy and give them the freedom to use internet services safely and privately. HOPR runs on top of the Ethereum blockchain and uses these mechanisms to ensure users privacy via incentivisation: sphinx packet format and packet mixing, proof-of-relay and probabilistic payments.

- *Privacy by design:* is an approach to systems engineering and calls for privacy to be taken into account throughout the whole engineering process. HOPR believes that the Internet is a public good – a digital commons that should be safe and secure for all its users. However, it is impossible to provide such privacy using the current Internet infrastructure. What is needed is a new privacy infrastructure on top of the existing Internet. This is what HOPR has built using the sphinx packet format and packet mixing.

- *Decentralization:* At the heart of HOPR, there is a global, decentralised network of nodes running the HOPR protocol. Decentralisation ensures that the network is independent, with no one single entity to influence its development or manipulate outcomes to their advantage. It also makes the network resilient, able to keep running even if a majority of nodes are damaged or compromised and very difficult, if not impossible, to shut down.

- *Incentivization:* This is biggest innovation of the HOPR design. Previous privacy technologies like Tor didn't incentivize node runners to provide service which discouraged new users to join. By incentivising users, HOPR will lower the barrier to adoption. Incentivisation also encourages good behavior as the only way to earn token rewards is to behave correctly and follow the protocol.

2 Threat Model

Although we assume that nodes in the HOPR network can communicate reliably, the network can still be damaged by malicious actors and node failures. We assume a threat model with byzantine nodes with either the ability to observe all network traffic and launch network attacks or inject, drop or delay messages.

There are different attack vectors which could threaten the security of HOPR network, in the following section we mention these attacks and the mitigation methods used by HOPR to resist them:

- **Sybil attacks:** An attacker uses a single node to forge multiple identities in the network, thereby bringing network redundancy and reducing system security. This attack is expensive to conduct in practice since the attacker must stake lots of HOPR tokens within each malicious node they create in order to increase their probability of being chosen as a relayer and thus attacking the network.

- **Eclipse attacks:** The attacker seeks to isolate and attack or manipulate a specific user that is part of the network. This is a common attack in peer-to-peer networks since nodes have a hard time identifying malicious ones as they don't have a global view of the whole network. The cost of launching an eclipse attack is high since HOPR nodes are constantly challenging other peers and keeping a reputation score for each node.

- **Camouflage attacks:** A malicious node pretends to be an honest one most of the time. When its reputation value reaches a high level, it occasionally attacks the system. Since the attacker needs a long time to gain enough reputation score and be selected. Based on this, the system can still perform well.

- **Observe-Act Attack:** The attacker observes the reputation score distribution of honest nodes, then control malicious nodes to act and have the same reputation score in order to increase the probability that most malicious nodes are chosen as relayers. This attack however reveals the identities of malicious nodes which conduct this attack and their reputation will be reduced if not loose their stake.

Security Goals

In addition to resisting the previous attacks, the HOPR protocol has been defined to meet these security goals which are inherited from the Sphinx packet format:

- Sender-receiver unlinkability: The inability of the adversary to distinguish whether $\{S_1 \rightarrow R_1, S_2 \rightarrow R_2\}$ or $\{S_1 \rightarrow R_2, S_2 \rightarrow R_1\}$ for any concurrently online honest senders S_1, S_2 and honest receivers R_1, R_2 of the adversary's choice.
- Resistance to active attacks: Resistance to active attacks like tagging and replay attacks where the adversary modifies and re-injects messages to extract information about their destinations or content.

3 Sphinx Packet Format

HOPR uses the SPHINX packet format [1] to encapsulate and route data packets in order to achieve sender and receiver unlinkability. The SPHINX packet format determines how mixnet packet are created and transformed in order to relay them to the next downstream node. Each sphinx packet consists of two parts:

