PolySphinx: Extending the Sphinx Mix Format With Better Multicast Support

Daniel Schadt[†] Karlsruhe Institute of Technology
Christoph Coijanovic[†] Karlsruhe Institute of Technology
Christiane Weis[‡] NEC Laboratories
Thorsten Strufe[†] Karlsruhe Institute of Technology

***Abstract*—Mix networks are a well-known technique to hide communication metadata, but incur a high overhead especially in group communication settings. This hinders their adoption in real-world usage, as group communication makes up a big part of modern communication patterns. In this paper, we introduce "PolySphinx", a mix format that is a step towards efficient anonymous multicasting and allows a mix node to replicate the message payload to multiple recipients. We prove that PolySphinx does not compromise on the anonymity offered to users, while considerably reducing the latency of group messages: In a group with 25 members, the average latency drops from 6.1 s using the state-of-the-art *Rollercoaster* approach to 4.1 s using PolySphinx.**

1. Introduction

Group communication plays an important role in modern life [1]. However, as with most digital communications, metadata leaked to malicious parties can pose serious threats. For example, metadata may reveal that a user is associated with a group such as Alcoholics Anonymous or the extra-parliamentary opposition. Such a revelation can have serious implications for the user's social status and even safety.

Mix networks, a concept introduced by David Chaum [2], can be used to hide such metadata. In a mix network, a message is sent through a series of *mix nodes*, each of which collects incoming messages and relays them to the next node along the path in such a way that an observer cannot link incoming and outgoing messages and thus cannot identify sender and recipient. For their operation, mix networks define a *mix format*, which describes how the message and path are prepared for the mix nodes to operate. Sphinx [3] provides a modern mix format that is used in several mix networks [4]–[6]. It lacks dedicated support for group communication.

Group communication technically translates to *multicast*, i.e. the ability to send a single message to multiple recipients. Any one-to-one communication channel can naïvely support multicast communication by having the sender replicate the message and send multiple copies individually. However, this approach does not scale well: First, the sender must send the messages one at a time, possibly with limited frequency, which increases latency. Second, the sender's outgoing bandwidth consumption is multiplied by the number of recipients of the message, since the payload has to be sent multiple times. 'True' multicast, where the sender sends a single message that is replicated as close to the recipients as possible, would eliminate both drawbacks.

Many modern group communication applications such as video calling, live streaming, and collaborative productivity tools require both low latency [7] and high bandwidth [8]. Future applications such as the metaverse will only increase these requirements [9]. Due to longer paths and limited sending rates, mix networks inherently increase latency over direct communication. If mix networks are to be used to anonymize these group communication applications, multicast support in mix networks must be improved. The goal of this paper is to significantly reduce both the bandwidth and latency overhead of multicast communication in mix networks while maintaining strong privacy guarantees.

In this paper, we introduce *PolySphinx*, a new mix format based on Sphinx. The PolySphinx format allows the sender to instruct a mix node to act as a "replication node", copying a single message to multiple recipients. This alleviates bandwidth usage for the client and significantly reduces message latency: We show that in a group of 25 members, the average group size required by users [10], we can reduce latency from 6.1 s using the state-of-the-art approach [11] to 4.1 s using PolySphinx. Furthermore, PolySphinx improves the goodput in a network even for small replication factors of 2 as long as messages are larger than 435 Bytes.

Revealing replication information to a potentially malicious third party is a serious privacy risk. PolySphinx avoids this risk by providing the replication node with precomputed information about the remaining paths to the recipients, of which it can only decipher the immediate next hops. In doing so, PolySphinx does not compromise privacy: We prove that PolySphinx provides the same anonymity as previous work, even against a global active adversary and adversarial mix nodes.

In particular, we make the following contributions

- We introduce the PolySphinx mix format, which allows efficient multicasting of messages in mix networks.
- We prove the security of PolySphinx against a global active adversary.
- We provide an open source implementation of PolySphinx and a prototype of a PolySphinx-based mix network.

†. firstname.lastname@kit.edu
‡. firstname.lastname@neclab.eu

- We use and extend the existing mix network simulator of Hugenroth *et al.* [11] to evaluate PolySphinx and demonstrate its superior performance over existing approaches.

In Section 2 we introduce the required background and in Section 3 the related work. In Section 4 we define our threat model, before giving the general idea of PolySphinx in Section 5. In Section 6 we detail the construction of PolySphinx, after which we prove its security in Section 7 and evaluate its performance in Section 8. We give our conclusion in Section 9.

In this paper, we discuss a vanilla version of PolySphinx. We sketch possible extensions that may further improve performance and flexibility in Appendix D.

2. Background

In this section, we will first introduce some general notation and primitives that we will use throughout this paper, before introducing the Sphinx format that forms the basis of our extension, and the Loopix system that provides the context for our evaluation.

2.1. Notation

The following notation is common throughout the paper (cf. Appendix A for a notation table). We use κ as our security parameter. The bigger κ is, the more work has to be done by an attacker to break our security guarantees.

In a mix network, a message is relayed through multiple mix nodes before arriving at its destination. To prevent messages from being linked based on their size, we assume that each payload has a size of $l \in \mathbb{N}$ bits. We also assume that headers have a fixed size, which leads to an upper bound on the length of a message path, denoted as $r \in \mathbb{N}$. For PolySphinx, we assume that a mix node acts as a "replication node", creating multiple copies of a single incoming message. To hide the exact size of the recipient set, we always replicate by a factor of $p \in \mathbb{N}$, which we call the "replication factor".

For working with bit strings, that is elements of $\{0,1\}^*$, we define 0_k to be the string consisting of k null-bits, 0_0 to be the empty string, we define $a \mathbin{+\!\!+} b$ to be the concatenation of a followed by b, and we define $a_{[x..y]}$ to be the substring of a ranging from indices x to y, inclusive.

We define the Decisional Diffie-Hellman Problem (DDH) as the problem of distinguishing between tuples (g^a, g^b, g^{ab}) and (g^a, g^b, g^c), where g is the generator of a group [12]. We say that the DDH-Assumption "holds" in a group if the DDH Problem cannot be solved efficiently in this group. For the remainder of this paper, we use $\mathcal{G}$ to be a prime-order cyclic group in which the DDH-Assumption holds, $\mathcal{G}^*$ to be $\mathcal{G}$ without identity element, g to be the generator of $\mathcal{G}$ and q to be the order of $\mathcal{G}$.

2.2. Cryptographic Primitives

We require several hash functions, which we model as random oracles:

- We use a hash function $h_\mu : \mathcal{G}^* \to \{0,1\}^\kappa$ to derive keys for a message authentication code.
- We use a hash function $h_\rho : \mathcal{G}^* \to \{0,1\}^\kappa$ to derive keys for a pseudorandom number generator.
- We use a hash function $h_b : \mathcal{G}^* \times \mathcal{G}^* \to \mathbb{Z}_q^*$ to generate "blinding factors".
- We use a hash function $h_\mathcal{K} : \{0,1\}^\kappa \to \{0,1\}^\kappa$ to re-randomize key material.

To verify the authenticity and integrity of a packet's metadata, we require a message authentication code (MAC), denoted as $\mu : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$. The MAC takes a key and input string and outputs a message "tag". We model μ with a fixed key as a random oracle.

To encrypt the metadata, we require a pseudorandom number generator, denoted as $\rho : \{0,1\}^\kappa \to \{0,1\}^\infty$. We view ρ as internally re-seeding itself and being able to output arbitrarily many bits. We use the substring notation $\rho(\cdot)_{[x..y]}$ to denote a specific range of output bits from ρ.

To encrypt the message payload, we require a pseudorandom permutation family, keyed by a key from $\{0,1\}^\kappa$. We denote the permutation mapping as $\textsc{Enc} : \{0,1\}^\kappa \times \{0,1\}^l \to \{0,1\}^l$, and the inverse as $\textsc{Dec} : \{0,1\}^\kappa \times \{0,1\}^l \to \{0,1\}^l$.

2.3. Sphinx

Sphinx is a mix packet format introduced by Danezis and Goldberg [3]. Due to its compactness, it has been used as a base for multiple anonymous communication networks (ACNs) [4]–[6] and is a good foundation for our format extension.

To identify network participants, Sphinx assigns each user $u \in U$ an address in $\{0,1\}^{2\kappa}$ and each mix node $n \in N$ an identifier in $\{0,1\}^\kappa$. Sphinx also requires each mix node n to have a private key $x_n \leftarrow^R \mathbb{Z}_q^*$ and a public key $y_n = g^{x_n} \in \mathcal{G}^*$ used for Diffie-Hellman key exchanges. It is assumed that each network participant can obtain the public key of each mix node, but the public key infrastructure is beyond the scope of this paper.

When a user wants to send a message to another user, she first chooses a random path of mix nodes through the network to generate the Sphinx packet. A Sphinx packet is created in two parts: First, the header is created independently of the message payload. This header contains the information needed to route the packet through the mix network. Second, the message payload is prepared by encrypting it with all the keys of the mix nodes the packet will pass through, in reverse order. This method of payload encryption is commonly referred to as *onion encryption*, as each layer of encryption is "peeled off" one by one at each hop. The header is consumed by the mix nodes as the packet is forwarded hop by hop. The final mix node acts as the *exit node*, delivering the message to the intended recipient.

The Sphinx format ensures that each mix node only has access to the information necessary to forward the packet to the next hop in the path, and that outgoing and incoming packets cannot be linked without the private key of the node that processed them. This is achieved by having a

shared secret between the message sender and each mix node, which is used as the basis for header encryption and MAC generation. The shared secret is generated using a Diffie-Hellman key exchange, where the sender uses a one-time private key and embeds its part of the key exchange in the header, from which the mix node can then generate the shared secret. The pre-computed Diffie-Hellman part is called the *Sphinx group element*.

2.4. Multicast Support in Sphinx

Multicast messaging can be built on top of existing unicast messaging strategies, albeit inefficiently. The "naïve" approach is for the sender to simply send a copy of the message to each recipient individually. We call this approach the *sequential unicast* strategy.

The disadvantages of the sequential unicast approach are twofold: First, senders in a mix network are usually rate-limited, so sending messages one at a time results in huge latencies. Second, the sender has to waste bandwidth by sending the same message content multiple times.

MultiSphinx [11] provides a way to reduce these latencies by allowing a sender to combine multiple outgoing packets into a single "big" packet. A mix node along the path then splits the packet again into its inner packets. While this approach can reduce latency, it does not help with bandwidth consumption as the sender still has to send multiple copies of the same payload: The "big packet" must contain full replicas of the multicast packet, and smaller packets must be padded to "big packet size". The main advantage of MultiSphinx is that "sending fewer but larger messages allows lower power consumption on mobile devices" [11, p. 10].

Rollercoaster [11] does multicasting by dividing the work among group members: Recipients who have already received the message help distribute it to the rest of the group. This is done via a deterministically generated schedule that can be reconstructed by any group member, so that each member knows her "delivery responsibility". While this scheme scales well to large groups due to the exponential growth of the number of people reached in each step, there is still a large latency for smaller groups before the benefits of distributed work kick in.

2.5. Loopix

Sphinx itself is only a packet format and cannot protect against traffic correlation or frequency analysis attacks. Therefore, we consider Sphinx in the context of the Loopix anonymity system [5]: The Loopix system uses Sphinx as a packet format, but adds rules describing how packets must be mixed at mix nodes to avoid correlation attacks, as well as mechanisms to detect other types of active attacks on the network. To evaluate PolySphinx, we will build a Loopix variant that uses PolySphinx as the packet format and add rules that send PolySphinx packets for multicast messages.

