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

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

December 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.

1.1 HOPR Design Ethos

HOPR is built with the following principles in mind: that privacy must be an integral part of the design process (privacy by design); that privacy cannot be reliably achieved in centralized systems; and that decentralized systems must

be properly incentivized to provide privacy at a sufficient scale and level of reliability. The following section explains these three principles in more detail, as well as how HOPR meets them.

1.1.1 Privacy by design

Privacy by design is an approach to systems engineering which considers privacy throughout the entire engineering process, not merely as an afterthought. The internet is a public good – a digital commons that should be safe and secure to use for all users. However, it is impossible to provide such privacy using current internet infrastructure, which attaches little or no importance to metadata privacy, and indeed in many cases relies on copious amounts of metadata to be made public in order to function. Therefore, new infrastructure with a focus on privacy must be created on top of the existing internet. HOPR provides an essential part of that infrastructure, and has been built with privacy as its foremost goal.

1.1.2 Decentralization

The internet is not private by design, although it is decentralized by design. In the early days of the internet, this decentralization provided a certain measure of privacy, by placing control in the hands of individual users. However, internet users must increasingly interact with services provided by central authorities. These central authorities control the privacy of individual users, often without their full informed consent.

In particular, interactions with third-party service providers regularly entail a loss of transport-layer privacy. Links can often be drawn between the user and the service provider, either by the service provider themselves, or by a third party who can observe public metadata revealed during the interaction or pressure the service provider to divulge stored data. With a sufficient number of such links, a profile can be constructed of a user's online activity.

This runs contrary to the requirements of privacy as a fundamental human right, since often the only truly private option is to abstain from using such services entirely. To provide privacy as a fundamental feature on top of the internet, a decentralized infrastructure is required.

The HOPR protocol runs on nodes within a decentralized network, therefore ensuring that the network is independent. No single entity can influence its development or manipulate its performance 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.

1.1.3 Incentivization

Although users are incentivized to use privacy technologies by the privacy they provide, most privacy technologies have no baked-in incentive framework for infrastructure providers: rewards either flow to a centralized service provider (and thus the service is not truly private, since the service provider will gather transport-level information on user activity) or the providers in a decentralized network receive no direct rewards. This constrains the growth of the network and limits its scope. In systems which leverage privacy through obscurity, this reduced scale compromises the privacy of the entire network, since fewer users means less data to confound observers. Although blockchains provide a way, for the first time, to incentivize a decentralized network without introducing a centralized entity responsible for payment provision, it is challenging to leverage blockchain technology without compromising either the privacy of network members or the reliability of the network. In brief, if node runners can rely on the anonymity of the technology they are providing, they can potentially use it to claim rewards without properly fulfilling their duties as a node runner. On the other hand, if node runners are required to interact with a blockchain to prove their service provision and claim their reward, the resultant on-chain data is liable to provide a permanent and growing source of metadata which can be used to erode the users' privacy.

HOPR's proof-of-relay mechanism threads this needle, and is the major innovation of the HOPR protocol. Every HOPR node can receive payment for each packet they process and forward, but only once their data payload arrives at the next node. Payments are probabilistic and cheat-proof, which ensures nodes do not (de-)prioritize other nodes and/or packets. To maximize received incentives, node operators must ensure good network connectivity, node availability, computational capacity, and committed funds (to reward other nodes). These combined incentives promote a broad, robust, and reliable network. At the same time, the probabilistic nature of payments obscures links between on-chain data and node runners.

1.2 Security Goals

Generally, the HOPR protocol aims to hide the fact that two parties are communicating with each other, as well as the contents of the communication. This information should be hidden from any party external to the network, all intermediaries involved in transferring the data between the communicating parties, and even the communicating parties themselves, if one or both of them desires it.

Specifically, the HOPR protocol aims to build a network with the following features (definitions based on [Backes et al., 2013], [Piotrowska et al., 2017] and [Danezis and Goldberg, 2009]) for senders, indicated by A_n and recipients,

indicated by Z_n.

1.2.1 Sender anonymity

In a network with sender anonymity, an observer is not able to tell if a particular packet was sent by any adversary-selected honest senders A_1 or A_2 to the honest recipient Z. Formally this is denoted as $\{A_1 \rightarrow Z, A_2 \not\rightarrow\}$ or $\{A_1 \not\rightarrow, A_2 \rightarrow Z\}$.

1.2.2 Recipient anonymity

In analogy to sender anonymity, here an observer is not able to tell if a particular packet was received by any adversary-selected honest recipients Z_1 or Z_2 from the honest sender A. Formally this is denoted as $\{A \rightarrow Z_1, \not\rightarrow Z_2\}$ or $\{\not\rightarrow Z_1, A \rightarrow Z_2\}$.

1.2.3 Sender-recipient unlinkability

For any pair of senders, A_1 and A_2, communicating with any pair of recipients, Z_1 and Z_2, an adversary must be unable to determine whether two packets travelled from $\{A_1 \rightarrow Z_1, A_2 \rightarrow Z_2\}$ or $\{A_1 \rightarrow Z_2, A_2 \rightarrow Z_1\}$.

These properties assume senders and recipients are honest, but they should hold for any honest senders and recipients of the adversary's choice.

To provide these features, the HOPR protocol builds on top of the Sphinx packet format [Danezis and Goldberg, 2009], the specifics of which are explained in Section 4. HOPR also employs cover traffic, explained in Section 7. The additional parameters required for HOPR's incentive scheme are encapsulated in the Sphinx header as described in section 6.2 as additional routing parameters and as such do not change the privacy guarantees of Sphinx.

1.3 Threat Model

Although we assume that nodes in the HOPR network can communicate reliably, the network must still be protected from malicious actors and node failures. We assume a threat model with Byzantine nodes with the ability to either observe all network traffic and launch network attacks or to inject, drop, or delay packets as follows:

1.3.1 Global passive adversaries (GPAs)

A global passive adversary (GPA) is an attacker who can observe the entirety of network traffic passing between users. A GPA is considered passive because their attacks are based on observation alone. A theoretical GPA is arbitrarily powerful, and their full abilities are unlikely to be manifest in any real-world attacker. Nonetheless, building a network which is GPA-resistant introduces extra security through redundancy and future-proofing.

Thanks to the properties of Sphinx (see Section 4) and proof of relay (see Section 6.2), HOPR is able to defend against GPAs. Nonetheless, it is worth mentioning some additional attacks that a decentralized mixnet like HOPR is particularly vulnerable to by virtue of its decentralized and anonymous nature.

1.3.2 Sybil attacks

In a Sybil attack, an attacker forges multiple identities in the network, thereby introducing network redundancy and reducing system security. The attacker can potentially de-anonymize packet traffic and thus link the sender and recipient's identities. HOPR mitigates Sybil attacks via the trust assumption of the Sphinx packet format: only a single honest relayer is needed to ensure integrity of the entire transmission chain, and since the traffic is source routed, users can choose routes themselves to ensure this minimal requirement of one honest relayer is met. Since path selection is dependant on a financial stake, launching a Sybil attack would be prohibitively expensive.

1.3.3 Eclipse attacks

Many decentralized networks suffer from a general unavailability of a general understanding of their topology, hence nodes cannot determine whether their respective local views are complete or accurate. As an attacker, it might be attractive to flood a victim with inaccurate information about collaborating nodes while withholding information about honest nodes.

HOPR mitigates this issue by using a different medium to announce entry nodes to the network. This is done using a smart contract on a blockchain and known within HOPR as DEADR (Decentralized Entry Advertisement and Distributed Relaying). DEADR nodes serve two functions: (1) providing access to the distributed hash table (DHT) that is used within the libp2p environment that HOPR leverages and (2) facilitating network address translation (NAT) or relaying traffic that cannot be sent directly between adjacent nodes who reside in separate internet sub-networks (e.g., computers without a public internet-facing internet protocol (IP) address behind typical home routers). A HOPR node only requires access to one honest DEADR node and it is significantly cheaper

in computational terms to check whether a DEADR node is honest than to conduct an eclipse attack, which requires on-chain transactions. A successful eclipse attack therefore requires controlling all announced DEADR nodes or forging a public blockchain, both of which we assume are impossible and beyond the scope of this work.

Thus we are satisfied that HOPR more than adequately protects users against attack. However, security cannot come at the expense of usability. To be appealing to users and node runners alike, it is important for HOPR to find a satisfactory compromise to what is known as the "anonymity trilemma" [Das et al., 2018], which states that a privacy network can provide at most two of the following properties: low latency communication, low bandwidth overhead, and strong anonymity.

The rest of this paper will outline the current state of the art among anonymous communication protocols and layer-2 scalability protocols, describe how the HOPR network is built, and explain how HOPR provides a satisfactory resolution to the anonymity trilemma.

1.4 State of the Art

HOPR has been built on top of, or as a reaction to, various technologies and research projects which try to meet some or all of the design ethos and security goals outlined above. These technologies and research projects can be broadly divided into two groups: (1) anonymous communication protocols which attempt to hide user information; (2) layer-2 scalability protocols which allow systems to perform micro-payments via the Ethereum blockchain. The following sections present the state of the art for each of these groups.

1.4.1 Anonymous Communication Protocols

Virtual Private Networks (*VPNs*) are point-to-point overlay networks used to create secure connections over an otherwise untrusted network, such as the internet [Venkateswaran, 2001]. Clients receive a public IP address from the VPN endpoint, which is then used for all outgoing communication. Since this public IP address can be almost anywhere in the world, VPNs are often used to bypass censorship [Hobbs and Roberts, 2018] and geolocalization blocks. However, VPNs provide dubious privacy, since clients must trust the VPN endpoint. The endpoint provider has full access to users' connection and communication metadata, since they control all communication going through it. In addition, VPN endpoints are often a single point of failure, since they are provided as centralized services. Both factors are major weaknesses when considering VPNs for privacy-preserving communication.

These privacy concerns have motivated the creation of a new model for VPNs, **Decentralized Virtual Private Networks** (*dVPN*s). dVPNs leverage blockchain technology to remove reliance on a central authority. Users act as bandwidth providers, allowing other users to access internet services via their node, in exchange for rewards.

Examples of dVPN projects include Orchid [Cannell et al., 2019], Sentinel [Sentinel, 2021], and Mysterium [Mysterium, 2017]. However, these projects still lack privacy, performance, and reliability guarantees, since bandwidth providers can potentially inspect, log, and share any of their traffic. This harms honest bandwidth providers as well as users: since bandwidth providers can theoretically monitor traffic, nodes whose bandwidth are used to enable illicit activity can potentially be held liable for those activities by government authorities. Indeed, there have been incidents where unaware dVPN users have been (ab)used as exit nodes through which DDoS attacks were performed.