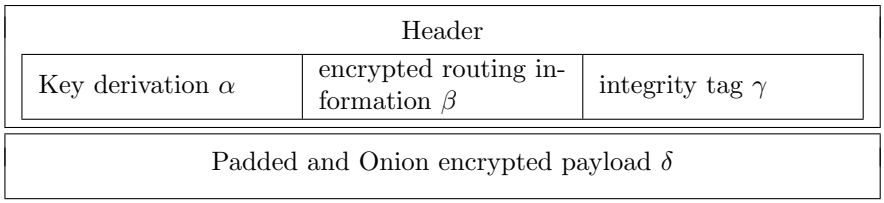

Figure 2: Sphinx packet format

3.1 Construction

We will explain in the following all the steps a sphinx packet goes through in order to arrive to its final destination starting from the key derivation in order to extract new shared keys with all the relay nodes to unblinding the routing information using those shared keys in order to find the public key of the next relay node and checking the routing information's integrity before sending it. Each node then deletes that information and replaces it with their own blinding

and decrypts one layer of the payload which has several layers of encryption similar to onion routing.

Notation: Let $\kappa = 128$ be the security parameter. An adversary will have to do about 2^κ operations to break the security of Sphinx with non negligible probability. 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, we use the secp256k1 curve. The element g is a generator of G and q is the (prime) order of G, with $q \approx 2^\kappa$. $G^*$is the set of non-identity elements of G. h_b is a pre-image resistant hash function used to compute blinding factors and modeled as a random oracle 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). We use the BLAKE2s hash function.

Each node n has a private key $x_n \in \mathbb{Z}_q^*$ and a public key $y_n = g^{x_n} \in G^*$. The α_i are the group elements which, when combined with the nodes' public keys, allow computing a shared key for each via Diffie-Hellman (DH) 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 DH shared secrets, b_i are the blinding factors.

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) and the final destination of the packet (Z).
(A) performs an offline Diffie-Hellman key exchange with each of these nodes and derives shared keys with each of them.
(A) computes a sequence of r tuples (in our case r=4)

$$(\alpha_0, s_0, b_0), ..., (\alpha_{r-1}, s_{r-1}, b_{r-1})$$

as follows:

- $\alpha_0 = g^x, s_0 = y_B^x, b_0 = h_b(a_0, s_0)$
- $\alpha_1 = g^{xb_0}, s_1 = y_C^{xb_0}, b_1 = h_b(a_1, s_1)$
- $\alpha_2 = g^{xb_0b_1}, s_2 = y_D^{xb_0b_1}, b_2 = h_b(a_2, s_2)$
- $\alpha_3 = g^{xb_0b_1b_2}, s_3 = y_D^{xb_0b_1b_2}, b_3 = h_b(a_3, s_3)$

Where y_B, y_C, y_D, y_Z are the public keys of the nodes B, C, D which we assume to be available to A .

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 (Z) that it is the final receiver of the message. The END message is a distinguished prefix byte that's added to the final receiver's Ethereum address. The serialization concatenates the x and y coordinates of the public key as follow:

$$02 + x - coordinate(16\ bytes) + y - coordinate(16\ bytes) if\ y\ is\ even$$

or

$$03 + x - coordinate(16\ bytes) + y - coordinate(16\ bytes) if\ y\ is\ odd$$

The routing information looks as the following:

$$\beta_{v-1} = (Z \| 0_{(2(r-v)+2)\kappa - |Z|} \oplus PRNG(s_{v-1})_{[0....(2(r-v)+3)\kappa-1]}) \| \phi_{v-1}$$

and

$$\beta_i = n_{i+1} \| \gamma_{i+1} \| \beta_{i+1}{}_{[0....(2r-1)\kappa-1]} \oplus PRNG(s_i)_{[0....(2r+1)\kappa-1]}$$

for $0 \leq i < v-1$

Such that Z is the destination address and $|Z|$ is its length. $v \leq r$ is the length of the path traversed by the packet where $|Z| \leq (2(r-v)+2)$. ϕ is a filler string such that

$$\phi_i = \{\phi_{i-1} \| 0_{2\kappa}\} \oplus PRNG(s_{i-1})_{[(2(r-i)+3)\kappa..(2r+3)\kappa-1]}$$

where $\phi_0 = \epsilon$ is an empty string.
ϕ_i is generated using the shared secret s_{i-1} and used to ensure the header packets remain constant in size as layers of encryption are added or removed. Upon receiving a packet, the processing node extracts the information destined for it from the route information and the per-hop payload. The extraction is done by deobfuscating and left-shifting the field. This would make the field shorter at each hop, allowing an attacker to deduce the route length. For this reason, the field is pre-padded before forwarding. Since the padding is part of the HMAC, the origin node will have to pre-generate an identical padding (to that which each hop will generate) in order to compute the HMACs correctly for each hop.