The goal of Loopix is that an adversary cannot trace a message from its sender to its recipient as long as there is at least one non-adversarial node on the path.

Loopix assumes the existence of a global adversary who can monitor traffic on any network link and inject arbitrary messages. It further assumes that a subset of the mix nodes cooperate with the adversary, i.e., such nodes share their secret keys with the adversary or change their functionality at the adversary's command.

In Loopix, message mixing is done by delaying incoming messages by a random amount of time, chosen by the sender. This is a technique known as *stop-and-go mixing* [13]. In addition to the mixing, Loopix uses *drop* messages to hide user activity (a technique commonly called "cover traffic"), and *loop* messages that a user sends to herself to help with the detection of active attacks. The different kinds of messages are sent according to different sending rates.

3. Related Work

ACNs can be based on several techniques [14]: *Mix networks* [2] hide sender-recipient relations by having mix nodes collect and "shuffle" messages before relaying them, while *onion-routing* [15] takes a similar approach of relaying messages through a series of nodes, but skips the shuffling. *Dining cryptographer networks* (DC-nets) [16] ensure anonymity by having each participant provide a small piece of information necessary to reconstruct the message. Networks based on *private information retrieval* (PIR) typically rely on homomorphic encryption [17] to allow clients to anonymously retrieve messages from a server [18].

There are a number of systems that allow users to communicate anonymously over shared "dead drops". Systems like Vuvuzela [19] and Karaoke [20] obfuscate dead drop access via a mix network. Express [21] allows users to anonymously write to dead drops via distributed point functions.

In general, dead drop-based systems lend themselves well to multicast communication, since a single dead drop can be accessed by many users. However, current approaches have significant drawbacks: Vuvuzela and Karaoke use server-generated noise to hide dead-drop access patterns. Due to their anytrust mix network design, *every* server must generate enough noise to hide accesses by itself. When dead drops can be accessed by more than two clients, the variability in dead drop access patterns increases, further exacerbating the amount of noise needed. Express only hides who is writing to a dead drop. An adversary can trivially determine who is accessing the same dead drop, posing a significant privacy risk. Provisioning separate dead drops for each member of the group maintains privacy guarantees, but imposes more overhead on the sender, since the message must be replicated to each member's dead drop.

DC-nets [16] rely on a broadcast medium, which makes them inherently suited to multicast messaging, but also acts as a barrier to the efficiency and scalability of such systems. Modern iterations of this protocol family (e.g., Dissent [22], D3 [23], and Verdict [24]) aim to improve efficiency, but still lag behind mix networks and onion routers when it comes to real-world performance. D3, for example, takes several minutes to set up a shuffle round between a few hundred clients, and then another minute per message sent.

Onion routers like Tor [15] allow for low-latency communication, but make the network vulnerable to global adversaries and traffic or frequency correlation attacks [25] because they do not hide timing or size information of the routed packets. This allows an adversary to deanonymize users. While Tor assumes a one-to-one messaging model, multicast messaging can be built on top by using multicast trees combined with anonymous routing, as done by AP3 [26] and MTor [27]. These approaches can provide low-latency and bandwidth-efficient multicasting, but they suffer from similar weaknesses against global adversaries as Tor.

Nym[1] is a deployed mix network with many available mix nodes. At its core, Nym uses the Loopix [5] mixing scheme and the Sphinx [3] format. As Loopix, Nym does not provide efficient group messaging.

XRD [28] provides horizontal scalability for mix networks by dividing the network into smaller mix chains and ensuring that each pair of users shares a chain. A user in this system naturally sends multiple messages in each round (one for each of the chains she is part of). However, setting up the system so that each pair of group members shares a different mix chain is computationally expensive and might reveal group membership during the setup process.

M2 [29] is a mix network with built-in multicast capabilities. However, its design allows mix nodes to learn information about group sizes, which together with a-priori information can be used to identify users. It also requires users to subscribe to the content they want to receive, which is infeasible in a group where every member sends messages.

Various approaches to ACNs promise strong anonymity [5], [16], efficiency [15], [29], scalability [19], [23], or real-world implementations [6], [15]. However, none of them provide low-latency, high-bandwidth group multicast with strong anonymity guarantees against a global, active adversary to support, for example, real-time group video calls in anonymous groups.

4. Threat Model & Goals

In this section, we will introduce our assumed network setting, as well as our threat model.

4.1. Network Setting

Our network consists of two main types of participants: The users U and the mix nodes N. Each user accesses the network through a mix node of her choice, called a "provider node", which acts as a gateway between the user and the mix network. A user sends new messages into the network through her provider node, and the provider holds messages for the user so that she can retrieve them.

We also assume that there are *groups* of users who wish to communicate with each other, where a group is a subset of all users. A message sent by one group member should be received by all other group members. Groups in PolySphinx are created ad hoc by sending a single message to multiple recipients. The "group members" are therefore the recipients and the sender of the message combined. We consider group management to be an interesting but orthogonal problem to our work, and do not discuss it further. Instead, we assume that each group member has a list of other group members who should receive the message.

To facilitate group messaging, a mix node can act as a "replication node". In this case, the task of the mix node is to generate multiple copies of a message to send to multiple recipients. The sender chooses a random path through the mix nodes, the first node acts as the replication node. There is a maximum of one replication node per packet.

For simplicity, we assume that a sender always sends a message to exactly p recipients $\{r_1, \ldots, r_p\}$, corresponding to the replication factor. If the group contains more people, multiple independent packets can be created until all group members are included, and dummy recipients can be used to ensure that each packet has exactly p recipients. We leave the concrete implementation of these mechanisms to the protocol using PolySphinx.

4.2. Threat Model

Similar to Sphinx and Loopix, we assume the existence of a global adversary who can read all network traffic. Furthermore, we assume that the adversary can collaborate with a subset of the mix nodes, gaining access to their secret keys and being able to control their behavior. We call nodes that collaborate with the adversary *corrupt*, and nodes that do not collaborate *honest*.

We also assume that the group members trust each other, i.e., the adversary cannot corrupt them. Since each group member has a list of group members, corrupt recipients would trivially allow an adversary to learn this list. This model of trust within the group is shared by other multicast-related work, such as Rollercoaster [11] and M2 [29].

Our goals are similar to those of Loopix. Intuitively, we want to hide the sender-recipient relationship, i.e. which sender is connected to which (individual) multicast recipient. We also want to hide to which recipients a given multicast message is sent.

Recall that in our model, group members trust each other. Therefore, we do not aim to hide the sender-recipient relationship or the other recipients from the group members, and we assume that a group member will not collaborate with the adversary.

Our goals therefore are:

- Given that there is at least one honest node on the path between a sender and recipient, this sender-recipient relationship is protected. We call this property *single sender anonymity* (SSA).
- Given that there is at least one honest node between a recipient and the replication node, this recipient is protected from being exposed as a recipient to the adversary. This holds even if the adversary has managed to track the message to the other recipients or the sender.We call this property *reception exposure protection* (REP).

1. https://nymtech.net/ — Accessed December 16, 2024

We note that protecting the set of recipients without at least one honest node between the replication node and the recipient cannot be achieved in a design based on replication nodes: If we assume that the replication node and all the nodes behind it are corrupt, the adversary could simply cooperate with all these nodes and then link the recipients together. However, since an honest replication node itself or honest nodes behind a replication node also protect the sender-recipient relationship, we put the replication node at the start of the path and keep the paths behind it long. This way we achieve anonymous replication without any additional latency.

5. Design Overview

We introduce *PolySphinx* as a new mix format based on Sphinx [3], adding capabilities that allow a mix node to replicate a message in transit. This approach shifts the communication overhead from the sender (an end device with limited resources) to a mix node (a server with more resources). We refer to a mix node that replicates a packet as a "replication node" for this path. As with other mix formats, the sender determines the path a PolySphinx packet will take when it is created. Thus, the sender alone determines the position of the replication node within the path.

The sender can think of message distribution as laying a tree over the network of mix nodes, where the vertices are the mix nodes that the message should traverse and the leaves are the exit nodes. If a vertex has multiple children, it acts as a replication node. For visualization, see Fig. 1.

The position of the replication node is a trade-off between sender and recipient protection: The sender prefers to have a long path between her and the replication node, since an honest node there is on the path from her to *every* recipient. This unlinks her from all communication partners at once. Similarly, each recipient prefers to have a long path between herself and the replication node, since an honest node there will unlink her from the other recipients (and the sender).

While PolySphinx supports replication at any node, it is beneficial for the anonymity sets that all senders decide on the same position of the replication node in the path. As argued in Section 4.2, we analyze PolySphinx with the first node chosen as replication node. This placement maximizes protection for all recipients while keeping the overall path length short. When all recipients are unlinked from the communication, the sender-recipient relationship is automatically protected as well. During an ongoing chat, group members act as both senders and recipients, so current senders are incentivized to behave in a non-egoistical manner. If a sender does not adhere to this node placement, they can be detected (and punished) by the recipients.

Like MultiSphinx [11], we require a fixed replication factor of p to hide the exact information about the number of recipients. Thus, all senders must always build packets for p recipients. If they want to send their message to fewer recipients, they must add dummy recipients. PolySphinx provides clients with p fixed dummy addresses. Packets addressed to these clients are detected and discarded by the last node in the path.

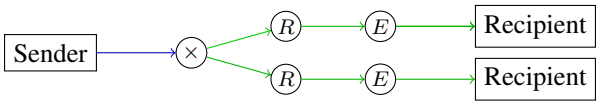

Figure 1: A multicast tree. The encircled $\times$ represents a replication node, the R a relay node and the E an exit node. Blue arrows denote the path pre-replication, green arrows post-replication.

For a mix network to provide anonymity, a mix node must not learn anything about the path of a message except for its immediate predecessor and successor nodes. This must also hold for the replication node, since we must assume that it may be corrupt. Therefore, we cannot simply send a list of recipients (in plain text) to the replication node, and we cannot rely on the replication node to generate new and correct onion-layered packets for us. Nor can we provide complete pre-generated packets to the replication node—as MultiSphinx does—because that would waste bandwidth. We therefore need to find a way for a node to replicate a packet so that it is both indistinguishable from its origin and its copies, and still decryptable by the intended recipient.

We solve the problem of how to get the recipient names to the replication node by providing it with pre-computed PolySphinx headers that encode the full path between the replication node and the final recipient. Since the sender generates these headers and chooses the paths that they encode, even a corrupt replication node cannot "peek" into them to learn more than the immediate next hop.

To solve the problem of actually replicating the payload, we build the onion "in reverse": Each mix node adds a layer of encryption to the packet, using a key that the sender provides in the header specifically for that node. For the replication node, the sender embeds multiple keys, so that the replication node can generate different copies of a single input packet.

To ensure that the recipients can decrypt the payload and access the plaintext, the sender could trivially embed the same keys that it provided to the mix nodes for the recipients. The recipients can then remove the encryption layers one by one. However, embedding many keys adds a lot of overhead, since each recipient would need κr bits of key material. We save space by generating the keys using a *key tree*, a deterministic scheme that allows a recipient to generate the same keys based on a single seed and a few bits of path information, adding only $\kappa + r \log_2 p$ bits of overhead.

6. PolySphinx

In this section, we will provide the detailed construction of the PolySphinx format.