A research project called VPN^0 [Varvello et al., 2019] aims to provide better privacy guarantees. VPN^0 leverages 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.

Onion Routing is another approach to network privacy, implemented by projects such as Tor [Dingledine et al., 2004]. Tor encapsulates messages in layers of encryption and transmits them through a series of network nodes called *onion routers*. However, Tor is susceptible to end-to-end correlation attacks from adversaries who can eavesdrop on the communication channels. These attacks reveal a wide range of information, including the identity of the communicating peers.

The Invisible Internet Project (*I2P*) is an implementation of onion routing with notably different characteristics than Tor [Astolfi et al., 2015]:

- **Packet switched instead of circuit switched** Tor allocates connection to long-lived circuits; this allocation does not change until either the connection or the circuit closes. In contrast, routers in I2P maintain multiple tunnels per destination. This significantly increases scalability and failure resistance since packets are used in parallel.
- **Unidirectional tunnels** Employing unidirectional rather than bidirectional tunnels makes deanonymization harder, because tunnel participants see half as much data 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.

Despite these improvements, I2P is vulnerable to eclipse attacks since no I2P router has a full view of the global network (similar to other peer-to-peer networks). Like Tor, I2P only provides protection against local adversaries, making it vulnerable to timing, intersection, and traffic analysis attacks. I2P has also been shown to be vulnerable to Sybil and predecessor attacks, in spite of various countermeasures implemented to thwart these.

Mixnets are overlay networks of so-called *mix nodes* which route packets anonymously, similarly to Tor [Chaum, 1981]. Mixnets originally used a cascade topology where each node would receive a batch of encrypted messages, decrypt them, randomly permute packets, and transfer 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 susceptible to traffic 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 packets among random noise. An external adversary able to observe the packet flow should not be able to distinguish communication packets from these cover traffic packets, increasing privacy.

The application of cover traffic provides metadata protection from global network adversaries, a considerable improvement on projects such as Tor and I2P. However, 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. For applications such as real-time video streaming, Tor may be more suitable, provided the user feels the latency improvements outweigh the increased privacy risks.

Loopix [Piotrowska et al., 2017] is another research project which uses cover traffic to resist traffic analysis while still achieving low latency. To achieve this, Loopix employs a mixing strategy called *Poisson mix*. Poisson mix nodes independently delay packets, making packets unlinkable via timing analysis.

1.4.2 Layer-2 Scalability Protocols

Blockchain technology (mostly public blockchains like Bitcoin and Ethereum) suffers from a major scalability issue: every node in the network needs to process every transaction, validate them, and store a copy of the entire chain state. Thus the number of transactions cannot exceed that of a single node. For Ethereum, this is currently around 30 transactions per second.

Multiple solutions have been proposed to resolve the scalability issue, including sharding and off-chain computation. Both approaches intend to create a second layer of computation to reduce the load on the blockchain mainnet.

Off-chain solutions such as Plasma [Poon and Buterin, 2017], Truebit [Teutsch and Reitwießner, 2019], 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 [Poon and Dryja, 2016], a payment channel is opened between two parties by committing a funding transaction. Those parties may then make any number of signed transactions that update the channel's funds without broadcasting those to the blockchain. The channel is eventually closed by broadcasting the final version of the settlement transaction. The channel balance is updated by creating a new set of commitment transactions, followed by trade revocation keys which render the previous set of commitment transactions unusable. Both parties always have the option to "cash out" by submitting the latest commitment transaction to the blockchain. If one party tries to cheat by submitting an outdated commitment transaction, the other party can use the corresponding revocation key to take all the funds in the channel.

As soon as closure is initiated, the channel can no longer be used to route payments. There are different channel closure transactions depending on whether both parties agree on closing the channel. If both agree, they provide a digital signature that authorizes this cooperative settlement transaction. In the case where they disagree or only one party is online, a unilateral closure is initiated without the cooperation of the other party. This is done by broadcasting a "commitment transaction". Both parties will receive their portion of the money in the channel, but the party that initiates the closure must wait for a certain delay to receive their money. This delay time is negotiated by the nodes before the channel is opened, for the protection of both parties.

The Raiden network [Raiden Team, 2016] is a layer-2 payment solution for the Ethereum blockchain. The project employs the same technological innovations pioneered by the Bitcoin Lightning Network by facilitating transactions off-chain, but provides support for all ERC-20 compliant tokens. Raiden differs in operation from its main chain because it does not require global consensus. However, to preserve the integrity of transactions, Raiden powers token transfers using digital signatures and hash-locks. Known as **balance proofs**, this type of token exchange uses payment channels. Raiden also introduces "mediated transfers", which allow nodes to send payments to another node without opening a direct channel to it. These payment channels are updated with absolute amounts, whereas HOPR employs relative amounts (more details in Section 5).

1.5 Assessment

Although there are numerous projects attempting to solve the problem of online privacy, most fail to satisfy the design principles and security goals outlined in Section 1.2.

Privacy The metadata privacy properties that a mixnet like HOPR provides are significantly beyond what VPNs, dVPNs, or even Tor can deliver. VPNs and dVPNs have single points of trust and failure. Tor and I2P have been shown to be subject to a large variety of de-anonymizing attacks, which a mixnet like HOPR is resilient against. The HOPR mixnet unlinks sender and recipient even when faced by powerful global passive adversaries (GPAs) which can monitor every packet in the network. This is achieved by packet transformation, mixing, and introducing delays before forwarding a packet, as well as bandwidth overhead in form of cover traffic and packet padding.

Decentralization Most projects presented above have decentralization as a core goal (with the exception of VPNs, where the service is provided by a centralized entity, at the expense of user privacy). However, many projects lack the impetus to scale, which prevents them from leveraging many of the advantages which come with a decentralized network, such as privacy through obscurity, while retaining all of the challenges, such as coordination and consensus issues. It is our opinion that the only way to solve the scalability problem is with the introduction of a robust incentivization scheme for node runners.

Incentivization Many projects referenced in this section lack incentivization of any kind. VPNs incentivize the service provider, but since this is a centralized entity this fails the privacy condition.

Bandwidth providers in dVPNs share their resources and are granted tokens accordingly as payment for their services. For example, Mysterium [Mysterium, 2017], an open-source dVPN built on top of a P2P architecture, uses a smart contract on top of Ethereum to ensure that the VPN service is adequately funded. However, the resultant liability risk for dVPNs means these incentives are likely to be insufficient.

Tor and I2P rely on donations and government funding, which only covers the cost of running a node, not any additional reward. This has discouraged volunteers from joining these networks and the number of relayers in both networks has stayed mostly static in recent years, even as usage and the demand for privacy has risen sharply.

Mixnet designs are generally unconcerned with the problem of incentivization, and most existing implementations rely on a group of volunteer agents who lack incentives to participate. However, it is possible to add incentivization on top of a mixnet design, which is what HOPR does. Some authors have proposed adding digital coins to packet delivery in mixnets [Atkinson and Silaghi, 2007]. However, here the anonymity provided by the mixnet acts as a double-edged sword: since there is no verification for whether a packet arrives at its final destination, and node identities are kept secret by the mixnet, selfish nodes can extract payment without performing their relaying task or without passing on acknoledgements to

the next node. Thus a naive implementation of this approach does not actually provide an incentive at all.

It is in this last area that HOPR provides its main innovation: HOPR's proof-of-relay mechanism ensures that node runners are only paid if they complete their relaying duties. The challenge is to enforce this without publicizing data which would allow an adversary to break the anonymity of the network, and to provide both incentivization and anonymity without introducing unacceptable latency and bandwidth costs. The remainder of this paper will explain how this is achieved.

2 Peer-to-peer Mechanisms

HOPR is designed as a decentralized network, hence there is no central coordinator and nodes need to interact with each other directly to organize the operation of the network. This causes various challenges: starting with the absense of a complete overview of the network, such that nodes need to operate with incomplete information about the nodes. In addition, it is expected that some nodes reside on publicly exposed hosts whilst others are hidden behind one or more network address translation mechanisms (NATs), hence it is necessary to traverse NATs in order to establish direct connections. Last but not least, nodes who are run by end users and which are therefore mostly used to send and receive messages, join and leave the network as they like to, hence the network need to deal with a significant amount of churn and nodes need to maintain an overview about other nodes' availability.

2.1 Addressing

Each node possesses an ECDSA public key which is used to derive unique identifier in the network, called *HOPR address*. The HOPR address uses the self-describing *multiformats* standard which creates a prefix how the address is supposed to be interpreted. For ECDSA public keys on the secp256k1 curve, this yields the binary multiformats prefix $0x002508021221$[1]. The string representation is the Base58-encoding of the bitstring.

$$id = base58.encode(0x002508021221 \;||\; pubKey)$$

Identifiers distinguish nodes from each other, whereas addresses allow nodes to establish a connection to each other. HOPR addresses follow the self-describing

[1] $0x002508021221_{16} = [0_{10}, 37_{10}, 8_{10}, 2_{10}, 18_{10}, 33_{10}]$, meaning 37 bytes, containing a compressed ECDSA public key on the secp256k1 curve and the key itself consists of 33 bytes

multiaddr standard. There are two types of addresses: *direct addresses* and *relay addresses*.

Direct Addresses Nodes that are able to gain control over a TCP socket can have one or more direct address which are determined by their network adapter. HOPR supports running multiple nodes on the same machine, multiple nodes in the same local network and nodes running behind public IP addresses. Following the *multiaddr* standard, an address includes the IP address family, e.g. IPv4, the actual host address, the transport protocol and the utilized port, followed by the HOPR address. The string representation is given by

$$addr = /\mathsf{ip4}/1.2.3.4/\mathsf{tcp}/9091/<id>$$

Relay Addresses HOPR nodes can also reach each other indirectly by asking other nodes in the network to forward their traffic to the desired destination. This is especially necessary if nodes are operating behind a NAT or a restrictive firewall. Using a relay address means first establishing a connection to the relay and asking the relay node to extend the connection to the final destination. If the relay node manages to reach the final destination, it opens a general-purpose connection that allows node to exchange data, such as mixnet packets or status messages. Following the *multiaddr* standard, the relay address starts with the HOPR address of the relay node, that is reached using a p2p connection, which attempts to establish a circuit by using a p2p connection to the destination.

$$/\mathsf{p2p}/<relayId>/\mathsf{p2p\text{-}circuit}/\mathsf{p2p}/<id>$$

To prevent congestion, relay nodes come with a limited number of relay slots and nodes need to register with the relay nodes to use them as relays.

At startup, each node fetches all nodes that have announced themselves on-chain with a routable address. From this list, each node selects those five relay nodes with the lowest latency and also have a free relay slot. Afterwards, the node creates for each relay node a relay address and publishes them within the DHT and keeps the connection to the relay node open which preserves the address mapping of the NAT.