β_i is computed as the concatenation of Z and a sequence of padding which is then encrypted by XORing with the output of a pseudo-random number generator seeded with shared key s_{v-1} of node $v-1$. The result is finally concatenated with ϕ to ensure the header packets remain constant in size.
In the original sphinx paper, Z is concatenated with an identifier I and 0 padding sequence where I is used for SURBs (Single-Use-Reply Blocks) such that $I \in \{0,1\}^\kappa$. We don't use I since we don't currently use SURBs .

Since (A) has a shared secret with each of the nodes along the path, it is able

to derive blindings for each of them.
Each node along the path receives an authentication tag γ_i in the form of a message authentication code (MAC) which is encoded in the header.
Some padding is added at each mix stage in order to keep the length of the message invariant at each hop.

The mix header is constructed as follow:

$$M_i = (\alpha_i, \beta_i, \gamma_i)$$

Once (B) receives the packet, it derives the shared key s_0 by computing

$$s_0 = (\alpha_0)^b = (g^x)^b = (g^b)^x = y_B^x$$

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

Integrity check By using the derived shared secret s_i, each node is able to recompute the authentication tag and check the integrity of the received packet as follows:

$$\gamma_i = HMAC(s_i, \beta_i)$$

(B) computes the keyed-hash of the encrypted routing information β_0 as

$$\gamma_0 = HMAC(s_0, \beta_0)$$

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 the next step.
This integrity check allows to verify whether or not the header was modified.

Unblinding The unblinding works as follow:
(B) decrypts the attached β_0 in order to extract the routing instructions. First, (B) appends a zero-byte padding at the end of β_0 and decrypts the padded block of routing information B by XORing it with $PRNG(s_0)$ as follows:

$$(\beta_0 \| 0_{2\kappa}) \oplus PRNG(s_0)$$

(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.

Delete and shift After that (B) extracts the public key of (C), it deletes the routing information from the packet. Afterwards, it fills the empty space with its own blinding which is different from the one received from (A) by setting the key share α_0 to $\alpha_1 = g^{xb_0}$. (B) also computes β_1 as follows:

β_0		
n_1	γ_1	β_1

The first κ bits of β_0 will be n_1 itself, the next κ bits will be γ_1, and the remaining $(2r-1)\kappa$ bits of β_0 are shifted left to form the leftmost $(2r-1)\kappa$ bits of β_1; the rightmost 2κ bits of β_1 are simply a substring of an output of the PRNG function.
The new mix header is now ready to be sent to (C) or as defined node n_1:

$$M_1 = (\alpha_1, \beta_1, \gamma_1)$$

Encrypt & Decrypt The payload δ is where the actual message is hidden and is computed in different layers using a wide block cipher encryption algorithm and is decrypted at each stage of mixing.
The payload is constructed as follow:

$$\delta = 0_l \| \tau \| \delta$$

where τ is the authentication tag string which is generated arbitrarily and $|\tau| = 4$ is its length (let's suppose $\tau =$ "$HOPR$"). l is the 0 padding length where $0 <= l <= |\delta| - 4$

Let m be the block size in bits, which will typically be large. Let $H_k(\tau)$ be a keyed hash function with the key k for the payload τ, k consists of four independent keys k_1, k_2,k_3 and k_4 which (A) derives from the master keys s_0, s_1, s_2 and s_3. The sphinx uses Lioness wide block cipher scheme for encryption and decryption purposes. Let $S(\tau)$ be a pseudo random function (stream cipher) which given the input τ will generate an output of arbitrary length. τ is divided into two parts left L and right R whose sizes are $|L| = w$ and $|R| = m - w$ so we get $\tau = R \| L$.