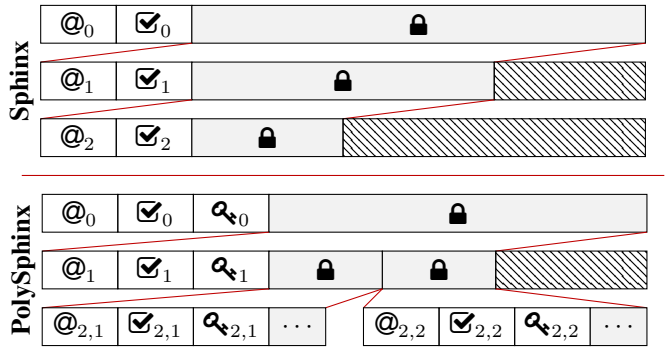

Figure 2: Header layout in Sphinx (top) and PolySphinx (bottom). The symbol @ denotes the address of the next node, ☑ a MAC, 🔑 an encryption key, and 🔒 a ciphertext. Hatched fields denote padding. Field sizes are not to scale.

6.1. Key Tree

The *key tree* is used by the sender and recipients to deterministically derive keys from a seed and path information. While there has been previous work on efficiently generating and updating group keys using *Asynchronous Ratcheting Trees* (ARTs) [30], [31], our requirements do not fit the use case for ARTs: Instead of a single shared group key, we need a deterministic way to generate multiple keys along a packet's path. We denote the key tree as $\mathcal{K}$ and define it recursively:

Definition 1 (Key Tree)**.** *Given a seed $S \in \{0,1\}^\kappa$, a hash function $h_\mathcal{K}$, and a replication factor $p \in \mathbb{N}$, we define the key tree $\mathcal{K}$ as follows:*

- *The root $\mathcal{K}[]$ is defined as $h_\mathcal{K}(S)$.*
- *Given a (possibly empty) path $x = x_1; x_2; \ldots$ and the corresponding $\mathcal{K}[x]$, the ith child $\mathcal{K}[x; i]$ is defined as $\mathcal{K}[x; i] \coloneqq h_\mathcal{K}(\mathcal{K}[x] + i)$ for $1 \leq i \leq p$.*
  The ; operator "concatenates" the path elements.

From the vertices of the key tree, the sender can now derive the keys for the mix nodes, and the recipient can use the seed S and the path choices to derive identical keys. It is important to note, however, that the key tree vertices $\mathcal{K}[\ldots]$ are not used directly as keys, since knowing such a value would allow the mix node to reconstruct the subtree below it, thereby compromising other keys. Instead, we treat the key tree as a confidential internal state and apply another hash function before embedding its values as keys.

6.2. Header Structure

Our header follows a similar general structure to the Sphinx header [3, p. 5] with the differences that 1) we can embed multiple lower level headers within a header and 2) we embed the derived key tree keys for each mix node — see Fig. 2 for a simplified visualization of the differences between Sphinx and PolySphinx. For the final recipient, we also embed the key tree seed and path information to allow reconstruction of the correct key tree.

We view our header as a 3-tuple $h = (\alpha, \beta, \gamma)$, where α is the so-called Sphinx group element, a precomputed Diffie-Hellman key exchange, β is the encrypted routing information, and γ is the message authentication code.

After decryption at a mix node, the routing information contains the address of the next mix node in the path n', the next message authentication code γ', the key σ with which to encrypt the message, and the routing information β' to pass to the next node. Additionally, there is a flag byte at the start to signal to the mix node whether it should act as a simple relay node, a replication node, or an exit node. For this we use three arbitrarily chosen but distinct bytes $f_\mathrm{R}, f_\times, f_\mathrm{E} \in \{0,1\}^8$.

6.3. Header Creation

We view header creation as a distinct step in the generation of a PolySphinx packet, since the header is created independently of the message content, and we generate headers recursively to embed them for use by the replication node.

The header creation algorithm is denoted as CREATEHEADER, which takes as input a path segment of mix nodes $n_0, \ldots, n_{\nu-1}$ that the packet should traverse, as well as the following additional information per node:

- For the final node $n_{\nu-1}$, the additional information is the routing data, denoted as Δ. If the created header is post-replication, this contains the recipient's address, such that the node can act as an exit node and deliver the message to the recipient. If the created header is pre-replication, then Δ contains the inner headers that the replication node must use.
- For the remaining nodes n_i, the additional information is the encryption key σ_i $(0 \leq i < \nu - 1)$ that the node should use to encrypt the message.

Now the sender first picks a random $x \leftarrow^R \mathbb{Z}_q^*$ and then computes the Sphinx group elements α_i, the shared secrets s_i and the blinding factors b_i for every node n_i along the path segment:

$$\begin{aligned}
&\alpha_0 = g^x, && s_0 = y_{n_0}^x, && b_0 = h_b(\alpha_0, s_0)\\
&\alpha_1 = g^{xb_0}, && s_1 = y_{n_1}^{xb_0}, && b_1 = h_b(\alpha_1, s_1)\\
&\quad\vdots && \quad\vdots && \quad\vdots\\
&\alpha_i = g^{xb_0b_1\ldots b_{i-1}}, && s_i = y_{n_i}^{xb_0b_1\ldots b_{i-1}}, && b_i = h_b(\alpha_i, s_i)
\end{aligned}$$

To build the routing information β, we need to know its size first, so we can properly encrypt it. For brevity, we define $\tau\,(\text{Pre})$ and $\tau\,(\text{Post})$ to be the size of the routing information $|\beta|$ in bits, pre- and post-replication respectively. The size $\tau\,(\cdot)$ depends on the maximum path length r and the replication factor p, and can be computed as follows:

$$\begin{aligned}
\tau\,(\text{Post}) &= r(8 + 3\kappa) + \kappa + r\lceil \log_2 p \rceil\\
\tau\,(\text{Pre}) &= (r-1)\cdot(8 + 3\kappa) + 8 + p\cdot(5\kappa + \tau\,(\text{Post}))
\end{aligned}$$

To ensure that mix nodes cannot tamper with the header, we include a Message Authentication Code (MAC) γ for each hop. The MAC is computed over the routing information β and embedded for each node along with the next hop address. Since each mix node will pad the header to ensure a constant length, we need to include the paddings in the MAC'ed data. This is done with filler strings ϕ_i that represent how the cumulative padding will arrive at node i, including the encryption that the mix nodes will apply. Filler strings are built iteratively:

$$\phi_i := \begin{cases} 0_0 & \text{if } i = 0 \\ (\phi_{i-1} \mathbin{+\!\!+} 0_{8+3\kappa}) \oplus \left(\rho\left(h_\rho\left(s_{i-1}\right)\right)_{[a..b]}\right) & \text{else} \end{cases}$$

where $a = \tau(\cdot) - (i-1)\cdot(8+3\kappa)$
and $b = \tau(\cdot) + 8 + 3\kappa - 1$

As the final step, the sender can now assemble the header:

$$f = f_{\mathrm{E}} \text{ if post-replication, } f_\times \text{ otherwise}$$
$$\beta_{\nu-1} = \left((f \mathbin{+\!\!+} \Delta) \oplus \rho\left(h_\rho\left(s_{\nu-1}\right)\right)_{[0..8+|\Delta|]}\right) \mathbin{+\!\!+} \phi_{\nu-1}$$
$$\text{For } 0 \le i < \nu - 1 : \beta_i = \hat{\beta}_i \oplus \rho(h_\rho(s_i))_{[0..\tau(\cdot)-1]}$$
$$\text{Where } \hat{\beta}_i = f_{\mathrm{R}} \mathbin{+\!\!+} n_{i+1} \mathbin{+\!\!+} \gamma_{i+1} \mathbin{+\!\!+} \sigma_i \mathbin{+\!\!+} \beta_{i+1\,[0..\tau(\cdot)-3\kappa-1]}$$
$$\text{For } 0 \le i \le \nu - 1 : \gamma_i = \mu\left(h_\mu\left(s_i\right), \beta_i\right)$$

The resulting header $(\alpha_0, \beta_0, \gamma_0)$ encodes the given path segment, and can be processed by the first node n_0. In the PolySphinx design, this header can not just be used to send a message directly, but it can also be embedded for a replication node to use.

6.4. Message Creation

Message creation in PolySphinx consists of three main steps: First, the sender must select paths through the mix network and determine where to replicate. Next, the sender creates the headers that encode the chosen paths. Finally, the sender attaches the payload to complete the packet and sends it to the first mix node.

If the sender wants to send a message m to p recipients $\{r_1, \ldots, r_p\}$, she first creates a *replication schedule*, i.e., a tree describing how the packet moves through the network, with the sender at the root, the mix nodes as vertices, and the recipients at the leaves (see Fig. 1). The replication schedule is generated by picking the randomly chosen first mix node as the replication node, and then for each recipient, picking a random path of mix nodes from the replication node to the recipient.

The sender then picks a random seed $S \leftarrow^R \{0,1\}^\kappa$ to generate the key tree $\mathcal{K}$ and maps the nodes of the replication schedule to the nodes of the key tree so that every path a message takes is mapped to a path in the key tree.

The sender then builds the headers from the inside out, first building the post-replication headers from the replication node to the exit nodes, and then wrapping them in the pre-replication header from the sender to the replication node.

For the post-replication headers, for each recipient r_i, the sender generates the header h_i using CREATEHEADER, passing the following parameters

- The routing information for the final node is $\Delta = S \mathbin{+\!\!+} P_i \mathbin{+\!\!+} r_i$, where P_i represents the path choices for this message in the key tree, and r_i represents the recipient's address.
- The sequence of mix nodes is the chosen path segment from the replication node to the exit node. For each mix node n_j, the associated key is $h_\mathcal{K}(\mathcal{K}[p_j])$, where p_j represents the path in the key tree up until n_j.

With the inner headers h_i generated, the sender can then go on to create the header for the path up until the replication node, again using our CREATEHEADER procedure. This time, the following parameters are used:

- The routing information is the concatenation of all inner headers, including the addresses of the next hops after the replication node and the encryption keys: $\Delta = (n_{0,0} \mathbin{+\!\!+} h_\mathcal{K}(\mathcal{K}[p_{0,0}]) \mathbin{+\!\!+} h_0) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} (n_{0,i} \mathbin{+\!\!+} h_\mathcal{K}(\mathcal{K}[p_{0,i}]) \mathbin{+\!\!+} h_i)$
- The sequence of mix nodes is the path segment from the sender's provider node up until the replication node, with the corresponding associated keys from the key tree.

To build the complete PolySphinx packet O, the sender encrypts her message m with the first key derived from the key tree and attaches the resulting payload ξ to the header:

$$\xi = \mathrm{ENC}\left(h_\mathcal{K}(\mathcal{K}[]), m\right) \quad O = ((\alpha, \beta, \gamma), \xi)$$

To hide their sending activity, senders must send at a fixed rate. In the absence of 'real' messages, cover messages must be generated. Cover messages are built as described above, but with a random m and dummy recipients. Note that in addition to PolySphinx's new group communications, usual unicast communications can be supported. In this case, two streams of cover traffic must be generated to hide the client's communication activity perfectly.

6.5. Message Processing at Mix Nodes

When a mix node n with private key x_n receives a packet $((\alpha, \beta, \gamma), \xi)$ where (α, β, γ) is the header and ξ is the payload, it starts by computing the shared secret $s = \alpha^{x_n}$ and the message authentication code $\gamma' = \mu(h_\mu(s), \beta)$.