2.2 Decentralized NAT Traversal

Due to the scarcity of IPv4 addresses in general as well as security concerns, it is expected that many, if not most, of the end users in the HOPR network operate behind one or more network address translation mechanisms (NATs).

As a result, nodes cannot rely on internal port to which the node is listening matching the external port that is perceived by other nodes. To establish a connection regardless, nodes need to find out to which port the destination got mapped and use that port instead of the canonical one.

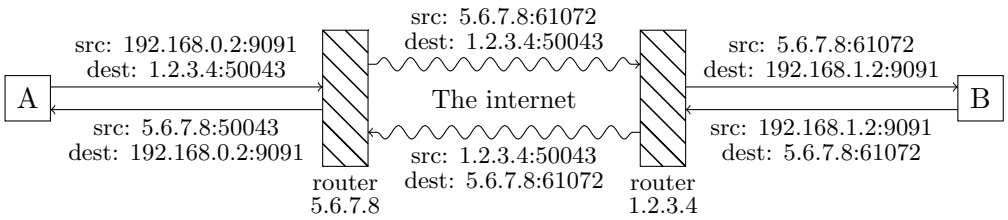


Figure 1: Node A operates in the local network 192.168.0.1/24 and listens on the canonical port 9091. A is connected to a router that applies a NAT and maps A's port to the dynamic port 50043. The packet travels through the internet and arrives at the router that forwards it to B. B is running behind a NAT as well, hence when sending a packet, B's source port is mapped by the router to the dynamic port 61072.

Hence, the crucial part is to provide the infrastructure to allow nodes to determine their public address and port, as well as a means of communication to interactively find out how to connect directly. Within the HOPR network, this is done in a decentralized way. First of all, all nodes answer STUN requests, hence nodes can find out their public address and port without relying on external resources such as public STUN servers. Secondly, nodes offer each other relay services such that connected nodes can use them as a rendezvous point to exchange connection information.

At startup, each node attempts to reserve a relay slot at several of the known relay nodes. Due to the dynamic port mapping, these nodes might be the only ones which are able to reach the node. It is therefore crucial, to preserve these connections.

Once the connections to the relay nodes have been established, the node publishes relay addresses to the DHT containing the relays to which it maintains a connection.

When connecting to a new node, the initiator of the connection first fetches the direct addresses of the node and attempts to reach the destination. In case that does not work, the initiator queries the DHT to fetch relay addresses of the destination and recursively establishes a connection. Once the connection to the relay is open, the relay attempts to reach the destination. If that works, initiator and destination have a channel to exchange information. This channel is then used to transfer payload such as mixnet packets but also signalling information that might give parties the opportunity to upgrade to a direct connection using WebRTC. Once both parties found out how to connect directly, they route the

traffic through the WebRTC connection. In case that is not possible, e.g. because both nodes are running behind bidirectional NATs, the relayed connection serves as fallback.

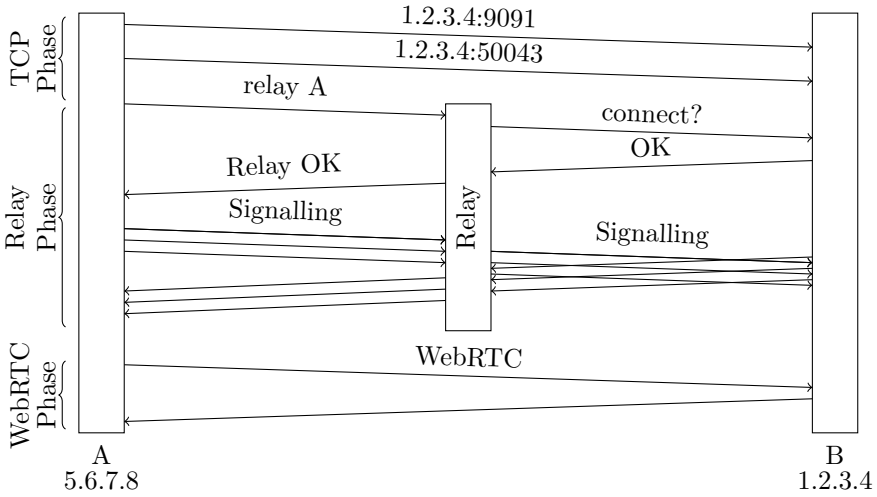


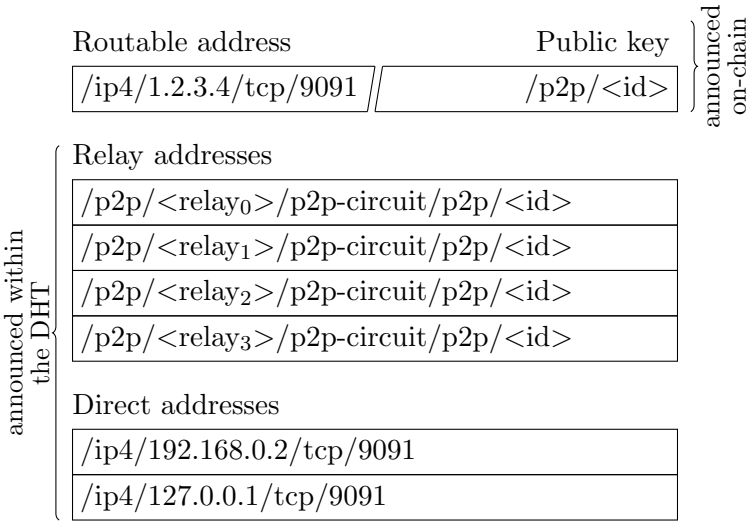
Figure 2: A intends to establish a connection to B. To do this, A runs through three phases: *TCP phase*, *Relay phase* and *WebRTC phase*. At first the node tries all available direct addresses. If any of them lead to a direct connection, the node is finished. Otherwise, A fetches relay addresses of B and connects to a relay. The relay tries to reach B. Once that is done, both nodes, gather information about their NAT situation and exchange with the other party. If that leads to a direct WebRTC connection, the nodes use this connection to proceed.

2.3 Peer Discovery

Newly connected nodes are lacking information about existing nodes in the network and thus need the help of others to introduce them to the rest of the network. This is done using a smart contract that creates a secondary channel and acts as a Schelling point holding a list of entry nodes to which new nodes can connect to and learn about other nodes in the network. At startup, nodes need either run their own Ethereum node a or connect to a Web3 provider to fetch a list of nodes that have announced themselves as entry nodes.

In addition, each node announces itself within the smart contract such that other nodes can about the existence the newly connected node. This step is necessary for all nodes because all nodes nodes need to maintain a mapping between Ethereum addresses and public key in order to construct mixnet packets, see section 4, and to handle incentives for relaying packets, see 6. HOPR thereby destinguishes two types of announcements: routable addresses of nodes who act as entry nodes and non-routable addresses that solely contain the nodes' public

key.



Apart from the altruistic benefit to the network if announcing as an entry node, it is expected that the announcement does not immediately lead to payout. But since path selection is based on availability, it becomes clear that nodes announce themselves on-chain and also have a good availability are chosen more likely as a relayer since they are well-known which thus increases their rewards.

Announcing on a public blockchain comes with the property that a potential adversary who attempts to run an eclipse attack against a newly connected node also needs to provide a sound and fake blockchain which is assumed to be hard. Hence, the attacker is unable to withhold entry nodes but nevertheless, but it is able to spam the victim with connection information about collaborating nodes.

Once nodes have established a connection to at least one of the entry nodes, they grow an understanding about those nodes who have not published a routable on-chain by using a DHT. Thereby, they also learn about relay addresses and use them to establish direct connections.

2.4 Availability Measurement

When learning about a new node, it remains unclear to the initiator of the connection whether this node is available or not. Since nodes might temporariliy or permanently leave the network, this also applies to already known nodes. Sending a mixnet packet requires a randomly sampled path through the network, it becomes necessary to estimate a nodes' availability and to provide a list of nodes the are expected to be online. Note that nodes cannot ping nodes just before they use them as a relay node because this would reveal the chosen path, hence nodes need to continuously and independently grow their view of availability of

other nodes.

Within the HOPR network, this is achieved by using a heartbeat mechanism that continuously sends ping requests to other nodes. A node is considered online if it replied to a *PING* attempt with a correct *PONG* response or if the node has recently received a *PING* request by that node.

Each node counts successful and unsuccessful *PING* attempts and uses this information to create a health score for other nodes. To prevent from continuously probing offline nodes, the heartbeat mechanism increases the probing interval in case the node was not reachable.

2.4.1 Ping mechanism

Ping messages are small messages that are sent in order to see if the destination is available. Each request should be unique in order to be able to clearly distinguish it from potential retransmissions, hence each ping request contains a random 16-byte string. To make sure that the destination has indeed processed the ping request, ping responses, "*PONG*" messages, are considered valid if they include the SHA256 hash of r.

2.4.2 Health Score

While running, each node maintains a list of "interesting nodes" from which it keeps measuring the availability. At startup, this list is initiated with all nodes to which the node has funded outgoing payment channel, see section describing the incentivization mechanism for further details. Nodes on the list start with a initial health score of 0.2. Whenever the node successfully pings a node on the list or the node passively gets pinged by a node, their health score increases by 0.1 up to a maximum of 1.0. Analogously, each unsuccessful attempt decreases the health score by 0.1 down to 0 which means that the node is considered offline.

Nodes with a health score greater than 0.5 are expected to have a sufficiently high availability and can be used as a relay when sampling a mixnet path.

2.4.3 Exponential Backoff

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 $n_{fail} = 5$.

3 Path Selection

HOPR 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.

3.1 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 security goals, 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.

3.2 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.

3.3 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 4) 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 $queue.peek()$

4 Sphinx Packet Format

HOPR uses the Sphinx packet format [Danezis and Goldberg, 2009] to encapsulate and route data packets its final recipient while achieving sender and recipient unlinkability. The Sphinx packet format determines how mixnet packets are created and transformed. This happens in a way that does not leak path information to relayers or other parties eavesdropping the communication. A Sphinx packet consists of two parts, a header and an onion-encrypted payload:

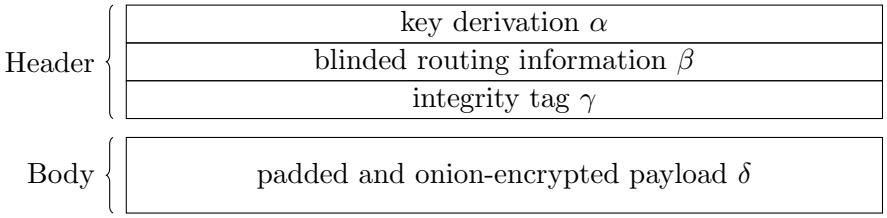


Figure 3: Schematic overview of a SPHINX packet

4.1 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 3), 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.