Encryption

Decryption The decryption happens as follow:

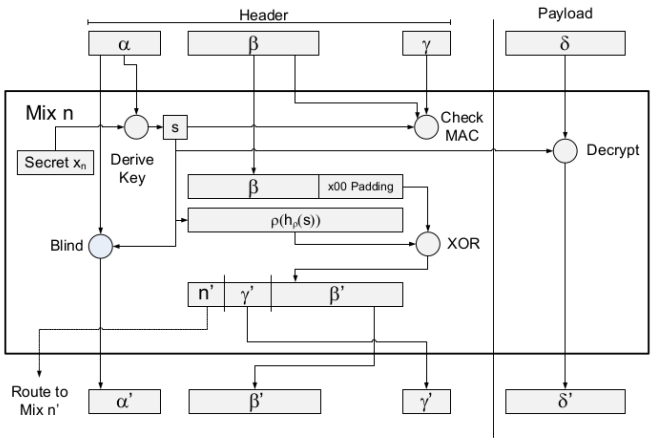

Figure 3: The processing of a Sphinx message

Same happens at (C) and (D): key derivation, unblinding, deleting, shifting, integrity check, decryption and blinding.

3.2 Implementation choices

Within HOPR, the following cryptographic primitives were used:

- **Cyclic group:** The cyclic group used in the HOPR Sphinx implementation is an elliptic curve group on the secp256k1 curve and thus operations will be done on the elliptic curve.
- **Hash function:** BLAKE2s hash function which is a cryptographic hash function faster than SHA-2 and SHA-3, yet is at least as secure as SHA-3 and produces digests of any size between 1 and 32 bytes.
- **MAC:** HMAC based on a hash function BLAKE2s.
- **Encryption scheme:** LIONESS [2] implementation, using BLAKE2s as a hash function and ChaCha20 as a stream cipher. We use the LIONESS PRP to implement π.
- **Padding:** In the original SPHINX paper, a sequence of only 0s is used for padding, this allows the last mix-node in the path to infer information about the length of the path and the last destination, hence breaking one of the security properties promised by Sphinx. In order to prevent this attack, We replace the 0-padding by a randomized padding for the last exit-mix node and always take $v = r$. This way, the exit node can't identify where the padding starts and thus won't be able to find the path length that preceeds the padding.

4 HOPR incentivization mechanism

HOPR incentivizes nodes in order to achieve correct transformation and delivery of mixnet packets. This is accomplished using a mechanism called "Proof-Of-Relay" with the following layer 2 solutions which are both cost effective and privacy preserving.

4.1 Probabilistic payments

In traditional payment channels, two parties A and B lock some funds within a smart contract, make multiple transactions off-chain and only commit the aggregation on-chain.

HOPR uses a concept called *acknowledgements* which allows every node to creates a message that acknowledges the processing of the packet to the previous node. This acknowledgement contains the cryptographic material to unlock the payout for the previous node. Note that the acknowledgement is always sent to the previous node - even if there was no payment.
The fact that we are using payment channels implies that the last HOPR acknowledgement contains all previous incentives plus the incentive for the most recent interaction

$$value_(ACK_n) = \sum_{i=1}^{n} fee_{packet_i}$$

where n is the total number of mixnet packets transformed.

If B received ACK_n before sending $packet_{n-1}$, it has no incentive to process $packet_{n-1}$ rather than $packet_{n-2}$.

To avoid this limitation of traditional payment channels, HOPR utilizes probabilistic payments

In probabilistic payments, the payouts use a concept called "tickets", a ticket can be either a win or a loss with a certain winning probability. This means nodes are incentivized to continue relaying packets as they don't know which ticket is a win.

HOPR uses a custom-made layer 2 solution. It is inspired by payment channels and probabilistic payments where incentives can be claimed independently:

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

Hence, there is no added value in pretending packet loss or intentionally changing the order in which packets are processed.

4.1.1 Channel management

Initially, each payment channel in HOPR is *closed* which means that in order to transfer packets, those channels have to be opened.

Opening a channel Nodes can open channels to other nodes by the following: A calls the method *fundChannel()*. Once the call has succeeded, an on-chain event *ChannelOpened* is thrown. The channel is now *open*.

$$fundChannel(A :< \lambda >, B :< \mu >)$$

where λ is the amount to be staked by A and μ is the amount to be staked by B. Both values can also be equal. A is also able to fund other channels which A is not part of.