If the mix node has seen the header before, the packet is discarded to prevent replay attacks. If the the MAC does not match ($\gamma \neq \gamma'$), the node also discards the packet to prevent tagging attacks.

Afterwards, the node can decrypt the routing information by applying the stream cipher ρ:

$$B = (\beta \mathbin{+\!\!+} 0_{8+3\kappa}) \oplus \rho(h_\rho(s))_{[0..\tau(\cdot)+7+3\kappa]}$$

The mix node can now cut off the flag byte $f = B_{[0..7]}$ from the beginning of B and use it to determine further action:

- If $f = f_{\mathrm{R}}$, the node should relay the packet to the next mix node. It extracts the first κ bits from the

beginning of B as n', then κ bits as γ', κ bits as σ and the remainder of B as β'. It then computes the next Sphinx group element $\alpha' = \alpha^{h_b(\alpha,s)}$ and can send the processed packet $((\alpha', \beta', \gamma'), \text{ENC}(\sigma, \xi))$ to n'.

- If $f = f_{\text{E}}$, the node is the designated exit node for a message. It extracts the first κ bits from the beginning of B as the seed for the key tree S, the next $r\lceil \log_2 p \rceil$ bits as the path P_i, and finally the next κ bits as the recipients address r_i.
  If the recipient's address corresponds to one of the designated dummy recipients, the node discards the packet. Otherwise, the node reconstructs the decryption keys $h_{\mathcal{K}}(\mathcal{K}[p_{i,0}])$, $h_{\mathcal{K}}(\mathcal{K}[p_{i,1}])$, ... using the seed S and path information P_i, decrypts the message using those keys, and delivers the plaintext to the intended recipient.
- If $f = f_{\times}$, the node is the designated replication node and should send p copies of the packet. To do that, the replication node extracts the p inner headers from B as shown in Fig. 3. Finally, for each extracted header h_i, it sends the packet $((\alpha_i, \beta_i, \gamma_i), \text{ENC}(\sigma_i, \xi))$ to the respective next hop n_i.

We write PROCESSPACKET as a shorthand for the algorithm described in this subsection. The output of PROCESSPACKET is a tuple with the flag byte and the extracted information:

- For packets to be relayed, it is the unwrapped inner packet and the address of the next mix node.
- For packets to be sent to the destination, it is the recipient's address and the payload.
- For packets to be replicated, it is a list of inner packets and corresponding addresses of the next mix nodes.

7. Proof of Security

Before presenting our proof, we introduce background on formal security in onion routing on which we base our properties. Afterwards, we will prove the security of PolySphinx in two steps: In the first step, we will prove the security at a non-replicating mix node, a property we call *intra-level indistinguishability*. In the second step, we will then prove that the security also holds at a replication node, a property we call *inter-level indistinguishability*. Since we assume the existence of an honest node on the path, and this node can be either a "normal" mix node or a replication node, our security holds in either case.

7.1. Formal Definitions

We will base our security proof on the formal work of Kuhn *et al.* [32]. In this subsection, we introduce their definition of *Layer-Unlinkability* (LU), and give intuition behind its meaning, so that we can adapt it to the multicast use case later.

The property of layer-unlinkability requires that a packet that passes through an honest node cannot be re-identified after it has been processed. Kuhn *et al.* formalize this property using a game that is played between a challenger and the adversary. Given a processed onion, the adversary has to decide whether it originates from a known onion or a random one generated by the challenger. If no efficient adversary can do so, the honest mix node successfully unlinked incoming (known) packets from outgoing ones.

Formally, the game is defined as follows:

Definition 2 (Layer-Unlinkability [32])**.**

1) *The adversary receives as input the challenge public key y_j, chosen by the challenger, and the name of the honest mix node.*
2) *The adversary may submit any number of packets O of her choice to the challenger. The challenger sends the output of the processed onion to the adversary.*
3) *The adversary submits a message m, a path $N = \{n_0, \ldots, n_\nu\}$ with the honest node at position j ($0 \leq j \leq \nu$) of her choice and key pairs for all nodes n_i, $i \neq j$.*
4) *The challenger ensures that the chosen values are valid and chooses $b \leftarrow^R \{0, 1\}$ randomly.*
5) *The challenger creates a packet O_0 with the adversary's input choice, as well as a second packet O_1 with a random message m' and a random, valid path N' that includes the subpath from the sender to the honest node n_j:*

   $$N' = \{n'_0, \ldots, n'_k = n_0, \ldots, n'_{k+j} = n_j, \ldots, n'_{\nu'}\}$$

   We denote the layers of O_0 and O_1 just before the honest node as O'_0 and O'_1.
6) *The challenger gives O_b and the output of processing O_0 at the honest node to the adversary.*
7) *The adversary may submit any number of packets O, $O \neq O'_0$, $O \neq O'_1$ to the challenger. The challenger sends back the output of processing O at the honest node.*
8) *The adversary produces guess b'.*

LU is achieved if any PPT adversary cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

Kuhn *et al.* also define *Tail-Indistiguishabilty*, which ensures that two onion-layered packets that share the same path after an honest node cannot be distinguished by an adversary, even if the recipient is corrupt. However, since our threat model does not consider adversaries that can corrupt recipients, this property does not apply to our work.

7.2. Intra-Level Indistinguishability

The first part of our security proof is to prove that messages that traverse an honest mix node are indistinguishable to an adversary.

We base our proof on Kuhn *et al.*'s security definitions. However, our adversary model differs from Kuhn *et al.*'s in

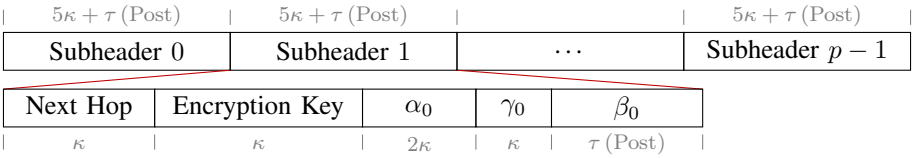

Figure 3: The structure of the inner headers. The gray numbers refer to the size of the element in bits.

that we assume trusted recipients. Therefore, we adapt their security properties to exclude attacks based on adversarial recipients. In addition to omitting Tail-Indistinguishability, we modify the Layer Unlinkability (LU) property to exclude tagging attacks on the payload. In such an attack, the adversary modifies the payload early in the path of the onion and detects the modification at the (adversarial) recipient.

Formally, we implement this by identifying the challenge packet only by its header, and always outputting an unmodified version of the challenge onion. Thus, the adversary cannot use observations from a challenge packet with a modified payload to win the LU game.

Note that in PolySphinx, the adversary can of course modify the payload of a captured onion. However, it gains no advantage from doing so because only the recipient can detect the altered payload. In our adversary model, this recipient is a trusted group member who will silently ignore the affected packet.

Note also that header protection is required by the properties and is implemented in PolySphinx via a MAC for each hop in the path.

We will provide our adapted version of the LU game in the following definition, and argue about its fitness for our purpose:

Definition 3 (Path-Untraceability (PU) game)**.**

1) *The adversary receives the challenge public key y_{n_j}, chosen by the challenger, and the identifier of the honest mix node n_j.*
2) *The adversary may submit any number of packets O to the challenger. The challenger sends the output of the processed packet as a reply.*
3) *The adversary submits a message m, a path $N = \{n_0, \ldots, n_j, \ldots, n_\nu\}$ with honest node at position j $(1 \leq j \leq \nu)$, key pairs (x_i, y_i) $(i \neq j)$ for the nodes, and final routing information Δ.*
4) *The challenger checks that the path is valid, that the public keys correspond to the private keys and that each node is assigned exactly one key pair. The challenger now draws $b \leftarrow^R \{0, 1\}$.*
5) *The challenger creates a packet O with the adversary's choices of message, path and destination, as well as a second packet O' with a random message m' $(|m'| = |m|)$, a random Δ' $(|\Delta'| = |\Delta|)$, and a random, valid path N' that includes the subpath from the sender to the honest node n_j:*

$$N' = \{n'_0, \ldots, n'_k = n_0, \ldots, n'_{k+j} = n_j, \ldots, n'_{\nu'}\}$$

Given the "outer packets" $O_0 = O$ and $O'_0 = O'$, we denote the "inner packets"

$$O_i = \textsc{ProcessPacket}(O_{i-1}) \text{ and}$$
$$O'_i = \textsc{ProcessPacket}(O'_{i-1})$$

6) *If $b = 0$, the challenger gives $(O_0, \textsc{ProcessPacket}(O_j))$ to the adversary. If $b = 1$, the challenger gives $(O'_k, \textsc{ProcessPacket}(O_j))$ to the adversary.*
7) *The adversary may submit any number of packets $O^? = (h^?, \xi^?)$ with $h^? \neq h_j$, $h^? \neq h'_{k+j}$. The challenger replies with $\textsc{ProcessPacket}(O^?)$.*
8) *The adversary wins the PU game if it correctly guesses the value of b.*

We say that our scheme provides PU-security if the advantage of any PPT adversary in winning the PU game is negligible.

Intuitively, if Path-Untraceability is achieved, then an attacker cannot link outgoing packets to incoming packets at an honest non-replicating node. This provides single sender anonymity (SSA) in cases where the honest node is not the replication node, and it is a necessary prerequisite for reception exposure protection (REP).

We can see that the modelled adversary matches our real-world expectations: By having access to the packets before and after the honest node, we capture the fact that the adversary can eavesdrop on the network traffic. By allowing the adversary to choose the key pairs of the mix nodes, we model the fact that the adversary can actively corrupt mix nodes. By allowing the adversary to submit packets to the challenger to get their processed form, we model the fact that the adversary can inject unrelated packets at the honest mix node and observe its behavior.

As explained above, the differences in payload tagging ability have no consequences due to our trust assumptions.

We will make the following assumptions in our security proof:

Assumption 1. *The underlying encryption scheme* ENC *is IND-CPA secure.*

Assumption 2. *The DDH-Assumption holds in $\mathcal{G}^*$.*

With these assumptions, we can give some intuition as to why PolySphinx achieves PU: The IND-CPA security of the underlying encryption scheme ensures that the adversary cannot learn information from the encrypted payload, and the DDH-Assumption tells us that she cannot learn anything from the group elements either. Since we model our hash

functions and MACs as random oracles, the adversary also cannot gain any information from their output values.

The full formal proof of the security is given in Appendix B.

Theorem 1. *The intra-level security of PolySphinx packets holds for any two messages if they are either both pre- or post-replication.*

Proof. The structure of PolySphinx headers pre- and post-replication only differs in the value and length of Δ. Since Path-Untraceability holds for arbitrary values of Δ_0 and Δ_1 as long as they are of the same length, it also holds for Δ_0 and Δ_1 that represent an inner PolySphinx header, or a recipient's address. □

7.3. Inter-Level Indistinguishability

In the previous section we proved that PolySphinx packets are indistinguishable as they pass through a mix node if they're both either pre- or post-replication. However, the replication node needs a closer look as it "unwraps" the inner headers and replicates the payload, turning pre- into post-replication packets.

We therefore define two additional games: The first game ensures that an outgoing packet at an honest replication node cannot be linked back to the incoming packet. This ensures that single sender anonymity (SSA) is achieved if the honest node is the replication node. We capture this requirement by having an adapted version of the PU game in which the challenger does not just encode a single path, but rather creates p inner headers and then wraps it in an outer header before giving it to the adversary.