4.1.1 Key Extraction

Path selection, see section 3, 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 $\mathsf{KDF}(salt, secret)$ 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 it has 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.

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

where y_i for $i \in \{B, C, D, Z\}$ refers to the nodes' private keys and s_i denotes the derived shared secret group elements.

Packets consist of (α, β, γ) where β is described in section 4.1.3 and γ in section 4.1.5, leaving both underspecified for the moment. Let further be $packet_i = (\alpha_i, \beta_i, \gamma_i)$, leading to:

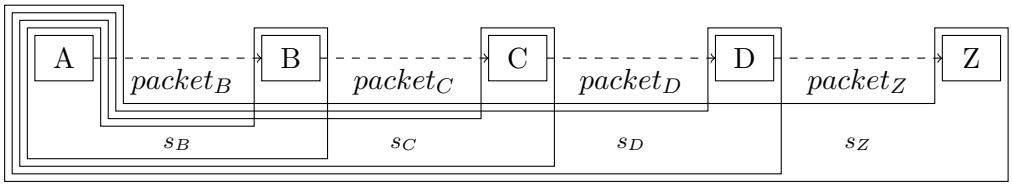


Figure 4: Node A derives shared keys s_B, s_C, s_D, s_Z with node B, C, D, Z using their public keys Y_B, Y_C, Y_D, Y_Z.

4.1.2 Replay protection

The creator of a packet picks a path that the packet is supposed to take. As seen in section 4.1.1, 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 4.1.1 using the key derivation described in section A.

4.1.3 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 being 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 4.1.5 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 4.1.1, 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.

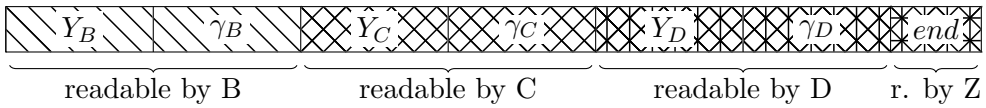


Figure 5: Blinded routing information sent to first relayer B.

4.1.4 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 $|\beta| + |y| + |\gamma|$.

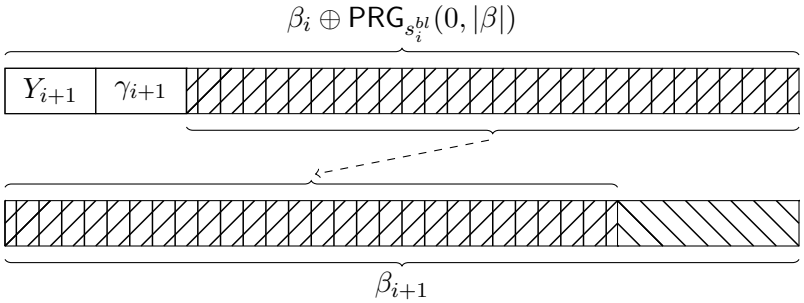


Figure 6: Shifting in the header

4.1.5 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 4.1.1, 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 4.1.3 and 4.1.4, 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.

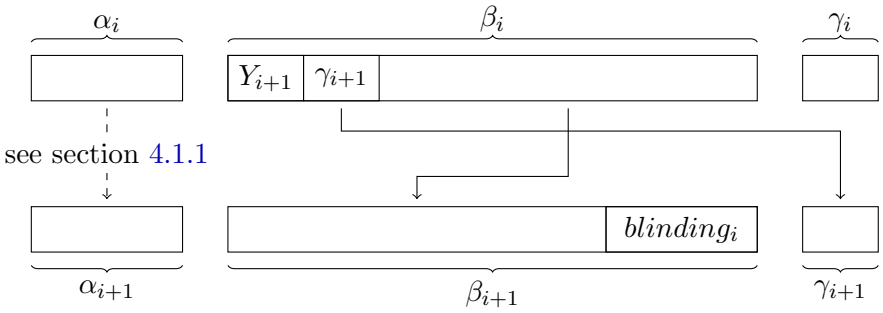


Figure 7: Extraction of γ_{i+1} and creation of the next packet

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.

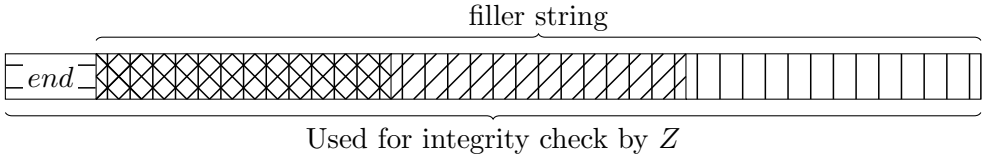


Figure 8: Node A is sending a packet through mix nodes B, C, D to final destination Z such that the readable part for Z includes the END tag and the blindings $blinding_B, blinding_C, blinding_D$ that were appended by the predecessors.

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 4.1.3 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

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

4.1.6 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 scheme, 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 4.1.1, 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.

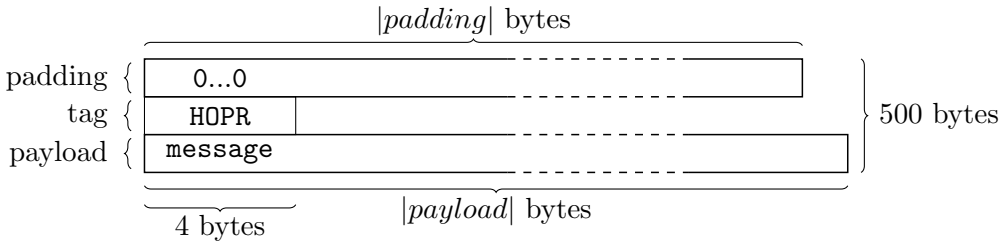


Figure 9: Padded message consisting of 0-padding, protocol tag τ (`0x484f5052`, UTF-8-encoded "HOPR"), and payload m.

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.

4.1.7 Shorter Paths

SPHINX packets define a maximum packet length r[2], 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.,

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

2019] with *random* data. This is necessary because the very last node, i.e. the one for which the message is destined, is otherwise able to determine the utilized path length.

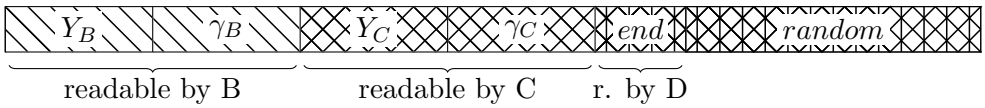


Figure 10: Node A has created routing information for B, C and the final recipient D, filling the empty part of β with random data.

By applying the transformations seen in section 4.1.4 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.

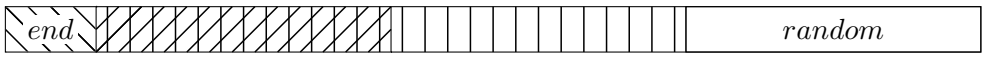


Figure 11: Node D receives a packet which was relayed by two intermediate nodes.

5 Tickets

The HOPR protocol makes use of a custom micropayment scheme to process its incentives. This section focuses on the utilized micro-payment scheme and serves as a building block for section 6.

Incentives are handled by a structure called *tickets* that is inspired by payment channels as well as probabilistic payments and allows nodes to issue asset transfers without requiring each time an on-chain interaction. Tickets are sent locked and get unlocked afterwards, i. e. after proving that a packet has been relayed. Nodes who receive a locked ticket are able to validate its validity. Once a node has received the required cryptographic material to unlock the ticket, it is able to claim the incentive by submitting the ticket to the smart contract.

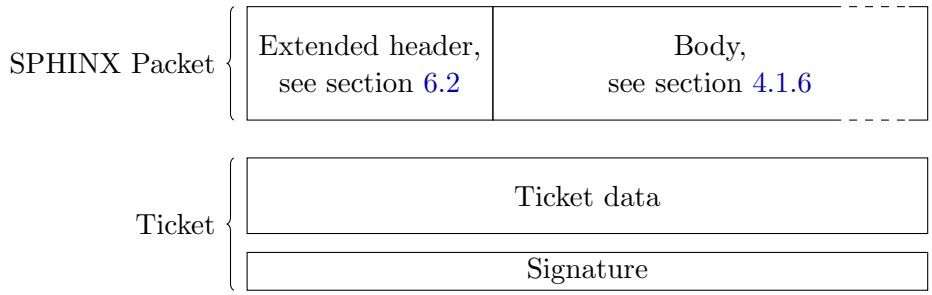


Figure 12: Schematic overview of a mixnet packet that is sent together with a ticket.

5.1 Ticket Issuance

Before a node is able to issue tickets for another node, it needs to lock funds on-chain to cover for the current as well as future tickets. Locking funds is considered equal to staking tokens in the HOPR network as it allows the node to create mixnet packets and act as relayer. By locking tokens in the smart contract, the node creates a unidirectional payment channel towards the recipient and is thus able to convince the recipient that it is eligible to issue tickets.

As ticket issuance happens without any interaction to the blockchain, it is the duty of the node who receives the ticket to check whether there were any tokens locked on-chain and to keep track about previously issued tickets. If there is no record on-chain about any locked funds or if the sum of the received tickets exceed the amount of tokens that were locked on-chain, the recipient should refuse the ticket.

Tickets are sent together with a mixnet packet and include the incentive for processing and forwarding the packet to the next downstream node. To meet the security goals, neither issuance nor redemption must be linkable to the creation or processing of mixnet packets. Therefore, each ticket is given a winning probability, which means that not every issued ticket leads to a claimable incentive and, moreover, those tickets who turn out to be a winner cover the missed incentives of losing tickets. It turned out that is not only beneficial in terms of privacy but also helps to keep transaction costs resulting from on-chain interactions minimal.

When issuing a ticket, the issuer picks a winning probability $winProb > 0$ and starts creating the data structure $ticketData$ by setting the intended value $value$ through

$$ticketData.value := \frac{value(ticket)}{winProb}$$

Note that the ticket issuer should not choose *winProb* too low as the recipient might refuse the ticket due to inappropriate winning probability.

	Value	Ethereum datatype	size (in bytes)
TicketData	Recipient	address	20 bytes
	Challenge	bytes32	32 bytes
	Ticket epoch	uint256	32 bytes
	Ticket value	uint256	32 bytes
	Winning probablity	uint256	32 bytes
	Ticket index	uint256	32 bytes
	Channel epoch	uint256	32 bytes
Sig	Signature r	bytes32	32 bytes
	Signature s	bytes32	32 bytes
	Recovery value v	uint8	1 byte

Figure 13: Structure of a ticket.