Redeem tickets As long as the channel remains open, nodes can claim their incentives for forwarding packets which is represented as tickets (see ticket section). Tickets are redeemed by dispatching a *redeemTicket()* call. The balance of the channel is then updated according to the balance defined in the ticket.

Closing a channel Nodes can close a payment channel in order to access their funds. The way to do so is using a timeout. A can initiate the process by calling *initiateChannelClosure()*. This changes the state to *pending_timeout*. Other nodes will have now time to claim not yet claimed tickets. Once the timeout is done, any of the involved parties can call *withdraw()*. Alice can then call *finalizeChannelClosure()* which turns the channel state into *closed*.

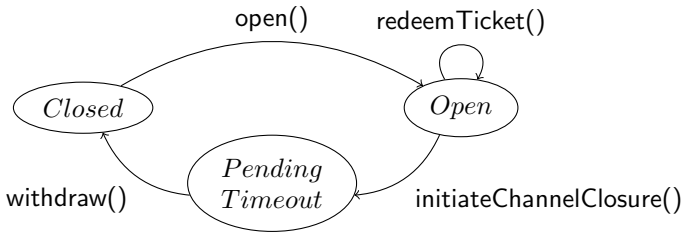

Figure 4: Channel workflow

4.2 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 discloses a secret that is unknown to the issuer of the incentive until it is claimed on-chain. The *opening* and the *response* to the PoR challenge are then used by the smart contract to determine whether the ticket has been redeemed or not.

Definition 4.2.1. *A commitment scheme* $Cm = (\mathsf{Commit}, \mathsf{Open})$ *is a protocol between two parties,* A *and* B*, that gives* A *the opportunity to store a value* $comm = \mathsf{Commit}(x)$ *at* B*. The value* x *stays unknown to* B *until* A *decides to reveal it to* B*.*
Hiding: *A commitment scheme is called* ***hiding*** *if it is infeasible for an adversary* Adv *to recover* x *from* $comm$*.*
Binding: *A commitment scheme is called* ***binding*** *if it is infeasible for an adversary* Adv *to find a value* x' *with* $x \neq x'$ *such that* $\mathsf{Open}(cm, x') \neq \bot$*.*

4.2.1 Setup phase

Once a node engages with another node in a payment channel and locks funds within that channel, it derives a master key $comm_0$ from its private key 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

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

The iterated commitment is computed as

$$comm_n = hash^n(comm_0)$$

where hash is a preimage-resistant hash function (we use keccak256 hash function which is used in Ethereum) and $comm_0$ is derived as

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

The master key is supposed to be pseudo-random 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 in order to ensure that the ticket issuer is unable to determine whether a ticket is a win 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 win.

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.

4.2.2 Opening phase

In order to redeem a ticket, a node has to 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 win, the ticket recipient should keep $comm_{i-1}$ until it is used to redeem a ticket. Tickets lead to a win if $\mathsf{hash}(t_h, r, comm_{i-1}) < P_w$ where $t_h = \mathsf{hash}(t)$ 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 $comm_{n-1} = \mathsf{hash}^{n-1}(comm_0)$. Once redeeming a ticket, the smart contract verifies that

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

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

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

Since this mechanism can be abused by the ticket recipient to tweak the entropy that is used to determine whether a ticket is a win or not, the smart contract keeps track on resets of the on-chain commitment and sets

$$channel.ticketEpoc[redeemer] \leftarrow channel.ticketEpoc[redeemer] + 1$$

and thereby invalidates all previously unredeemed tickets.

4.3 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).

5 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 or when the commitment is reset. A commitment is secret on-chain value that is used to verify whether a ticket is a win or not when it's revealed in order to redeem it.

5.1 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:

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

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.