Definition 4 (Multicast-Indistinguishability game)**.**

1–2) *as in Definition 3.*
3) *The adversary submits a message m, p paths $N_0, \ldots, N_{p-1}$, key pairs (x_i, y_i) for the nodes and final routing information $\Delta_0, \ldots, \Delta_{p-1}$ ($|\Delta_0| = \ldots = |\Delta_{p-1}|$).*
4) *as in Definition 3.*
5) *The challenger creates an outer packet O with inner paths $N_0, \ldots, N_{p-1}$, replication node n_j and the adversary's choices of message and destination, as well as a second packet O' with a random message m' ($|m'| = |m|$), random Δ'_i ($0 \leq i < p$, $|\Delta'_i| = |\Delta_i|$), and random, valid paths N'_i ($0 \leq i < p$). The path of the outer packet must end in the honest replication node n_j in both cases.*
6–8) *as in Definition 3.*

We say that our scheme is Multicast-Indistinguishable if the advantage of any PPT adversary in winning the Multicast-Indistinguishability game is negligible.

Intuitively, we expect Multicast-Indistinguishability to hold for PolySphinx: The inner headers are previously part of the routing information Δ, which we know is "private" and does not help the adversary in linking packets. The IND-CPA security of the payload encryption still holds, ensuring that no information is leaked.

More formally, we can prove this reduction:

Theorem 2. *PolySphinx achieves Multicast-Indistinguishability (Definition 4) against a PPT adversary.*

Proof. Our proof works by reduction: We show that given an adversary $\mathcal{A}$ that breaks Multicast-Indistinguishability, we can use it to construct $\mathcal{A}'$ that breaks the Path-Untraceability (PU) game for PolySphinx:

- $\mathcal{A}'$ calls $\mathcal{A}$ to get the choices of m, N_i, and Δ_i ($0 \leq i < p$).
- $\mathcal{A}'$ creates p headers h_i with values N_i and Δ_i, as described in Section 6.3.
- $\mathcal{A}'$ concatenates the headers h_i to generate the pre-replication header H, as described in Section 6.4.
- $\mathcal{A}'$ passes m, a random path N' and H (as Δ) to the challenger to receive the challenge packet O_c
- Now, O_c has the same structure that a multicast packet has in the Multicast-Indistinguishability game, so $\mathcal{A}'$ forwards O_c to $\mathcal{A}$ to get the guess b'.
- $\mathcal{A}'$ forwards b' to the challenger.

□

The second game ensures that two copies that originate from the same incoming packet at a replication node cannot be linked to each other. This ensures reception exposure protection (REP) in case there is no further honest node between the replication node and the recipient. We model this by asking the adversary to distinguish whether two packets have been independently created or if they originate from a single multicast packet.

Definition 5 (Multicast-Unlinkability game)**.** *The adversary selects two paths of mix nodes, N_0 and N_1, two final node information values Δ_0 and Δ_1 as well as a payload m. The challenger now chooses a random bit b and does one of two things:*

- *If $b = 0$, the challenger creates an outer packet with inner packets (N_0, Δ_0, m) and (N_1, Δ_1, m). The challenger then processes this packet and returns the two resulting packets (m'_0, m'_1) to the adversary.*
- *If $b = 1$, the challenger creates two separate packets with (N_0, Δ_0, m) and (N_1, Δ_1, m) and returns those to the adversary.*

The adversary should determine the value of b. The adversary may choose the secret keys of all selected mix nodes in N_0 and N_1.

Theorem 3. *PolySphinx achieves Multicast-Unlinkability (Definition 5) against a PPT adversary.*

Proof. The only difference in the creation of two separate messages versus a multicast message is how the embedded encryption keys are generated. However, as the encryption keys are the output of a random oracle, there's only a $1/2^{\kappa}$ chance that the adversary guesses the right input. Without

the input to the random oracle, the keys are indistinguishable from randomly chosen keys. □

7.4. Header Level

Finally, we want to note that an adversary can determine if an observed packet is pre- or post-replication. This is because PolySphinx achieves secure replication by embedding multiple headers in the replication node's layer. Thus, all headers prior to replication are larger than all headers after replication.

Without additional measures, an adversary can gain information about a user's activity (e.g., if the user is engaged in group communication) by observing their outgoing traffic. This issue can be mitigated by ensuring that sufficient cover traffic of both packet types is sent. Once a packet has been replicated, the resulting post-replication packets are indistinguishable from all other post-replication packets.

7.5. Security Summary

Recall that PolySphinx specifies that the replication node is the first node in the path. If the replication node is malicious, but there is at least one honest mix node between it and each recipient, PolySphinx achieves both it's privacy goals of single sender anonymity (SSA) and reception exposure protection (REP) due to Theorem 1. If the replication node is honest, PolySphinx achieves SSA and REP, even if *all* other nodes are malicious due to Theorems 2 and 3.

8. Evaluation

To evaluate the performance of PolySphinx, we will consider three different aspects:

- *Encryption benchmarks* to assess the computational effort that message processing takes, especially for larger messages.
- *Overhead analysis* to compare the bandwidth usage of PolySphinx compared to previous approaches.
- *Latency simulations* to assess the speedup we get in message delivery.

For a more concise evaluation, we evaluate PolySphinx for only one replication node per path. The possibility of arbitrary tree structures is discussed in Appendix D.2.

We note that latency and bandwidth are intertwined in a practical mix network like Loopix [5]: Since clients send messages at a fixed rate, they have a fixed bandwidth that they consume. Increasing the message rate also increases bandwidth usage, but reduces latency.

8.1. Encryption Benchmarks

We have performed encryption benchmarks to compare the computational overhead of PolySphinx with existing approaches and implementations. This benchmark was done to assess the computational effort required by users and mix nodes to process a message, and to ensure that our construction is usable in practice.

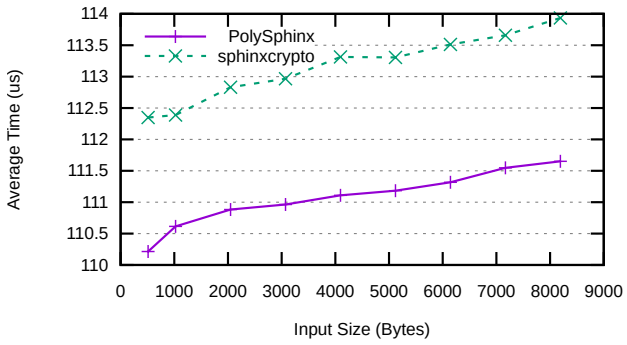

Figure 4: Message processing/unwrapping benchmark.

We expect PolySphinx to perform well here because the bulk of the packet is encrypted using a symmetric AES cipher, which benefits from hardware acceleration. This is especially true for processing packets at mix nodes, where there is little additional work beyond encrypting the payload. For header generation, the sender has to do extra work to compute all the shared secrets, which requires a lot of elliptic curve multiplications, and to derive the keys from the key tree. We therefore expect packet creation to be slower than current mix formats, but still negligible compared to other latencies.

For our benchmark, we have used an implementation of PolySphinx in Rust, as well as the `sphinxcrypto`[2] implementation of Sphinx. Note that any measurements for Sphinx are also applicable to MultiSphinx, as the same processing steps are taken in both protocols. Rust was chosen because it provides a memory-safe, fast language, and the Nym system is also implemented in Rust[3]. The benchmarks were executed on a modern laptop[4], and the results were collected via `criterion`[5].

For packet creation, we measured an average time of $1.56\,\mathrm{ms}$ for PolySphinx and $0.68\,\mathrm{ms}$ for Sphinx. For packet processing, we benchmarked the implementation for different payload sizes. The average time is between $110.2\,\mu\mathrm{s}$ for 512 Byte messages and $111.6\,\mu\mathrm{s}$ for 8192 Byte messages. In comparison, the Sphinx implementation takes $112.3\,\mu\mathrm{s}$ and $113.3\,\mu\mathrm{s}$ for the same message sizes. The packet processing results are summarized in Fig. 4.

Our benchmarks show that a practical PolySphinx implementation is fast, and that computation is not the bottleneck. We suspect that the total communication delay is dominated by other factors, such as the mixing delay or network delays.

2. https://crates.io/crates/sphinxcrypto — Accessed December 16, 2024
3. https://github.com/nymtech/nym — Accessed December 16, 2024
4. Lenovo Thinkpad E14 AMD G4, Ryzen 5 5625U, 16 GiB RAM
5. https://crates.io/crates/criterion — Accessed December 16, 2024

8.2. Overhead Analysis

Sending messages over a mix network increases bandwidth usage in two ways: First, the size of each message is increased because the mix format adds routing overhead. Second, cover traffic is added to obscure the sending activity.

Ignoring payload deduplication, PolySphinx increases both of these overheads compared to Sphinx: Packets are larger due to the inclusion of keys and subheaders. The size of the cover traffic increases to match the size of the real packets.

To see if the benefit of payload deduplication with PolySphinx makes up for the increased overhead, we compare the *goodput* of PolySphinx with that of Sphinx (using sequential unicast) in different network settings. We define goodput as the amount of payload bytes that reach the recipients compared to the total bytes transmitted by the sender, including cover traffic.

We expect that for small replication factors and small payload sizes, PolySphinx reduces goodput because the increased overhead in packets and cover traffic offsets the savings from de-duplicating the payload. However, for larger messages and larger replication factors, we expect PolySphinx to produce more goodput than Sphinx because the sender does not need to send copies of the payload.

For our evaluation, we wrote a script to compute the expected bandwidth consumption and goodput as a function of the send rates λ_P of payload messages and λ_C of cover messages, the replication factor p, the payload size l, and the fraction of multicast traffic ψ.

We set the sending rates to $\lambda_P = 2$ and $\lambda_C = 4$, consistent with the Rollercoaster evaluation [11]. We vary ψ to get a better understanding of how PolySphinx performs in non-optimal scenarios (i.e., $\psi < 1$). While λ_P includes all payload messages, ψ controls what fraction of these are multicast messages, i.e., messages sent to multiple recipients. In the case of PolySphinx, multicast messages are sent using pre-replication packets. The remaining $1 - \psi$ are unicast messages, sent directly using post-replication-like packets. The cover traffic for the different PolySphinx packet sizes is scaled accordingly. We limit our analysis to the bandwidth and goodput of one sender because we expect the bottleneck to be there, and we do not consider the total traffic in the mix network.

The experiment confirms our expectations that PolySphinx outperforms Sphinx. We see that for both formats, the proportion of goodput in the network increases with larger payload sizes, as the relative overhead of the packet header decreases.

For replication factors $p > 1$, the goodput of PolySphinx increases faster than that of Sphinx, and eventually overtakes Sphinx for all parameter combinations. For example, for $p = 2$, PolySphinx generates more goodput than Sphinx for messages larger than 435 Bytes when the sender sends only multicast messages. For $p = 5$, this break-even point is reduced to 198 Bytes. These results are shown in Fig. 5.

We further evaluated different combinations of sending rates. Increasing the amount of cover traffic per payload message adds more overhead to PolySphinx than it does to Sphinx, and thus increases the break-even point. However, the overall shape of the goodput curves does not change, they are simply stretched or compressed.