Recipient The Ethereum address of the recipient, derived from the recipient's public key. This nails down the ticket to one specific payment channel, the one from ticket issuer to ticket recipient. Note that Ethereum addresses are computed as

$$ethAddr : pubKey \in \{0,1\}^{64} \mapsto \mathsf{keccak256}(pubKey).\mathsf{slice}(12, 32)$$

(the last 20 bytes of the keccak256 hash of the uncompressed ECDSA public key).

Challenge Tickets are issued locked, hence the embedded incentive is not yet claimable by the ticket recipient but their validity can be verified. Locked means that the ticket states a challenge which needs to be solved before being able to claim the embedded incentive. This mechanism servers as a building block for proof of relay.

Ticket epoch Ticket redemptions relies on providing the value *opening* to a series of commitments that have previously been stored on-chain by the ticket recipient. To make sure that the party who wants to redeem tickets, is always able to compute the opening to a commitment, there is the opportunity to renew the on-chain commitment. As this allows the ticket recipient to change the entropy that is used to determine whether a ticket is a winner, the smart contract stores a counter that increases on every renewal and the ticket issuer signs the current value. This makes sure, that each commitment renewal invalidates all previously issued but not yet redeemed tickets.

Ticket value The ticket value is given by the intended *value* divided by the winning probability *winProb* in the base unit of the token, which is 10^{-8}. Hence, sending 1 HOPR with a winning probability of 1 leads to $ticket.value = 10^8$.

Winning probablity The proportion of tickets which lead to an actual payout is determined by their winning probability. To prevent from issues resulting from roundings, *ticketData* includes the inverse winning probabilty that is normalized with the common base of Ethereum, which is $2^{256} - 1$. Hence,

$$ticketData.invWinProb := winProb * (2^{256} - 1)$$

Ticket index Each ticket is labeled by an ongoing serial number named ticket index i and its current value is stored in the smart contract. Whenever a ticket gets redeemed, the stored value is updated to the value given by the redeemed ticket and thus invalidates all tickets with index $i' \leq i$. This is necessary to have each ticket being valid exactly once. Since ticket issuance does not change the value stored in the smart contract and tickets and tickets with unchanged ticket index are worthless, it is the duty of the ticket recipient to make sure that ticket index keeps increasing.

Channel epoch Payment channels might run throug multiple *open* and *close* sequences, see section 6.5 for more information. To make sure that tickets from previous channel incarnations lose their value once the channel is reopened, tickets include the current channel epoch counter and the smart contract considers the ticket invalid if the signed channel epoch does not match the stored channel epoch.

Signature As a last step, the issuer creates a signature over the hash of the ticket with

$$\begin{aligned} ticketHash = keccak256(&recipient \;||\; ethAddr(challenge) \;||\; ticketEpoch \;||\\ &amount \;||\; invWinProb \;||\; index \;||\; channelEpoch) \end{aligned}$$

yielding the ticket $t = (ticketData, Sig_{Issuer}(ticketHash))$.

5.2 Ticket Validation

Tickets are used to convince its recipient that it will receive the promised incentive, once the challenge is solved. As ticket issuance happens without any

on-chain interaction, it is the duty of the recipient to decide whether it accepts the ticket or resuses it.

Ticket validation runs through two states: receiving the ticket without knowing the response to the given challenge stated as *ticket.challenge*, validation of locked tickets, and once the response is known, validation of unlocked tickets.

Validation of Locked Tickets Due to the lack of a response to the stated challenge, the node is neither able to decide whether the ticket is going to be a win nor claim it on-chain to receive the incentives. Nevertheless, the node can use the embedded information to validate the ticket economically. Therefore, the node at first extracts the winning probability as

$$ticket.winProb = \frac{ticket.invWinProb}{2^{256} - 1}$$

which leads $value(ticket) = ticket.value \cdot ticket.invWinProb$. If the node considers $value(ticket)$ inappropriate, i.e. because it does match the expected amount, or if winning probability is set too high or too low, it should refuse the ticket.

As ticket issuance happens without any on-chain interaction and thus, there is no guarantee that there is any payment channel at all and that this payment channel has enough tokens Locked. Therefore, the recipient needs to check that before considering a ticket valid. In addition, there might be previous tickets, denoted as *stored*, that are not yet redeemed. Hence, the recipient needs to check that

$$channel.amount \leq value(ticket) + \sum_{t \in stored} value(t)$$

In addtion, as tickets are issued using an ongoing serial number, the recipient must check that $ticket_i.index > \max(ticket_{i-1}.index, 0)$ and refuse the ticket otherwise.

It remains to show that the ticket issuer indeed knows any *response* that solves *ticket.challenge*. This is especially relevant if the ticket issuer was given the challenge by a third party, i.e. the creator of a mixnet packet. For this section, this specific topic is out scope and covered in section on proof of relay.

Validation of Unlocked Tickets Once the *response* to *ticket.challenge* is known, i.e. after receiving a packet acknowledgement, the node is able determine whether the ticket is going to be a winner. To check this, the node first computes the next *opening* to the current value *commitment* stored in the smart contract and checks whether

$$\mathsf{keccak256}(\ \mathsf{keccak256}(ticketData)\ ||\ solution\ ||\ opening\ ) < ticket.winProb$$

If true, the node can consider the ticket to be a winner and store it for later use. In case the ticket turned out to be a loss, there is no added value to it and the node can safely drop it. Note that losing tickets are an integral part of the mechanism and do not reduce the average payout to the ticket recipient. This is the case because $value(ticket)$ is given by the expected value and hence the asymptotic payout does not change.

5.3 Ticket Redemption

After running through the validations of the previous section, the node n ends up with a set of stored tickets which it considers to be a win, hence

$$stored := \{t \in Tickets \mid isWinner(t) \wedge t.recipient = n\}$$

Each ticket t is given a ticket index, which means tickets need to be redeemed in order which is why the node first creates an ordered set *ordered* out of the set *tickets* and proceeds with the first ticket.

Redemption means that the node now proves for each $t \in stored$ one-by-one to the smart contract that t is indeed a win. If successful, the smart contract transfers the stated incentives to the account of the node, see paragraph asset transfer.

In contrast to ticket recipients, the smart contract considers a ticket only valid if the redeemer is able to provide a *response* that solves *ticket.challenge* and a value *opening* that opens the most recent *commitment* stored on-chain. Note that the smart contract thereby acts as a trusted third party that forces the node reveal additional cryptographic material despite the signature of the ticket is valid. This is possible because the blockchain consensus makes it infeasible to add state changes which have not been the result of a method execution in the smart contract.

Challenge Solving a challenge C means finding a value $r \in \mathbb{F}$ such that $r \cdot G = C$. Hence, in order to check this equation, the smart contract needs to compute a scalar multiplication of an elliptic curve point, which is as of writing of the paper not directly available within Ethereum.

Instead, Ethereun allows to efficiently implement a function mul':

$$mul' : x \in \mathbb{F} \mapsto ethAddr(x \cdot G)$$

where $ethAddr : \{0,1\}^{64} \mapsto \{0,1\}^{20}$ maps uncompressed elliptic curve points to Ethereum addresses. Hence, the smart contract compares the computed Ethereum address against the challenge stated in the ticket.

Issuer signature By computing $C' = mul'(response)$ the smart contract is able to recompute the hash of the ticket as

$$\begin{aligned} ticketHash = keccak256(&recipient \mid\mid C' \mid\mid ticketEpoch \mid\mid amount \mid\mid \\ &invWinProb \mid\mid index \mid\mid channelEpoch) \end{aligned}$$

and is thus able by using the provided signature to recover the public key of the ticket issuer. By now having both Ethereum addresses, the one of the issuer and the one of the recipient, the smart contract is able to compute the identifier $channelId$ of the utilized payment channel.

Payment channel validation As the previous steps were computed without any feedback to the computed values, the computed $channelId$ will either lead to a non-existing entry in case there is no such channel known to the blockchain, or to a record of a payment channel. Due to the usage of a collision-resistant hash function, it is assumed to be infeasible for an attacker to find a second pre-image that maps the ticket hash to a specific $channelId$. Hence, either the challenge is correct and there is a payment channel or any of the previous conditions is not met.

If $channelId$ leads to a payment channel record, the smart contract checks that $channel.state = OPEN$ and $channel.amount \leq ticket.amount$ and rejects the ticket otherwise.

Replay protections A ticket can be valid due to valid signature and correct *response*, but as it controls an asset transfer and thereby initiates on-chain state changes, it must be valid exactly *once*.

Each ticket is given an ongoing serial number i and each ticket redemption sets the on-chain value $channel.index$ to $ticket.index$ if $ticket.index > channel.index$. Otherwise, the ticket is rejected.

Analogously, each reincarnation of the payment channel, i.e. a sequence of *open* and *close*, increases the channel epoch counter. To turn tickets issued for previ-

ous incarnations of the payment channel invalid, the smart contract rejects all tickets with $ticket.channelEpoch \neq channel.epoch$

Last but not least, each renewal of the on-chain commitment increases the ticket epoch counter. To prevent from the ticket issuers from ticket recipients resetting the ticket epoch counter to values that they find more beneficial, e.g. in order to tweak the ticket's winning probability and turn previously losing tickets into winning ones, the smart contract rejects all tickets for which $ticket.ticketEpoch \neq channel.ticketEpoch$.

Commitment By opening a payment channel from ticket issuer to ticket recipient, the ticket recipient needs to store a series of commitments in the smart contract. Let *commitment* be the most recent one. By submitting a ticket, the ticket recipient recipient peals off one of the previously deposited commitments, hence need to check that *open* is a valid opening for *commitment*. This done by checking if

$$channel.commitment = keccak256(opening)$$

If false, the smart contract rejects the ticket. See section 6.4 for further details on the commitment scheme.

Ticket luck Redeeming a ticket includes two sources of entropy of entropy: *response* to the stated *challenge* that is known by the ticket issuer, and *open* that opens the most recent *commitment* and is solely known to the ticket recipient. By submitting both values to the smart contract, the smart contract is able to determine whether ticket is a winner. It checks

$$\mathsf{keccak256}(\ \mathsf{keccak256}(ticketData) \ ||\ solution \ ||\ opening\ ) < ticket.winProb$$

and causes a revert otherwise.

Asset transfer If none of the previous checks have failed, the smart contract transfers the included tokens in the ticket to the recipient. This can happen in two ways: either there is an open payment channel in the other direction, namely from ticket recipient to ticket issuer and *ticket.amount* is credited to that payment channel or the tokens are directly transferred to the recipient's account.