The issuer (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 win or not.
(A) sets content of a ticket to:

$$t := (\sigma, P_w, R, I, T_c, \zeta, c_{Id}, \tau, V)$$

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

Recipient's adress R: is a unique identifier that is derived from the recipient's public key.
Ticket Index I: is set by the ticket issuer and increases with every issued ticket. The recipient verifies that the index increases with every packet and drops packets if 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 unchanged index.
Ticket challenge T_c: is set by the ticket issuer and used to ckeck whether a ticket is redeemable before the packet is been relayed. The packet is dropped if that's not the case.
Acount counter α_c: is used as a mechanism to prevent cheating by turning non-winning tickets into winning ones. This is done by increasing the value of α_c whenever a node resets a commitment which helps keeping track of updates to the on-chain commitments.
Channel iteration ζ: 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.
ChainId c_{Id}**:** The channel identifier which is defined by the ticket issuer in order to determine which channel will be used between issuer and recipient. For example, tickets that are valid on xDAI are not valid on Ethereum.
Tag τ is given as a constant and depends on the utilized blockchain. It is used to distinguish HOPR tickets from others with the same structure that are meant for different payment channels and invalidates their usage in HOPR.
Version V is given as a constant and depends on the utilized blockchain. It is used to invalidate tickets that were issued for previous versions of HOPR from being used in future iterations of the protocol.

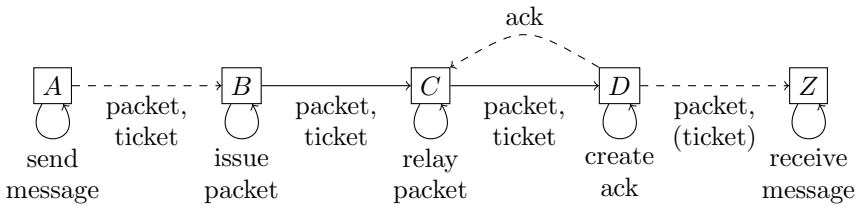

Figure 5: Ticket workflow

5.1.1 Challenge

(A) creates a shared secret s_i with all the relay nodes in the channel (B-C-D-Z) by using an offline version of the Diffie-Hellman key exchange. This shared key is a session key that's generated from the master DH key sphinx key.

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

$$T_{c_i} := (s_i + s'_{i+1}) * G$$

and a hint for B,C,D,Z of how the promised value s'_C, s'_D, s'_Z is going to look like. The value "hint" or H is computed as

$$H_i := s'_{i+1} * G$$

where $*$ is the curve multiplication operation and G is a generator of the curve (the same used in the sphinx section).

5.2 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 (A) receives $s_{i+1}^{(1)}$ from (B) by the 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

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

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

$$H_i = s_{i+1} * G$$

which is stored in the sphinx packet header.
The unacknowledged ticket is stored in the database under the hint to the promised value to make sure that the acknowledgement can be afterwards linked to the unacknowledged ticket.
Together with s_i, the node can verify that

$$T_{c_i} = s_i * G + H_i$$

with

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

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.

5.3 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 $T_{c_i} = ack_i * G$.
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

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

Additionally, the node retrieves the opening value *open* to the current on-chain commitment (used to verify whether a ticket is a win) and checks whether r, *open* leads to a winning ticket. This is the case if

$$h(t_h, r_i, open) < P_w$$

where $t_h = h(t)$ is the ticket hash and h is a hash function. 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.
The node also checks the following:

- Challenge check
$$r * G = T_{c_i}$$
- The ticket has been already redeemed (replay protection):
$$C_I < I$$
where C_I is the channel index and I the ticket index.
- Channel **exists** and is **open**. The check happens locally and not on-chain using the blockchain indexer in order not to reveal any metadata. If the node does not have a record about the channel or considers the channel to be in a state different from **open**, the ticket is dropped and the reception of the accompanying packet is rejected.
- Channel balance towards the ticket recipient is sufficient to cover the costs for the ticket:
$$C_b > \sigma$$
where σ is the ticket amount and C_b is the channel balance.
- Ticket index is strictly greater than the current value in the smart contract (reorder protection). This is valid because redeeming a *ticket* with index $I = n$ requires $I < n + 1$ and sets $I = n + 1$. So the new index becomes greater than n.

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 tickets 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 $h(ticket)$ 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 $ticket_C$.