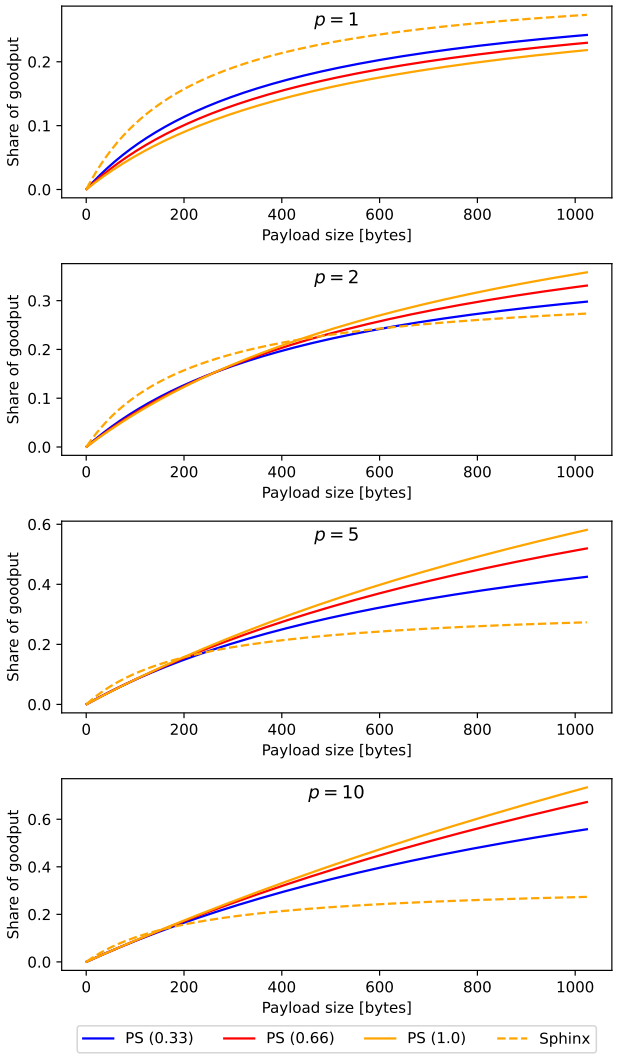

Figure 5: Share of goodput that the client sends, in relation to the replication factor p, the payload size and different shares of multicast traffic (fraction in parenthesis). In most configurations, PolySphinx produces more goodput than Sphinx, especially at high replication factors and large payload sizes. Note that the scale of the y-axis changes.

In addition, we can see that for favorable parameters (a large replication factor and a large payload size), a sender using PolySphinx can generate more goodput than the bandwidth she consumes. This is because a single message of her is replicated into many messages with large payloads at the replication node, which more than compensates for the cover traffic she needs to send.

We can say that PolySphinx improves the goodput of a network in realistic scenarios, especially for large payloads.

For the best savings, the parameters must be chosen according to the expected group sizes, payload sizes, and the amount of multicast traffic in the network. Additional evaluation comparing the bandwidth consumption and achieved sending rates of PolySphinx and Sphinx can be found in Appendix C.

8.3. Latency Simulation

We want to learn the average end-to-end latency of group messages for various group sizes to assess how well PolySphinx achieves its goal of enabling low-latency multicast messaging.

We expect PolySphinx to perform well in small groups, as our savings are linear: If without PolySphinx the sender has to send n messages, with PolySphinx she now has to send n/p messages.

For our simulations we used the simulation tool developed by Hugenroth *et al.*[6]. This tool simulates a Loopix mix network and user behavior, and it captures the end-to-end latencies of messages sent between group members. We extended the tool to support a PolySphinx-like distribution scheme and simulated 24 hours of mix network activity. We chose a group size of 25 as a minimum, as it is common in real-world environments [10], as well as some larger group sizes to analyze scaling behavior and to match the Rollercoaster simulations.

We compare PolySphinx to Rollercoaster as well as a naïve baseline where each users sends multiple packets per round. We note that we normalize the sending rates for PolySphinx and Rollercoaster to have the same bandwidth — that is, we assume a bandwidth of 6 Sphinx packets with 3 KiB of payload per second (matching the default Rollercoaster setup of 2 payload messages, 2 drop messages and 2 loop messages per second), and adjust the PolySphinx sending rates to match the same bandwidth usage. The baseline is not normalized for bandwidth and instead serves as a comparison point for optimal linear scaling.

In the simulated scenario, we assume that all protocol participants are always online. While the on/offline behavior of the participants does not affect PolySphinx, it does affect our comparison to Rollercoaster. Therefore, we chose the scenario where Rollercoaster performs best.

Our results show that for the smallest simulated group size of 25, the average latency drops from $6.1\,\mathrm{s} \pm 8.6\,\mathrm{ms}$ ($\pm$ standard error of the mean) using Rollercoaster to $4.1\,\mathrm{s} \pm 6.9\,\mathrm{ms}$ using PolySphinx, in a group of 32 from $6.4\,\mathrm{s} \pm 7.6\,\mathrm{ms}$ to $4.6\,\mathrm{s} \pm 7.1\,\mathrm{ms}$ and in a group of 64 from $7.1\,\mathrm{s} \pm 6.8\,\mathrm{ms}$ to $5.7\,\mathrm{s} \pm 7.6\,\mathrm{ms}$. For larger groups, Rollercoaster can provide lower average latencies: With 128 members, Rollercoaster achieves $9.0\,\mathrm{s} \pm 4.6\,\mathrm{ms}$ versus PolySphinx's $12.0\,\mathrm{s} \pm 10.6\,\mathrm{ms}$, and with 256 members, Rollercoaster achieves $10.5\,\mathrm{s} \pm 3.6\,\mathrm{ms}$ versus PolySphinx's $21.8\,\mathrm{s} \pm 13.9\,\mathrm{ms}$. These results are also summarized in Fig. 6.

6. https://github.com/lambdapioneer/rollercoaster — Accessed December 16, 2024

Note that we have omitted a comparison with Multi-Sphinx as well as Rollercoaster-with-MultiSphinx. As we normalize for bandwidth, MultiSphinx does not provide an advantage over Sphinx other than reducing the total number of packets sent. The simulation published with Rollercoaster disregards this, as it normalizes for outgoing message rate instead [11].

We can therefore say that PolySphinx reduces the latency of group messages in small to medium-sized groups. While PolySphinx has larger packets than Sphinx leading to a decreased sending rate, this is compensated for by the fact that a single message is replicated to multiple recipients at the replication node. To provide good scaling for large groups, we refer to Appendix D.1 for a discussion of a combination of PolySphinx and Rollercoaster.

9. Conclusion

Efficient anonymous group messaging is difficult, and existing solutions do not provide the desired guarantees of anonymity, efficiency, or scalability. We present the new *PolySphinx* design based on the idea of mix networks and *replication nodes*: Instead of naïvely having the sender perform message replication, we offload this task to a mix node, thereby reducing bandwidth consumption for the sender and improving message latency.

In designing PolySphinx, we must ensure that the replication mechanism does not leak any information about the sender or recipients to the replication node, since we must assume that this node can be corrupted by the adversary. This rules out simple solutions such as having the replication node create new onion-encrypted packets. We solve this challenge by having the sender prepare most of the necessary parts, and providing the replication node with precomputed headers and key material to use. We formally show that our construction produces packets that are indistinguishable to an adversary.

Using the Hugenroth *et al.* simulator, we evaluated the performance of PolySphinx in practice. For a realistic group size of 25 [10], we can reduce the end-to-end latency from 6.1 s using Rollercoaster to 4.1 s, and we increase goodput compared to the naïve approach for all messages larger than 198 B. The latency savings over Rollercoaster diminish with increasing group size, and break even at about 128 members.

The simulations, along with the benchmarks and overhead evaluations, show that PolySphinx provides an efficient way to add multicast messaging to mix networks, significantly reducing latency and bandwidth overhead. This enables the implementation of efficient group communication services such as messaging and media sharing applications in anonymous contexts.

Acknowledgements

We thank the anonymous reviewers for their feedback and their valuable input. This work has been funded by the Helmholtz Association through the KASTEL Security Research Labs (HGF Topic 46.23), and by funding of the

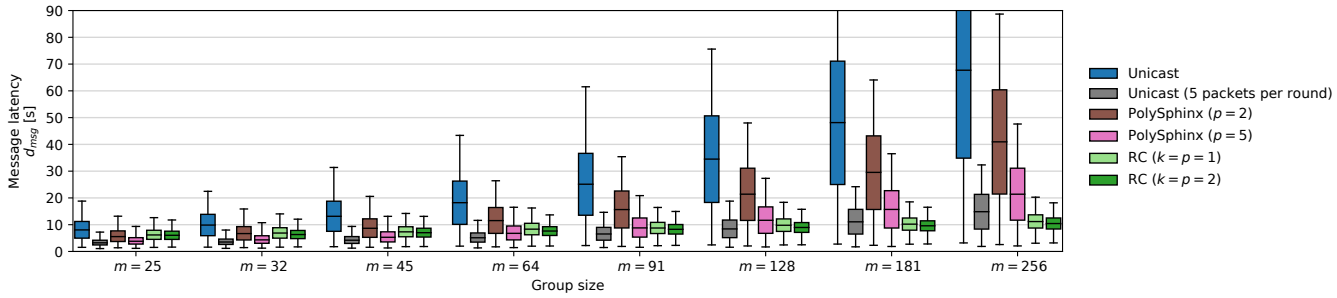

Figure 6: End-to-end latency of various multicast strategies. Note that the unicast with 5 packets per round is not normalized for bandwidth, the other strategies are. p is the multiplication factor for Rollercoaster, and the replication factor for PolySphinx. k is Rollercoaster's branching factor.

German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

Availability

We have uploaded the code for this paper to https://github.com/PolySphinx. This includes our PolySphinx implementation in Rust, the code for the benchmarks, the extended version of the simulator from Hugenroth *et al.* and the Jupyter notebooks used to do the overhead analysis. An overview of our API is given in Appendix E.

References

[1] M. Seufert, A. Schwind *et al.*, "Analysis of group-based communication in WhatsApp," in *MONAMI*, 2015.

[2] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, 1981.

[3] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *IEEE S&P*, 2009.

[4] C. Chen, D. Asoni *et al.*, "HORNET: High-speed onion routing at the network layer," in *ACM CCS*, 2015.

[5] A. Piotrowska, J. Hydes *et al.*, "The Loopix anonymity system," in *USENIX Security*, 2017.

[6] C. Diaz, H. Halpin *et al.*, "The Nym network: The next generation of privacy infrastructure," White paper, version 1.0, 2021.

[7] C.-L. Ignat, G. Oster *et al.*, "How do user groups cope with delay in real-time collaborative note taking," in *ECSCW*, 2015.

[8] J. Han, B. Li *et al.*, "A technical overview of AV1," *Proc. IEEE*, 2021.

[9] M. S. Elbamby, C. Perfecto *et al.*, "Toward low-latency and ultra-reliable virtual reality," *IEEE Network*, 2018.

[10] H. Halpin, K. Ermoshina *et al.*, "Co-ordinating developers and high-risk users of privacy-enhanced secure messaging protocols," in *SSR*, 2018.

[11] D. Hugenroth, M. Kleppmann *et al.*, "Rollercoaster: An efficient Group-Multicast scheme for mix networks," in *USENIX Security*, 2021.

[12] D. Boneh, "The Decision Diffie-Hellman problem," in *Algorithmic Number Theory*, 1998.

[13] D. Kesdogan, J. Egner *et al.*, "Stop- and- Go-MIXes Providing Probabilistic Anonymity in an Open System," in *Workshop on Information Hiding*, 1998.