6 Incentivization Mechanism

Nodes in the HOPR network receive incentives for providing services that allow others to achieve privacy. The HOPR protocol incentivizes nodes for transforming and delivering mixnet packets. To turn this into a trustless solution, HOPR uses a custom Proof of Relay scheme which makes the nodes' work verifiable. Incentives are handled by a micropayment scheme based on probabilistic payment channels. This requires the issuer of incentives to not know whether a specific incentive lead to a payout or not. The receiver of an incentive therefore deposits in advance an iterated on-chain commitment to the smart contract and reveals its openings whenever claiming an incentive.

6.1 Incentivized Behavior

Relaying a mixnet packet requires computation to transform it such that it can either get sent to the next downstream node n_{i+1} or forwarded to the user. In case the packet is relayed, the relayer needs to establish a connection to node n_{i+1} and cache the transformed mixnet packet for a short period of time until it is able to deliver it - or drop the transformed packet because node n_{i+1} appeared to be unreachable.

Transforming and delivering packets are those mechanisms that are most crucial to keep the HOPR network running, therefore they are incentivized by the protocol. On the other hand, receiving a packet is not incentivized.

6.2 Proof of Relay

The Sphinx Packet Format guarantees that packets can only get processed in the order that has been chosen by the creator. As a result, after applying the required transformations, the next downstream node can either decode the packet or ends up with some random bitstring. Hence it is the next downstream node who is able to verify the correctness of the previously applied tranformations which it confirms by sending an acknowledgement.

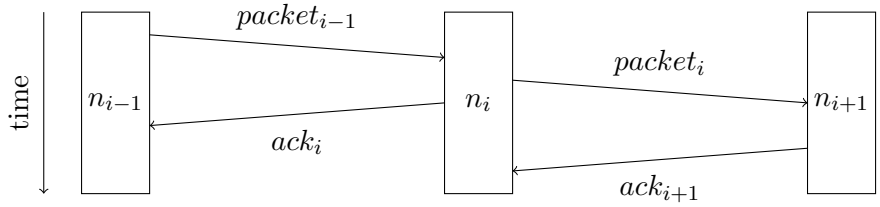


Figure 14: Node n_{i-1} sends $packet_{i-1}$ to node n_i. Node n_i transforms the incoming packet, resulting in $packet_i$ and sends it to node n_{i+1}. Afterwards it acknowledges the processing of $packet_{i-1}$ to node n_{i-1}.

Note that acknowledgements are sent immediately once the processing of the incoming packet has been completed. This makes sure that sending acknowledgements does not create any observable pattern in reverse to the chosen path. It is rather that each acknowledgement depends on one incoming packet but it does not depend on the chosen path.

Construction Each ticket includes a challenge C_i which requires a response $response_i$ to be claimable on-chain. The creator of the packet samples all of them and uses the first one for the ticket sent to the first relayer. All subsequent ones are put into the section of the routing information, see section 4.1.3, that is visible by the corresponding node. Each relayer n_i and their corresponding next downstream node n_{i+1} engage in a 2-out-of-2 secret sharing of $response_i$ picked by the creator of the packet. To convince each relayer that the creator of the packet knows any value $\widetilde{response}_i$ that solves C_i, each relayer n_i receives a value $hint_i$ that serves as an argument of knowledge.

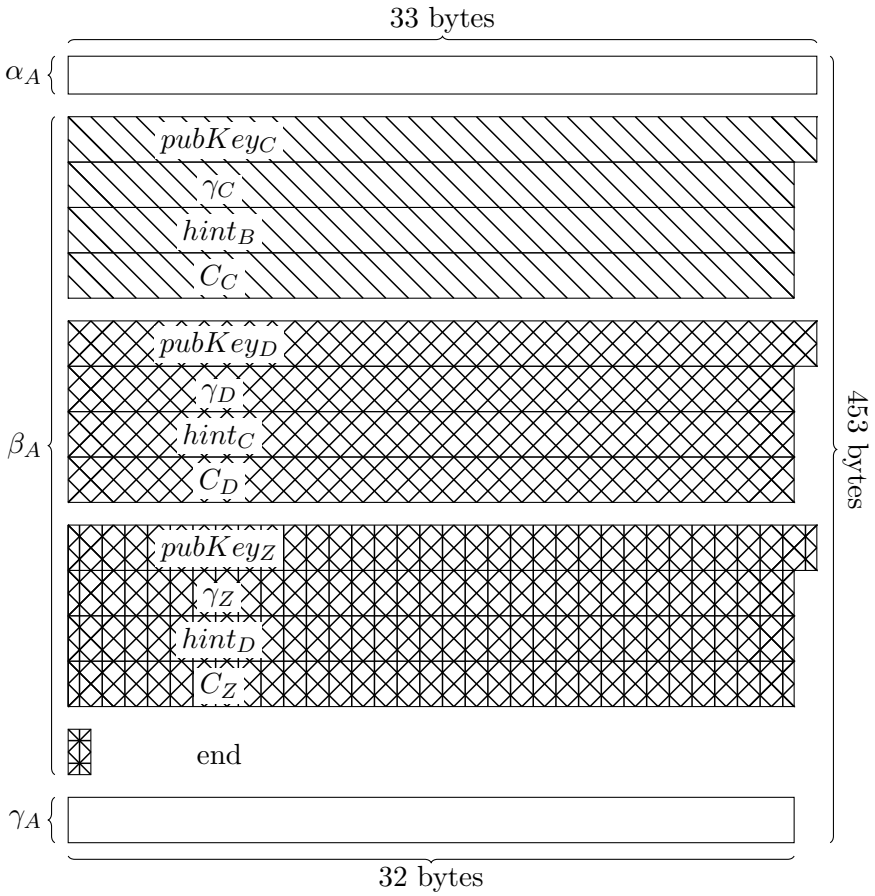


Figure 15: Mixnet packet header with PoR fields that is sent from the sender A to B and supposed to be forwarded through nodes C, D to Z.

Secret sharing Each node derives two keys, s_i^{own} and s_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.

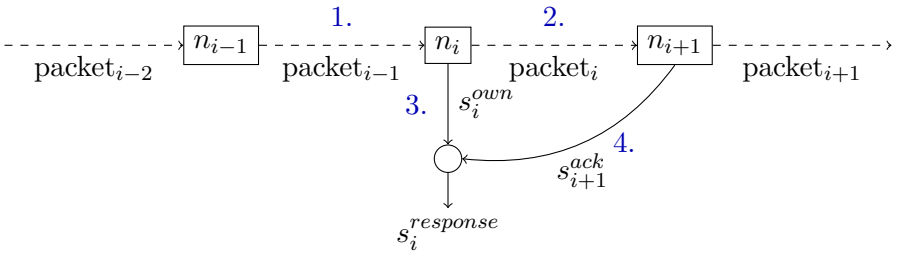


Figure 16: 1. Node n_i receives $packet_{i-1}$, validates it, transforms it and 2. sends it to node n_{i+1}. 3. While processing $packet_i$, node n_i derives s_i^{own} and 1. once node n_{i+1} considers $packet_i$ valid, it *acknowledges* the receipt of $packet_i$ and thereby reveals s_{i+1}^{ack} to 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[3], the node n_i not only needs to transform the packet but must also issue a ticket to 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}.

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

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.

6.3 Probabilistic Payment Channels

Payment channels give two nodes, A and B, the opportunity to lock funds on-chain and later on handle who of them possesses which fraction. At each moment, it holds that

$$balance(A) + balance(B) = const.$$

The state becomes final once one of the nodes submits the most recent state to the smart contract. Therefore it requires a proof by the counterparty that the proclaimed state is in their interest, which is given by a digital signature and named *update transaction*. Hence an update transaction that transfers $x > 0$ digital assets from A to B is approved by,

$$update_i = Sig_A(balance(A) - x \mid\mid balance(B) + x)$$

By doing so, payments can only in one direction, turning the payment channel into a unidirectional payment channel. This is possible because digital assets will only be transferred from A to B and thus B will always pick in its own the interest the most recent update transaction since $balance(B)$ is strictly increasing.

If the channel allowed asset transfers in the opposite direction, namely from B to A the update transactions need to include a versioning element that prevents any of the parties from rollback to the most beneficial state since the smart contract is unable to determine the most recent transaction otherwise. Hence,

$$update_i = Sig_A(i \mid\mid balance(A) - x \mid\mid balance(B) + x)$$

In addition, this requires certain timeframes in which the counterparty is given the opportunity to present their most recent update transaction. This is necessary because none of the nodes can prove that there does *not* exist a more

recent update transaction. Once the timeout ends, and none of the parties have submitted any more recent update transaction, the last submitted state becomes final.

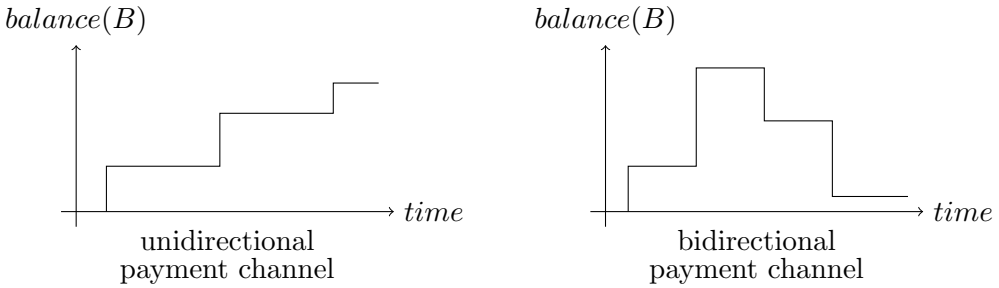


Figure 17: Node A and B have a payment channel. Whilst in case of unidirectional payment channels, $balance(B)$ is strictly increasing, the derivative of $balance(B)$ *can* change sign if the channel is bidirectional.

Within the HOPR protocol, payment channels are implemented as unidirectional channels, hence there *can* be one from $A \to B$ and another one from $B \to A$. Whenever, an update transaction is submitted to the blockchain, the assets get transferred to the channel in the opposite direction, or directly to the counterparty if there is no such channel. This obviously controverses the original purpose of payment channels, which is *aggregation* of asset transfers.

When talking about unidirectional payment channels, aggregation means that $value(update_i) = value(update_{i-1}) + \Delta x$, hence $update_i$ makes $update_{i-1}$ obsolete as it already includes the additional assets Δx. So, if a node manages to receive $update_i$ before $update_{i-1}$, there is no need to provide the services the which were supposed to be compensated by $update_{i-1}$.

This is problematic for HOPR because it uses Proof of Relay to unlock payment made from one node to the other and the state of a payment channel between nodes n_{i-1} and n_i therefore relies on third-party actions, namely by n_{i+1} and n'_{i+1}. Both of them acknowledge the reception of the packet and the correct transformation done by node n_i. Using update transactions arises the question which update transaction node n_{i+1} and node n'_{i+1} should sign because they cannot know which acknowledgement reaches n_i first. Hence, the incentives for packet need to be done independently and thus should not rely on update transactions.