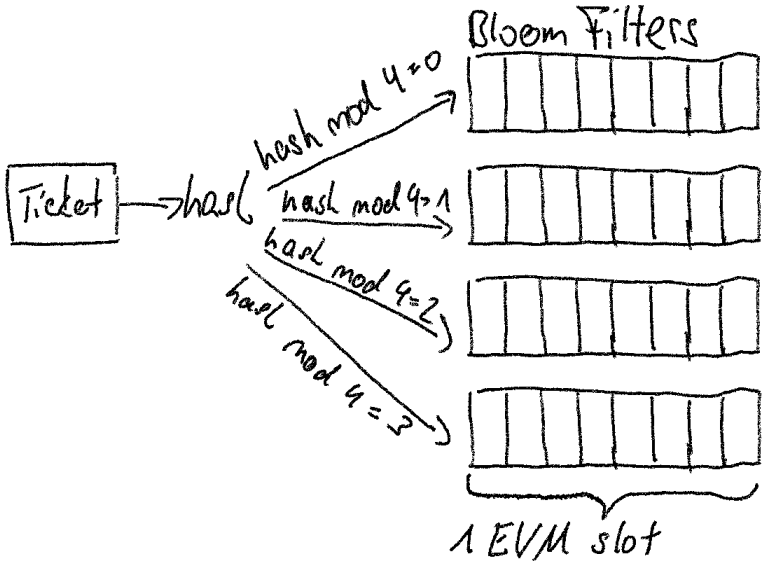

Figure 6: Ticket aggregation

7 Event propagation

HOPR uses a decentralized event propagation method (blockchain indexer) to allow information to be aggregated at many points in the network and shared with other nodes. This method prevents single point of failure vulnerabilities that service providers like infura or Alchemy could create. It also allows HOPR end users to use the HOPR network without needing any additional computational and bandwidth resources.

HOPR decentralized trustless event propagation

HOPR uses Ethereum full nodes to fetch events and forward them to their HOPR instance which improves network latency. HOPR then aggregates the events and store them chronologically:

- $Open_i := (c_{Id}, open, balance, balance_a)$
- $Close_i := (c_{Id}, close)$

The nodes then publish the update independently to the DHT and allow HOPR end users to download it and verify the validity of the data.

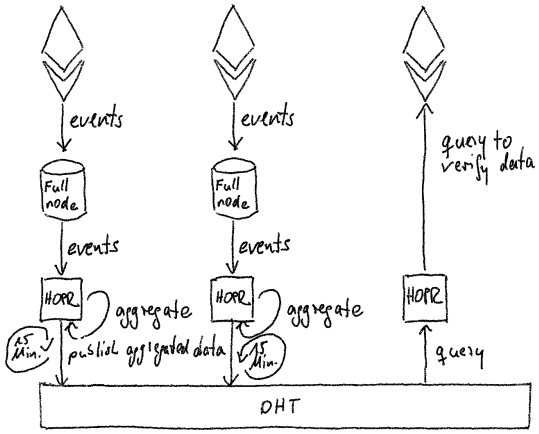

Figure 7: Event propagation

HOPR is a privacy preserving protocol that guarantees sender-receiver unlinkability while still providing high availability.
In order to achieve the aforementioned properties, nodes use DHT to place information in the network and ask selected other nodes to replicate it and update that list every 5 minutes. Since the indexer retrieves on-chain information, there is no leakage about HOPR nodes' knowledge of the network.
Event logs are set to:

$$log_i := h(log_{i-1}||Open) \quad or \quad log_i := h(log_{i-1}||Close)$$

where $log_0 := h(0)$.
When a node receives log_i from another node, the validity can be checked by recreating the hash and comparing the value with the on-chain value.

8 Path Selection

HOPR uses a random selection algorithm to determine the identities of nodes participating in the network relaying service.
For each round, a subset k-of-n nodes will be chosen where n is the total number of nodes in the HOPR network. The selection is divided into two steps:

1. Pre-selection: During this phase, a subset $m << n$ nodes will be selected based on different factors:
    - Availability
    - Payment channel graph

- Stake

Each node gets a score that is proportional to the previously mentioned factors.