[14] S. Sasy and I. Goldberg, "SoK: Metadata-protecting communication systems," Cryptology ePrint Archive, Paper 2023/313.

[15] R. Dingledine, N. Mathewson *et al.*, "Tor: The second-generation onion router," in *USENIX Security*, 2004.

[16] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *J. Cryptol.*, 1988.

[17] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Symposium on the Theory of Computing*, 2009.

[18] H. Corrigan-Gibbs, D. Boneh *et al.*, "Riposte: An anonymous messaging system handling millions of users," in *IEEE S&P*, 2015.

[19] J. V. D. Hooff, D. Lazar *et al.*, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *ACM SOSP*, 2015.

[20] D. Lazar, Y. Gilad *et al.*, "Karaoke: Distributed private messaging immune to passive traffic analysis," in *USENIX OSDI*, 2018.

[21] S. Eskandarian, H. Corrigan-Gibbs *et al.*, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *USENIX Security*, 2021.

[22] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable anonymous group messaging," in *ACM CCS*, 2010.

[23] D. Wolinsky, H. Corrigan-Gibbs *et al.*, "Scalable anonymous group communication in the anytrust model," in *EUROSEC*, 2012.

[24] H. Corrigan-Gibbs, D. I. Wolinsky *et al.*, "Proactively accountable anonymous messaging in Verdict," in *USENIX Security*, 2013.

[25] J. Nurmi and M. Niemelä, "Tor de-anonymisation techniques," in *NSS*, 2017.

[26] A. Mislove, G. Oberoi *et al.*, "AP3: Cooperative, decentralized anonymous communication," in *ACM SIGOPS European Workshop*, 2004.

[27] D. Lin, M. Sherr *et al.*, "Scalable and Anonymous Group Communication with MTor," *Proc. Priv. Enhanc. Technol.*, 2016.

[28] A. Kwon, D. Lu *et al.*, "XRD: Scalable messaging system with cryptographic privacy," in *USENIX NSDI*, 2020.

[29] G. Perng, M. Reiter *et al.*, "M2: Multicasting mixes for efficient and anonymous communication," in *ICDCS*, 2006.

[30] K. Cohn-Gordon, C. Cremers *et al.*, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *ACM CCS*, 2018.

[31] K. Emura, K. Kajita *et al.*, "Membership privacy for asynchronous group messaging," Cryptology ePrint Archive, Paper 2022/046.

[32] C. Kuhn, M. Beck *et al.*, "Breaking and (partially) fixing provably secure onion routing," in *IEEE S&P*, 2020.

	Meaning		Meaning
0_k	Bit string of k zero bits	u	User
B	Decrypted routing data	U	Set of all users
f_*	Flag bytes	x_n	Mix node private key
g	DH group generator	y_n	Mix node public key
$\mathcal{G}$	DH group	α	Sphinx group element
$\mathcal{G}^*$	DH group without identity	β	Packet routing data
h_*	Hash functions	γ	Packet MAC tag
$\mathcal{K}$	Key tree	Δ	Final routing data
l	Payload size (bits)	σ_i	Node encryption key
m	Message	κ	Security parameter
n	Mix node identifier	λ_*	Sending rates
N	Set of all mix nodes	μ	MAC
O	Onion-wrapped packet	ν	Path length of the packet
p	Replication factor	ξ	Packet payload
$p_{i,j}$	Single path choice in P_i	ρ	Pseudorandom generator
P_i	Path choices for a packet	τ	Length of the header
r	Maximum path length	ϕ	Filler string
r_i	i'th recipient	ψ	Fraction of multicast traffic
S	Key tree seed		

TABLE 1: Notation overview

Appendix A. Notation

An overview of the symbols used is provided in Table 1.

Appendix B. Security Proof

We prove that PolySphinx achieves PU via a hybrid argument over five games. In this process, we will show that we can modify the PU game gradually to remove any values from the packets that could make them distinguishable, arriving at a game that provides no information that could give the adversary any advantage. We also show that our adversary cannot distinguish those modified games, and therefore, it must also have at most a negligible advantage in the original PU game.

The games of our proof are as follows, each game having a slight modification to the previous one:

Hybrid (Path-Untraceability)**.** *In game H_0, the challenger creates the header and message as specified in PolySphinx, without changes.*

Hybrid. *In game H_1, the challenger modifies the creation process and replaces the keys from the key tree with randomly chosen values.*

Since PROCESSPACKET *only uses those embedded keys, there is no need to further modify the behavior of* PROCESSPACKET *or the mix nodes.*

Hybrid. *In game H_2, the challenger changes the way the shared secret at the honest node is computed and chooses $s_j \leftarrow^R \mathcal{G}^*$ instead of using $s_j = y_{n_j}^{xb_0 \dots b_{j-1}}$.*

In step 6 of the game, we modify PROCESSPACKET *to use the new value s_j to process the challenge packet at node n_j.*

This is equivalent to $\mathbf{G}_1$ in the Sphinx proof.

Hybrid. *In game H_3, the challenger further modifies the creation process to not include proper encryptions of the message payload, but instead output the encryption of random bytes as payload, therefore removing any information that the payload might have carried.*

Since the content of a packet has no influence on its processing at a mix node, there is no need to further modify the behavior.

Hybrid (Ideal World)**.** *In game H_4, the challenger replaces the header values β_j and γ_j with random values, and saves the original values as β'_j and γ'_j.*

We modify PROCESSPACKET *that if β_j is given, it will reject the packet if the extracted MAC does not equal γ_j, and it will internally use β'_j to process the packet.*

This is equivalent to $\mathbf{G}_2$ in the Sphinx proof.

We now begin the hybrid argument by showing that for any PPT adversary, each of the above games is indistinguishable (up to a negligible probability) from its predecessor:

Lemma 1. *No PPT adversary can distinguish H_1 from H_0.*

Proof. In H_0, the key that the sender embeds for the mix node is the hash of a key tree node, $H(\mathcal{K}[\dots])$. In H_1, it is a random value. Since we model hash functions as random oracles, the only way to distinguish between those two cases is if the adversary knows the input to the hash function. However, $\mathcal{K}[\dots]$ itself is either the output of a random oracle, or the randomly chosen initial seed — neither of which is embedded in the message.

As the adversary cannot distinguish between the output of a random oracle with unknown input, and a randomly chosen value, the adversary also cannot distinguish H_1 and H_0. □

Lemma 2. *No PPT adversary can distinguish H_2 from H_1.*

Proof. The difference between H_2 and H_1 is the way in which the shared secret is calculated. If a distinguisher $\mathcal{D}$ exists that could differentiate between those two games, we could use it to construct a DDH distinguisher $\mathcal{D}_{\text{DDH}}$:

- $\mathcal{D}_{\text{DDH}}$ receives the challenge tuple (p, q, r).
- $\mathcal{D}_{\text{DDH}}$ generates a key pair $x_j \leftarrow^R \mathbb{Z}_q^*$, $y_j = g^{x_j}$.
- $\mathcal{D}_{\text{DDH}}$ passes y_j as the challenge public key to $\mathcal{D}$ to receive the values for m, N and Δ.
- $\mathcal{D}_{\text{DDH}}$ constructs a PolySphinx packet using q as the j'th Sphinx group element ($\alpha_j = q$) and r as the j'th shared secret ($s_j = r$), and the values of m, N and Δ.
- $\mathcal{D}_{\text{DDH}}$ now passes the constructed message to $\mathcal{D}$ and receives the response bit b.
- $\mathcal{D}_{\text{DDH}}$ passes b to its challenger.
- Any valid oracle queries for PROCESSPACKET can be answered by $\mathcal{D}_{\text{DDH}}$ by using the private key x_j.

As per Assumption 2 however, no such distinguisher $\mathcal{D}_{\text{DDH}}$ can exist, and therefore the two games are indistinguishable.

We note that the adversary cannot use the oracle to gain an advantage, as passing the challenge message is forbidden,

and constructing a different message (α, β', γ) would require it to forge the MAC — which would require knowledge of the shared secret. □

Lemma 3. *No PPT adversary can distinguish H_3 from H_2.*

Proof. The only difference between H_3 and H_2 is the fact that in H_3, random bytes are encrypted instead of the actual payload.

If a distinguisher $\mathcal{D}$ exists that could differentiate between those two games, we could use it to construct an IND-CPA distinguisher $\mathcal{D}_{\text{IND}}$:

- $\mathcal{D}_{\text{IND}}$ generates a key pair $x_j \leftarrow^R \mathbb{Z}_q^*, y_j = g^{x_j}$ and a random node identifier n_j.
- $\mathcal{D}_{\text{IND}}$ sends y_j as the challenge public key to $\mathcal{D}$.
- $\mathcal{D}_{\text{IND}}$ receives from $\mathcal{D}$ the choices of m_0, the path $N_0 = \{n_0, \ldots, n_{\nu-1}\}$ and Δ_0.
- $\mathcal{D}_{\text{IND}}$ draws a message m_1 randomly with $|m_1| = |m_0|$.
- $\mathcal{D}_{\text{IND}}$ forwards m_0, m_1 to the IND challenger and receives the challenge c.
- $\mathcal{D}_{\text{IND}}$ creates a PolySphinx header $h = \text{CREATEHEADER}(N_0, \Delta_0)$ and then sends (h, c) to $\mathcal{D}$.
- $\mathcal{D}_{\text{IND}}$ receives the guess b from $\mathcal{D}$ and forwards it to the IND challenger.
- Any valid oracle queries for PROCESSPACKET can be answered by $\mathcal{D}_{\text{IND}}$ by using the private key x_j.

We note that $\mathcal{D}_{\text{IND}}$ "acts" as if the secret key that the challenger uses internally is the first randomly drawn key in H_1. Since the key is never exposed to the adversary, the adversary cannot know whether $\mathcal{D}_{\text{IND}}$ actually has the key.

Our $\mathcal{D}_{\text{IND}}$ would win as often as $\mathcal{D}$, however no such distinguisher $\mathcal{D}_{\text{IND}}$ can exist per Assumption 1. Therefore $\mathcal{D}$ can not exist and the two games are indistinguishable. □

Lemma 4. *No PPT adversary can distinguish H_4 from H_3.*

Proof. Per definition, β is combined with the output of a pseudorandom generator ρ and as such indistinguishable from randomness. Likewise, μ is the output of a random oracle with a random input key and therefore indistinguishable from randomness as well. □

Lemma 5. *No adversary can have a non-negligible advantage to win H_4.*

Proof. No value in the header or payload of the packet in H_4 is dependent on the challenge bit b. Therefore, no adversary can have an advantage over random chance. □

Theorem 4. *PolySphinx fulfills the definition of Path-Untraceability from Definition 3 under Assumption 1 and Assumption 2.*

Proof. The proof follows from Lemma 1 – Lemma 5. □

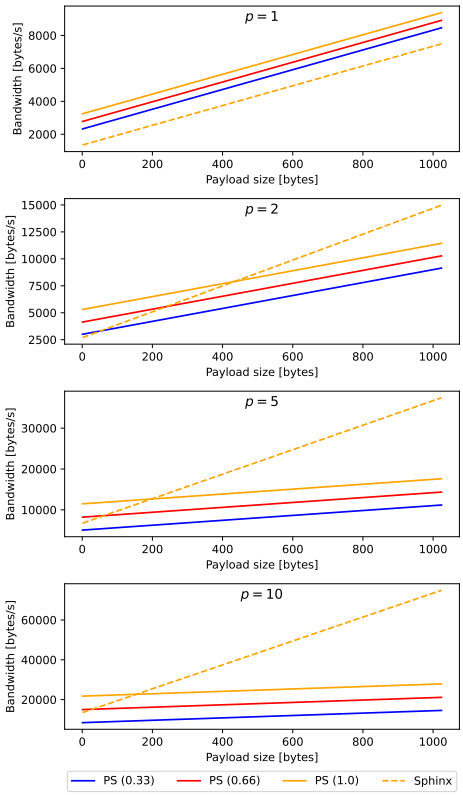

Figure 7: Bandwidth consumption of PolySphinx and Sphinx for comparable sending rates. The number in parenthesis gives the share of multicast packets.