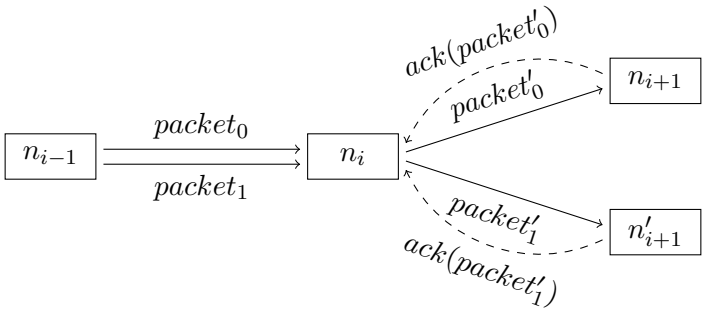


Figure 18: Node n_{i-1} send packet $packet_0$, $packet_1$ to node n_i which forwards them to node n_{i+1} and node n'_{i+1}. Both nodes acknowledge the validity of $packet'_0$ and $packet'_1$ to node n_i.

Probabilistic payments Incentives relate to packets which are assumed to be sent very often in the HOPR network. Nodes can therefore expect that they will shortly receive another micropayment from the same node. It is thus not necessary to claim the incentive for each packet individually. This is why nodes issue each other probabilistic tickets that have the same asymptotic payout but do neither require in-order processing nor require an on-chain operation for every incentive. Hence when picking the same relay fee, it holds that

$$\forall i \; \forall j \;\; : \;\; value(ticket_i) = value(ticket_j)$$

Moreover, the nodes estimate the usage of the link to the nodes with whom they have engaged in a payment channel and pick a ticket winning probability that leads to a continous payout. Connections with higher througput can use lower probabilities whilst those with fewer traffic are supposed to use winning probabilities closer to 1.

Probabilistic payments rely on the necessity for the ticket issuer to not know whether a ticket that is issued will turn into an asset transfer or not. Hence, the ticket redemption must rely on some entropy provided by the ticket redeemer and need to be kept secret until the ticket is redeemed. This is necessary, because by knowing which ticket is going to become a winner, the ticket issuer would solely issue losing tickets and thereby withhold the incentives for the next downstream node.

Analogously, the ticket receiver should not know whether a ticket will be a win before being able to reconstruct the response to the challenge stated in the ticket. If not, the ticket receiver would just process those packet that come with a winning ticket and drop all other packets.

A ticket is a winner if

$$keccak256(ticketHash \;||\; porSecret \;||\; opening) < ticket.invWinProb$$

where $ticketHash$ is the hash of the received ticket, see section 5.3, and thus known by both ticket issuer and recipient whereas $porSecret$ is known by the ticket issuer and can be reconstructed as part of the $Proof of Relay$ scheme by the ticket recipient once it receives the acknowledgement for the forwarded ticket. On the other hand, *opening* refers to the value that opens the most recent commitment that is stored on-chain, see section 6.4. The on-chain commitment is chosen by the ticket recipient and due to the hiding property of the utilized commitment scheme, the opening values are solely known by the ticket recipient.

6.4 On-chain Commitment

As seen in previous section 6.3, ticket luck relies on two sources of entropy, one provided by the ticket issuer and one provided by the recipient. This section focuses on the entropy provided by the ticket recipient.

The entropy need be kept hidden until the ticket gets redeemed. In addition, the ticket recipient must not be able to change the entropy just before redeeming a ticket turn previously losing tickets into winners. Furthermore, as redeeming the ticket reveals the entropy and thus loses its value, it need to be reset with every ticket redemption. Last but not least, the mechanism must be feasible to implement it within Ethereum.

To match the aforementioned properties, HOPR uses an iterated on-chain commitment scheme that is initialized once the channel is opened. The scheme is iterative, so by revealing a commitment, the ticket recipient sets the upcoming commitment and the smart contract keeps this value for the next ticket redemption.

Commitment scheme The commitment scheme in HOPR is a tuple of three algorithms, $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open})$. KeyGen takes a security parameter and returns a seed x. Commit takes x and produces the commitment as $cm = Commit(x)$. Open takes a commitment cm and r and fails if x does not fit to cm, otherwise it return 1.

The commitment scheme is called binding if it is infeasible for an adversary to find a value $x' \neq x$ such that $\mathsf{Open}(cm, x') = 1$. It is called hiding if is infeasible for an adversary to derive the commited value x from cm.

The iterated commitment scheme is computed as

$$comm_i = keccak256^i(x) = keccak256^{i-1}(keccak256(x)) \quad \text{for } i > 0$$

where $comm_0 = x$ and thus servers as a master secret. The value x is chosen randomly by the ticket recipient, hence due to the pseudorandomness of the utilized cryptographic hash function, for every $i > 0$, the resulting value $comm_i = keccak256(comm_{i-1})$ is pseudorandom as well and therefore infeasible to predict by the ticket issuer.

The algorithm Open is therefore quite simple as it solely checks that

$$comm_i = keccak256(\widetilde{comm_{i-1}})$$

Commitment phase Once a node is the destination of a payment channel, it samples a master key $comm_0 \in \{0,1\}^{32}$ and computes $comm_n = keccak256^n(comm_0)$ and submits a transaction that stores $comm_n$ on-chain within the payment channel. The value $n > 0$ is chosen by the ticket recipient and should reflect the amount of tickets that are expected to be sent using this channel.

Obviously, the ticket recipient can run out of openings after redeeming a huge amount of tickets - or losing the master secret x. Therefore, the ticket recipient has the opportunity to renew the stored on-chain commitment. As this would allow the recipient to alter the opening to a more pleasant value just before redeeming a ticket, i.e. to turn a previously losing ticket into a winning one. Therefore, each payment channel includes a counter that gets increased on every renewal.

$$channel.ticketEpoch = channel.ticketEpoch + 1$$

Increasing that counter invalidates all previously issued tickets, hence there is no incentive for the ticket recipient to renew it more than necessary.

Opening phase In order to redeem a ticket, the ticket recipient must reveal the opening $comm_{i-1}$ to the current commitment $comm_i$ stored within the smart contract.

The smart contract therefore checks that

$$channel.commitment = keccak256(\widetilde{comm_{i-1}})$$

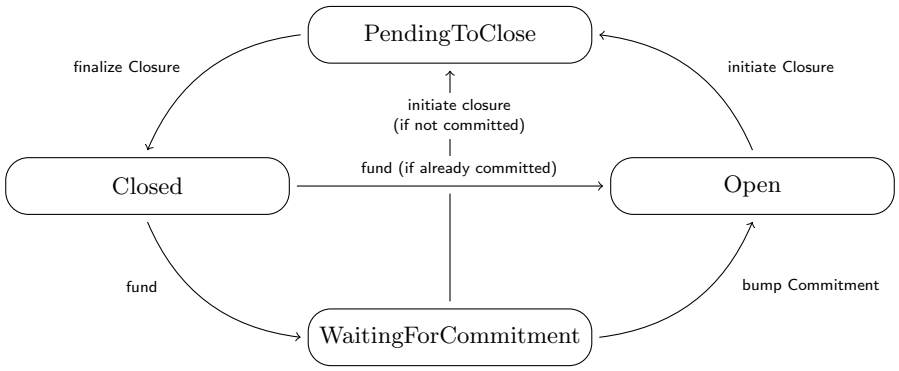
If the values match, the smart contract sets $channel.commitment = comm_{i-1}$, hence the ticket recipient needs to reveal $comm_{i-2}$ to redeem the next ticket.

6.5 Payment Channel Management

Sending a packet requires the node n_{open} to issue a ticket to the next downstream node n_{dest} which cannot be done without an open payment channel towards n_{dest}. So, in order to send a packet, the n_{open} first needs open a payment channel and thereby deposit assets to cover for upcoming tickets that will be issued towards n_{dest}.

A payment channel runs throug multiple states during its lifecycle: Initially, each payment channel is *Closed*. Node n_{open} can create one by locking some assets in the smart contract, which leads to *WaitinForCommitment*. This step is necessary because the payment channel becomes usable once n_{dest} has set a commitment. Once that is done, a payment channel is considered *Open*. If the counterparty has already set commitment, then funding immediately changes the state to *Open*. As long as a payment channel is open, the counterparty can submit ticket to redeem them.

Closing a payment channel means accessing the previously locked funds and transfering them to the nodes' accounts. As the smart contract cannot know whether the most recently issued ticket is the last one that got issued and turned out to be a win, payment channels go into *PendingToClose* once n_{open} intends to close it and need to await a timeout once n_{open} is able to retrieve the assets. Before the end of the timeout, the node n_{dest} has the opportunity redeem its stored winning tickets since they lose its validity once the payment channel gets closed. Once the timeout is done, any of the nodes finalize the closure and turn the payment channel state into *Closed*.



TODO: add redeemTicket arrows

Figure 19: Payment channel states and possible state transitions.

In rare cases it can happen that a node n_{open} intends to open a payment channel, but n_{dest} does not submit an on-chain commitment. Under this circumstance, n_{open} has the opportunity to immediately turn the channel into *PendingToClose*.

State Each channel is assigned a unique identifier, computed from the addresses of n_{open} and n_{dest},

$$channelId = keccak256(\ ethAddr(n_{open})\ ||\ ethAddr(n_{dest})\ )$$

where *ethAddr* maps public keys to Ethereum addresses. Note that using this scheme leads to different *channelIds* for the channel from $n_{open} \rightarrow n_{dest}$ and the one from $n_{dest} \rightarrow n_{open}$.

Once n_{open} locks funds, the smart contract instantiates the data structure *ChannelData* and stores it under the previously computed *channelId* within its storage. Instantiation of *ChannelData* can happen it two ways: either there has been a previous instance of the channel of not. If there has been one, the smart contract keeps the values *commitment* and *epoch* from the previous instance. For *epoch* this is necessary because tickets from previous instances must lose their value once the channel gets closed. This is achieved by incrementing *epoch* on every incarnation of the channel. On the other hand, commitments are kept to simplify channel reopenings.

	Value	**Ethereum datatype**	**size (in bytes)**
ChannelData	Balance	uint256	32 bytes
	Commitment*	bytes32	32 bytes
	Ticket Epoch	uint256	32 bytes
	Ticket Index	uint256	32 bytes
	Status	enum	32 bytes
	Epoch*	uint256	1 bytes
	Closure Time	uint32	4 bytes

Figure 20: The structure of a stored channel. Values marked with * survive reincarnations of the channel.

Closing a channel sets *closureTime* to current UNIX timestamp plus the timeframe[4] in miliseconds in which node n_{dest} is allowed to submit previously unredeemed tickets.