2. Random selection: Each edge (from node a to b) within the subset m is assigned a random number r_i. Edges are then sorted by $r_i * score(edge_i)$

Once an edge is selected, it is added to the current path. All paths are then sorted according to their weight and the path with the highest score is expanded next.

Algorithm 1: Path selection in HOPR

$V := Nodes$
$E := \{(x, y) \in V \times V \mid channel(x, y).state = OPEN\}$

$queue \leftarrow new\ PriorityQueue()$
$queue.addAll(\{(x, y) \in E \mid x = self\})$
$deadEnds \leftarrow \emptyset$
$iterations \leftarrow 0$
while $queue.isEmpty() \wedge iterations < maxIterations$ **do**
  $current \leftarrow queue.pop()$
  $currentNode \leftarrow lastNode(current)$
  if $|current| = l$ **then**
    return $current$
  end
  $open \leftarrow \{(x, y) \in E \mid x = currentNode \wedge y \notin deadEnds\ \}$
  $open \leftarrow open.sort(weight)$
  if $open = \emptyset$ **then**
    $deadEnds \leftarrow deadEnds \cup currentNode$
  else
    $toPush \leftarrow \{(current_0, ..., current_{|current|-1}, o) \mid o \in open\})$
    $queue.push(toPush)$
  end
  $iterations \leftarrow iterations + 1$
end
return $\perp$

8.1 Availability

Availability is estimated using the heartbeat protocol. Each node maintains a list of neighbor nodes (logical neighbors in the heartbeat ring) in the network and either ping or passively listen to them in order to determine whether they are online or offline. A node is considered online if the ping response ("PONG") comes back within a certain timeframe. Otherwise, the node is considered offline and its waiting time for the next PING attempt is doubled.

8.2 Payment channel graph

Every node that intends to send messages needs to have a basic understanding about the topology of the network which means whether the channel is open and funded with enough HOPR tokens for the relaying service. In case no existing payment channel is open, the sender creates a new channel and funds it with enough HOPR tokens

8.3 Stake

The HOPR tokens are used to create payment channels with other nodes in the network and thereby staked in the HOPR network. The node will then use the HOPR token to cover transaction costs when interacting with the blockchain. The more stake a node locks the higher probability that it would be chosen as a relayer.

9 Conclusion

9.1 Future work

9.1.1 Path position leak

In HOPR, payments are performed hop-by-hop along a packet's route. The incentives break the unlinkability guarantees inherited from the SPHINX packet format as they reveal the identity of the packet origin who transfers those incentives in the channel using their signature.
To solve this problem, HOPR forward incentives next to the packet.

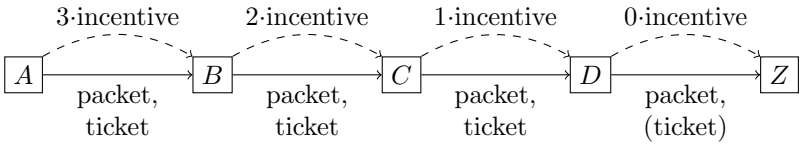

Figure 8: Incentive workflow

This however 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 leakage is considered to have a low severity but further research will be conducted on the subject.

9.1.2 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 attract the network by:

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

Since nodes need to monitor the network to select paths, they need to filter nodes that behave inappropriately. In order to do so, HOPR plans to implement a transitive reputation system which gives a score to each node that acts as a relayer.
The node's reputation either increases or decreases its probability of being chosen depending on its behavior.

Transitive trust evaluation

The reputation can be defined as: "a peer's belief in another peer's capabilities, honesty and reliability based on the other peers recommendations." 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.

References

[1] George Danezis and Ian Goldberg. 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, 2009.

[2] Ross J. Anderson and Eli Biham. Two practical and provably secure block ciphers: BEARS and LION. In Dieter Gollmann, 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, 1996.

10 Appendix

Name	Definition	DataType	Size in bytes	usage
Recipient		address		
Amount		uint256		
TicketIndex		uint256		
Iteration		uint256		
WinProb		uint256		
Epoch		uint256		
Challenge		bytes32		
ChainId		uint8	1	
Version		uint8	1	
Tag		uint8	1	
SignatureX		bytes32	32	
SignatureY		bytes32	32	