Appendix C. Extended Evaluation

The evaluation in Section 8.2 compares PolySphinx and Sphinx in terms of the goodput share of the total bandwidth, assuming that sending rates are constant. We see that PolySphinx produces a bigger share of goodput, but the overall bandwidth consumption increases as well.

Therefore, we provide the raw bandwidth values for PolySphinx and Sphinx in Fig. 7. We can see that the bandwidth increases for both formats, but the increase for Sphinx is way steeper: As the sender has to send p copies of the payload, increasing the size of the payload leads to a p-fold bandwidth increase.

In addition to evaluating goodput and bandwidth, we further evaluate how many messages PolySphinx can send given a constant bandwidth limit that must not be exceeded.

We expect that using PolySphinx means that less packets can be sent due to the increased packet size, but due to the replication mechanism, a single packet will result in multiple payload packets. This mechanism makes up for the decreased sending rate.

For this part of the evaluation, we change our experiment slightly: Instead of fixing the sending rates, we now assume a fixed bandwidth at the sender's link and calculate how

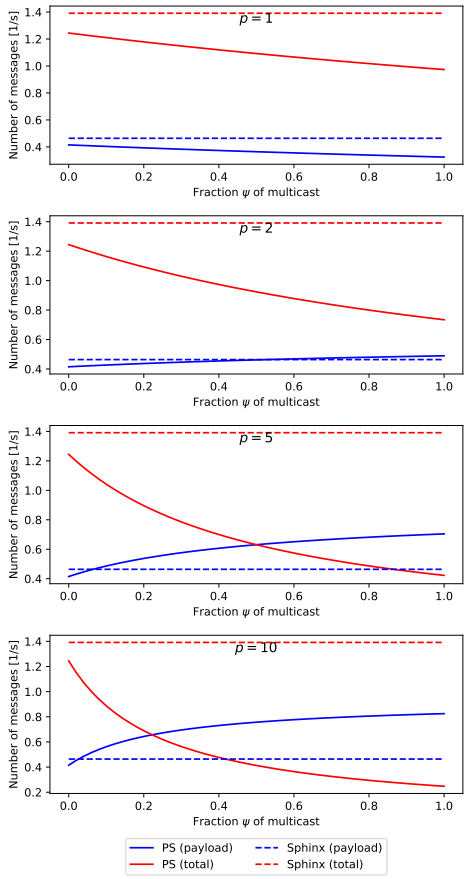

Figure 8: Number of messages sent per second for a fixed bandwidth of $1\ \frac{\text{KiB}}{\text{s}}$ and a payload size of $512\,\text{B}$.

many packets (Sphinx or PolySphinx) she can send using that bandwidth. Out of those packets, we consider $1/3$ to be payload messages, and $2/3$ to be cover traffic, which is in line with the payload to cover ratio in Section 8.2. We then vary the remaining parameters, i.e. the payload size l, the replication factor p and the share of multicast traffic ψ.

Our results show that the number of payload packets that PolySphinx produces quickly surpasses the number of payload packets that Sphinx produces when multicast is used. For a bandwidth limit of $1\ \frac{\text{KiB}}{\text{s}}$, a payload size of $l = 512\,\text{B}$ and a replication factor of $p = 2$, PolySphinx produces more payload packets when around half of the packets are multicast packets. For bigger replication factors and payload sizes, PolySphinx performs even better.

The detailed results for our experiment with $1\ \frac{\text{KiB}}{\text{s}}$ and $l = 512\,\text{B}$ are shown in Fig. 8. In addition, we provide Fig. 9 with an overview of how the improvement increases when increasing the payload size and the replication factor.

In summary, we can say that our extended evaluation confirms our previous findings. Figure 10 shows a summary for which parameter combinations PolySphinx performs better than Sphinx, for comparable sending rates.

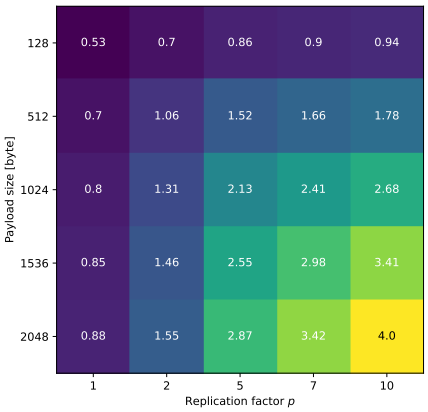

Figure 9: Factor by which the number of payload messages increases when comparing PolySphinx and Sphinx. Numbers less than 1 represent parameter combinations where PolySphinx performs worse, numbers greater than 1 represent parameter combinations where PolySphinx performs better. All numbers were calculated with a bandwidth limit of $1\ \frac{\text{KiB}}{\text{s}}$ and a multicast fraction of $\psi = 1$.

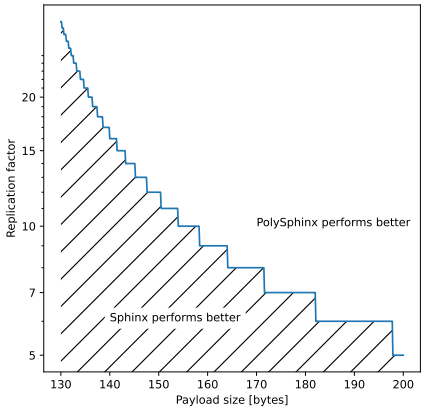

Figure 10: Summarizing results of how to pick the replication factor for a given payload size. Note that the y-axis is in logarithmic scale.

Appendix D. Extensions

There are multiple ways to extend PolySphinx, to make it more efficient, more flexible, or more practical. In this section, we want to describe possible extensions that could be a part of future work or practical PolySphinx implementations.

Packet Format	Description
Sphinx	Base Format
MultiSphinx	Combines multiple Sphinx packets into one
PolySphinx	Deduplicates payload for multicast packets
Sending Strategy	
Sequential Unicast	Replicate all copies at the sender
Rollercoaster	Group members help with distribution

TABLE 2: Categorization of Rollercoaster and PolySphinx as orthogonal technologies.

D.1. Combining Rollercoaster and PolySphinx

Rollercoaster and PolySphinx are two orthogonal techniques: Rollercoaster can be thought of as a *sending strategy*, while PolySphinx is a *mix format*. The sending strategy defines the order in which the messages are sent, while the format defines how the messages are packed. Examples of this categorization are shown in Table 2.

Currently, we compare Rollercoaster combined with (Multi)Sphinx [11] and sequential unicast combined with PolySphinx. A system combining Rollercoaster with PolySphinx inherits the advantages of both: It scales well for large groups by distributing the sending tasks among the group members, and each member sends the payload only once by using the multicast feature of PolySphinx.

However, Rollercoaster requires a small amount of data per recipient, different in each payload, to signal the role that the recipient should play in the distribution of the message. Therefore, a slight modification to PolySphinx is necessary to allow a small amount of data to be sent to each recipient individually in the header.

The performance evaluation and security aspects of such a combined system are interesting future work.

D.2. Nesting PolySphinx Arbitrarily

Instead of using one replication node only, it is possible to extend the idea of PolySphinx and allow arbitrary nesting. Generalizing the post- and pre-replication packets, we introduce the concept of "levels", denoted as λ. The post-replication packets then correspond to a level of $\lambda = 0$, while the pre-replication packets correspond to a level of $\lambda = 1$. A packet of level $\lambda = i$ ($i > 0$) is split into p packets of level $\lambda = i - 1$, each of which is later split into p packets again (if $i > 1$). This allows a single packet from the sender to reach p^λ recipients without the need for duplicated payloads.

The downside of this approach is that the different levels are distinguishable by packet size. This means that the network needs to have more cover traffic to accommodate all possible levels in the system, or each packet needs to be padded to the same size (e.g., via dummy recipients).

An evaluation of such a system needs to account for the increased overhead in cover traffic when comparing arbitrary nesting. Building and evaluating a system that is tailored to a specific distribution of multicast traffic is an interesting challenge for future work.

Furthermore, the threat model becomes more complicated as we now have different segments between the sender and the first replication node, the first replication node and the second replication node, ..., and the last replication node and the recipient. Analyzing the different possible locations for honest nodes, as well as which properties of the group communication they protect, provides further directions for research.

Appendix E. API

We provide an implementation of the PolySphinx format in Rust. This implementation allows an user to create PolySphinx headers and packets and use them as a base for a mix network. In this section, we give a quick overview over the (simplified) interface to our PolySphinx library.

To represent paths through the mix network, we introduce the `Path` type:

```
enum Path {
    Direct(Vec<Mixnode>, Recipient),
    Multi(Vec<Mixnode>, Vec<(Mixnode, Path)>),
}
```

A `Direct` path represents a path through the mix network with a single recipient at the end, while a `Multi` path represents a path to a replication node, with a number of "inner" paths.

The main interface consists of two functions, representing the algorithms described in Section 6.4 and Section 6.5:

```
fn create_polyheader(path: &Path)
    -> Result<(Header, EncryptionKey)>

fn unwrap_header(priv_key: &PrivateKey, header:
↪  &Header)
    -> Result<Command>
```

The first function `create_polyheader` implements the functionality to create a (possibly nested) header for a given path through the mix network. It draws a random seed for the key tree and recursively creates the inner headers. The return values are the created header and the first encryption key, which can be used to prepare the packet's payload.

The second function `unwrap_header` implements the decryption of a layer of the header given a node's private key. The return value depends on the node's task: It can be a `Command::Relay`, a `Command::Destination` or a `Command::Multicast` — similar to the possibilities described in Section 6.5.

To aid the implementation of clients and mix nodes, we also provide functions that deal with the payload encryption:

```
pub fn prepare_payload(key: &EncryptionKey, data:
↪  &[u8]) -> Result<Vec<u8>>;
pub fn recrypt_payload(key: &EncryptionKey, data:
↪  &[u8]) -> Result<Vec<u8>>;
pub fn decrypt_payload(key: &DecryptionKey, data:
↪  &[u8]) -> Result<Vec<u8>>;
```

Appendix F.
Meta-Review

F.1. Summary

This paper presents a new mix-net package format, PolySphinx, that can be used for efficient multi casting in mix networks for mix networks. Multi casting is particularly important for group communication where users send the same message to all other users in the group. When using PolySphinx a special mix-net node – instead of the client – is responsible for replicating the package, reducing latency for receivers and bandwidth consumption for the client.

F.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) This paper constructs an interesting new method for letting a mix-node create p independently routed copies of a single broadcast message. Prior work either replicated the entire package, including payload, resulting in no reduction in communication cost; or instead relied on other recipients to forward messages.
2) Performance evaluations of the new proposal confirm practical performance of the new package format in the setting of group communication, especially when the group size is not too big.
3) The paper includes a privacy proof that shows that the desired properties of the original Sphinx package format are maintained by PolySphinx.