7 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

[4] As of now, the timeframe is set for debugging purposes to five minutes.

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.

7.1 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.

7.2 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.

7.3 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 3 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 5). 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., Sybils or nodes with insignificant stake who would thus not be able to relay a lot of traffic.

7.4 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*.

8 Conclusion

8.1 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.

8.2 Future work

8.2.1 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.

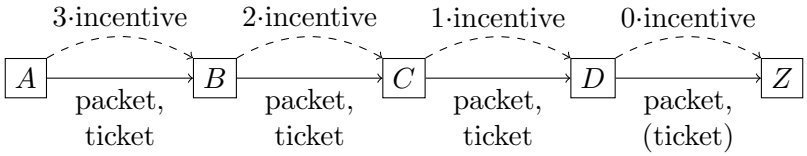


Figure 21: Incentive workflow

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.

8.2.2 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.

[Astolfi et al., 2015] Astolfi, F., Kroese, J., and Van Oorschot, J. (2015). I2p - the invisible internet project. Web technology report, Media Technology, Leiden University. https://staas.home.xs4all.nl/t/swtr/documents/wt2015_i2p.pdf.

[Atkinson and Silaghi, 2007] Atkinson, T. and Silaghi, M. (2007). Guarantees for customers of incentive anonymizing networks. Cryptology ePrint Archive, Report 2007/460. https://ia.cr/2007/460.

[Backes et al., 2013] Backes, M., A., K., Manoharan, P., Meiser, S., and Mohammadi, E. (2013). Anoa: A framework for analyzing anonymous communication protocols. *Computer Security Foundations Symposium*, pages 163–178. https://eprint.iacr.org/2014/087.pdf.

[Cannell et al., 2019] Cannell, J. S., Sheek, J., Freeman, J., Hazel, G., Rodriguez-Mueller, J., Hou, E., Fox, B. J., and Waterhouse, S. (2019). Orchid: A decentralized network routing market. https://www.orchid.com/assets/whitepaper/whitepaper.pdf.

[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., 2004] Dingledine, R., Mathewson, N., and Syverson, P. F. (2004). Tor: The second-generation onion router. In Blaze, M., editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX. http://www.usenix.org/publications/library/proceedings/sec04/tech/dingledine.html.

[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.

[Hobbs and Roberts, 2018] Hobbs, W. R. and Roberts, M. E. (2018). How sudden censorship can increase access to information. *American Political Science Review*, 112(3):621–636. https://www.cambridge.org/core/journals/american-political-science-review/article/how-sudden-censorship-can-increase-access-to-information/A913C96E2058A602F611DFEAC43506DB.

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

[Mysterium, 2017] Mysterium (2017). Mysterium network project. https://mysterium.network/whitepaper.pdf.

[Piotrowska et al., 2017] Piotrowska, A. M., Hayes, J., Elahi, T., Meiser, S., and Danezis, G. (2017). The loopix anonymity system. *CoRR*, abs/1703.00536. http://arxiv.org/abs/1703.00536.

[Poon and Buterin, 2017] Poon, J. and Buterin, V. (2017). Plasma: Scalable autonomous smart contracts. https://plasma.io/plasma.pdf.

[Poon and Dryja, 2016] Poon, J. and Dryja, T. (2016). The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf.

[Raiden Team, 2016] Raiden Team (2016). Raiden network - high-speed asset transfers for ethereum. https://raiden.network.

[Sentinel, 2021] Sentinel (2021). Sentinel: A blockchain framework for building decentralized vpn applications. https://sentinel.co/whitepaper.

[Teutsch and Reitwießner, 2019] Teutsch, J. and Reitwießner, C. (2019). A scalable verification solution for blockchains. *CoRR*, abs/1908.04756. http://arxiv.org/abs/1908.04756.

[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.

[Varvello et al., 2019] Varvello, M., Querejeta-Azurmendi, I., Nappa, A., Papadopoulos, P., Pestana, G., and Livshits, B. (2019). VPN0: A privacy-preserving decentralized virtual private network. *CoRR*, abs/1910.00159. http://arxiv.org/abs/1910.00159.

[Venkateswaran, 2001] Venkateswaran, R. (2001). Virtual private networks. *IEEE Potentials*, 20(1):11–15. https://ieeexplore.ieee.org/document/913204.

[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 compressed 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	Name	Length	Hash Key (UTF-8)
SPHINX packet	PRG	s^{prg}	32	`HASH_KEY_PRG`
	PRP	s^{prp}	128 + 64	`HASH_KEY_PRP`
	Packet tag	s^{tag}	32	`HASH_KEY_PACKET_TAG`
	Blinding	s^{bl}	32 + 12	`HASH_KEY_BLINDING`
PoR	acknowledgement	s^{ack}	32*	`HASH_KEY_ACK_KEY`
	ownKey	s^{own}	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.

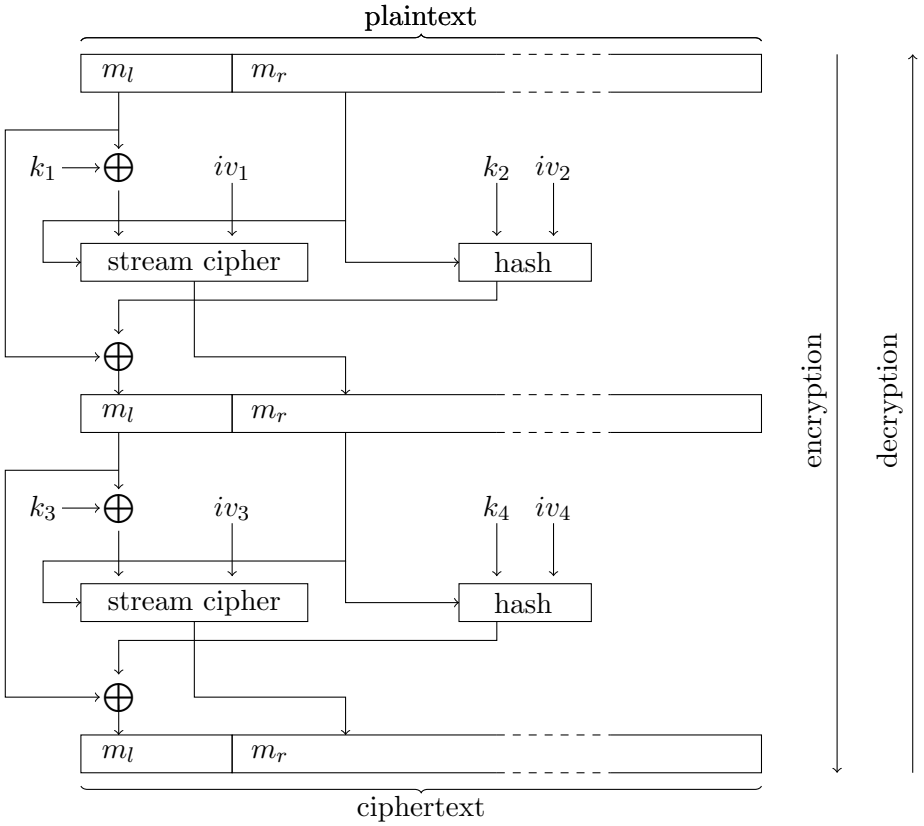


Figure 22: Schematic overview of encryption and decryption of the LIONESS [Anderson and Biham, 1996] scheme

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 Pseudorandomness generator - PRG

A pseudorandomness generator (PRG) consumes a key k and creates an output of an arbitrary length that is indistinguishable from random numbers of the same length. Within HOPR, it is used to create the blindings necessary to hide the routing information of the SPHINX packet which is covered in section 4.1.3.

$$PRG : k \in \{0,1\}^{32} \times start \in \mathbb{N} \times end \in \mathbb{N} \mapsto \{0,1\}^{end-start}$$

PRG takes a key $k \in \{0,1\}^{32}$, a natural number $start$ and a natural number end with $start \leq end$ and creates an output of size $end - start$. For $start$ values strictly greater than 0, it returns the PRG output beginning with $start$.

The PRG output is generated using AES with 256-bit key length in counter mode (AES-CTR-256), encrypting zeros and using k as encryption key and $iv_{pre} \in \{0,1\}^{12}$ as an initiailization vector where $iv := iv_{pre} \mid\mid blockNumber$ and $blockNumber \in \{0, \ldots, 2^{32} - 1\}$, yielding

$$block_i = \mathsf{AES\_CTR\_256}_{k,\ iv_{pre} \mid\mid i}(0)$$

The final output is generated as

$$(block_{\lfloor \frac{start}{bLength} \rfloor} \mid\mid \ \ldots \ \mid\mid block_{\lceil \frac{end}{bLength} \rceil}).slice(bStart,\ bEnd)$$

where $bStart = start \ \mathbf{mod}\ bLength$ and $bEnd = start - end + bStart$.

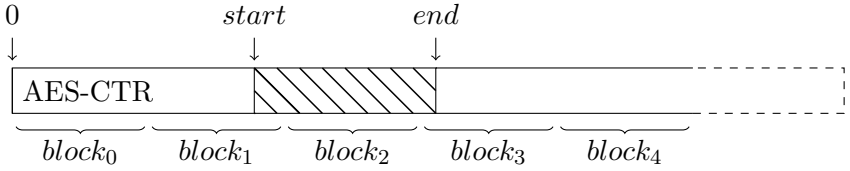


Figure 23: Pseudorandomness generator used within the HOPR protocol

D Hopr architecture

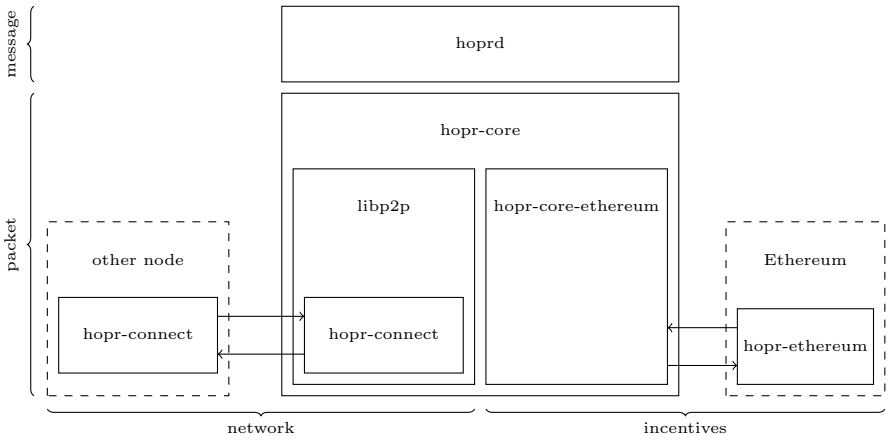


Figure 24: Architecture of a HOPR client